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 2016–23 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\textsuperscript{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, acknowledgements, and disclaimer 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.\textsuperscript{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.

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

\textsuperscript{1}The first edition was published in 2003, though it experienced a number of edits and adjustments in subsequent years.

\textsuperscript{2}The article by David M. Cook titled “Computation in Undergraduate Physics: The Lawrence Approach” and appearing in the American Journal of Physics (Am. J. Phys. 76, 321–326 (April-May 2008) describes these efforts in some detail.
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—are 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 ifthen and imakeidx packages, to assemble the PDF files that permit the selected assemblage subsequently to be viewed on a screen or printed.

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 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 $HEAD, $IDLHEAD, and $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 Local Guide when commands or statements illustrated in the book are submitted to the user’s machine.

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 introduce4

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.

4The 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
• 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.

Chapter 12 begins with a brief orientation to LSODE—a large and well-tested publicly available FORTRAN program for solving systems of ordinary differential equations—and Chapter 16 begins with a general discussion of multigrid techniques for solving partial differential equations and then provides an orientation to MUDPACK—a large collection of publicly available FORTRAN programs for solving partial differential equations using those techniques. Each chapter then illustrates the use of the tools on which it focuses to address several representative problems. 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).

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 either IDL or MATLAB, either MAPLE or Mathematica, and LATEX; and finally covers either the IDL or MATLAB and either the MAPLE or the Mathematica portions of the chapters on ordinary PYTHON explain why Chapters 4–12 in the first edition have become Chapters 6–14 in the second edition.
differential equations (ODEs), integration and root finding. The chapter on Numerical Recipes, the
FORTRAN and/or C portions of the chapters 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.

David M. Cook
Appleton, Wisconsin
14 February 2023
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. Stemmer, LU '97, Mark Nornberg, LU '98, Scot Shaw, LU '98, Jim Truitt, LU '98, Eric D. Moore, LU '99, Teresa K. Hamele, LU '00, Danica Dralus, LU '02, Ryan T. Peterson, LU '03, Scott J. Kansinski, LU '04, Michelle L. Milne, LU '04, Lauren E. Kost, LU '05, Claire Weiss, LU '07, and Erik Garbackik, LU '08—all Lawrence students who, as undergraduates, contributed during the first decade and a half to the evolution of this book and the accompanying manual of solutions to representative exercises. While current versions in most instances deviate—often considerably—from the initial drafts to which these students contributed, there can be no denying that their contributions have played an important role in the evolution of the use of computational resources in the Lawrence curriculum and the evolution of this book. I thank them warmly and sincerely for their assistance. All of these students have given permission for me to use whatever has over the years become of their contributions in this publication.

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

- numerous students (beyond those specifically named in the previous paragraph) who have enrolled in my courses over the years and who, directly and indirectly, have made comments that have influenced this book.
- my departmental colleagues J. Bruce Brackenridge, John R. Brandenberger, Jeffrey A. Collett, Matthew R. Stoneking, and Paul W. Fontana, who have listened to my sometimes lengthy observations about matters computational and curricular and offered valuable criticisms.
- Wolfgang Christian (Davidson College), Robert Ehrlich (George Mason University), and A. John Mallinckrodt (California Polytechnic Institute–Pomona), each of whom participated in a 1990 Sloan-supported conference on computation in advanced undergraduate physics that I organized at Lawrence and each of whom as a consultant throughout the writing project has offered many suggestions and constructive criticisms.
- all those faculty members\(^1\) 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. Walsek (Lebanon Valley College), William H. Ingham (James Madison University), and Tim Young (University of North Dakota). In July, 2002 (first workshop): Albert Batten (United States Air Force Academy), Andrea Cox LU '91 (Beloit College), Brian Cudnik (Prairie View A and M), Craig Gunsul (Whitman College), Kevin Lee (University of Nebraska), Kam-Biu Luk (University of California, Berkeley), Daryl Macomb (Boise State University), Viktor Martisovits (Central College), Donald Miller (Central Missouri State University), Dorn Peterson (James Madison University), Robert Ragan (University of Wisconsin – Lacrosse), Shafiq Rahman (Allegheny College), Rahmathullah Syed (Norwich University), Jorge Talamantes (California State University – Bakersfield), Mark Timko (Elmhurst College), Paul Tjossem
providing a critical examination of the evolving manuscript and offered numerous suggestions for improvement—in addition to finding many previously undetected typographical glitches.

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

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

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

- Donald Knuth, originator of TeX;
- Leslie Lamport, originator of \LaTeX;
- Han The Than, originator of pdfTeX, 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 \makeidx 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 xdv, 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 Petea (University of Richmond), David Grumbine (St. Vincent College), Kristi Hendrickson L ’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 Pogorzelski (SUNY at Geneseo), Chandra Prayaga (University of West Florida), Bruce Richards (Oberlin College), David Seely (Albion College), Adam Smith (Hillsdale College), Walther Spjeldvik (Weber State University), Julie Talbot (State University of West Georgia), Paul Thomas (University of Wisconsin–Eau Claire), Srinivasa Venugopalan (SUNY at Binghamton), and Alma Zook (Pomona College).
• The many contributors to the MiKTeX project;
• The developers, authors, and maintainers of WinEdt, the text editor used during the latter years of writing CPSUP; and
• The developers and maintainers of ps2pdf for converting PostScript files to PDF and the developers and maintainers of pdfcrop for pruning excessive white borders from PDF files.
• Radical Eye Software, which holds the copyright on dvips, a program for converting .dvi files to PostScript.

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

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

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

• The MathWorks, Inc., for permission to incorporate in this book and distribute IDL source code for the routines ludiffeq.23 and ludiffeq.45, which code uses algorithms patterned after those used in 1991 in the MATLAB routines ode23 and ode45.

• Wayne Landsman, author of the IDL routines qsimpson and trapzd in the IDL Astronomy User’s Library, for permission to use those routines as the basis for the routines luqsimp and lutrapzd and to distribute the source code for luqsimp and lutrapzd as supplements to this book.

• Research Systems (later Exelis Visual Information Solutions and now part of Harris Geospatial Solutions), Incorporated, for permission to use portions of any RSI-supplied and/or edited .pro code—most particularly evident in RSI contributions to ludiffeq.23.pro, ludiffeq.45.pro, and luqsimp.pro—and to use the IDL name and trademark.

• Numerical Recipes Software (a) for permission to use the names and calling sequences of several Numerical Recipes routines at various places in this book, (b) for permission to refer to the C header files nr.h and nrutil.h and the file nrutil.c containing assorted utilities used by various C recipes, and (3) for permission to use the names and calling sequences of several IDL routines that are derived from Numerical Recipes routines (and for the use of which Research Systems Incorporated has permission from Numerical Recipes Software).

• William Chia-Wei Cheng, author of TGIF, for permission to reproduce in the appendix on that program several of the icons used in its many screen displays.

Sixth, I wish to acknowledge and thank the W. M. Keck Foundation for three grants [#880969 (1988), #931248 (1993), #0115795 (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. Any small conference held at Lawrence in 1990 and supported by the Alfred P. Sloan Foundation stimulated thinking among the participants about

2 Specifically flmoon, xflmoon, caldat, julday, xjulday, asevar, xasevar, rk4, xrk4, rkq, rkck, mmid, bsstep, rkdumb, odeint, trapzd, xtrapzd, qtrap, xqtrap, qsimp, qromb, point, rtbis, xrtbis, rtnewt, xrtnewt, rsafe, xrtsafe, zbrak, gaussj, ludcmp, lubksb, tridag, svdcmp, svbksb, mnewt, newt, and broydn (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.
uses of computers in upper-division undergraduate physics. All of these grants have contributed in many ways to the developments at Lawrence that have culminated in the writing of this book. In particular, the NSF CCLI-EMD grant made in February, 2000, supported my sabbatical while I finalized the text of (the first edition of) this book. That grant also supported four week-long summer faculty workshops that have, on the one hand, provided constructive feedback on a succession of drafts and, on the other hand, enhanced awareness nationally of this book and of the developments at Lawrence.

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

The statements described in the various chapters of this book have been tested extensively but have certainly not been tested with all versions of all software packages on all possible platforms with all possible versions of the underlying operating systems. Differences from version to version of the software packages, from operating system to operating system, and from platform to platform exist. This brief section identifies the versions of the various programs that have been tested and the operating systems and platforms on which those tests have been carried out. That the behavior of other combinations of version, operating system, and platform will conform in every detail to that herein described can, of course, not be guaranteed. One can, however, have some confidence that the behavior in combinations not explicitly tested will not differ enormously from that described herein—except that newer versions of a software package may well have features not implemented in earlier versions (and occasionally a feature or specific syntax available in an earlier version has been removed altogether from more recent versions). With reasonable confidence, one can presume that the commands and syntax and features described in this book will work on other platforms with the tested versions of the programs and with subsequent versions. Statements herein that exploit features implemented for the first time in the tested versions will, of course, not be accepted in earlier versions, but those “glitches” should not be numerous or extensive. Where, in the months and years since the original draft was created, I have become aware of such glitches, I have made appropriate updates and subsequent productions have incorporated those updates.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 MAXIMA codings herein have been fully verified with
  - MAXIMA Version 5.36.1 and wxMAXIMA Version 15.04.0 on a Hewlett-Packard platform running Windows 7,
  - MAXIMA Version 5.38.1 and wxMAXIMA Version 16.04.2 on a Hewlett-Packard platform running Windows 7, and
  - MAXIMA Version 5.36.1 and wxMAXIMA Version 12.01.0 on a Hewlett-Packard platform running the Fedora 17 implementation of LINUX.

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

¹The date of production of each version of CPSUP is displayed at the top of the cover page on that version. I have maintained a dated list of edits made to the source files, so changes made after the date of production of a particular version of CPSUP and a subsequent production of that version can readily be identified for anyone who wishes to update an outdated production. Generally, updated productions fairly promptly replace the previous production at parc.aapt.org/curricula/cpsup. Versions dated between 10 and 31 January 2021 provide the base. Edits made after 31 January 2021 are recorded in the file of edits.
- MAXIMA 5.42.0 and wxMAXIMA 18.10.1

on a Hewlett-Packard platform running Windows 10.

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

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

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

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

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

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

- Except where otherwise noted in the text, the PYTHON codings have been fully verified with
  - PYTHON 2.7.16 from the Anaconda2 distribution installed on a Hewlett-Packard platform running Windows 10, using the Anaconda2 prompt and also using the Anaconda2 Python Shell.\(^2\)
  - PYTHON 3.7.3 from the Anaconda3 distribution installed on a Hewlett-Packard platform running Windows 10, using the Anaconda3 prompt and also using the Anaconda3 Python Shell.\(^3\)

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

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

\(^2\)See the Local Guide for ways to bring up the prompt and the shell in your environment.

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

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

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

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

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

## 1 Preliminaries  
1.1 An Orientation to Computers  
1.1.1 A Simple Responsive Machine  
1.1.2 Character Codes  
1.1.3 The ASCII Character Set  
1.1.4 Representation of Data in a Computer  
1.2 Files and Directories  
1.3 Operating Systems  
1.4 Glossary, Conventions, and Understandings  
1.5 Assumed Background  
1.5.1 The Gamma Function  
1.5.2 The Laplace Transform  
1.6 Licensing Issues  

## 5 Introduction to PYTHON  
5.1 Beginning a PYTHON Session  
5.2 Basic Entities in PYTHON  
5.2.1 Data Types  
5.2.2 Variable Names  
5.2.3 Assignment of Values to Variables  
5.2.4 Commands  
5.2.5 PYTHON Modules  
5.3 A Sampling of PYTHON Capabilities  
5.3.1 Using PYTHON Interactively  
5.3.2 Creating and Examining Strings, Tuples, Lists, and Sets  
5.3.3 Interrogating and Adjusting the Symbol Table  
5.3.4 Creating and Examining Arrays  
5.3.5 Creating and Manipulating Matrices  
5.3.6 Complex Numbers  
5.3.7 Solving Linear Equations  
5.3.8 Data and Method Attributes of Variables  
5.3.9 A First Graph  
5.4 Loops, Logical Expressions, and Conditionals  
5.5 Eigenvalues and Eigenvectors  
5.6 The PYTHON Edit Window and Stored Programs  
5.6.1 A Quick Example of a Stored Program  
5.6.2 Reading Data From a File; Radioactive Decay  
5.6.3 Writing Data To a File; Formatting Output  
5.7 Defining Functions  

Table of Contents xv
5.7.1 Format of a Function Definition .................................................. 44
5.7.2 The Function `luplot` ................................................................. 45
5.7.3 Using Global Variables ............................................................... 46
5.8 On-Line Help ..................................................................................... 48
5.9 Graphing Scalar Functions of One Variable ....................................... 48
  5.9.1 The Basic Strategy ....................................................................... 50
  5.9.2 Plotting Several Graphs on One Set of Axes ............................... 50
  5.9.3 Polar Plots and Custom Axes ....................................................... 52
  5.9.4 Multiple Plots In a Single Window .............................................. 53
  5.9.5 Plotting Experimental Data ......................................................... 55
5.10 Making Hard Copy ........................................................................... 58
  5.10.1 . . . of Text ................................................................................ 58
  5.10.2 . . . of Graphs ............................................................................ 58
5.11 Graphing Scalar Functions of Two Variables ...................................... 59
  5.11.1 A Preliminary: The Function `meshgrid` .................................... 60
  5.11.2 Surface Plots: The Commands `plot_wireframe` and `plot_surface` . 61
  5.11.3 Contour Plots: The Command `contour` ..................................... 64
  5.11.4 Shaded Surfaces: The Command `plot_surface` Again ............... 66
  5.11.5 Features of the Figure Window .................................................. 67
  5.11.6 Functions of Two Variables in Polar Coordinates ....................... 68
5.12 Graphing Scalar Functions of Three Variables .................................... 70
  5.12.1 Reduction to Two-Dimensional Displays .................................... 71
  5.12.2 Isosurfaces ................................................................................ 72
5.13 Graphing Vector Fields ..................................................................... 75
  5.13.1 The Function `quiver` ............................................................... 76
  5.13.2 More Elaborate Displays ............................................................. 77
  5.13.3 Three-Dimensional Vector Fields .............................................. 78
5.14 Animation ....................................................................................... 79
  5.14.1 The Basic Command .................................................................. 80
  5.14.2 Motion of a Vibrating String I ................................................... 80
  5.14.3 Motion of an L-Shaped Drumhead ............................................. 82
  5.14.4 Trajectory of a Projectile in Uniform Gravity .............................. 84
  5.14.5 Motion of a Vibrating String II .................................................. 86
5.15 Advanced Graphing Features ............................................................ 87
  5.15.1 Mathematical Symbols and Greek Characters ............................. 87
  5.15.2 Space Curves ............................................................................ 87
  5.15.3 Using Multiple Windows ......................................................... 88
  5.15.4 Scaling so Circles are Circles ..................................................... 89
  5.15.5 Customizing Axes .................................................................... 89
  5.15.6 Alternative Specifications of Color ............................................ 91
  5.15.7 Interactive Plotting ................................................................. 92
5.16 Miscellaneous Occasionally Useful Tidbits ....................................... 93
  5.16.1 Find, Change Current Directory .............................................. 93
  5.16.2 Setting the Search Path ............................................................. 94
  5.16.3 Customizing PYTHON ............................................................. 94
  5.16.4 Restoring PYTHON’s Initial State ............................................. 96
  5.16.5 Reading Data from the Keyboard .......................................... 97
  5.16.6 Error Trapping ................................................................. 98
5.17 References ..................................................................................... 98
5.18 Exercises ....................................................................................... 99
  5.18.1 Writing PYTHON Statements .................................................. 99
  5.18.2 Finding Eigenvalues and Eigenvectors .................................... 100
  5.18.3 Graphing Scalar Functions of a Single Variable ....................... 101
  5.18.4 Graphing Scalar Functions of Two Variables ......................... 103
  5.18.5 Graphing Scalar Functions of Three Variables ....................... 104
  5.18.6 Graphing Vector Fields ........................................................... 105
  5.18.7 Animation .............................................................................. 107
  5.18.8 Animation .............................................................................. 108
5.A Listing of `plotradio.py` ................................................................. 109
5.B Listing of `rcfilter.py` ................................................................. 109
| 5.C | Listing of stwave.py | 110 |
| 5.D | Listing of drumhead.py | 111 |
| 5.E | Listing of projectile.py | 113 |
| 5.F | Listing of stringanim.py | 113 |

### 6 Introduction to MAXIMA

| 6.1 | Beginning a MAXIMA Session | 116 |
| 6.2 | On-Line Help | 117 |
| 6.3 | Basic Entities in MAXIMA | 119 |
| 6.4 | Variable Names in MAXIMA | 119 |
| 6.5 | Expressions in MAXIMA | 119 |
| 6.6 | Binding Expressions to Variables; Defining Functions | 121 |
| 6.7 | Using MAXIMA | 122 |
| 6.7.1 | ... for Arithmetic | 123 |
| 6.7.2 | ... for Algebra | 124 |
| 6.7.3 | ... for Complex Variables | 126 |
| 6.7.4 | ... for Trigonometry | 127 |
| 6.7.5 | ... for Algebraic Equations | 128 |
| 6.7.6 | ... for Basic Calculus | 130 |
| 6.7.7 | ... for Laplace Transforms | 131 |
| 6.7.8 | ... for Ordinary Differential Equations | 132 |
| 6.7.9 | ... for Lists, Matrices, and Arrays | 135 |
| 6.7.10 | ... for Vector Calculus | 137 |
| 6.8 | Loops, Logical Expressions, and Conditionals | 139 |
| 6.9 | Command Files | 141 |
| 6.10 | High-Resolution Plotting | 145 |
| 6.11 | Making Hard Copy | 152 |
| 6.11.1 | ... of Text | 152 |
| 6.11.2 | ... of Graphs | 154 |
| 6.12 | Output in \LaTeX Format | 154 |
| 6.13 | Animation | 155 |
| 6.14 | Using the Workbook | 155 |
| 6.15 | Pattern Matching | 156 |
| 6.15.1 | Search and Replace: Small Angle Approximations | 156 |
| 6.15.2 | Simplification Rules | 158 |
| 6.15.3 | Pattern Matching Functions | 159 |
| 6.16 | Miscellaneous Occasionally Useful Tidbits | 159 |
| 6.16.1 | Specifying directories | 159 |
| 6.16.2 | MAXIMA’s Default Directory | 159 |
| 6.16.3 | MAXIMA’s Search Path | 160 |
| 6.16.4 | Customizing MAXIMA | 161 |
| 6.16.5 | Restoring MAXIMA’s Initial State | 161 |
| 6.16.6 | Placing Text in Graphs | 161 |
| 6.16.7 | Space Curves | 162 |
| 6.17 | Miscellaneous Commands | 162 |
| 6.18 | References | 163 |
| 6.19 | Exercises | 164 |

### 9 Introduction to Programming

| 9.1 | Components of a Programming Language | 176 |
| 9.1.1 | Variables, Variable Names, and Use of Memory | 176 |
| 9.1.2 | Essential Action Statements | 178 |
| 9.1.3 | Logical Expressions and Conditions | 179 |
| 9.1.4 | Syntactic Wrinkles | 180 |
| 9.1.5 | Essential Control Structures | 180 |
| 9.2 | Sample Short Algorithms | 184 |
| 9.3 | Two Larger Algorithms | 186 |
| 9.3.1 | Solving Laplace’s Equation | 186 |
| 9.3.2 | File Output/Input | 188 |
| 9.11 | Solving Laplace’s Equation with PYTHON | 191 |
# Table of Contents

11 Solving Ordinary Differential Equations ........................................... 215

11.1 Sample Problems ............................................................................. 216
  11.1.1 Projectile in a Viscous Medium .................................................. 216
  11.1.2 Chain Radioactive Decay ............................................................. 219
  11.1.3 Exponential and Logistic Growth .................................................. 220
  11.1.4 Forced, Driven, Damped Harmonic Oscillation ........................... 220
  11.1.5 An LRC Resonant Circuit ............................................................ 222
  11.1.6 Coupled Oscillators .................................................................... 222
  11.1.7 Motion under Central Forces ....................................................... 223
  11.1.8 Standing Waves in a String ........................................................... 225
  11.1.9 The Schrödinger Equation in One Dimension .............................. 226

11.2 Laplace Transforms ........................................................................ 227

11.3 Solving ODEs Symbolically with MAXIMA ..................................... 228
  11.3.1 Projectile in a Viscous Medium .................................................... 230
  11.3.2 Logistic Growth .......................................................................... 232
  11.3.3 Damped Harmonic Oscillator ....................................................... 234
  11.3.4 Chain Radioactive Decay ............................................................. 234
  11.3.5 Coupled Oscillators .................................................................... 236
  11.3.6 Standing Waves in a String ........................................................... 237
  11.3.7 Infinite Depth Quantum Well ........................................................ 238
  11.3.8 Finding Solutions by Series Methods .......................................... 239

11.6 Algorithms for Solving ODEs Numerically .................................... 241
  11.6.1 Euler’s Method ............................................................................ 242
  11.6.2 Improved Euler Method ............................................................... 244
  11.6.3 Runge-Kutta Methods ................................................................. 245
  11.6.4 Assessing Accuracy .................................................................... 247
  11.6.5 Adaptive Methods ...................................................................... 249
  11.6.6 Multistep Methods ..................................................................... 250

11.10 Solving ODEs Numerically with PYTHON ..................................... 250
  11.10.1 Using Elementary Commands ..................................................... 251
  11.10.2 The PYTHON Command `odeint` ................................................. 254
  11.10.3 Defining ODEs for `odeint` ......................................................... 254
  11.10.4 Chain Radioactive Decay ............................................................ 255
  11.10.5 Damped Harmonic Oscillator ....................................................... 258
  11.10.6 Planetary Orbits ......................................................................... 260
  11.10.7 Standing Waves in a String ........................................................... 263
  11.10.8 The Quantum Harmonic Oscillator ......................................... 266
  11.10.9 Keywords that Modify `odeint` .................................................... 271
  11.10.10 py-files in the Public Library ................................................... 272
  11.10.11 Other ODE Solvers in PYTHON .................................................. 273

11.11 Solving ODEs Numerically with MAXIMA .................................... 274

11.16 Exercises ....................................................................................... 276
# TABLE OF CONTENTS

15 Solving Partial Differential Equations 383

15.1 Sample Problems ........................................... 385
  15.1.1 Motion of a String: The Wave Equation .................... 385
  15.1.2 Heat Flow: The Diffusion Equation ......................... 388
  15.1.3 Steady State Heat Flow: The Laplace Equation ................. 389
  15.1.4 Other Situations ......................................... 389
  15.1.5 Classification of Second-Order PDEs ...................... 390
  15.1.6 Equations of Fluid Dynamics: Navier-Stokes’ Equations ...... 391
    15.1.6.1 Conservation of Mass: The Equation of Continuity .... 392
    15.1.6.2 Newton’s Second Law .................................. 393
    15.1.6.3 A Special Case: Non-viscous Flow .................... 395
    15.1.6.4 A Second Special Case: Sound Waves ............... 396
  15.1.7 A General 1D Equation ................................ 397
  15.1.8 A General 2D Equation ................................ 398

15.2 Finite Difference Methods (FDMs) in One Dimension ............. 399

15.6 Using PYTHON to Solve 1D PDEs via an FDM .................... 401
  15.6.1 A General Coding ....................................... 401
  15.6.2 An Example: Simple Harmonic Motion ....................... 402

15.9 Finite Element Methods (FEMs) in One Dimension ............... 405
  15.9.1 Discretizing the Domain: Preprocessing ..................... 406
  15.9.2 Selecting Interpolation or Shape Functions ................. 406
  15.9.3 Formulating the Equations for a Single Element .......... 408
  15.9.4 Assembling the System of Equations ....................... 410
  15.9.5 Incorporating the Boundary Conditions .................... 412

15.13 Using PYTHON to Solve 1D PDEs via an FEM .................... 414
  15.13.1 A General Coding ....................................... 414
  15.13.2 An Example: Simple Harmonic Motion ....................... 416

15.16 Finite Difference Methods (FDMs) in Two Dimensions .......... 419
  15.16.1 The Wave Equation ...................................... 419
  15.16.2 The Diffusion Equation .................................. 422
  15.16.3 The Laplace Equation .................................... 424

15.20 Using PYTHON to Solve 2D PDEs via an FDM .................... 427
  15.20.1 The Wave Equation ...................................... 427
  15.20.2 The Diffusion Equation .................................. 434
  15.20.3 The Laplace Equation .................................... 441

15.23 Finite Element Methods (FEMs) in Two Dimensions ............. 448
  15.23.1 Discretizing the Domain: Pre-processing .................. 448
  15.23.2 Selecting the Interpolation or Shape Functions ........... 450
TABLE OF CONTENTS

15.23.3 Formulating the Equations for a Single Element ........................................ 451
15.23.4 Assembling the System of Equations ............................................................. 453
15.23.5 Incorporating the Boundary Conditions ......................................................... 455
15.27 Using PYTHON to Solve 2D PDEs via an FEM .................................................... 457
15.27.1 A General Coding .............................................................................................. 457
15.27.2 An Example: Isotropic Heat Flow ...................................................................... 464
15.30 Exercises ............................................................................................................... 467
15.A Program(s) for FDM Approach to 1D Problem .................................................... 475
15.A.4 Listing of fdm1d.py (PYTHON) ........................................................................ 475
15.B Evaluating Integrals for 1D FEM Problem ............................................................ 476
15.B.1 ... using MAXIMA ............................................................................................ 476
15.C Program(s) for FEM Approach to 1D Problem .................................................... 476
15.C.4 Listing of fem1d.py (PYTHON) ........................................................................ 476
15.D PYTHON Programs for FDM Approach to 2D Problems ....................................... 478
15.D.1 Listing of fdmwave1d.py .................................................................................. 478
15.D.2 Listing of fdmwave1d_anim.py .......................................................................... 479
15.D.3 Listing of fdmdiffus1d.py ................................................................................. 481
15.D.4 Listing of fdmdiffus1d_anim.py ......................................................................... 482
15.D.5 Listing of fdmlap2d.py ..................................................................................... 483
15.D.6 Listing of fdmlap2d_plot.py ............................................................................. 484
15.D.7 Listing of fdmlap2d_anim.py ............................................................................ 485
15.E A Useful Integral .................................................................................................... 486
15.F Evaluating Integrals for 2D FEM Problem ............................................................ 487
15.F.1 ... using MAXIMA ............................................................................................ 487
15.G Program(s) for FEM Approach to 2D Problems .................................................. 491
15.G.4 Listing of fem2d.py (PYTHON) ........................................................................ 491
A Introduction to L\TeX ................................................................. 495
A.1 Creating a Simple Document .................................................................................. 496
A.1.1 Structuring a L\TeX Source File ........................................................................ 496
A.1.2 Creating a PostScript File .................................................................................. 498
A.1.3 Creating a PDF File ........................................................................................... 499
A.1.4 Displaying the Document on the Screen ............................................................ 500
A.1.5 Printing the Document ...................................................................................... 500
A.2 Specification of Global Style: The Preamble .......................................................... 501
A.3 In-Text Specification of Local Style ....................................................................... 503
A.3.1 Type Style ......................................................................................................... 503
A.3.2 Type Size ............................................................................................................ 504
A.3.3 White Space ....................................................................................................... 505
A.3.4 Environments ..................................................................................................... 506
A.3.5 \TeX Packages .................................................................................................. 507
A.3.6 Miscellaneous Other Capabilities ..................................................................... 507
A.4 Including Equations ............................................................................................... 508
A.5 Including Lists ....................................................................................................... 509
A.6 Including Tables ..................................................................................................... 510
A.7 Including Illustrations ............................................................................................ 512
A.7.1 Using Cut and Paste .......................................................................................... 512
A.7.2 Using the graphics Package .............................................................................. 513
A.7.3 Using the tikz Package ...................................................................................... 515
A.7.4 Using the picture Environment and pict2e Package ........................................ 518
A.8 Including TOC, LOF, and LOT .............................................................................. 518
A.9 Including an Index .................................................................................................. 519
A.10 Including Hyperlinks ............................................................................................ 521
A.11 Converting .eps and .ps Files to .pdf .................................................................. 522
A.12 Using Conditional Expressions in \TeX ................................................................. 524
A.13 Error Messages Generated by \TeX ..................................................................... 525
A.14 The Page Previewer for .dvi Files ........................................................................ 526
A.15 The Spell Checker in UNIX .................................................................................. 527
A.16 A Sample Document ............................................................................................. 528
A.17 Miscellaneous Other Features ............................................................................. 529
# TABLE OF CONTENTS

A.18 References ................................................................. 536
A.19 Exercises ................................................................. 537
A.A Listings ................................................................. 540
  A.A.1 ... for Windows ......................................................... 540
    A.A.1.1 Batch File ceps2pdf.bat ................................... 540
    A.A.1.2 Batch File cps2pdf.bat ................................... 540
    A.A.1.3 Batch File rdfileeps.bat ................................ 540
    A.A.1.4 Batch File rdfileps.bat ................................ 540
  A.A.2 ... for UNIX ........................................................... 541
    A.A.2.1 Shell Script ceps2pdf .................................... 541
    A.A.2.2 Shell Script cps2pdf .................................... 541
    A.A.2.3 Shell Script rdfileeps ................................ 541
    A.A.2.4 Shell Script rdfileps ................................ 541
  A.A.3 ... for Windows and UNIX .......................................... 542
    A.A.3.1 Python Script ExtractFileName.py ......................... 542

Z Contacting Software Vendors ........................................... 543

Index .............................................................................. 547
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, including¹

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

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

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

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

¹Many of the specific examples in this list are identified by names that are trademarks belonging to the vendor of the identified software and registered in the United States Patent and Trademark Office. Those that the author knows to have that status are identified with the symbol ® at the first occurrence of the name. Full contact information for the vendors of the software (and the owners of the trademarks) is compiled in Appendix Z.
students launched into the computational world as sophomores, say, will, as juniors and seniors, be motivated to use computational resources intelligently and successfully on their own initiative, whenever it seems to them appropriate to exploit those tools. As a resource, the computer should parallel the library; this book aims to help students develop the skills to support that view.\textsuperscript{2}

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

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

2. You learn the basic commands for one or more tools (how to start the tool, how to stop the tool, how to construct the primary entities—mathematical expressions, numerical arrays, …—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 \textit{Local Guide}, which supplements this book with detailed information that relates specifically to your site.

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

\section{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.

\textsuperscript{2}Uses of internet resources are conspicuously absent from the list of skills in this opening paragraph. While such uses are playing an increasingly important role both in education and in professional life, they are explicitly excluded from the purview of this book.
1.1. AN ORIENTATION TO COMPUTERS

1.1.1 A Simple Responsive Machine

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

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

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

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

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

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

```
3The special words LOOP and END_LOOP bracket a group or block of instructions that are as a block to be executed repeatedly. We shall here ignore concerns about stopping the loop.
4The special words IF, THEN, and END_IF convey a conditional execution of one or more statements. The statement(s) between the THEN and the END_IF will be executed only if the condition following the IF is true when the entire construction is encountered.
```
PROGRAM TYPEWRITER

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

END_PROGRAM

In this listing, we have introduced the word program to identify a complete set of instructions for the performance of some task, and we have introduced the special words PROGRAM and END_PROGRAM to bracket a program. We have also provided a way to designate an appropriate name for the program.

Note that, while a particular code is always associated with a character, not all codes are associated with printing characters. Non-printing characters are called control characters. When received by a printer (or other peripheral device), they result not in the display of a particular symbol but in the performance of some other function. We have already met \langle CR \rangle and \langle LF \rangle. Other control characters familiar to the user of an ordinary typewriter are the backspace \langle BS \rangle, which causes the printhead to back up one space; the horizontal tab \langle HT \rangle, which causes the printhead to advance to the next pre-set (horizontal) tab position; and the vertical tab \langle VT \rangle, 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.\(^5\)\(^6\) Further, declare that pressing a particular key on the keyboard turns some of the bulbs on and leaves the rest off, and endow the CPU with a capacity to sense which bulbs are on and which off. If we represent a light bulb that is off by the symbol 0 and a light bulb that is on by the symbol 1, then we can convey a particular pattern by a string of eight 0’s and 1’s. The string 10011101, for example, represents the sequence on-off-off-on-on-on-off-on.

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

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

\(^6\)Actually, the codes will be sent as a stream of bits, each of which is an electrical voltage level that will be either “high” or “lo”, often said to be “on” or “off”.
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, 00000010, ..., 11111111). Our choice of the byte for internal coding therefore permits us to distinguish 256 codes. Internally, the CPU sees only binary patterns (light bulbs that are on or off; electrical signals that are either high or low; areas on a magnetic tape that are either magnetized or unmagnetized; etc.), and these patterns are conveniently represented by sequences of bits. Externally, binary integers are cumbersome, so various more compact representations are often used. The binary pattern can be interpreted as a decimal integer (as above), but the conversion from binary to decimal is awkward. A more convenient but still compact notation involves grouping the bits in an eight-bit binary integer in the pattern xx-xxx-xxx and using the eight symbols 0, 1, 2, ..., 7 to represent the three-bit binary integers 000, 001, 010, 011, 100, 101, 110, and 111. The integer 10011101, for example, would then have the translation

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

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

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

1.1.3 The ASCII Character Set

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

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

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

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

- an eight-bit byte as an **unsigned eight-bit integer**, assigning its 256 different patterns to the (positive) integers ranging (in decimal) from 0 to 255.
- an eight-bit byte as a **signed eight-bit integer**, assigning its 256 different patterns to the (negative and positive) integers ranging (in decimal) from −128 to +127. (The range is not symmetric because we must assign one of the patterns to the integer 0.) The highest order bit normally conveys the sign of the value and the remaining seven bits convey the value, though the connection between bit patterns and values—especially negative values—is not always as straightforward as one might naively assume.\(^8\)
- a sixteen-bit combination of two consecutive bytes as an **unsigned sixteen-bit integer**, with its \(2^8\) = 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 \(2^{16}\) = 4294967296 values assigned to the (positive) integers ranging (in decimal) from 0 to 4294967296.
- a 32-bit combination of four consecutive bytes as a **signed 32-bit integer**, with its 4294967296 values assigned to the (positive) integers ranging (in decimal) from −2147483684 to +2147483683.

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

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

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

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

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

<table>
<thead>
<tr>
<th>Code</th>
<th>Character</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>NUL</td>
</tr>
<tr>
<td>16</td>
<td>DLE</td>
</tr>
<tr>
<td>32</td>
<td>SP</td>
</tr>
<tr>
<td>48</td>
<td>0</td>
</tr>
<tr>
<td>64</td>
<td>@</td>
</tr>
<tr>
<td>80</td>
<td>P</td>
</tr>
<tr>
<td>96</td>
<td>~</td>
</tr>
<tr>
<td>112</td>
<td>p</td>
</tr>
<tr>
<td>1</td>
<td>SOH</td>
</tr>
<tr>
<td>17</td>
<td>DC1</td>
</tr>
<tr>
<td>33</td>
<td>!</td>
</tr>
<tr>
<td>50</td>
<td>2</td>
</tr>
<tr>
<td>66</td>
<td>B</td>
</tr>
<tr>
<td>82</td>
<td>R</td>
</tr>
<tr>
<td>98</td>
<td>b</td>
</tr>
<tr>
<td>114</td>
<td>r</td>
</tr>
<tr>
<td>3</td>
<td>EOX</td>
</tr>
<tr>
<td>19</td>
<td>DC3</td>
</tr>
<tr>
<td>35</td>
<td>#</td>
</tr>
<tr>
<td>52</td>
<td>4</td>
</tr>
<tr>
<td>68</td>
<td>D</td>
</tr>
<tr>
<td>84</td>
<td>T</td>
</tr>
<tr>
<td>100</td>
<td>d</td>
</tr>
<tr>
<td>116</td>
<td>t</td>
</tr>
<tr>
<td>4</td>
<td>EOT</td>
</tr>
<tr>
<td>20</td>
<td>DC4</td>
</tr>
<tr>
<td>36</td>
<td>$</td>
</tr>
<tr>
<td>52</td>
<td>4</td>
</tr>
<tr>
<td>68</td>
<td>D</td>
</tr>
<tr>
<td>84</td>
<td>T</td>
</tr>
<tr>
<td>100</td>
<td>d</td>
</tr>
<tr>
<td>116</td>
<td>t</td>
</tr>
<tr>
<td>5</td>
<td>ENQ</td>
</tr>
<tr>
<td>21</td>
<td>NAK</td>
</tr>
<tr>
<td>37</td>
<td>%</td>
</tr>
<tr>
<td>53</td>
<td>5</td>
</tr>
<tr>
<td>69</td>
<td>E</td>
</tr>
<tr>
<td>85</td>
<td>U</td>
</tr>
<tr>
<td>101</td>
<td>e</td>
</tr>
<tr>
<td>117</td>
<td>u</td>
</tr>
<tr>
<td>6</td>
<td>ACK</td>
</tr>
<tr>
<td>22</td>
<td>SYN</td>
</tr>
<tr>
<td>38</td>
<td>&amp;</td>
</tr>
<tr>
<td>54</td>
<td>6</td>
</tr>
<tr>
<td>70</td>
<td>F</td>
</tr>
<tr>
<td>86</td>
<td>V</td>
</tr>
<tr>
<td>102</td>
<td>f</td>
</tr>
<tr>
<td>118</td>
<td>v</td>
</tr>
<tr>
<td>7</td>
<td>BEL</td>
</tr>
<tr>
<td>23</td>
<td>ETB</td>
</tr>
<tr>
<td>39</td>
<td>'</td>
</tr>
<tr>
<td>55</td>
<td>7</td>
</tr>
<tr>
<td>71</td>
<td>G</td>
</tr>
<tr>
<td>87</td>
<td>W</td>
</tr>
<tr>
<td>103</td>
<td>g</td>
</tr>
<tr>
<td>119</td>
<td>w</td>
</tr>
<tr>
<td>8</td>
<td>BS</td>
</tr>
<tr>
<td>24</td>
<td>CAN</td>
</tr>
<tr>
<td>40</td>
<td>(</td>
</tr>
<tr>
<td>56</td>
<td>8</td>
</tr>
<tr>
<td>72</td>
<td>H</td>
</tr>
<tr>
<td>88</td>
<td>X</td>
</tr>
<tr>
<td>104</td>
<td>h</td>
</tr>
<tr>
<td>120</td>
<td>x</td>
</tr>
<tr>
<td>9</td>
<td>HT</td>
</tr>
<tr>
<td>25</td>
<td>EM</td>
</tr>
<tr>
<td>41</td>
<td>)</td>
</tr>
<tr>
<td>57</td>
<td>9</td>
</tr>
<tr>
<td>73</td>
<td>I</td>
</tr>
<tr>
<td>89</td>
<td>Y</td>
</tr>
<tr>
<td>105</td>
<td>i</td>
</tr>
<tr>
<td>121</td>
<td>y</td>
</tr>
<tr>
<td>10</td>
<td>LF</td>
</tr>
<tr>
<td>26</td>
<td>SUB</td>
</tr>
<tr>
<td>42</td>
<td>*</td>
</tr>
<tr>
<td>58</td>
<td>:</td>
</tr>
<tr>
<td>74</td>
<td>J</td>
</tr>
<tr>
<td>90</td>
<td>Z</td>
</tr>
<tr>
<td>106</td>
<td>j</td>
</tr>
<tr>
<td>122</td>
<td>z</td>
</tr>
<tr>
<td>11</td>
<td>VT</td>
</tr>
<tr>
<td>27</td>
<td>ESC</td>
</tr>
<tr>
<td>43</td>
<td>+</td>
</tr>
<tr>
<td>59</td>
<td>;</td>
</tr>
<tr>
<td>75</td>
<td>K</td>
</tr>
<tr>
<td>91</td>
<td></td>
</tr>
<tr>
<td>107</td>
<td>k</td>
</tr>
<tr>
<td>123</td>
<td>{</td>
</tr>
<tr>
<td>12</td>
<td>FF</td>
</tr>
<tr>
<td>28</td>
<td>FS</td>
</tr>
<tr>
<td>44</td>
<td>,</td>
</tr>
<tr>
<td>60</td>
<td>&lt;</td>
</tr>
<tr>
<td>76</td>
<td>L</td>
</tr>
<tr>
<td>92</td>
<td>\</td>
</tr>
<tr>
<td>108</td>
<td>l</td>
</tr>
<tr>
<td>124</td>
<td>}</td>
</tr>
<tr>
<td>13</td>
<td>CR</td>
</tr>
<tr>
<td>29</td>
<td>GS</td>
</tr>
<tr>
<td>45</td>
<td>-</td>
</tr>
<tr>
<td>61</td>
<td>=</td>
</tr>
<tr>
<td>77</td>
<td>M</td>
</tr>
<tr>
<td>93</td>
<td>]</td>
</tr>
<tr>
<td>109</td>
<td>m</td>
</tr>
<tr>
<td>125</td>
<td>}</td>
</tr>
<tr>
<td>14</td>
<td>SO</td>
</tr>
<tr>
<td>30</td>
<td>RS</td>
</tr>
<tr>
<td>46</td>
<td>.</td>
</tr>
<tr>
<td>62</td>
<td>&gt;</td>
</tr>
<tr>
<td>78</td>
<td>N</td>
</tr>
<tr>
<td>94</td>
<td>~</td>
</tr>
<tr>
<td>110</td>
<td>n</td>
</tr>
<tr>
<td>126</td>
<td>-</td>
</tr>
<tr>
<td>15</td>
<td>SI</td>
</tr>
<tr>
<td>31</td>
<td>US</td>
</tr>
<tr>
<td>47</td>
<td>/</td>
</tr>
<tr>
<td>63</td>
<td>?</td>
</tr>
<tr>
<td>79</td>
<td>O</td>
</tr>
<tr>
<td>95</td>
<td>-</td>
</tr>
<tr>
<td>111</td>
<td>o</td>
</tr>
<tr>
<td>127</td>
<td>DEL</td>
</tr>
</tbody>
</table>

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

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

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

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

Clearly, many different **data types** are in common use. Most importantly, the information stored in a particular byte or aggregate of bytes contains nothing at all to identify its data type. The four bytes of a character string are indistinguishable from the four bytes in a 32-bit unsigned integer and both are indistinguishable from the four bytes in a single-precision floating point number. The bit pattern in those four bytes can be interpreted in any of these ways (and in others as well). It is the programmer’s responsibility to make sure that the program treats stored values in a way appropriate to their data types, usually by referring to memory cells with names that convey the data type. When conversion from one form to another—e.g., character to associated numerical ASCII code—is necessary, the programmer must invoke an appropriate routine to effect the conversion.
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.\(^\text{10}\) 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, \(\textit{those}\) into still higher-level aggregates, \(\ldots\). The process is analogous to the aggregating of individual documents into a file folder, of these folders into file drawers, of the drawers into file cabinets, \(\ldots\). In the computer world, we need then not only the files themselves but a new type of file that basically lists the contents of the aggregate that it represents. The resulting structure for keeping track of files looks like a tree. At the highest level, the tree has a single file—the root directory—that contains the names of the files it contains (and information about their locations on the disks of the computer). Some of those files may themselves describe (sub)directories, in each of which are listed the names of the files it contains. Some of those files in turn may describe (subsub)directories.\(^\text{11}\) Locating a specific file in the entire structure then requires not only giving the file name but also describing its \textit{path}—the sequence of directories through which we must pass from the root directory to reach the file. In UNIX, the root directory for the entire storage system is named `/`; the forward slash is also used to separate directory paths in an extended path. Thus, to specify the location of a file buried several directories down from this universal starting point, we would have to supply an identifier like

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

which indicates that the file \texttt{intro.tex}—the \LaTeX{} source file for this chapter—will be found in the \texttt{intro} directory in the \texttt{CCLI} directory in the \texttt{cook} directory in the \texttt{people} directory in the \texttt{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 \textit{Local Guide}.\(^\text{12}\) That document also explains conventions about (and restrictions imposed on) file names and types, user accounts, and other matters that vary so much from site to site that this book cannot sensibly explain them all.

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

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

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

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

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

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

1.4 Glossary, Conventions, and Understandings

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

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

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

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

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

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

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

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

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

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

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

• 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 preceding with a prompt can—and should—be executed as you work your way through the material. Statements without a prompt should not, and most often could not, be executed.

• Especially in constructing statements for computer algebra systems and presenting their output, we will not always present the output in exactly the form or with exactly the appearance it will actually have. In particular, we will frequently use unsubscripted variables, e.g. \(x_1\) or \(xf\), in the statement to be executed but render these variables as subscripted, e.g., \(x_1\) or \(xf\), in the displayed output.

• At various points in this book, we define, use, and/or refer to a variety of files specific to the text. All such files are stored on your local computer system and are available for your use. The head of the directory tree in which those files are stored is identified explicitly in the

\(^{14}\text{In some languages, procedures can optionally be written}\quad \text{err = SQRTPRO}( x^2 + y^2, R ) , \text{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

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

$$n! = n \times (n - 1) \times (n - 2) \times (n - 3) \times \cdots \times 2 \times 1$$  \hspace{1cm} (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

$$n!! = n \times (n - 2) \times (n - 4) \times (n - 6) \times \cdots$$  \hspace{1cm} (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

$$10!! = 10 \times 8 \times 6 \times 4 \times 2 = 2^5 \times 5 \times 4 \times 3 \times 2 \times 1 = 2^5 5!$$

$$\Rightarrow \quad (2n)!! = 2^n n!$$ \hspace{1cm} (1.3)

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

$$9!! = 9 \times 7 \times 5 \times 3 \times 1 = \frac{9 \times 8 \times 7 \times 6 \times 5 \times 4 \times 3 \times 2 \times 1}{8 \times 6 \times 4 \times 2} = \frac{9!}{8!!} = \frac{9!}{2^4 4!}$$

$$\Rightarrow \quad (2n + 1)!! = \frac{(2n + 1)!}{2^n n!}$$ \hspace{1cm} (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

$$\Gamma(\nu) = \int_0^\infty t^{\nu-1} e^{-t} dt$$ \hspace{1cm} (1.5)

We quickly conclude that

$$\Gamma(1) = \int_0^\infty e^{-t} dt = 1$$ \hspace{1cm} (1.6)

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

$$\Gamma\left(\frac{1}{2}\right) = \int_0^\infty \frac{e^{-t}}{\sqrt{t}} dt = \int_0^\infty \frac{e^{-x^2}}{x} dx = \int_0^\infty e^{-x^2} dx$$ \hspace{1cm} (1.7)

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

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

$$= \int_0^\infty \int_0^\infty e^{-(x^2+y^2)} dx \, dy = \int_0^{2\pi} e^{-r^2} r \, dr \int_0^\infty d\phi = \pi$$ \hspace{1cm} (1.8)
and we conclude that
\[ \Gamma \left( \frac{1}{2} \right) = \sqrt{\pi} \] (1.10)

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

The Gamma function, however, has a particularly interesting property that we can deduce if we apply integration by parts to the definition. Provided \( \nu > 1 \), we find that
\[
\Gamma(\nu) = -\int_0^\infty t^{\nu-1} e^{-t} \left| \right|_0^\infty + (\nu - 1) \int_0^\infty t^{\nu-2} e^{-t} dt = (\nu - 1) \Gamma(\nu - 1) \] (1.11)

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

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

and we have indeed succeeded in finding a function that is the natural extension of the factorial function to non-integral arguments. Indeed, one often sees the notation \( \nu! \) as an alternative to the notation \( \Gamma(\nu + 1) \)—and the latter in fact provides a formal definition of the former.\(^\text{15}\) Note that, since we know quite explicitly that \( \Gamma(1) = 1 \), this connection between the Gamma and factorial functions supports what is sometimes an assertion of convenience, namely that \( 0! = 1 \).

### 1.5.2 The Laplace Transform

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

this transform has several important properties:

- The Laplace transform of a linear combination of functions is that same linear combination of the Laplace transforms of the separate functions,
\[
\mathcal{L} \left( af(t) + bg(t) \right) = \int_0^\infty e^{-st} \left( af(t) + bg(t) \right) \, dt \\
= a \int_0^\infty e^{-st} f(t) \, dt + b \int_0^\infty e^{-st} g(t) \, dt \\
= a \mathcal{L} \left( f(t) \right) + b \mathcal{L} \left( g(t) \right) \] (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>( f(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>( \sin \omega t )</td>
<td>( e^{at} )</td>
</tr>
<tr>
<td>( \frac{dx}{dt}(t) )</td>
<td>( s \tilde{x}(s) - x(0) )</td>
<td>( \frac{d^2x}{dt^2}(t) )</td>
<td>( s^2 \tilde{x}(s) - s x(0) - \frac{dx}{dt}(0) )</td>
</tr>
</tbody>
</table>

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

\[
\frac{df}{dt}(s) = \int_0^{\infty} e^{-st} \frac{df(t)}{dt} \, dt = e^{-st} f(t) |_{0}^{\infty} + s \int_0^{\infty} e^{-st} f(t) \, dt = s \tilde{f}(s) - f(0) \quad (1.16)
\]

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

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

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

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

1.6 Licensing Issues

Much of the software on every device in computational facilities around the world is proprietary and subject to the provisions both of the applicable copyright laws and of license agreements between the local institution and the vendors of the software. Usually—but not always, the licenses acquired by a given institution will permit simultaneous use on all of the devices in a laboratory at that institution. Almost certainly, the licenses limit use to projects and activities at that institution and prohibit copying of the software, except for purposes of system maintenance and backup. All users of all devices must be constantly mindful of the proprietary nature of much of the available software and must abide by the restrictions imposed by the copyright laws and by the license agreements. Those restrictions for each software package available at your site are described in the Local Guide.
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 package1 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

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

---

1See 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.

2See Appendix Z for full contact information.

3A detailed comparison of PYTHON 2.7 with PYTHON 3.6 is provided on the web at the link wiki.python.org/moin/Python2orPython3. Review this document to help you decide which version will be best for your applications.
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 \4 ‘python Path/idle.py’ (for the GUI) at the prompt from the operating system, \6 (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, \7 either in a UNIX or Windows shell to the operating system (CLI) or in a newly created PYTHON Shell (GUI). \8 \9 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 \langle \text{RETURN} \rangle, 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 \langle ''' Comments ''' \rangle 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 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.

\4 The PYTHON shell is defined by a PYTHON program, so bringing up the shell involves running a PYTHON program!
\5 Since 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.
\6 In these commands to the operating system, case may be important.
\7 Note that local configurations may have changed this prompt from the off-the-shelf default.
\8 The label in the PYTHON Shell may include an identification of the version in use.
\9 Be 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.
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 \texttt{int} for signed integers, \texttt{float} for floating point values, \texttt{str} for character strings, and \texttt{tuple} or \texttt{list} or \texttt{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,\footnote{A string value (e.g., ‘cook’) that does not represent a valid number cannot be converted to a number.} the PYTHON commands \texttt{int}, \texttt{float}, and \texttt{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.\footnote{As described in Section 5.3.1, rounding floating values up or down to integers is accomplished by functions in the \texttt{math} module.} During some conversions, e.g., the conversion of the floating point value 3.67 with the expression \texttt{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 \texttt{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,

\begin{verbatim}
long(a)   converts an “ordinary” integer (data type \texttt{int} to a ”long” integer (data
type \texttt{long}). Long integers will be displayed with a suffix L,\footnote{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 \texttt{int}, not data type \texttt{long}, is used. The command \texttt{long} does not
exist in PYTHON 3.} bool(a) converts a to type \texttt{bool} (Boolean for logical operations with value either
\texttt{True} or \texttt{False}),\footnote{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.} hex(a) converts an integer a to hexadecimal representation of type \texttt{str}, and
oct(a) converts an integer a to octal representation of type \texttt{str}.
\end{verbatim}

The PYTHON command \texttt{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.\footnote{The URL \url{developer.rhino3d.com/guides/rhinopython/python-datatypes/} links to a more complete discussion of basic data types in PYTHON.}

5.2.2 Variable Names

The simplest variables in PYTHON represent either single numbers (type \texttt{int} or \texttt{float}), character strings (type \texttt{str}), or aggregates of several of these types referred to by a single name (type \texttt{tuple} or type \texttt{list} or type \texttt{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 \texttt{True}, \texttt{False}, \texttt{sin}, and \texttt{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

\[ x = \alpha + \beta \]

will add the values currently stored in the variables `alpha` and `beta` and store the result in the variable `x`, creating `x` if 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

\[ \text{command}( \text{argument}, \text{argument}, \text{Keyword}=	ext{Name}, \ldots) \]

15Technically, the length of a variable name is limited only by the available memory, but testing that limit is unwise.
16If a string contains quotation marks of one type, the entire string should be contained in marks of the other type.
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

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

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`.\(^{17}\)

The module `os`,\(^{18}\) 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\(^{19}\) 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 \(<\text{RETURN}\>\), 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:\(^{20}\)

---

\(^{17}\)Full documentation of `matplotlib` can be found at the URL `matplotlib.org/?`, where `?` is a specific version number or the word `stable` for the current stable version.

\(^{18}\)See Section 5.16.1.

\(^{19}\)See Section 5.4.

\(^{20}\)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
1.6
>>> 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

Do simple arithmetic.
Ask for type of value.
Do integer division, which ignores anything after a decimal.\(^\text{21}\)
Find remainder after integer division.
Do floating division, which gets 8.0/5.0 correct.
Raise integer to power.
Raise to fractional power.
Ask for type of value.
Invoke built-in function pow.

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),

\(^{21}\) 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 use the statement 8//5 or 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.\footnote{A full listing of the functions included in the 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.}

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
golden
>>> a = math.sqrt(9)
>>> a
3.0
>>> print a  or  >>> print(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.\footnote{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.}

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

Entities of type `str`, `tuple`, `list`, and `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 `single` or `double` quotation marks, though it will most often in either case be subsequently displayed with `single` quotation marks.\footnote{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.} We illustrate both assigning a `string` to a variable and examining it with the statements
5.3. A SAMPLING OF PYTHON CAPABILITIES

>>> name = 'David'
Create a string name

>>> name
'David'
Display value on the screen.

>>> print( name )
David
Display using print; note absence of quotation marks.

>>> type( name )
<type 'str'>
Display data type of variable.

>>> len( name )
5
Find number of characters in string.

>>> name[2]
'v'
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

>>> b = ( 1, 0, 3, 7, 10 )
Create a tuple b.

>>> b
(1, 0, 3, 7, 10)
Display it on the screen.

>>> type(b)
<type 'tuple'>
Display data type of variable.

>>> b[3]
7
Display third element; remember that indices begin with 0.

>>> len(b)
5
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

>>> c = [ 1, 0, 3, 7, 10 ]
Create a list c.

>>> c
[1, 0, 3, 7, 10]
Display it on the screen.

>>> type(c)
<type 'list'>
Display data type of variable.

>>> c[3]
7
Display third element; remember that indices begin with 0.

>>> c[3] = 5
Edit third element.

>>> c
[1, 0, 3, 5, 10]
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.

---

25See Section 5.3.4.
In a similar way, a 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

```python
>>> 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
```
Create a list of lists 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 \([1][1]\), not \([1,1]\) or \((1,1)\).

### 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:

```
dir()
```
which will return an alphabetically sorted 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:

```
b = None or del b or del b, name, beta
```
Here, the first statement retains the variable name but renders the value stored therein empty while the second removes the variable name and its value altogether. Both release some memory for subsequent use.\(^{26}\)

### 5.3.4 Creating and Examining Arrays

Unfortunately, because of the format of their internal storage, strings, lists, tuples, and sets are 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,\(^{27}\) are added if the module **numpy** (**numerical python**) is imported with the statement\(^{28}\)

```
>>> import numpy
```

\(^{26}\)Two other commands—`globals()` and `locals`—return dictionaries (data type `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, `dir()` is sufficient.

\(^{27}\)And many other features, including many additional data types and many additional routines for numerical processing.

\(^{28}\)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.
Once this module has been imported, creation of an array involves invoking the command \texttt{numpy.array} which has (normally) a tuple or a list as its sole argument. For example, the statements

\begin{verbatim}
>>> 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')
>>> [ a.ndim, a.size, a.itemsize ]
[1, 5, 4]
\end{verbatim}

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 (\texttt{numpy.ndarray} and \texttt{int32}) and four (data) attributes\footnote{See Section 5.3.8.} (\texttt{dtype}, \texttt{ndim}, \texttt{size}, and \texttt{itemsize}) possessed by every array.

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

\begin{verbatim}
>>> b1 = [ 1, 2, 3, 4 ]
>>> b2 = [ 5, 6, 7, 8 ]
>>> b3 = [ 9, 10, 11, 12 ]
>>> b = numpy.array( [ b1, b2, b3 ] )
>>> print( b )
[[ 1, 2, 3, 4 ],
 [ 5, 6, 7, 8 ],
 [ 9, 10, 11, 12 ]]
>>> type(b)
<type 'numpy.ndarray'>
>>> b[2,3]
12
>>> b.dtype
dtype('int32')
>>> [ b.ndim, b.size, b.itemsize ]
[2, 12, 4]
>>> b.shape
(3L, 4L)
\end{verbatim}

The additional (data) attribute \texttt{shape} has here subtly been introduced.

The array \texttt{b} created somewhat laboriously in the previous paragraph can be created more easily with the single statement

\begin{verbatim}
>>> c = numpy.arange(1, 13, 1).reshape(3,4)
\end{verbatim}
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.\(^{30}\) 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 $n \times m$ column array of zeroes.
- `numpy.ones([n,m])` will create an $n \times m$ column 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 $n \times m$ column 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\(^ {31}\)

```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]]
```

\(^{30}\)Note that, if the terminating value is exactly reached, that value will be omitted from the array.

\(^{31}\)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  

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

>>> print( a[1,:] )  
[4 5 6]  

>>> a[1,2] = 22  

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

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

Print the second column of \( a \) as a row.

Print the first row of \( a \).

Edit element 1,2 and display result

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.\(^{32}\)

*Caution:* Sooner or later, we will want to make an independent copy, say \( arr2 \), of an existing array \( arr1 \). While it is tempting to invoke a compact statement like \( arr2 = arr1 \), that statement will not achieve the desired objective because it simply creates a second name that points to the *same* internal storage area as is pointed to by the first name. With that approach, changing an element in \( arr2 \) will simultaneously change that element in \( arr1 \). To assure the independence of the two created arrays, the more sophisticated statement

\[
>> arr2 = arr1.copy()
\]

must be invoked. *Beware!*

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 \( 2 \times 2 \) matrices with the statements

\[
>>> A = numpy.matrix([[1,2],[3,4]])
\]

Create and display matrix \( A \)

\[
>>> A
\]

\[
matrix([[ 1, 2],
 [ 3, 4]])
\]

Display data type of \( A \)

\[
>>> type(A)
\]

\[
<class 'numpy.matrix'>
\]

Create and display matrix \( B \)

\[
>>> B = numpy.matrix([[5,6],[7,8]])
\]

Display the element in the *first* row and *second* column of \( B \)

\[
>>> B
\]

\[
[[5 6],
 [7 8]]
\]

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

\(^{32}\) The URLs [docs.scipy.org/doc/numpy-1.13.0/reference/](http://docs.scipy.org/doc/numpy-1.13.0/reference/) and [docs.scipy.org/doc/numpy/user/quickstart.html](http://docs.scipy.org/doc/numpy/user/quickstart.html) point to more detailed enumerations and illustrations of the components in `numpy`. 
>>> A + B
    matrix([[ 6,  8],
            [10, 12])

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

multiplied element-by-element with the statement\(^{33}\)

>>> numpy.multiply(A, B)
    matrix([[ 5, 12],
            [21, 32])

or multiplied by the rules for matrix multiplication with the statement

>>> 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\(^{34}\)

>>> numpy.linalg.inv( A )
    matrix([[ 1.5, -0.5],
            [-0.5, 2. ]])

with the result having the data type of a matrix. We can evaluate the transpose of a matrix with the statement\(^{35}\)

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

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

>>> 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

\(^{33}\)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.

\(^{34}\)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.

\(^{35}\)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

>>> a = 3 + 4j
Assign a complex value to a. Note there is no multiplication sign before the j.

>>> print(a)
Display value.
(3+4j)

>>> type(a)
Display data type.
<type 'complex'>

>>> a.real, a.imag
Display real and imaginary parts.
(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 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.36

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

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

which 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.]]

>>> b = np.matrix([[3.0], [-4.0], [1.0]])
```

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)
>>> print(ainv * b)
[[ 0.85294118]
 [−0.14705882]
 [ 0.67647059]]
```

36 The URL docs.python.org/?/library/cmath.html, where ? is either 2 or 3, links to a full discussion of the cmath module.
This solution correctly solves the original equations, as hand substitution—and the statement 
\texttt{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 \texttt{a} and \texttt{b} as above, we might simply have

invoked the statement

\begin{verbatim}
>>> print( np.linalg.solve(a,b) )
\end{verbatim}

\[
\begin{bmatrix}
0.85294118 \\
-0.14705882 \\
0.67647059
\end{bmatrix}
\]

\subsection*{5.3.8 Data and Method Attributes of Variables}

PYTHON variables have what are called \textit{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 \textit{instance} attributes, identify information about the

variable that is stored with the variable. For example, the array \texttt{A} defined in Section 5.3.5 has,

among others, the attributes \texttt{ndim}, \texttt{size}, \texttt{itemsize}, \texttt{dtype}, and \texttt{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

\begin{verbatim}
>>> A.ndim
2
>>> A.size
4
>>> A.itemsize
4
>>> A.dtype
\texttt{dtype('int32')}  
>>> A.shape
\texttt{(2L, 2L)}
\end{verbatim}

Display number of dimensions in \texttt{A}.

Display number of elements in \texttt{A}.

Display number of bytes occupied by each element in \texttt{A}.

Display data type of each element in \texttt{A}.

Display size of each dimension in \texttt{A}, which here is a 2 \times 2 matrix. The output in

PYTHON 3 will be (2,2).

Alternatively, we can exploit the built-in command \texttt{getattr} with the statements

\begin{verbatim}
>>> getattr( A, 'ndim' )
2
>>> getattr( A, 'dtype' )
\texttt{dtype('int32')}  
\end{verbatim}

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 \textit{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

\texttt{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.

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

\footnote{Unless clarity requires otherwise, we will usually refer to both types simply as attributes.}
5.3. A SAMPLING OF PYTHON CAPABILITIES

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

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

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

```python
>>> x = np.arange( -3, 7, 2 )
>>> print( x )
[ -3 -1  1  3  5 ]
```
creates the indicated array, starting at \(-3\) and incrementing by 2 until the next result equals or exceeds 6. The parameter \( Step \) defaults to 1 and the parameter \( Start \) defaults to 0. Exploiting the defaults, the statement

```python
>>> x = np.arange( 5 )
>>> print( x )
[0 1 2 3 4]
```
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. Alternatively (and somewhat more transparently), the desired array can be created with a statement like

\[
Var = \text{np.linspace}( Start, Stop, NoVals )
\]
to store in \( Var \) a one-dimensional array of \( NoVals \) values starting with \( Start \) and ending with \( Start \). More specifically, the statement

```python
>>> y = np.linspace( -3, 5, 5 )
>>> print( y )
[ -3. -1.  1.  3.  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 \( NoVals \) defaults to 50 but \( Start \) and \( Stop \) 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

```python
>>> x = np.linspace(-3.0, 3.0, 61 )
>>> y = np.cosh( x )
```

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.

```python
>>> y = np.cosh( x )
```

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

```python
>>> plt.plot( x, y )
>>> plt.show()
```

to set the independent and dependent variables and then the statements\(^{38}\)

```python
>>> plt.plot( x, y )
>>> plt.show()
```

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 *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.\(^{39}\)

Taking control of some of the defaults adopted by the command `plt.plot` or otherwise embellishing the graph can be accomplished in several ways. We might, for example,

- use the commands `plt.xlim` and `plt ylim`—each of which has a single argument—a tuple specifying the desired ranges \((x_{\text{min}}, x_{\text{max}})\) on the \( x \) axis and \((y_{\text{min}}, y_{\text{max}})\) on the \( y \) axis.

- use the command `plt.title` to place a title on the graph.

- use the commands `plt.xlabel` and `plt.ylabel` to put labels on the \( x \) and \( y \) axes.

- use the command `plt.grid` to replace short tics along the axes with full grid lines.

\(^{38}\)The command `plt.plot` is more fully discussed in Section 5.9.

\(^{39}\)In PYTHON 3, this statement can be embellished to `plt.show(block=False)` to keep the graph on the screen while returning control to the PYTHON shell. Further, interactive plotting, which is discussed in Section 5.15.7, is available in some PYTHON backends and permits overriding this behavior.
5.3. A SAMPLING OF PYTHON CAPABILITIES

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\footnote{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.}

\begin{verbatim}
>>> plt.plot( x, y, color='black', linewidth=4 )
>>> plt.xlim( (-4.0, 4.0) ); plt.ylim( (0.0, 12.0) )
>>> plt.grid( color='black' )
>>> plt.title( 'Hyperbolic Cosine Function', fontsize=20 )
>>> plt.xlabel( 'x', fontsize=16 ); plt.ylabel( 'cosh(x)', fontsize=16 )
>>> plt.show()
\end{verbatim}

In these statements, each of the pairs \texttt{color='black'}, \texttt{linewidth=4}, \texttt{fontsize=20}, and \texttt{fontsize=16} assigns the value given by what follows the equal sign to the \textit{keyword} identified by what precedes the equal sign. Specifically, they override the default color (blue) for the graph with black, the default linewidth (0.5 pt) with 4 pt, and the default character size (10 pt) with 20 pt or 16 pt.

We have here introduced the keywords \texttt{fontsize}, \texttt{linewidth} and \texttt{color}. The keyword \texttt{fontsize} stipulates the size of the type used in points. The keyword \texttt{color} is discussed more fully and the keyword \texttt{linestyle} is introduced in Section 5.9.2.

Note, incidentally, the power of the statement \texttt{y = np.cosh(x)}. The argument of the function \texttt{np.cosh} is an \textit{array} and contains several values. In response to this statement, PYTHON generates a \textit{second} array \texttt{y} having the same number of elements as \texttt{x}. Each element in \texttt{y} is the hyperbolic cosine of the corresponding element in \texttt{x}. In many languages, we would be obliged to use a more elaborate construction to instruct the computer to work on each element individually. PYTHON automatically understands that intention with this simpler statement. Beyond simpler coding, PYTHON also achieves faster execution than would be achieved in more traditional coding.
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 (because of the function range) steps through the values in the list:

```python
[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
>>> for x in range(10): y[x] = (x+1)**2
```

of the loop.

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

```python
... for x in range(10): y[x] = (x+1)**2
```

---

41 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.

42 If you had imported the module numpy as np, we could alternatively have used the statement `y = np.zeros(10)`, though that would have created an array of floating zeros.

43 The statement `np.arange(10)` would produce the same list.

44 In PYTHON 2, the command range actually returns the indicated list. In PYTHON 3, the command range creates what is called an iterable object. When controlling a loop, that object has the same effect as an actual list but require less memory for its storage. The statement `list(range(10))` in PYTHON 3 will convert the iterable object into an actual list.

45 Remember that indices in PYTHON start at 0.

46 Note that, even here, (RETURN) must be pressed twice after the last statement in the loop.
>>> 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 changed by the execution of the loop.

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 = []
>>> 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.47

The second loop available in PYTHON is the while loop. Once started, a while loop continues executing until the condition controlling the loop changes from True to False—which means that the statements in the loop must assure that, sooner or later, that condition changes to False. A while loop that accomplishes the the same task as in the previous paragraph involves the statements48

```python
>>> 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 blocking of two or more statements is communicated to the PYTHON interpreter by the indentation in the formatting of the statements.

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

47PYTHON also admits (1) the statement list.insert( index, value) to insert value at the position index in an existing list, moving all following values one position later in the list and (2) the statement list1.extend( list2 ) to add a second list to the end of a first list.

48The substance of footnote 41 applies also here.
Within PYTHON, logical conditions have values and, if set to a variable, the variable for each such condition will have type bool. Only two values, however, are allowed: True and False. The statements

```python
>>> 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,

---

49 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.

50 Here, the lines in the body of the if/else structure are shown as they would be entered interactively one at a time at the PYTHON prompt in the Python Shell. Evidently, the prompt and its following space are not seen by the interpreter, so the i in the first line of the structure is seen by the interpreter as above the e in else. If the lines are entered in the CLI, the characters ... will appear at the beginning of each line after the first and necessary indentation will have to be inserted explicitly as shown at the right.
finding principal axes and moments of inertia, ...). The PYTHON function \texttt{eig} in the module \texttt{numpy.linalg} takes as arguments the name of the matrix whose eigenvalues and eigenvectors are desired. The function returns \textit{either} the eigenvalues \textit{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 \texttt{eig} with statements like\footnote{To find only the eigenvalues, use the statement \texttt{evals = np.linalg.eig(A)}.}

```python
>>> 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 \texttt{evals} is one of the eigenvalues of \texttt{A} and each \textit{column} in the array \texttt{evecs} is one of the \textit{normalized} eigenvectors of the matrix \texttt{A}, with \texttt{evals[0]} corresponding to \texttt{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

\begin{equation}
A \mathbf{x} = \lambda \mathbf{x}
\end{equation}

where \(\lambda\) is an eigenvalue and \(\mathbf{x}\) is the corresponding eigenvector. For the first eigenvalue and eigenvector, we find that\footnote{The name \texttt{lambda} is reserved in PYTHON.}

```python
>>> 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 \(A \mathbf{x}\) and \(\lambda \mathbf{x}\) is straightforward, but arranging the two arrays in parallel columns is harder. We use \texttt{np.concatenate} with keyword \texttt{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

```python
>>> lhs = A * evecs
>>> print(lhs)
[[ -2.9551691 -0.3469682 -1.0707888]
 [ -7.82377363 -0.87335404  0.16068274]
 [ -3.16594128  3.15776112  0.18806199]]
```

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

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

```python
[[-2.9551691 -0.3469682 -1.0707888]
 [ -7.82377363  0.16068274 -0.87335404]
 [ -3.16594128  3.15776112]]
```
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 \texttt{diag} and \texttt{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

\begin{verbatim}
>>> rhs = evecs * diag; print(rhs)

\[
\begin{bmatrix}
-2.9551691 & -0.3469682 & -1.0707888 \\
-7.82377363 & 0.16068274 & -0.87335404 \\
-3.16594128 & -0.0732154 & 3.15776112
\end{bmatrix}
\end{verbatim}

\end{verbatim}


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)

\[
\begin{bmatrix}
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{bmatrix}
\end{verbatim}

\end{verbatim}

Evidently, the two matrices, whose elements are all on the order of 1, differ in absolute value order of magnitude $10^{-15}$ of less, 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 computers 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 \textit{Python Shell} window (GUI) at the PYTHON prompt >>> 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 \textit{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 \textit{File} menu in the \textit{Python Shell}. Launched this way, the label in the bar at the top of the window providing the editor will be ‘Untitled’.\footnote{The \textit{File} menu in the \textit{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 \textit{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 \textit{File} menu in the \textit{Python Shell} window. In the resulting \textit{Python Edit} window, enter the lines
a = 'David'
b = 3.5
c = 100
d = (a, b, c)
print(d)

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.py), 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. Note that PYTHON is restarted, so via this route, everything in PYTHON’s work space prior to running this stored program will be lost.

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

\begin{verbatim}
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)
\end{verbatim}

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

\begin{verbatim}
[1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
\end{verbatim}

will appear in the Python Shell window. Note that, via the route described in this paragraph, 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

\begin{verbatim}
python FileName.py
\end{verbatim}

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.

55Any 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.
56If 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.
57Since 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.
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

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

in PYTHON 2 or the single statement

```python
>>> 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. PYTHON is not restarted when a command file is executed in this way.

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

---

58 See Section 5.16.1.

59 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 $HEAD/python/radio.dat to the file. See the note at the beginning of this chapter.

60 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.

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

62 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.
import matplotlib.pyplot as plt
import numpy as np
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)

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

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. 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.

63 The statement print(dataarray.shape) will return (201L, 4L).
64 See the note at the beginning of this chapter.
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

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

will open an empty file in the default directory,\(^65\) 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.\(^66\) 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

\(^65\)Destroying any existing file of the specified name.

\(^66\)Writing binary files is beyond the scope of this book.
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 time and, separated from that entry by some number of spaces, the corresponding entry from position. 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
' )
```

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])+ '
' )
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 have ragged right edges. 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,\(^{67}\) 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.\(^{68}\)

\(^{67}\)Remember that PYTHON starts counting at zero.

\(^{68}\)See the URL `docs.python.org/?/tutorial/inputoutput.html`, where ? is either 2 or 3, for more information.
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
```

The reserved word `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 `return` is necessary only if the function is to return values to the calling program. For example, the simple function

```python
def sumdiff(x,y):
    sum=x+y
    diff=x-y
    return sum, diff
```

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

```python
sumdiff(5,10)
(15, -5)
a,b = sumdiff(5,10)
a
15
b
-5
```

with PYTHON illustrates how to use the function. The somewhat more complicated function

```python
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]
```

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 `numpy` has not yet been imported, but the function itself cannot be executed without prior execution of the statement

```python
import numpy as np
```

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

```python
import numpy as np
```

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

---

69 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.
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.
'''
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

quad(1,-1,-6)
[-2.0,3.0]

illustrates how quad might be called.

5.7.2 The Function luplot

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

>>> import numpy as np
>>> import matplotlib.pyplot as plt
>>> execfile( 'luplot.py' )
>>> 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 luplot is not enclosed in quotation marks.

What if, however, we wanted to use luplot 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

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

(5.3)

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
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
```

```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
written, however, we cannot simply add these parameters as arguments to lineshape. Were we to
adopt that approach, we would want to set the parameters a, b, and x0 at PYTHON’s command
level and have them known not within luplot itself but within a function that luplot calls. The
easiest means to achieve this passing of a parameter from PYTHON’s command level into a function
that is not directly called from the command level involves using global variables. In essence, at
the command level, we assign values to appropriate variables—here a, b, and x0—at the command level.
Then, we declare these same variables to be global in the function that is to access those values
and use them symbolically. In this way, we circumvent PYTHON’s automatic isolation of variables
in functions from variables of the same name in other functions, rendering the variables known in
any function that declares them global—and only in those functions. To this end, we might rewrite
lineshape to be as listed in Table 5.3 and store it in the file lineshape1.py. Here, the statement
global a, b, x0 tells the function that it is to find the values of the variables a, b, and x0 in
variables named a, b, and x0, which will have been assigned values at command level. Then, we
invoke the already-defined luplot with this function using the statements

```python
>>> a = 1.0; b = 1.0; x0 = 2.0
```
Assign values to global variables.

```python
>>> execfile( 'lineshape1.py' ) or exec(open('lineshape1.py').read() )
```
Define lineshape1.py

```python
>>> luplot( lineshape1, 200, -10.0, 10.0 )
```
Graph function.
5.7. DEFINING FUNCTIONS

Figure 5.4: The Lorentz lineshape.

Table 5.3: The PYTHON function lineshape1.py.

```python

Program lineshape1.py

LINESHAPE1 - Plots Lorentz line shape.
The function lineshape returns the value of the Lorentz
distribution for given argument x. The parameters a, b, and
x0 must be declared assigned values prior to invoking
lineshape1, which then uses the values in those
global variables.

```edef lineshape1(x):
    global a, b, x0
    y = a**2/(b**2 + (x-x0)**2)
    return y

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

```python
>>> help( FunctionName )
```

where FunctionName is presented without quotation marks and without the file type, will display on
the screen at least the calling statement of the function 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. For some functions, it may be necessary to hover the cursor over a “button”
that is displayed and follow the instructions that pop up.

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 at least three links,
  including:
  - '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 explain-
  ing 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.\(^70\) 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.\(^71\)

- 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

\(^70\) In PYTHON 2, the information is displayed immediately; in PYTHON 3, you may have to double-click the left
mouse button on a button to bring up the information.

\(^71\) See the previous footnote.
5.9. GRAPHING SCALAR FUNCTIONS OF ONE VARIABLE

A 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
```

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`...
CHAPTER 5. INTRODUCTION TO PYTHON

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)

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()
```

Import needed modules.
Create array of 101 values in interval.
Evaluate B, also an array of 101 values.
Generate internal graph of $B_z$ versus $z$.
Add title.
Label x and y axes.
Add grid.
Display graph.

This sequence of statements will produce the graph shown in Fig. 5.5. 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 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 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
```

74 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.

75 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.

76 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.
5.9. GRAPHING SCALAR FUNCTIONS OF ONE VARIABLE

Figure 5.5: On-axis magnetic field of a circular current loop.

Figure 5.6: Undamped and damped sine waves.

```
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,\textsuperscript{77} 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

\textsuperscript{77}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.
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 5.5: 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</th>
</tr>
</thead>
<tbody>
<tr>
<td>-, solid</td>
<td>Solid (default)</td>
<td>b, blue</td>
<td>blue</td>
</tr>
<tr>
<td>:, dotted</td>
<td>Dotted</td>
<td>orange</td>
<td>orange</td>
</tr>
<tr>
<td>--, dashed</td>
<td>Dashed</td>
<td>g, green</td>
<td>green</td>
</tr>
<tr>
<td>-.-, dashdot</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>

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’\textsuperscript{78} and ‘k--’\textsuperscript{79} 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.

\textbf{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

$$r(\theta) = a(1 - \cos \theta)$$ (5.5)

\textsuperscript{78}Equivalent to color=’black’.
\textsuperscript{79}Equivalent to color=’black’ and linestyle=’--’. 
Figure 5.7: Polar plot of the cardioid

```
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()
```

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()
```

Here, in the command `plt.polar`, the radial coordinate comes first and the azimuthal coordinate—in radians—comes second. Note also that polar graphs are automatically displayed with equal 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\(^{80,81}\)

```
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)
```

\(^{80}\)Note that the name `fig` and the names `ax*` are entirely arbitrary. They are not reserved words.

\(^{81}\)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.
This statement creates a *Figure* window and establishes names for the sub-windows in the pattern

\[
\begin{align*}
\text{ax1} & \quad \text{ax2} \\
\text{ax3} & \quad \text{ax4}
\end{align*}
\]

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

\[
\text{fig.subplots_adjust(left=??, bottom=??, right=??, top=??, wspace=??, hspace=??)}
\]

...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
z = np.linspace(-4.0, 4.0, 101)
v = np.sin(z); w = np.cos(z); x=np.sinh(z); y=np.cosh(z)
fig, ( (ax1, ax2), ( ax3, ax4 )) = plt.subplots(2,2)
fig.subplots_adjust(wspace=0.4, hspace=0.5)
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.

The coding is a bit simpler if we invoke the command `plt.subplot` (note singular) by substituting the statements
Figure 5.8: Graphs of sine, cosine, hyperbolic sine, and hyperbolic cosine on the interval \(-4 \leq x \leq 4\).

```python
plt.subplot(221)
plt.plot(z,v, 'k', linewidth=3)
plt.title('Sine', fontsize=14)
plt.xlabel('$z$'); plt.ylabel('$\sin(z)$')

plt.subplot(222)
plt.plot(z,w, 'k', linewidth=3)
plt.title('Cosine', fontsize=14)
plt.xlabel('$z$'); plt.ylabel('$\cos(z)$')

plt.subplot(223)
plt.plot(z,x, 'k', linewidth=3)
plt.title('Hyperbolic Sine', fontsize=14)
plt.xlabel('$z$'); plt.ylabel('$\sinh(z)$')

plt.subplot(224)
plt.plot(z,y, 'k', linewidth=3)
plt.title('Hyperbolic Cosine', fontsize=14)
plt.xlabel('$z$'); plt.ylabel('$\cosh(z)$')
plt.show()
```

after the first four statements in the coding in the previous paragraph. The resulting figure is identical to Fig. 5.8. Note that, in contrast to the previous coding, all statements here relating to a particular subplot must be grouped together.

### 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
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>

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. The data are read into the PYTHON workspace with the statements:

```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)
```

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.

---

82 You can, of course, copy this file from the directory `$HEAD/python` into your current directory and examine its structure with a text editor.

83 Make sure the default (working) directory is set to the directory containing the file `rcdata.dat`. See Section 5.16.1.

84 The command `numpy.loadtxt` has numerous keywords, all of whose defaults are appropriate here. More information can be found by Googling `numpy.loadtxt`. 

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%,\(^85\) we add the error bars with the statements

```python
unc = 0.05*data[:,1]
plt.errorbar( data[:,0], data[:,1], unc, ecolor='black', linestyle='none', capsize=2 )
```

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

\[
G = \frac{V_{\text{out}}}{V_{\text{in}}} = \frac{1}{\sqrt{1 + \left(\frac{f_0}{f}\right)^2}} \quad \text{where } f_0 \text{ is defined as } f_0 = \frac{1}{2\pi RC}
\]  

(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.

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

(5.7)

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.\(^86\)

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

```python
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 \(R = 15\) kΩ and \(C = 0.0442\) µF 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

\(\text{\textsuperscript{85}}\)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.\(\text{\textsuperscript{86}}\)See also the function `numpy.logspace` described in the PYTHON manuals.
Figure 5.9: Data from the RC high-pass filter experiment.

```python
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')
```

The first of these statements uses `plt.plot`—remember, we have already set the logarithmic scale on the horizontal axis—to draw a line from data coordinate (500.0, 0.325) to the data coordinate (900.0, 0.325). The second then uses `plt.text` to place the label “Theoretical” near to the line, with the label starting at the point (1000.0, 0.3). The remaining statements place a circular plotting symbol and label it “Experimental”.

### 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. Then, we must send that file to the printer. The second

---

87 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, 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. 

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,

$$ z(x, y) = \sin(2\pi x) \sin(3\pi y) $$  \hspace{1cm} (5.8)

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

$$ I(\xi, \eta) = \left( \frac{\sin \xi}{\xi} \right)^2 \left( \frac{\sin \eta}{\eta} \right)^2 $$  \hspace{1cm} (5.9)

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, 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 \((y, \text{eta}, \ldots)\) containing the second coordinate \((y, \eta, \ldots)\) of those points.

---

88.pdf, .png, .ps, and .eps are usually available.
89 See the URL matplotlib.org/api/as_gen/matplotlib.pyplot.savefig.html for a full description of these options.
90 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

   \[ z = \text{np.sin}(2.0*\text{np.pi}*x) \times \text{np.sin}(3.0*\text{np.pi}*y) \]

   \[ I = (\text{np.sin(xi)}/\text{xi})^2 \times (\text{np.sin(eta)}/\text{eta})^2 \]

   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 = numpy.linspace( 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

```plaintext
print( x )
[[ -3. -1.  1.  3. ]
 [ -3. -1.  1.  3. ]
 [ -3. -1.  1.  3. ]
 [ -3. -1.  1.  3. ]]

print( y )
[[ -1. -1. -1. -1. ]
 [ -0.5 -0.5 -0.5 -0.5 ]
 [  0.   0.   0.   0. ]
 [  0.5  0.5  0.5  0.5 ]
 [  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

\[ z = x**2 + y**2 \]

```plaintext
print( z )
[[ 10.   2.   2.  10. ]
 [ 2.   0.   0.  2. ]
 [ 2.   0.   0.  2. ]
 [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

```python
xx = np.linspace(0.0,1.0,40); yy = np.linspace(0.0,1.0,40)
x, y = np.meshgrid( xx, yy )
z = np.sin(2.0*np.pi*x) * np.sin(3.0*np.pi*y)
```

...and create refined arrays $z$ and $I$ for the two examples of Eqs. (5.8) and (5.9).

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

```python
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import axes3d 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()
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()
```

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

---

91 We shift at this point to assuming the growing program will be created as a file in a PYTHON Edit window.

92 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 $I$.

93 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, and they 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 $I$.\(^{94}\)

Specifically, we use the statements

\(^{94}\)See Section 5.15.1 for a more detailed discussion.
5.11. GRAPHING SCALAR FUNCTIONS OF TWO VARIABLES

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).

![Graph of $I$ function](image)

```
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()
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()
```

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

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

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.
Figure 5.12: Alternative views of the wireframe representations of the functions \( z \) and \( I \) in Eqs. (5.8) and (5.9), respectively.

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 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

```
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 contour

The command `contour`, whose general syntax is similar to that of the command `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

```
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

```
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 `colors` (note the ‘’) 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. 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. 

---

5.11. GRAPHING SCALAR FUNCTIONS OF TWO VARIABLES

Figure 5.13: A rescaled surface representation of the function $I$ in Eq. (5.9).
Figure 5.14: Contour plot of the function $z$ in Eq. (5.8). This view shows the membrane from above, where solid lines represent regions in which the membrane is displaced towards the viewer and dashed lines—which are invoked automatically without user request—represent regions in which the membrane is displaced away from the viewer.

5.11.4 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()
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 as cm
```

which provides a multitude of color maps (cm) from which to choose.\footnote{See the URL matplotlib.org/examples/color/colormaps_reference.html for a full enumeration of the available color maps.} We can then create a more
interpretable surface plot with the statements

```python
fig = plt.figure()
ax4 = plt.axes(projection='3d')
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()
```

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.\(^{100}\)

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 left end of the bar at the bottom of the window (PYTHON 2) or along the right end of the bar across the top of the window (PYTHON 3).\(^{100}\)

\(^{100}\)For the coding for the Greek letters, see the text associated with Fig. 5.11.
CHAPTER 5. INTRODUCTION TO PYTHON

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

- 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 left of the bottom edge (PYTHON 2) or the right of the top edge (PYTHON 3) 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.

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

- 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

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

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
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

\[
x = r \cdot \cos(\phi); \quad y = r \cdot \sin(\phi)
\]

Finally, for a crude graph, we complete the task of generating a wire frame graph with the statements

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

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

\[
fig = plt.figure()
plt.contour(x, y, z, 20, colors='black')
plt.show()
\]

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, 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

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

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

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

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
import numpy as np  # Import numpy.
xx = np.linspace(-10.0, 10.0, 33)  # Calculate grid.
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)  # Calculate premultiplier.
zs = z**2  # Calculate z^2.
rhos = x**2 + y**2 + zs  # Calculate ρ^2.
 rho = np.sqrt(rhos)  # Calculate ρ.
p = pfactor * np.exp(-2.0*rho/3.0) * (9.0*zs**2 - 6.0*zs*rhos + rhos**2)  # 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\(^{101}\)

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

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

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

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()
```

will produce mesh surface, shaded surface, and contour representations of the probability density in the plane parallel to the \( xy \) plane at the specified value of \( z \). These figures are not shown.

Extracting data in a plane perpendicular to the \( x \) or \( y \) axes is easily achieved in PYTHON\(^{102}\) by invoking \( p[15, :, :] \) in the statements

\(^{101}\)We must extract two-dimensional arrays from all three arguments, since these statements will not behave properly with three-dimensional arrays in any position.

\(^{102}\)Not all graphing tools admit arbitrary positioning of the wildcard—: or equivalent—in a multidimensional array.
Figure 5.19: Wireframe representation of the function $p$ in Eq. (5.11) in the plane $x = x_0 + 15 \times dx = -10.0 + 15 \times 20.0 / 32.0 = -0.625$.

$$px = p[15,:,:]$$
$$yyy = np.linspace(-10.0,10.0,33)$$
$$yy,zz = np.meshgrid( yyy, yyy )$$

```python
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()
```

```python
plt.contour( yy, zz, px, 20, colors='black' )
plt.xlabel( 'y', fontsize=14 )
plt.ylabel( 'z', fontsize=14 )
plt.axis('square')
plt.show()
```

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.

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

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

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

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

---

103 Refer to the URL plot.ly/python/isosurfaces-with-marching-cubes/ for full detail on the plotly module.
104 The variable $verts$ is a two-dimensional numpy array with three columns, each row of which contains the coordinates of one of the vertices in the isosurface, and $faces$ is a two-dimensional numpy array each row of which contains the indices of the three vertices composing one of the triangles in the mesh isosurface. The variables $normal$ and $values$ play no role in the present context.
105 In some versions of PYTHON, the invoked function has a slightly different name and the first statement might have to be cast as $verts$, $faces$, $normals$, $values = measure.marching_cubes( p, level=1.2e-4 )$.
106 This statement may not execute instantly.
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

```python
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 rescaling $x_1, y_1,$ and $z_1$ to span this new range with statements like

```python
x2 = np.array( x1 ); x11 = 0.625*x2 - 10.0
y2 = np.array( y1 ); y11 = 0.625*y2 - 10.0
z2 = np.array( z1 ); z11 = 0.625*z2 - 10.0
```

With this change, executing the two statements

```python
fig = ff.create_trisurf(x11, y11, z11, faces, title="Isosurface", \
    show_colorbar=False, colormap=['rgb(255,255,255)'])
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 re-oriented 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 **\LaTeX** 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 **\LaTeX** to include **.jpg** files in the **\includegraphics** command.

Sometimes, you may wish to override the automatic labeling of the axes in the display produced by the statements leading to Fig. 5.21. The statement

```python
fig.update_layout( scene = dict(
    xaxis = dict(nticks=9, range=[-10.0,10.0],),
    yaxis = dict(nticks=9, range=[-10.0,10.0],),
    zaxis = dict(nticks=9, range=[-10.0,10.0],),
) )
```

inserted just before the **offline.plot** statement will achieve that objective. Here, **nticks** specifies the number of tick marks to be placed along the corresponding axis\(^{107}\) and **range** specifies the range of the variable to be displayed on that axis. Numerous other available keywords, all of which will be displayed in a report printed on the screen when this statement is executed, are described in the documentation for the **update_layout** command.

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

---

\(^{107}\) The coefficients in these linear transformations were obtained by solving the equations $-10.0 = 0.0 \ast a \ast b$ and $10.0 = 32.0 \ast a \ast b$ by the method described in Section 5.3.7. Note that converting tuples $x1, y1,$ and $z1$ to arrays is necessary before the linear rescaling can be achieved.

\(^{108}\) It appears as if, whatever number is specified, the actual number of tick marks displayed will be rounded down to 3, 5, 11, or 22, as if tick marks can be labeled only with integers.
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\textsuperscript{109}

$$
E_z = \cos \kappa_x b \bar{x} \sin n \pi \bar{y}
$$

$$
H_x = -\frac{\kappa_x b}{n \pi} \cos \kappa_x b \bar{x} \sin n \pi \bar{y}
$$

$$
H_y = \sin \kappa_x b \bar{x} \cos n \pi \bar{y}
$$

(5.13)

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.

\textsuperscript{109}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.
within the guide, and
\[
\left( \frac{\kappa_x b}{n\pi} \right)^2 = \left( \frac{\omega b}{n\pi c} \right)^2 - 1
\]
(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 \( \kappa_x b \), where \( \kappa_x b \) is determined from Eq. (5.14) by \( \omega \), \( b \), and \( c \). Since \( \kappa_x b \) must be real if the wave is to propagate down the guide, we must require that \( (\omega b/n\pi c)^2 > 1 \) or that \( \omega > n\pi c/b \). For the sake of a specific example, we choose \( b = 1 \) and \( n = 2 \), and then we choose \( \omega \) so that \( \kappa_x b/2\pi \) turns out to have the value 1. With these choices, the fields are given by

\[
\begin{align*}
E_z &= \cos 2\pi \bar{x} \sin 2\pi \bar{y} \\
H_x &= \sin 2\pi \bar{x} \cos 2\pi \bar{y} \\
H_y &= -\cos 2\pi \bar{x} \sin 2\pi \bar{y}
\end{align*}
\]
(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\(^\text{110}\)

```
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 \texttt{quiver}

The PYTHON function \texttt{quiver} produces a two-dimensional vector field plot. At each grid point, \texttt{quiver} draws an arrow which conveys both the direction and the magnitude of the field at that

\(^{110}\text{From here on in this section, we drop the overbars.}\)
5.13. GRAPHING VECTOR FIELDS

Figure 5.23: Magnetic field of Eq. (5.15) at $t = 0$ in a rectangular wave guide using \texttt{quiver}. The illustrated structure, of course, propagates along the guide toward positive $x$ as time unfolds.

![Graph of magnetic field](image)

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 $H$ in PYTHON’s memory can be displayed in this form by executing the statement

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

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 \texttt{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('Electromagnetic Field in a Rectangular Wave Guide', fontsize=16 )
```
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 electric field is directed out of the paper where the contour lines are solid and into the paper where the contour lines are dashed. The illustrated structure, of course, propagates along the guide toward positive \( x \) as time unfolds.

![Electromagnetic Field in a Rectangular Wave Guide](image)

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.

### 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)
```

and then creates arrays providing three-dimensional grids for the coordinates \( x, y, z \) and calculates the three components \( A_x, A_y, A_z \) for the vector field to be displayed. For example, the statements
5.14 Animation

**Python** offers at least three ways to create animated views:

- The module `animation.FuncAnimation`, the details of which are described in Subsection 5.14.1 and illustrated in Subsections 5.14.2 and 5.14.3.
- The module `vpython`, which is available only in Python 3 and is described and illustrated in Subsection refPython:TPUG.

```python
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)
ax.quiver( x,y,z, Ax,Ay, Az, length=0.2, normalize=True, color='black' )
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.

Figure 5.25: A three-dimensional vector field.
• The **pause** command in the `matplotlib.pyplot` module, which is described and illustrated in Subsection 5.14.5.

### 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 I

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

$$u(x,t) = \cos(\omega t) \sin(n \pi x) \quad (5.17)$$

describes that motion. We begin by executing the statements

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

to provide the several needed modules for the task at hand. Then, we prepare an empty figure with the statements
5.14. ANIMATION

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

```python
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\(^\text{111}\) and create the animation with the statements

```python
harm = int(input('Specify harmonic: '))
am = animation.FuncAnimation( fig, animate, init_func=init, 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 × in the upper right corner of the Figure window.\(^\text{112}\)

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 \(\text{⟨F5⟩}\). Alternatively, provided the current directory is properly set in each case, the program can be run from within the Python Shell with the statement

\(^{111}\)Explicit conversion of the value returned by the `input` command works in both PYTHON 2 and PYTHON 3, and is necessary in PYTHON 2. 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.

\(^{112}\)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.
execfile( 'stwave.py' ) or exec( open('stwave.py').read() )

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

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

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

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

\begin{verbatim}
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
\end{verbatim}

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

\begin{verbatim}
[0.0, 0.0, 0.0, 0.0, 0.0, 0.0]
[0.0, 0.0, 0.0, 0.0, 0.0, 0.0]
[0.0, 0.0, 0.0, 0.0, 0.0, 0.0]
[0.0, 0.0, 0.0, 0.0, 0.0, 0.0]
[0.0, 0.0, 0.0, 0.0, 0.0, 0.0]
[0.0, 0.0, 0.0, 0.0, 0.0, 0.0]
\end{verbatim}

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

\begin{verbatim}
i = 0; k = 0
drm=np.zeros( (41,33,33) )
while k <= 40:
    tmp = []
    j = 0
\end{verbatim}
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 from 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,
```

and at last create the animation 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 statement

```python
execfile( 'drumhead.py' ) or exec( open('drumhead.py ').read() )
```

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

```python
python drumhead.py
```

---

114 We place this definition at the beginning of the file.
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.\textsuperscript{115} 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

\[ x(t) = v_{x0} t \quad ; \quad y(t) = -\frac{1}{2} g t^2 + v_{y0} t \quad , \quad z(t) = 0.0 \quad (5.18) \]

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

In this example, we are using the Anaconda distribution of PYTHON 3.9.13 with VPYTHON 7.6.4.\textsuperscript{116} We construct our animation from components in the \texttt{vpython} module, which must be installed 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 \texttt{scene} is created automatically. Note also that, while your program is run as always in PYTHON, \textit{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 \texttt{vpython} are made available by importing everything in the \texttt{vpython} module with the statement\textsuperscript{117}

\begin{verbatim}
from vpython import *
\end{verbatim}

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

\begin{verbatim}
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' )
\end{verbatim}

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\textsuperscript{119}

\begin{verbatim}
bball = sphere( pos=vector(0.0,0.0,0.0), radius=0.3, color=color.red, 
make_trail=True, trail_color=color.blue)
\end{verbatim}

\begin{samepage}
\textsuperscript{115} Full details can be found by going to the URL vpython.org and selecting 'Documentation' in the resulting screen.

\textsuperscript{116} Check the website glowscript.org for information on how to use VPYTHON within your favorite browser.

\textsuperscript{117} We import \texttt{vpython} using the construction "from vpython import *", allowing us to use components from the module without prefacing each with \texttt{vpython}. Had we used the construction import \texttt{vpython} as \texttt{vp}, say, the characters \"vp,\" would have to precede any invocation of a component, e.g. \texttt{vp.scene}, \texttt{vp.text}, \texttt{vp.color.red}, …

\textsuperscript{118} 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.

\textsuperscript{119} We here use the built-in objects \texttt{sphere} and \texttt{box}. The VPYTHON literature describes many other available objects.
\end{samepage}
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:\(^\text{120}\)

\[
g = 9.8; \; vx0 = 8.0; \; vy0=12.0
\]

\[
\text{ball.velocity} = \text{vector}(vx0,vy0,0.0)
\]

Note that the second statement also adds a previously non-existent attribute to the object ball.

Finally, we create the while loop

\[
t=0.0
\]
\[
deltat=0.01
\]
\[
\text{while ball.pos.y} > -0.001:
\]
\[
\text{rate(25)}
\]
\[
\text{ball.pos} = \text{vector}( \text{ball.velocity.x*t}, \\-
-0.5*g*t**2+\text{ball.velocity.y*t}, \; 0.0 )
\]
\[
t = t + \text{deltat}
\]

controlled by the \( y \) position of the ball to update the velocity. In addition, the statement \text{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.\(^\text{121}\) 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 \textit{projectile.py} is presented in Section 5.E.

Once \textit{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

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

We have here barely scratched the surface of the capabilities of VPython. Numerous objects can be introduced into the scene.\(^\text{122}\) 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.\(^\text{123}\)

\(^{120}\) Some trial and error preceded the selection of the values for the initial velocity.
\(^{121}\) The delay will, of course, be longer if the processing of the loop requires more than 0.04 s.
\(^{122}\) See footnote 119.
\(^{123}\) More complete details can be found at any of several on-line references, including documentation and examples from links at the URL \url{www.vpython.org}.\n
5.14.5 Motion of a Vibrating String II

The `pause` command in the `matplotlib.pyplot` module is well built to facilitate the creation of animations. Basically, replacing the statement `plt.show()` in a program that generates a sequence of displays with the statement `plt.pause(delay)` allows the program to go on to the next display without requiring a manual closing of the display already on the screen. Adding an appropriate statement to clear the display in an appropriate position in the program will clear the current display before creating the next display in the animation. Finally, in PYTHON 2 (but not in PYTHON 3), the statement `plt.show()` must be inserted after the last display has been generated so that manual closing of the final display in an animation will prepare for a re-execution of the program to generate a subsequent animation.

To illustrate more specifically the use of the `pause` command to create an animation, we return to the vibrating string described by Eq. (5.17) in Subsection 5.14.2. Supposing the needed modules will be imported with the statements

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

before we execute the program creating the animation, we begin that program with the statements

```python
n = input( 'Mode number: ' ) # Request mode of interest
n = int(n)
delay = input( 'Delay between plots: ' ) # Request delay between plots
delay = float(delay)
```

to request the number of the mode of interest and the delay time between displays. Then, assuming the string to have length 1.0, we start by creating an array dividing that interval into 100 segments with the statement

```python
x = np.linspace(0.0,1.0,101) # Set equally spaced points along string
```

Next, we include the statements

```python
f0 = np.sin(n*np.pi*x) # Set initial shape of string
fig, ax = plt.subplots() # Create figure and axes
for t in range(301):
f = f0*np.cos(2.0*n*np.pi*t/100.0) # Evaluate shape at t
ax.clear() # Clear previous figure
ax.plot(x,f,color='black', linewidth=2) # Plot shape
plt.ylim([-1.0,1.0]) # Standardize y axis range
plt.pause(delay) # Pause
```

to evaluate the shape \( f(x,t) \) of the string as a function of \( x \) at 301 values of \( t \) equally distributed over three periods (0 \( \leq t \leq 100 \)) of the fundamental oscillation of the string and display that shape at each instant. Finally, we add the statement

```python
if int(sys.version[0])==2: plt.show() # Add plt.show() if PYTHON 2
```

to provide for a clean exit when the final frame is manually closed before the program is re-executed to animate another mode. Because the statement `plt.show()` is needed only in PYTHON 2, we make its inclusion conditional on PYTHON 2 being in use, so we must also import the module `sys` before executing the program. The resulting program named `stringanim.py` is listed in Section 5.F and is stored in the default directory. After the statements
import numpy as np
import matplotlib.pyplot as plt
import sys

have been executed, the animation of a particular mode is produced with the statement and sample input

execfile('stringanim.py') or exec(open('stringanim.py').read()
Mode number: 3
Delay between plots: 0.1

After this animation has completed, the Figure window should be closed manually before the program is re-executed to create a subsequent animation. Note that this route to an animation is far richer than here illustrated.

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 \LaTeX math mode. e.g., '...$B_z$...' and '...$z$...', in the character strings creating those textual displays. PYTHON also recognizes the names of major mathematical functions and Greek letters placed in \LaTeX math mode, though any backslash characters must be “escaped” by duplicating them, e.g., '...$\sin(\theta)$...'.

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

\[ x = \cos t \quad y = \sin t \quad z = \alpha t \] (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

---

124 Alternatively duplicating backslashes in the character string can be avoided by preceding the string with the character r, e.g., 'r'$x'\ldots$\sin(\theta)$\ldots'. This inclusion tells the PYTHON interpreter to pass the character string literally to the construction of the display.
Figure 5.26: Space curve of a charged particle moving in a constant magnetic field along the \( z \) axis.

```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

```python
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(r'$x$ versus $\cos(x)$', fontsize=14)
ax2 = fig2.add_subplot(111)
ax2.plot(x,y2, linewidth=2, color='black')
ax2.set_title(r'$x$ versus $\sin(x)$', fontsize=14)
```
5.15. ADVANCED GRAPHING FEATURES

Figure 5.27: The effect of different axis configurations.

```python
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

---

125 Additional options include `off` and `on`.
Table 5.7: Coding to illustrate making circles circles.

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

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

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()
```

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 grided, 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)

- `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])`\(^{126}\) 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 all are empty, tick marks and labels will be suppressed altogether. Whether the lists are empty or not, the shaded planes will remain.

\(^{126}\)The symbol | is the standard symbol for 'or'. Thus, `[x|y]ticks` conveys either `xticks` or `yticks`. 
Table 5.8: Coding to produce 2D and 3D graphs with default axis positions.

```python
# 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()
```

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()
```

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 `color='blue', 'green', 'red', 'cyan', 'magenta', 'yellow', 'black', and 'white'`. 
Figure 5.28: Graph of $\sin(x)$ vs. $x$.

- one of 'b', 'g', 'r', 'c', 'm', 'y', 'k', and 'w' (no keyword color)
- an RGB tuple, e.g., color=(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, (0.0,0.0,0.0) corresponds to black, (0.5,0.5,0.5) corresponds to a gray, and (1.0,1.0,1.0) corresponds to white.
- a hexadecimal RGB string, e.g., color='#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.\(^\text{127}\)

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\(^\text{128}\) plt.show() before displaying anything. Many—though not all—PYTHON backends admit the statement plt.ion() to turn on interactive mode.\(^\text{129}\) Once this statement has been executed with a background that supports interactive mode,\(^\text{130,131}\) 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()
```

\(^\text{127}\) See the URL matplotlib.org/users/colors.html for greater detail.

\(^\text{128}\) We assume that the statement import matplotlib.pyplot as plt has been executed.

\(^\text{129}\) You can find out the backend in use in your environment by typing the statement plt.get_backend().

\(^\text{130}\) You can determine if interactive mode is functioning by executing the two statements import matplotlib; matplotlib.is_interactive() after the statement plt.ion() has been executed.

\(^\text{131}\) Caution: Interactive mode will work only in the PYTHON command window. It is not available in the PYTHON Shell (IDLE).
5.16. MISCELLANEOUS OCCASIONALLY USEFUL TIDBITS

\[
x = \text{np.linspace}(0, 10, 101) \\
y = \text{np.cos}(x) \\
\text{plt.plot}(x, y) \\
\text{plt.xlabel}('x', \text{fontsize}=16) \\
\text{plt.ylabel}('\cos(x)', \text{fontsize}=16)
\]

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

\[
\text{plt.figure}(2) \\
\text{plt.plot}(x, y, \text{color}='\text{black}', \text{linewidth}=3)
\]

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

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

Once this module has been imported, the following commands—among many others—provide frequently used actions:

- \texttt{print( os.getcwd() )}—report the current directory.
- \texttt{os.chdir( 'Path' )}—change the current working directory to the directory identified by \textit{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.
- \texttt{os.system( 'OSCommand' )} invokes the system command \textit{OSCommand}.
- \texttt{os.listdir( 'Path' )}—displays a list in the PYTHON window of all files in the directory identified by \textit{Path}. Unfortunately, the list is hard to read because it sprawls across the entire screen, folding at the screen width (often in the middle of a file name). If the module \texttt{numpy} has been imported as \texttt{np}, an easier-to-read listing can be created with the statement

\[
\text{print( np.array( os.listdir( 'Path' ) ) )}
\]

Many more features are provided in the \texttt{os} module.\footnote{\textsuperscript{132} See the URL docs.python.org/2/library/os.html.}
CHAPTER 5. INTRODUCTION TO PYTHON

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

```python
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

```python
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. 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

```python
sys.path.append( 'Path' )
```

will add the specified path at the end of the current path and the statement

```python
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

```python
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 and you

---

133 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.
5.16. MISCELLANEOUS OCCASIONALLY USEFUL TIDBITS

- 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

```bash
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.\(^\text{134}\)

If, instead, you are working in a UNIX environment in a C-shell or T-shell\(^\text{136}\) 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 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.\(^\text{138}\)

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).\(^\text{139}\)

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.

\(^{134}\) The simple statement `set` will display a list of all environment variables; the statement `echo %PYTHONSTARTUP%` will display only the specified environment variable.

\(^{135}\) 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.

\(^{136}\) The commands are slightly different in other shells. See the manual for the shell you are using or consult with a system administrator.

\(^{137}\) The simple statement `setenv` will display a list of all environment variables; the statement `echo $PYTHONSTARTUP` will display only the specified environment variable.

\(^{138}\) 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.

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\(^{140,141}\)

\begin{verbatim}
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'])
\end{verbatim}

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),\(^{142}\)

Alternatively (though this route is more complicated), you can copy the file `matplotlibrc` from the directory in which PYTHON itself is stored,\(^{143}\) 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
- 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.\(^{144}\)

### 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,

\(^{140}\) 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.

\(^{141}\) 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.

\(^{142}\) See footnote 139.

\(^{143}\) You can find the location of the standard `matplotlibrc` file by importing `matplotlib` as `mpl` and then executing the statement `print( mpl.matplotlib_fname() )`.

\(^{144}\) See the URL `matplotlib.org/users/customizing.html` for greater detail on the topic of this paragraph.
whenever you type \( \langle F5 \rangle \) 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 \( \langle \text{CONTROL}-F6 \rangle \)). 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

\[
\text{Variable(s)} = \text{input}( 'Prompt: ' )
\]

The format of the input is dictated by the nature of the specified variables. For example, the statement

\[
a = \text{input}( 'Enter a value: ' )
\]

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

\[
a,b = \text{input}( 'Enter two values, separated by a comma: ' )
\]

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.\(^{145}\) In any given context, it is prudent to include in the prompt a stipulation of the data expected as input.

\(^{145}\)See the PYTHON attribute `split` encountered first in Section 5.6.2.
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

```python
a = 'string'
while isinstance(a, int) == False:
    a = 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 `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.\(^{146}\) A better route in PYTHON 3 involves exploiting the `try/except` control structure. For example, the coding\(^{147}\)

```python
a = None
while a == None:
    try:
        a = int(input("Please enter an integer: "))
    except ValueError:
        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 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](http://www.bn.com) or [www.amazon.com](http://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](http://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](http://docs.python.org/?/tutorial/index.html), where `?` = 2 or 3, points you to the official PYTHON tutorial.
- [www.numpy.org](http://www.numpy.org)—the numpy home site, which provides full information about numpy.

---

\(^{146}\) Entering 3 will set `a` to the string '3'.

\(^{147}\) `ValueError` is one of the types of error that can be tested. Others include `IOError`, `ImportError`, `EOFError`, and `KeyboardInterrupt`. Using `except` without any error type will trap all errors.
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 × 8 unit matrix, and (c) a 10 × 10 matrix all of whose elements are zero except those on the main diagonal (which are all 2) and those on the diagonals just above and just below the main diagonal (which are all −1). Search for a route more efficient than laboriously setting each of the 100 elements in the 10 × 10 matrix individually.

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

$$LUx = b \quad \Rightarrow \quad \begin{pmatrix} 1 & 0 & 0 \\ L_{21} & 1 & 0 \\ L_{31} & L_{32} & 1 \end{pmatrix} \begin{pmatrix} U_{11} & U_{12} & U_{13} \\ 0 & U_{22} & U_{23} \\ 0 & 0 & U_{33} \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} b_1 \\ b_2 \\ b_3 \end{pmatrix} \quad (5.20)$$

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 $\sum a_i$ 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

- 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, 22, 32, 36, 68, 84, 89, 98, 99, 119, 123, 127, 132, and 144, in Table 5.6, and at the end of Sections 5.14.1, 5.14.4, and 5.16.1.
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. (d) Explore and explain the difference between the statements `numpy.sum(b)`, `numpy.sum(b, axis=0)`, and `numpy.sum(b, axis=1)` when applied to an array `b` 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 reflects 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 $[1,1,2,3,5,8,13,21,34,55]$ 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/fibob.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 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 by placing an element 0.0 both before the first element and after the last element in the eigenvector. If, for example, the eigenvectors are in the columns of `evecs`, then the statements

```python
y = numpy.append([0.0], evecs[:,6] )
y = numpy.append( y, [0.0] )
matplotlib.pyplot.plot( y )
```

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

```python
x = numpy.linspace(0,1.0,12)
matplotlib.pyplot.plot( x, y )
```

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

(d) (Optional) Repeat parts (b) and (c) but with a similarly constructed matrix that is $50 \times 50$. 

5.10. When a (weak) constant external electric field of magnitude \( F \)—we reserve \( F \) 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 \( \langle nlm|eF|nl'm' \rangle \), where \( l \), \( m \), \( l' \), and \( m' \) range over all possible values of those quantum numbers allowed by the particular value of \( n \). If the states by which the rows and columns are labeled are ordered \( |2, 0, 0 \rangle, |2, 1, -1 \rangle, |2, 1, 0 \rangle, \) and \( |2, 1, 1 \rangle \), then the matrix for the state \( n = 2 \) is

\[
3e a_0 F = \begin{pmatrix}
0 & 0 & -1 & 0 \\
0 & 0 & 0 & 0 \\
-1 & 0 & 0 & 0 \\
0 & 0 & 0 & 0
\end{pmatrix}
\]

where \( e \) is the magnitude of the charge on the electron and \( a_0 \) is the Bohr radius. Similarly, if the states by which the rows and columns are labeled are ordered \( |3, 2, 2 \rangle, |3, 1, 1 \rangle, |3, 2, 1 \rangle, |3, 0, 0 \rangle, |3, 1, 0 \rangle, |3, 2, 0 \rangle, |3, 1, -1 \rangle, |3, 2, -1 \rangle, \) and \( |3, 2, -2 \rangle \), then the matrix for the state \( n = 3 \) is

\[
3e a_0 F = \begin{pmatrix}
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & -9/2 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & -9/2 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & -3\sqrt{6} & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & -3\sqrt{6} & 0 & -9/\sqrt{3} & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & -9/\sqrt{3} & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & -9/2 \\
0 & 0 & 0 & 0 & 0 & 0 & -9/2 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0
\end{pmatrix}
\]

Find the eigenvalues and eigenvectors of these matrices. The eigenvalues give the energy shifts for the Stark effect for \( n = 2 \) and \( n = 3 \) and the eigenvectors give the linear combinations of the base states (i.e., the states in the absence of the external field) out of which the states in the presence of the field emerge as the field is turned on.

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 \( m/m_0 \), \( p/m_0c \), and \( K/m_0c^2 \) 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 $\xi = E_0/mc^2$. (c the speed of light.) 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

$$E(\theta) = \frac{q}{4\pi\varepsilon_0 r^2} \frac{1 - \beta^2}{(1 + \beta^2 \sin^2 \theta)^{3/2}}$$

where $\theta$ is the polar angle of the observation point, $r$ is the radial coordinate at that point, and $\beta = v/c$. (c is the speed of light.) Obtain graphs of $E/(q/4\pi\varepsilon_0 r^2)$ 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

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

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 $T_0 = \hbar c/(\lambda_0 k)$ 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

$$u(\lambda, T) = \frac{1/\Lambda^5}{e^{1/(\Lambda\tau)} - 1}$$

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 $B(z)/B_0$ (a) as a function of $z/L$ for various values of $a/L$, (b) as a surface over the $(z/L)(a/L)$ plane, and (c) as a contour over the $(z/L)(a/L)$ plane. Write a paragraph describing these graphs.
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

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

where \(B_0\) is the magnetic field at the origin. Plot graphs showing \(B(z)/B_0\) 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

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

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 \(0\, {\degree}\ C\) and placed in an oven at \(u_0 = 200\, {\degree}\ C\). The temperature \(u(r,t)\) at a point a distance \(r\) from the center of the potato at time \(t\) is given by

\[
u(r,t) = u_0 - 2 \sum_{n=1}^{\infty} j_0(\beta_n r/a) \frac{\kappa \beta_n^2}{\kappa \beta_n^2 + \omega/\omega_0} e^{-\kappa \beta_n^2 t/(ca^2)}
\]

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 \(175\, {\degree}\ C\), i.e., to a value such that \(u(0,t)/u_0 = 0.875\). Hints: (1) Note that

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

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 \([\kappa = 0.63\, J/(m \cdot s \cdot K)], \quad c = 4.2 \times 10^6\, J/(K \cdot m^3)]\), 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

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

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

5.18.6 Graphing Vector Fields

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

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

where \( l, m, \) and \( n \) are positive integers. Obtain several presentations of the pressure distribution inside this box at \( t = 0 \) and at \( t = \pi/\omega_{mn} \) 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

\[
x_{old} \leftarrow x_0 \quad \text{Set starting point.}
\]
\[
y_{old} \leftarrow y_0 \quad \text{Choose step.}
\]
\[
ds \leftarrow \text{chosen step size}
\]
\[
\text{LOOP}
\]
\[
FX \leftarrow F_x(x_{old},y_{old}) \quad \text{Start loop to generate steps.}
\]
\[
FY \leftarrow F_y(x_{old},y_{old}) \quad \text{Calculate components of force.}
\]
\[
FM \leftarrow \text{SQRT}(FX^2 + FY^2) \quad \text{Calculate magnitude of force.}
\]
\[
x_{new} \leftarrow x_{old} + ds*FX/FM \quad \text{Step \( dx \) along force line.}
\]
\[
y_{new} \leftarrow y_{old} + ds*FY/FM \quad \text{Draw segment of force line.}
\]
\[
\text{Draw line from} \ (x_{old},y_{old}) \ \text{to} \ (x_{new},y_{new}) \quad \text{Shift focus to new point.}
\]
\[
x_{old} \leftarrow x_{new}
\]
\[
y_{old} \leftarrow y_{new}
\]
\[
\text{EXIT_LOOP WHEN DONE}
\]
\[
\text{END_LOOP}
\]

Cast the loop in this algorithm as a PYTHON procedure \texttt{trace\_field} that would be called with a statement like

\texttt{trace\_field( xold, yold, ds, n )}

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

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

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

\[
\begin{align*}
    x_{\text{new}} & \leftarrow x_{\text{old}} - ds*F_Y/F_M \\
y_{\text{new}} & \leftarrow y_{\text{old}} + ds*F_X/F_M
\end{align*}
\]

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

\[
\texttt{MAG = numpy.sqrt( Hx**2 + Hy**2 )}
\]

for example, will generate an array \texttt{MAG} containing the magnitudes of the magnetic field at each available grid point. This array then conveys a \textit{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 \texttt{quiver} to explore the velocity field

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

(which happens not to depend on \(z\)) for various values of \(\omega\) and \(\alpha\). Then write a paragraph or two describing the field. \textit{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. \textit{Hint}: In
5.18. EXERCISES

establishing a scale, you may find it useful to assume a length scale $a$, recast the field in the form

$$v = a\omega \left( -\frac{y}{a} \hat{i} + \frac{x}{a} \hat{j} \right) + \alpha \hat{k} \implies \frac{v}{a\omega} = \left( -\frac{y}{a} \hat{i} + \frac{x}{a} \hat{j} \right) + \frac{\alpha}{a\omega} \hat{k}$$

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

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

of its normal modes of oscillation. Here, $A_n$ is the amplitude of the $n$-th harmonic (and may be negative to convey a $180^\circ$ 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

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

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

$$f(x,t) = e^{-\left(x-ct\right)^2} \quad \text{and} \quad f(x,t) = \sin(x) \cos(t)$$

but feel free to invent others of your own choosing. Here, we suppose that $x$ and $t$ have been cast in appropriate dimensionless units.

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 in PYTHON 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.
5.18.8 Animation

5.40. Write a program to (a) accept as input an arbitrary two- to four-digit integer with at least two different digits, (b) rearrange the four digits to create (i) the largest and (ii) the smallest integers from the four digits, (c) subtract the smallest number from the largest numbers, and (d) repeat steps (b)–(c) until the same number occurs twice in succession. As a guard against an infinite loop, limit the number of iterations to 10. You should find, probably with some surprise, that the end result is always 6174, known as a Kaprekar Constant, and that no more than seven iterations will be required. Note: If the input number or any result after you complete step (c) has only three digits, add a leading zero before moving to the next iteration. Note: If you code appropriately, the adding of zeroes may happen automatically. Optional: Explore whether numbers with three or five digits exhibit a similar property and, if they do, identify the corresponding Kaprekar Constant. Google “Kaprekar Constant” to locate several articles on this and similar interesting numbers.
5.A  Listing of plotradio.py

'''
Program plotradio.py

This PYTHON command file reads the file \texttt{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 \texttt{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 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')
# 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' )  # 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

# 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
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
```

5.F Listing of stringanim.py

```python
n = input('Mode number: ')  # Request mode of interest
n = int(n)
delay = input('Delay between plots: ')  # Request delay between plots
```
delay=float(delay)

x = np.linspace(0.0, 1.0, 101) # Set equally spaced points along string

f0 = np.sin(n*np.pi*x) # Set initial shape of string

fig, ax = plt.subplots() # Create figure and axes

for t in range(301):
    f = f0*np.cos(2.0*n*np.pi*t/100.0) # Evaluate shape at t
    ax.clear() # Clear previous figure
    ax.plot(x,f,color='black', linewidth=2) # Plot shape
    plt.ylim([-1.0, 1.0]) # Standardize y axis range
    plt.pause(delay) # Pause

if int(sys.version[0])==2: plt.show() # Add plt.show() if PYTHON 2
Chapter 6

Introduction to MAXIMA

Note: All program (.mac) files referred to in this chapter are available in the directory $HEAD/maxima, 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 MAXIMA's default search path. If so, the files can be found by MAXIMA 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.

MAXIMA, a large program for manipulating expressions symbolically, is a descendant of one of the earliest computer algebra systems, specifically of MACSYMA, which originated in the late 1960s at the Massachusetts Institute of Technology.\footnote{MACSYMA, which is no longer available, was a proprietary—and expensive—program but MAXIMA is freely available for many operating systems, including Unix, Linux, Windows, Macintosh, and even Android. (See Appendix Z for full contact information.) The terms of the GNU General Public License (GPL) to which users of MAXIMA are subject are explained in the Local Guide.} MAXIMA can perform symbolic algebra; evaluate derivatives, integrals, and Taylor series; solve algebraic and differential equations; manipulate complex numbers; find eigenvalues and eigenvectors; produce high-resolution two- and three-dimensional graphs; and accomplish numerous other tasks. In this chapter, we describe some ways by which expressions can be defined for MAXIMA and then illustrate some of the statements by which MAXIMA can be instructed to manipulate these expressions. Users should also work through the on-line MAXIMA primer (Section 6.2) and examine several of the built-in demonstrations and examples (also Section 6.2). Further details can be found in printed MAXIMA documentation (Section 6.18) and in on-line help messages accessible from within MAXIMA itself (Section 6.2). We shall refer to this documentation collectively as the MAXIMA manuals.

A typical installation of MAXIMA will make available three different user interfaces at different levels of sophistication. For the most part, we ignore the interface that runs directly in a shell window to the operating system (OS-shell or the maxima window) and limit ourselves to the two interfaces that exploit pop-up windows directly controlling the underlying program: a command-line interface (CLI, most often referred to as the xmaxima window), which—similar to the OS shell—provides coarse resolution and displays equations as they would be constructed with a typewriter, and a graphical user interface (GUI, most often referred to as the wxmaxima window), which displays equations in a more elegant form, admits display of Greek letters and other symbols, and offers menu-based assistance for those who have trouble remembering the available commands. In this chapter, we confine ourselves to describing the structure of statements without addressing the ways in which the GUI facilitates assembling them. While the statements we describe can be entered in any interface, for the most part we work in the GUI.
6.1 Beginning a MAXIMA Session

Detailed instructions for initiating a session with MAXIMA will be found in the Local Guide. In general, MAXIMA will be started either (1) by typing an appropriate command (e.g., \texttt{maxima} for the in-shell interface, \texttt{xmaxima} for the CLI, \texttt{wxmaxima} for the GUI) at the prompt from the operating system,\(^2\) (2) by double-clicking the left mouse button on an appropriate icon\(^3\) on the desktop, or (3) by selecting \texttt{X-MAXIMA} (for the CLI) or \texttt{WX-MAXIMA} (for the GUI) from a menu. Presently, a window labeled \texttt{Xmaxima console} (for the CLI) or \texttt{wxMaxima} (for the GUI) will appear. This window will offer a variety of menus. In the CLI, the window will display a message about MAXIMA and then the prompt (\%i1), which will also label the first input line (i-line); in the GUI, the window will be blank, the label (\%i1) for the first input line appearing only after that line has been submitted to MAXIMA for execution, though the string --> will appear as the line is being typed.\(^4\) In either case, MAXIMA is ready to accept a succession of statements to control what MAXIMA does on your behalf. We must from the outset be aware that

1. Internally, MAXIMA is case sensitive.

2. Lines of MAXIMA code entered at its prompt can be terminated in any of three ways, depending on whether we want to

   (a) submit the statement for execution and instruct MAXIMA to display its response, in which case we terminate the statement with a semicolon (;),

   (b) submit the statement for execution and instruct MAXIMA to suppress explicit display of its response, in which case we terminate the statement with a dollar sign ($), or

   (c) continue entering the statement on a new line, in which case we simply type (ENTER) \textit{on the main keyboard} (no semicolon or dollar sign) and continue typing the statement, terminating it with a semicolon or a dollar sign when the full statement has been typed.

   Once the full statement has been typed and terminated with a semicolon or a dollar sign, the statement is submitted for execution by typing the character from the proper row and column of Table 6.1. Whether the output is actually displayed or not, the output line (o-line) will be given the label (\%on), where \textit{n} is the appropriate sequential number corresponding to the number on the i-line.\(^5\)

3. To place two or more statements in a single physical line, terminate each statement with a semicolon or a dollar sign but hold off actually submitting the statement for execution until the final statement has been typed and terminated. All statements will be executed and each output line (o-line) will be assigned its own label, whether or not the line is displayed. Further, the label on the next i-line will have a number incremented by one from that of the last o-line.

4. Selecting ‘Interrupt’ from the FILE menu (CLI) or the MAXIMA menu (GUI) causes MAXIMA to abort its present activity and return to a top-level input line, though there may be a short delay because MAXIMA takes time to clean up its workspace before actually returning to its top level. Alternatively, typing \langle\text{CONTROL}/g\rangle (CLI) or \langle\text{CONTROL}/G\rangle (GUI) will achieve the same objective.

5. Selecting ‘Exit’ from the FILE menu, double-clicking ML on the MAXIMA icon in the extreme upper left corner of the \texttt{MAXIMA} window, or clicking ML on the ‘\times’ button in the upper right corner of the \texttt{MAXIMA} window causes MAXIMA to terminate execution, returning control to the computer’s operating system. You may be asked whether to confirm or cancel this action before the action is actually taken.

\(^2\)In these commands to the operating system, case may be important.

\(^3\)The icon for the CLI is likely to be named \texttt{X-MAXIMA}; the icon for the GUI is likely to be named \texttt{WX-MAXIMA}.

\(^4\)The MAXIMA system variable \texttt{inchar} defines the character used to label input lines. By default, \texttt{inchar} has the value \texttt{i}, but that default value can be changed by assigning a different string to \texttt{inchar}. See Section 6.6.

\(^5\)The MAXIMA system variable \texttt{outchar} defines the character used to label output lines. By default, \texttt{outchar} has the value \texttt{o}, but that default value can be changed by assigning a different string to \texttt{outchar}. See Section 6.6.
Table 6.1: Characters used to submit statements to MAXIMA. Here, ⟨ENTER⟩ refers to the ⟨ENTER⟩ key in the main part of the keyboard, ⟨NUM/ENTER⟩ refers to the ⟨ENTER⟩ key in the numeric keypad, and ⟨CONTROL/ENTER⟩ means to hold down the ⟨CONTROL⟩ key while typing the ⟨ENTER⟩ key in the main part of the keyboard.

<table>
<thead>
<tr>
<th>OS-shell</th>
<th>X-MAXIMA</th>
<th>WX-MAXIMA</th>
</tr>
</thead>
<tbody>
<tr>
<td>Windows</td>
<td>⟨ENTER⟩ or ⟨CONTROL/ENTER⟩</td>
<td>⟨ENTER⟩ or ⟨NUM/ENTER⟩ or ⟨CONTROL/ENTER⟩</td>
</tr>
<tr>
<td>LINUX, UNIX, MAC</td>
<td>⟨ENTER⟩ or ⟨NUM/ENTER⟩ or ⟨CONTROL/ENTER⟩</td>
<td>⟨ENTER⟩ or ⟨CONTROL/ENTER⟩</td>
</tr>
</tbody>
</table>

6. In the course of a lengthy MAXIMA session, MAXIMA’s workspace may become cluttered with things no longer needed. Sometimes, we would like to restore MAXIMA to its initial state. The simplest way to achieve this objective is to exit from MAXIMA and then restart it. To restore MAXIMA (more or less) to its state immediately upon launch without exiting and restarting, invoke any of the following:

- Execute the pair of statements

  (%i25) reset(); kill(all);

  on a single line in the listed order. Together, these statements will remove all defined labels and reset most global variables and options to their default values.

- Select ‘Restart Maxima’ from the MAXIMA menu in the wxMaxima window or ‘Restart’ from the File menu in the Xmaxima window.

- Select Clear Memory from the MAXIMA menu in the wxMaxima window or ‘kill(all)’ from the MAXIMA→CLEAR MEMORY menu in the Xmaxima window

6.2 **On-Line Help**

MAXIMA makes several sorts of on-line help available:

1. A description of any specific command will be displayed if the command ? CommandName; e.g., ? expand; is typed at the MAXIMA prompt. Note that the space after the question mark is mandatory.

2. Users can peruse on-line manuals by selecting whatever of ‘MAXIMA Manual’, ‘Xmaxima Manual’, ‘MAXIMA Help’ and/or ‘wxMaxima Help’ is available in the Help menu. Further, the MAXIMA Primer pops up automatically when the CLI is invoked and can, alternatively, be displayed by opening the file $MAXHEAD/maxima/5.36.1/xmaxima/intro.html in a browser.6

6Here, $MAXHEAD$ identifies the top-level directory in which files associated with MAXIMA are stored; it is defined for your site in the Local Guide. Further, the folder 5.36.1 in this path identifies the version of MAXIMA in use and will be different if an alternative version is installed in your system.
3. The Help menu in the GUI offers the option to display built-in examples of some of the MAXIMA functions. For example, in the GUI, selecting ‘Example...’ from the Help menu brings up a pop-up box in which you can type the name of a command, e.g., diff, and then click ML on ‘OK’ to display one or more examples of the use of the specified command. Alternatively, in either interface, the statement

```
example( ExampleName );
```

will also display the specified example. However invoked, the statement effects a reading of one or more statements from the selected example file and displaying the output of each statement, doing so without user intervention at any point. Some of these examples are quite long. Scrolling through the individual components after they have all been displayed is, of course, possible. The statement `example();` returns a long list of all available examples.

Your attention is drawn particularly to the examples `complex`, `diff`, `expand`, `integrate`, `matrices`, `radcan`, `ratexpand`, `ratsimp`, `solve`, `trig`, and `ode2`.

4. A built-in demonstration having the name `DemonstrationName` and illustrating the use of a particular MAXIMA capability can be invoked by typing the statement

```
demo( DemonstrationName );
```

e.g., `demo( cf );`, on any input line.\(^7\) When this statement is invoked on an available demo, the components are displayed one at a time. To continue to the next component, the user must type ; and then ⟨ENTER⟩ on the numeric keypad or ⟨CONTROL/ENTER⟩ on the main keyboard each time the display pauses.

In the CLI, a demonstration may be stopped before it comes to its natural end by selecting ‘Input prompt’ (equivalent to typing ⟨ALT-s⟩) or ‘Restart’ from the File menu; in the GUI, a demonstration may be stopped prematurely by selecting ‘Restart Maxima’ from the MAXIMA menu.

There are very few useful built-in demonstrations. Note, however, that most built-in examples can be displayed line by line rather than non-stop by typing the statement

```
demo( "ExampleName.mac" );
```

5. By selecting ‘Tutorials’ in the Help menu in the GUI, the user can bring up the `wxMaxima documentation` window, which provides links to numerous tutorials.

6. The statements

```
describe( CommandName );  or  ? CommandName
```

e.g., `describe(diff);` or `diff;` typed at any input line will bring up either a description of the command or a list of topics available for the specified command and offer the option of viewing one or more of those topics.

\(^7\)When no path is specified, `demo` searches in the directories provided in the option variable `file_search_demo`, which has a default value but which can also be modified by statements described in Section 6.9.
6.3 Basic Entities in MAXIMA

At the very beginning, we point out that MAXIMA distinguishes among a variety of different entities, including:

- **expressions, lists, and arrays**, which are the primary quantities with which MAXIMA works;
- **functions**, which accept zero, one, or more arguments as input and return various outputs, thereby providing the means by which expressions, lists, and arrays are manipulated;
- **option variables** (some of which are called *plot options*), whose values—default or user-specified—modify the behavior of associated functions;
- **system variables**, which are maintained by MAXIMA but contain possibly useful information;
- **special symbols**, including \%pi for \(\pi\), \%i for \(\sqrt{-1}\), \%e for \(e\), \%gamma for the Euler-Mascheroni constant, inf for \(+\infty\), and minf for \(-\infty\);
- **keywords**, which may appear among the arguments to functions and modify the output produced by the functions.

6.4 Variable Names in MAXIMA

Legal variable names, sometimes called *identifiers*, begin with an alphabetic character (upper or lower case), a percent sign, an underscore character, or a backslash followed by a numeral. Later characters may be any of these characters, but only an initial numeral requires a preceding backslash. Variable names can be of any length. Lower case and upper case letters are treated as distinct, so the variable ABC is different from the variable abc.

Greek letters cannot be displayed as such in the CLI (*xmaxima* window), though alpha or Alpha, for example, can certainly be the name of a variable. In the GUI (*wxmaxima* window), with three exceptions, spelling out the name of a Greek letter in lower-case letters in an i-line will result in the display of the lower-case Greek letter in the associated o-line, e.g., alpha in an i-line will produce \(\alpha\) in the corresponding o-line. Also in the GUI, if the initial letter of the name of a Greek character in an i-line is made upper case, the corresponding upper-case Greek letter is produced in all cases (i.e., with no exceptions).\(^9,10\)

Note particularly that MAXIMA will by default assume that any variable to which no explicit value has been assigned represents a real quantity.

6.5 Expressions in MAXIMA

Expressions are written in MAXIMA using a syntax very much like that used in ordinary algebra and are supplied merely by typing the expression at a input line,\(^11\) e.g.,

\[8\]

\(^8\) The three exceptions are (1) \%gamma produces an upper-case \(\Gamma\), (2) lambda is left as lambda, and (3) psi produces an upper-case \(\Psi\).

\(^9\) Whether the initial letter is lower or upper case, preceding the Greek name with a percent sign \% will display the corresponding lower or upper case Greek letter in all cases, though \%gamma, displayed \(\gamma\) is a system variable that is bound to the value 0.577215..., the value of the Euler-Mascheroni constant, \%pi, displayed \(\pi\), is a system variable that is bound to the value 3.14159265..., and \%phi, displayed \(\varphi\), is a system variable bound to the value of the golden mean \((1 + \sqrt{5})/2 = 1.618033...\).

\(^10\) In the GUI, preceding the (spelled-out) name of a Greek letter with a backslash ‘\’ in an input line will generate the corresponding upper- or lower-case Greek letter in the output line, though with the same three exceptions enumerated above (\%gamma produces \(\Gamma\), \%lambda produces lambda, and \%psi produces \(\Psi\)).

\(^11\) In this section, we use non-sequential statement numbers so as 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.
to which, since the expression was terminated with a semicolon rather than a dollar sign, MAXIMA will respond:\(^\text{12}\)

\[
(\%o12) \quad a \cos(b x) + x
\]

The symbols +, -, \(*\), /, and ^ are used for addition, subtraction, multiplication, division, and exponentiation.\(^\text{13}\) Except for those in quoted strings, spaces are ignored by MAXIMA and may be freely inserted for legibility. Functions available for use in expressions include:

- General functions, e.g., abs(), float(), bfloat(), fix(), round(), sqrt(), signum(), unit_step(), and random().
- Trigonometric, hyperbolic and exponential functions and their inverses, e.g., sin(), cos(), tan(), atan(), atan2(), sinh(), cosh(), tanh(), atanh(), exp(), and log().
- Special functions, e.g., Bessel and modified Bessel functions, Airy functions, elliptic integrals, gamma and beta functions, error function, Legendre polynomials, Hermite polynomials, and Laguerre polynomials.

Full details on these and other functions, including information about the form of their arguments and option variables that affect their behavior, will be found in the MAXIMA manuals.

Internally, MAXIMA sees simple expressions as lists. The first element (known as part zero of the expression) indicates the operator involved, and the remaining elements (known as parts one, two, ...) indicate the operands. Thus, for example, the two expressions

\[
a + b + c + d
\]

and

\[
a \ast b \ast c
\]

are stored internally as the lists

\[
[ +, a, b, c, d ] \quad \text{and} \quad [ *, a, b, c ]
\]

respectively. More complicated expressions, such as

\[
a + b - c + d \quad \text{and} \quad a + b(c + d) + e
\]

are stored as lists of lists, being

\[
[ +, a, b, [-, c], d ] \quad \text{and} \quad [ +, a, [+b, [+c, d]], e ]
\]

respectively. As a final example, which anticipates a later need, suppose the expression of interest is

\[
(x + 3)^2 (x - a)(x + a)
\]

We would generate MAXIMA’s internal representation in several steps finding

\[
[ *, (x+3)^2, x-a, x+a ]
\]

\[
[ *, [-, x+3, 2], [+x, -a], [+x, a] ]
\]

\[
[ *, [-, [+x, 3], 2], [+x, [-, a]], [+x, a] ]
\]

\(^\text{12}\) Output of MAXIMA on different platforms may not be character for character identical to what is displayed in this book. Also note that we will occasionally rearrange the display of MAXIMA output lines to improve legibility. No substance is lost in those rearrangements.

\(^\text{13}\) Following the pattern in FORTRAN, MAXIMA also recognizes the symbol ** for exponentiation.
Here, the first line shows parts 0, 1, 2, and 3 of the original expression while the second line reveals that part 1 of the original expression is itself a list having part 0 (part 0 of part 1 of the original expression; *), part 1 (part 1 of part 1; x+3), and part 2 (part 2 of part 1; 2). Indeed, the third line reveals that some of the parts of the parts themselves have parts. Thus, for example, part 2 of part 1 of part 1 of the original expression is 3 while part 1 of part 2 of part 2 of the original expression is a. These relationships can be involved and confusing, but some manipulations depend critically on being able to identify the parts, subparts, subsubparts, and... of a complicated expression. Note that, in many cases, part 0 of an expression or subexpression is the operator to be applied to the remaining parts of the expression.

6.6 Binding Expressions 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. MAXIMA provides for this shorthand in two ways. First, as a session proceeds, MAXIMA maintains an internal record of the contents of each i-line and each o-line and identifies each such item with the label on the line on which it appeared. Each item can then be referred to in subsequent statements by that label (%i?) or (%o?), where ? stands for the number assigned to the line. For example, a later request that MAXIMA square the expression on line (%i12) in the previous section could be communicated to MAXIMA with the statement

(%i18) %i12^2;
(%o18) (a cos(b x) + x)^2

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 chosen variable. The statement

(%i20) bigexp : x + a * cos(b*x);
(bigexp) a cos(b x) + x

for example, binds the expression following the colon to the variable bigexp. Subsequently, whenever that variable is included in an expression, it will automatically be replaced by the expression to which it has been bound, and the example on line (%i18) above might then have been written

(%i30) bigexp^2;
(%o30) (a cos(b x) + x)^2

Binding an expression to a variable is preferred over using line labels because the chosen variable can have mnemonic significance, and it is decidedly easier to remember the variable to which a particular expression has been bound than to remember the label on an i-line or o-line, especially after the associated line has scrolled out of the MAXIMA window.

Even if a mnemonic name is used, we can remind ourselves of the value bound to a particular variable simply by asserting its name. The statement

(%i47) bigexp;

for example, will request MAXIMA to display the current value of bigexp on (%o47). Further, if we have no further need for a particular variable and wish to tell MAXIMA to forget that it once had a value bound to the variable, we could enter the statement

\[ ^{14}\text{In recent versions of MAXIMA, the label o-label that would normally be displayed when a variable is explicitly assigned a value may be replaced with the name of that variable, though the corresponding o-label—here (%o20)—will still be assigned and may be used to refer to that output in subsequent statements. Throughout CPSUP, we will probably not consistently use either format of the label on an output line.} \]
(\%i83) \text{remvalue( bigexp );}

to remove the variable and free some of MAXIMA’s workspace.

When special functions of significance to a particular session would be useful, we might exploit
the MAXIMA operator \text{:=} to append any desired functions to the built-in functions. For example,
the statement

(\%i121) \text{spectrig(x) := 1 + sin(x)^2;}

typed at any input line defines the function \text{spectrig()} for subsequent use just as we would use
any of MAXIMA’s built-in functions. Further, if we have no further need for a particular function
and wish to tell MAXIMA to forget that it once associated a function with a particular variable, we
could enter the statement

(\%i175) \text{remfunction( spectrig );}

to remove the function and free some of MAXIMA’s workspace.

\section*{6.7 Using MAXIMA \ldots}

At base, a session with MAXIMA involves entering one or more expressions and then instructing
MAXIMA to process these expressions in some way. The following transcripts of MAXIMA sessions
illustrate some of the simpler manipulations. The transcripts are presented in two columns, the one
on the left recording the statements submitted to MAXIMA and MAXIMA’s responses, the one on
the right containing explanatory comments.

As a statement is being entered at a MAXIMA prompt, both in the CLI and in the GUI, the
arrow, home, and end keys can be used to move the cursor around in the line and on the screen, the
backspace and delete keys can be used to remove characters, and all printing keys can be used to
insert characters at the point of the cursor. Further, the cursor need not be at the end of a line for the
line to be submitted for execution. After a statement has been submitted for execution, the cursor
can be moved back into that line, edits can be made, and the line can be resubmitted. Depending
on the particular implementation of MAXIMA in use, the record of the activity on the screen may
become unclear in one way or another. For example, in the CLI in Windows, the original i-line label
will remain on the screen and continue to identify the original content of the line even though the
statement displayed on the line will reflect the edits made. In contrast, in the GUI in Windows,
the original i- and o-line labels will disappear from the screen altogether and be replaced by new
labels, even though the no-longer-displayed labels remain identified with their original meaning.
Editing and resubmitting existing lines may create a confusing, out-of-order on-screen record of the
unfolding activity.

To keep the several segments in this section somewhat separated from one another, we shall
periodically either (1) exit from MAXIMA as described in item 5 of Section 6.1 and reenter MAXIMA
or (2) restart MAXIMA as described in item 6 of Section 6.1, thereby clearing MAXIMA’s workspace
and restarting the i- and o-line labels. This action is taken primarily to avoid having a single string of
line labels throughout this section. The action is 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 MAXIMA.

Finally, note that, throughout this section, various components of MAXIMA (variable names,
functions, option variables, plot options, system variables, operators, \ldots) are introduced simply
by presenting statements in which they are used, frequently with \textit{brief} comment. Details on all
mentioned entities and on many others can be found in the MAXIMA manuals.
6.7. USING MAXIMA ...

6.7.1 ... for Arithmetic

We illustrate first some of the ways in which MAXIMA can be instructed to perform simple arithmetic operations. Consider, for example, the statements

(\%i1) 2 + 3;
(\%o1) 5

Enter a simple numeric expression. MAXIMA returns a value, doing the arithmetic.

(\%i2) 4/6;
(\%o2) \frac{2}{3}

Enter another expression. MAXIMA does not automatically generate a floating point evaluation but did reduce the fraction.

(\%i3) sqrt(9);
(\%o3) 3

Compute \sqrt{9}.

(\%i4) sqrt(12);
(\%o4) 2\sqrt{3}

Compute \sqrt{12}. MAXIMA simplifies the expression but keeps it exact.

(\%i5) factorial(9);
(\%o5) 362880

Find 9! = 9 \times 8 \times 7 \times 6 \times 5 \times 4 \times 3 \times 2 \times 1, with factorial.

(\%i6) 9!;
(\%o6) 362880

Find 9! with postfix operator !.

(\%i7) 9!!;
(\%o7) 945

Find 9 double-factorial, 9!! = 9 \times 7 \times 5 \times 3 \times 1.

(\%i8) float( \%o2 );
(\%o8) 0.6666666666666666

Ask for floating value. By default, the value is computed as a double-precision value. When the option variable fpprintprec has a value between 2 and 16 inclusive, that value determines the number of digits actually printed by float. For other values of this variable (including the default value 0), 16 digits will be printed. Note, however, that trailing zeroes will be suppressed.

(\%i9) bfloat( \%o2 );
(\%o9) 6.6666666666666667b-1

Ask for bigfloat precision floating value. That MAXIMA has returned such a value is conveyed by the b preceding the exponent in the output. The number of digits actually displayed is controlled by fpprintprec unless fpprintprec exceeds the option variable fpprec, in which case fpprec, with default value 16, stipulates the number of digits to be displayed. Again trailing zeroes are suppressed.

(\%i10) [fpprec, fpprintprec];
(\%o10) [16, 0]

Display current values of controlling option variables.

(\%i11) fpprec:24$ fpprintprec:30$

Reset those values. Here, the dollar signs in line (\%i11) suppress display but not behind-the-scenes evaluation of MAXIMA’s response. Note that the labels \%o11 for fpprec and \%o12 for fpprintprec are still assigned.

(\%i12) bfloat(t);
(\%o12) 6.6666666666666667b-1

Display bfloat(t) again, this time with the number of digits controlled by fpprec, since fpprintprec > fpprec.

(\%i13) reset( fpprec, fpprintprec )$

Reset default values of option variables.
We have, of course, illustrated only some (+, /, sqrt, factorial, !, !!, float, bfloat) of the 
operators and functions that might be used to persuade MAXIMA to manipulate with numbers and 
only two (fpprec and fpprintprec) of the option variables whose values affect the behavior of these 
functions. You should make it a point to read about these and other functions, option variables, 
and operators in the MAXIMA manuals.

6.7.2 . . . for Algebra

MAXIMA is, of course, capable of much more than simple arithmetic. We next illustrate how 
algebraic manipulations can be requested, beginning with the statements

(\%i1) (x^2-a^2) * (x+3)^2;
(\%o1) (x + 3)^2(x^2 - a^2)

Enter an expression. Note in the o-line 
that the order of factors may—and here 
does—differ from that entered. MAXIMA 
arranges expressions internally to facilitate 
their subsequent manipulation.

(\%i2) expand(\%);
(\%o2) x^4 + 6x^3 - a^2x^2 + 9x^2 - 6a^2x - 9a^2

Expand the expression. Here, the symbol \% 
refers to the most immediately past o-line.

(\%i3) factor(\%);
(\%o3) (x + 3)^2(x - a)(x + a)

Factor the expression on line \(\%o2\).

We note, of course, that this factoring hasn’t quite returned us to our starting point. To achieve that 
end, we need to multiply out the second and third factors without expanding the first. In essence, 
we will have to carve the expression apart, manipulate with its parts, and then reconstruct the 
expression. As discussed in Section 6.5, MAXIMA sees this expression as lists of lists. Indeed, we 
can persuade MAXIMA to reveal that structure by exploiting the MAXIMA command part in the 
statement

(\%i4) [part(%o3,0), part(%o3,1), part(%o3,2), part(%o3,3)];
(\%o4) [ *, (x + 3)^2, x - a, x + a ]

or we could exploit not only the function part but also the command print and MAXIMA’s ability 
to construct a loop in the statement\(^{15}\)

(\%i5) for i from 0 thru 3 do print( part(%o3,i) );
  *
   (x + 3)^2
   x - a
   x + a
(\%o5) done

Using these capabilities, we might achieve the entire objective with the single nested statement

(\%i6) part(%o3, 1) * expand( part(%o3,2)*part(%o3,3) );
(\%o6) (x + 3)^2(x^2 - a^2)

\(^{15}\)In more recent versions of MAXIMA, a colon : can be substituted for from in this statement. We will adhere to 
the older syntax, which still works in these more recent versions.
in which we extract parts 1, 2, and 3 from \(\%o3\), expand the product of parts 2 and 3, and then reconstruct the expression by multiplying that result by part 1. This time, we have succeeded in reconstructing the original expression. As here illustrated, beating MAXIMA’s version of a complicated expression into a more conventional form may require a fair bit of creative massaging.

We continue with some additional manipulations. First, we set the constant \(a\) in the previous result to the specific value 2 with the statement

\[
(\%i17) \text{ev( } %, \text{ a=2 );}
\]

Evaluate \%o6 with \(a = 2\). The simpler statement \(\%, \text{ a=2};\) would accomplish the same end without invoking \text{ev}.

Then we find a partial fraction expansion of the reciprocal of this polynomial with the statement

\[
(\%i18) \text{partfrac( 1/%, x );}
\]

Find partial fraction expansion of \(1/%o7\) expanding with respect to \(x\).

\[
(\%o8) \frac{6}{25(x + 3)} + \frac{1}{5(x + 3)^2} - \frac{1}{4(x + 2)} + \frac{1}{100(x - 2)}
\]

Numerous rewrites of this expression are now possible. For example, replacing \(x\) with \(3z\) would be accomplished with the command \text{subst}, as for example in the statement

\[
(\%i19) \text{subst(3*z, x, % );}
\]

Substitute \(3z\) for \(x\) in \%o8.

\[
(\%o9) \frac{6}{25(3z + 3)} + \frac{1}{5(3z + 3)^2} - \frac{1}{4(3z + 2)} + \frac{1}{100(3z - 2)}
\]

To illustrate a more complicated recasting, suppose we wanted to square the denominator of the second term in \%o8 without affecting any other term. The statement \text{expand(\%o8)} does more than we want; it will expand every denominator. (Try it.) We need a more sophisticated operation that rearranges only the second term (part 2 of the original expression). We extract that term, carve it up, manipulate that term alone, and substitute the result of that manipulation into the original expression as the second term. We identify the proper part with the statements

\[
(\%i10) \text{part( \%o8, 2 );}
\]

Extract second part of \%o8.

\[
(\%o10) \frac{1}{5(x + 3)^2}
\]

\[
(\%i11) \text{part( \%o8, 2 );}
\]

Extract second part of second part of \%o8.

\[
(\%o11) \frac{6x}{x + 3}
\]

\[
(\%i12) \text{part( \%o8, 2 );}
\]

Extract second part of second part of second part of \%o8.

\[
(\%i13) \text{part( \%o8, 2, 2 );}
\]

Alternatively, extract the second part (third 2) of the second part (second 2) of the second part (first 2) of \%o8 in one step.

\[
(\%o13) \frac{(x + 3)^2}{x + 3}
\]

Then we expand the extracted piece of the denominator of the second term in the original expression and substitute it into \%o8 as part 2,2,2 with the statement

\[
(\%i14) \text{subpart( expand( \% ), \%o8, 2, 2, 2 );}
\]

\[
(\%o14) \frac{1}{5(x^2 + 6x + 9)} + \frac{6}{25(x + 3)} - \frac{1}{4(x + 2)} + \frac{1}{100(x - 2)}
\]

\[
(\%i15) \text{reset()$ kill(all)$}
\]

See item 6 in Section 6.1.

Note once again that MAXIMA has a mind of its own regarding the order of terms in an expression.

We have, of course, only mentioned a few (\text{expand, factor, part, ev, partfrac, subst, subpart}) of the numerous functions MAXIMA makes available for manipulating expressions algebraically. We have also mentioned the function \text{print} and the control structure \text{for-do}. The
differences among \texttt{expand, combine, multthru, distrib,} and \texttt{xthru} are explored in one of the exercises. Additional important functions include \texttt{ratexpand, factorsum, lhs, rhs, coeff, pickapart, rempart, num, denom, radcan, ratsimp, fullratsimp,} and \texttt{ratsubst}. Further, the action of one or more of these functions may be influenced by one or more of the option variables \texttt{algebraic, dontfactor, logexpand, radexpand, ratexpand, simplify_products,} and perhaps others. You should make it a point to read about these functions and option variables in the MAXIMA manuals.

6.7.3 ... for Complex Variables

MAXIMA is also capable of manipulating complex variables. We illustrate some of MAXIMA’s main capabilities in this category with the statements

\begin{verbatim}
(%i1) fct : sin(x+y*%i);
(%o1) sin(%iy + x)

(%i2) realpart( fct );
(%o2) sin(x) cosh(y)

(%i3) imagpart( fct );
(%o3) cos(x) sinh(y)

(%i4) conjugate( fct );
(%o4) -sin(%iy - x)

(%i5) (x+%i*y)/(x-%i*y);
(%o5) %iy + x
x - %iy

(%i6) [realpart(%), imagpart(%)];
(%o6) [x^2 - y^2, 2xy]
   y^2 + x^2

(%i7) abs( %o5 );
(%o7) |%iy + x|
   |%iy - x|

(%i8) cabs( %o5 );
(%o8) 1

(%i9) q : realpart(%o5) + %i*imagpart(%o5);
(%o9) x^2 - y^2
   y^2 + x^2 + 2%ixy
   y^2 + x^2

(%i10) q1 : carg( q );
(%o10) atan2(2xy, x^2 - y^2)
   y^2 + x^2

(%i11) q2 : carg( %o5 );
(%o11) atan2(2xy, x^2 - y^2)
   y^2 + x^2

(%i12) q3 : carg( %o5 )
   (%i13) q3 : carg( %o5 )
\end{verbatim}

\texttt{Define complex expression.} MAXIMA’s symbol \texttt{%i} stands for $\sqrt{-1}$ in both i- and o-lines.

\texttt{Extract real and imaginary parts.} Note that MAXIMA assumes x and y to represent \emph{real} quantities.

\texttt{Find complex conjugate.}

\texttt{Define another complex expression.}

\texttt{Extract real and imaginary parts as a list.}

\texttt{Find absolute value of \texttt{%o5}.} MAXIMA assumes x and y are real but notes \texttt{%i} = $\sqrt{-1}$, so punts on the evaluation.

\texttt{Find complex absolute value of \texttt{%o5}.} \texttt{cabs} also assumes x and y to be real but knows how to deal with \texttt{%i} = $\sqrt{-1}$. \texttt{cabs} also simplifies the result obtained from simply the square root of the sum of the squares of the real and imaginary parts!

\texttt{Recast \texttt{%o5} as sum of real and imaginary parts.}

\texttt{Find complex argument of \texttt{q}.} Note that \texttt{carg} gives a different—but presumably equivalent—result if used in the form \texttt{carg(\texttt{%o5})};.

In the evaluation of \texttt{atan2}, only the ratio of the two arguments is important. Recognizing that the first and second arguments of the \texttt{result} in \texttt{q1} are parts 1 and 2 of \texttt{q1}, we can simplify the result just obtained by invoking the statement
6.7. USING MAXIMA ...

(%i11) [p1:part(q1,1), p2:part(q1,2)];  
Extract and name arguments of q1.

(%i12) p1/p2;  
Calculate ratio of those arguments.

(%i13) q2: atan2(num(%), denom(%));  
Reconstruct arctangent using numerator and denominator of the ratio of the two original arguments.

(%i14) x + %i*y;  
Define a complex number.

(%i15) polarform( %o14 );  
Convert to polar form.

(%i16) r*exp(%i*theta);  
Define another complex number.

(%i17) rectform(%o16 );  
Convert to rectangular form.

(%i15) reset()$ kill(all)$  
See item 6 in Section 6.1.

We have, of course, only mentioned some (realpart, imagpart, conjugate, abs, cabs, carg, polarform, rectform) of the functions MAXIMA makes available for manipulating complex numbers. You should make it a point to read about these functions and any associated option variables in the MAXIMA manuals.

6.7.4 ... for Trigonometry

MAXIMA 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

(%i11) trigexp : sin(x)*cos(x);  
Define a trig expression.

(%o11) cos(x)sin(x)

(%i12) exponentialize(trigexp);  
Convert to exponential form.

(%o12) \(-%i \left(\%e^{ix} - \%e^{-%i x}\right)\left(\%e^{ix} + \%e^{-%i x}\right)\) / 4

(%i13) demoivre(%);  
Convert to standard trig functions.

(%o3) cos(x)sin(x)

(%i14) trigreduce(trigexp);  
Recast with multiple angle identities.

(%o4) \(\sin(2x)\) / 2

(%i5) trigexpand(%);  
Expand functions of multiple angles.

(%o5) cos(x)sin(x)


\[(\%i6) \sin(2x+y); \]
\[%o6\] \(\sin(y+2x)\)

\[(\%i7) \text{trigexpand}(\%); \]
\[%o7\] \(\cos(2x)\sin(y)+\sin(2x)\cos(y)\)

\[(\%i8) \text{trigexpand}(\%); \]
\[%o8\] \(2\cos(x)\sin(x)\cos(y)\)

\[(\%i9) (\cos(x)+\sin(x))^2; \]
\[%o9\] \(\sin^2(x)+2\cos(x)\sin(x)+\cos^2(x)\)

\[(\%i10) \text{trigsimp}(\%); \]
\[%o10\] \(2\cos(x)\sin(x)+1\)

To illustrate the use of a loop once again, let us create a list of the values of the sine of integer multiples of \(\pi/2\). We use the statements

\%(i12) \text{for n from 0 thru 10 do a[n]: sin(n*%pi/2);} \quad \text{MAXIMA's symbol %pi stands}
\%(i12) \text{done}
\%(i13) \text{makelist( a[n], n, 0, 10 );}
\%(i13) \[0, 1, 0, -1, 0, 1, 0, -1, 0, 1, 0\]

\%(i14) \text{reset(); kill(all);} \quad \text{See item 6 in Section 6.1.}

We have, of course, only mentioned a few (\textit{exponentialize}, \textit{demoivre}, \textit{trigreduce}, \textit{trigexpand}, \textit{expand}, and \textit{trigsimp}) of the functions MAXIMA makes available for doing trigonometry. We have also illustrated again the use of the control structure \textit{for/from-thru/do}. Additional important functions include \textit{exp}, \textit{log}, \textit{logarc}, and \textit{logcontract}. Further, the action of one or more of these functions may be influenced by one or more of the option variables \%emode, \textit{exponentialize}, \textit{numer}, \textit{trigexpandplus}, \textit{trigexpandtimes}, \textit{triginverses}, \textit{logexpand}, \textit{logarc}, \textit{demoivre}, and \textit{halfangles}, among others. You should make it a point to read about these commands and option variables in the MAXIMA manuals.

### 6.7.5 ... for Algebraic Equations

We turn next to illustrating a few MAXIMA commands for manipulating expressions algebraically. To define a polynomial and find its roots, for example, we would use statements like

\%(i11) \text{poly : expand( (x^2-a^2) * (x+3)^2 );}
\%(i12) \text{x^4 + 6x^3 - a^2x^2 + 9x^2 - 6a^2x - 9a^2}
\%(i12) \text{solve( poly = 0, x );}
\%(i12) [x = -a, x = a, x = -3]

\%(i13) \text{multiplicities;}
\%(i13) [1, 1, 2]

\%(i14) \text{ev( poly, x = -a );}
\%(i14) 0

Define a polynomial.

Define another trig function.

Expand using trig addition formulae.

Do it again.

Define yet another trig function.

Expand it.

Simplify it. Clearly, \textit{trigsimp} knows the standard trigonometric identities.

Verify first root. Alternatively, we could have submitted the statement \textit{poly, x=-a;} with the same end result.
Verify all roots, using the for-do structure to code a loop.

Supply a quadratic equation.

Solve \( b^2 - 4ac \) for \( x \). Clearly, MAXIMA knows the quadratic formula.

Evaluate \( \frac{b^2 - 4ac + b}{2a} \) with indicated values of \( a, b, \) and \( c \). Note that MAXIMA does not detect the common factor of 2 in numerator and denominator of these expressions, though factor applied to \%o8 will effect that simplification.

In the second statement, we have invoked expand to multiply the initial factor in each solution into each term of the following binomial.

Finally, let us illustrate MAXIMA’s capability to solve simultaneous linear equations. To define and solve a representative system, we invoke the statements

Give MAXIMA a pair of simultaneous linear equations, binding each to a variable to facilitate subsequent reference.

Note that each solution—here there is only one—is presented as a list. That there are two opening and closing square brackets anticipates situations in which there are two or more solutions. The structure of the output would then be \([,], [,], \ldots\), and the pair of square brackets at the extreme ends would seem less redundant.

Try it with a more symbolic system.

Remove values from eqn1, eqn2.
CHAPTER 6. INTRODUCTION TO MAXIMA

We have, of course, only mentioned a few \texttt{(solve, multiplicities)} of the functions and system variables MAXIMA makes available for solving algebraic equations exactly. Behind the scenes, \texttt{solve} actually invokes either \texttt{linsolve} or \texttt{algsys}, as appropriate to the equations presented to it. Further, the functions \texttt{realroots, allroots, and newton} can find roots numerically. These functions may be influenced by one or more of the option variables \texttt{algepsilon, algexact, rootsepsilon, breakup, programmode, realonly, globalsolve, and solveexplicit}, among others. You should make it a point to read about these functions, option variables, and system variables in the MAXIMA manuals. Root finding is discussed in greater depth in Chapter 14.

6.7.6 ... for Basic Calculus

Differentiation, indefinite and definite integration, and determination of series representations of symbolic expressions are also among MAXIMA’s capabilities. To evaluate derivatives and integrals, for example, we would use statements like

\begin{verbatim}
(%i1) x^2*exp(-x^2);
(%o1) x^2 \cdot e^{-x^2}
(%i2) diff( %, x );
(%o2) 2x \cdot e^{-x^2} - 2x^3 \cdot e^{-x^2}
(%i3) integrate( %, x );
(%o3) -(-x^2 - 1) \cdot e^{-x^2} - e^{-x^2}
(%i4) ratsimp( % );
(%o4) x^2 \cdot e^{-x^2}
(%i5) factor( diff( %o1, x, 2 ) );
(%o5) 2(2x^4 - 5x^2 + 1) \cdot e^{-x^2}
(%i6) x/(a^2+x^2)^(3/2);
(%o6) x/(a^2+x^2)^{3/2}
(%i7) integrate( %, x, 0, inf );
Is a zero or nonzero? n;
(%o7) -1/a
(%i8) integrate( %o6, x, 0, inf );
Is a zero or nonzero? z;
(%o8) 1/a
\end{verbatim}

Enter an expression.

Differentiate \%o1 with respect to \(x\).

Evaluate indefinite integral of \%o2 on \(x\).

Simplify result to return to the original expression.

Differentiate expression twice with respect to \(x\) and simplify the result by factoring it.

Enter another expression.

Evaluate definite integral of \%o6 on \(x\) from \(x = 0\) to \(x = +\infty\). MAXIMA needs help. The first letter of the appropriate response is sufficient.

Evaluate the integral again with the other option.

This example has been included to draw attention to the need to be careful in using symbolic manipulating programs, since

\[
\int_0^\infty \frac{x}{(x^2 + a^2)^{3/2}} \, dx = \begin{cases} 1/|a|, & a \neq 0 \\ \text{divergent}, & a = 0 \end{cases}
\]

\begin{equation}
(6.1)
\end{equation}

MAXIMA’s value for this integral is incorrect. The question asked by MAXIMA is correct, but MAXIMA should return \(1/|a|\) for \(a \neq 0\) and something like ‘divergent’ for \(a = 0\).\footnote{The developers of MAXIMA have been alerted to this bug and will surely have corrected it in future versions of MAXIMA.}
6.7. USING MAXIMA …

Finally (for this section), we point out that MAXIMA can generate series expansions of prescribed functions. To illustrate, we ask MAXIMA to process the statements

\[
\frac{1}{1+\exp(-x)};
\]

Enter yet another expression.

\[
\frac{1}{e^{-x} + 1}
\]

Expand the expression in a Taylor series in \( x \) about \( x = 0 \), retaining terms to 5-th order in \( x \). The notation /T/ at the beginning of the o-line indicates that the line contains a Taylor series.

\[
\frac{1}{480} x^5 + \frac{-1}{48} x^3 + \frac{1}{4} x + \frac{1}{2} + \ldots
\]

We have, of course, only mentioned a few (\texttt{diff}, \texttt{integrate}, \texttt{taylor}) of the numerous functions MAXIMA makes available for doing calculus. Important additional functions include \texttt{changevar}, \texttt{gradeq}, \texttt{lddefint}, \texttt{limit}, and \texttt{romberg} (for numerical integration). Further, the action of one or more of these functions may be influenced by one or more of the option variables \texttt{derivabbrev}, \texttt{intanalysis}, \texttt{logabs}, \texttt{romberg}, and \texttt{romberg}, among others, and MAXIMA provides some information about what it has done in the system variable \texttt{dependencies}. You should make it a point to read about these functions, option variables, and system variables in the MAXIMA manuals.

6.7.7 … for Laplace Transforms

In Section 1.5.2, we defined the Laplace transform \( \hat{f}(s) \) of a function \( f(t) \) by the integral

\[
\hat{f}(s) = \int_0^\infty e^{-st} f(t) \, dt
\]

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 of the original problem. The initial conditions are automatically incorporated in this approach. Note, unfortunately, that inverting the Laplace transform of the solution to find the solution itself is rarely easy.

Within MAXIMA, the function \texttt{laplace} will calculate a Laplace transform and the function \texttt{ilt} will invert a transform to return to the original space. We illustrate the use of those functions with the statements

\[
\texttt{laplace( \sin(omega*t), t, s )};
\]

Calculate the Laplace transform of the function \( \sin(\omega t) \), where \( t \) is the primary variable and \( s \) is the variable in terms of which the transform is to be expressed.
CHAPTER 6. INTRODUCTION TO MAXIMA

(${\text{i2}})$ \text{ilt( } s, t );
Is $\omega$ zero or nonzero? \text{n};

(${\text{o2}})$ \text{sin(}\omega t)

(${\text{i3}})$ \text{ilt( } s, t );
Is $\omega$ zero or nonzero? \text{z};

(${\text{o3}})$ $\omega t$

Then, invert the transform to return to the original function. The intermediate question recognizes a distinction based on the value of the parameter $\omega$.

Seek result for the other option in the question asked. Here is another apparent error in MAXIMA: If $\omega$ is a (hard) zero, then the Laplace transform involved ($\%o1$ with $\omega = 0$) is zero and the inverse transform should also be zero.

Calculate the Laplace transform of the function $e^{-at}$.

(${\text{i4}})$ \text{laplace( } \text{exp(-a*t), t, s );}

(${\text{o4}})$ \text{1} \text{s} + \text{a}

(${\text{i5}})$ \text{ilt(1/( (s+a)*(s+b) ), s, t );}

(${\text{o5}})$ \text{1/}\text{e}^{-at} \text{b-a} - \text{1/}\text{e}^{-bt} \text{b-a}

(${\text{i6}})$ \text{reset()}$\text{kill(all)}$

See item 6 in Section 6.1.

6.7.8 ... for Ordinary Differential Equations

Among MAXIMA’s strongest suits is its ability to solve ordinary differential equations (ODEs). We will postpone introducing the more sophisticated of those capabilities until Chapter 11, limiting ourselves now to a quick illustration of the simplest of them. We first must tell MAXIMA that the function we seek—say $x(t)$—depends on the independent variable $t$ with the statement

(${\text{i1}})$ \text{depends( } x, t )$

Then, we tell MAXIMA the differential equation. For the sake of a specific example, we here choose the motion of an object of mass $m$ attached to a spring having constant $k$ and to a dashpot (shock absorber) having damping constant $b$. The appropriate equation of motion is communicated to MAXIMA with the statement

(${\text{i2}})$ $m \text{'diff(x,t,2) + b \text{'diff(x,t) + k*x = 0;}$

(${\text{o2}})$ $m \frac{d^2x}{dt^2} + b \frac{dx}{dt} + kx = 0$

Here $\text{diff(x,t,2)}$ specifies the second derivative of $x$ with respect to $t$ while $\text{diff(x,t)}$ specifies the first derivative. The opening tic (’) suppresses immediate evaluation of the derivative (and is recommended even though, in this specific situation, it is unnecessary). Next, to solve the equation for $x$ as a function of $t$, we invoke the MAXIMA command \text{ode2} by executing the statement

(${\text{i3}})$ \text{ode2( } s, x, t );
Is $m^2(4km - b^2)$ positive, negative, or zero? \text{p};

(${\text{o3}})$ $x = \%e^{-\frac{bt}{2m}} \left( \%k1 \sin \left( \frac{4k}{m - m^2} \frac{t}{2} \right) + \%k2 \cos \left( \frac{4k}{m - m^2} \frac{t}{2} \right) \right)$
Along the way, MAXIMA has requested further information about the value of a particular quantity. This request reflects MAXIMA's recognition that the character of the solution depends on whether the motion is underdamped, overdamped, or critically damped; by typing `p`, we selected the underdamped case. Note, incidentally, that `ode2` uses the symbols `%k1` and `%k2` for the initially undetermined integration constants in the solution to a second-order differential equation.\footnote{Note that, despite its name, `ode2` can solve both first- and second-order ODEs. The symbol `%c` is used for the (single) integration constant in the solution to a first-order equation.}

Now, let's clean up the expression a bit by introducing $\omega$ for the frequency $\frac{1}{2} \sqrt{\frac{4k}{m} - \frac{b^2}{m^2}}$ and $\beta$ for the combination $b/2m$ with the statements\footnote{The function `subst` will substitute its first argument for every occurrence of its second argument in the expression identified by the third argument. To work, the second argument must identify a subexpression that actually occurs as a part (in the technical sense; see Section 6.5) of the full MAXIMA expression. We could not, for example, substitute $\omega$ for $\frac{1}{2} \sqrt{\frac{4k}{m} - \frac{b^2}{m^2}}$ because this latter expression does not occur as a recognizable entity. Since the argument of the trigonometric functions would be stored internally as \([/,*,[/,,2]],2])\], the quantity \([/,*,,2])\] does not appear and will not be found.}.

\begin{verbatim}
(%i4) subst( 2*omega, sqrt( 4*k/m - b^2/m^2 ), %);
(%o4) x = %e^{bt/2m} (\%k1 sin(\omega t) + \%k2 cos(\omega t))

(%i5) subst( 2*m*beta, b, %);
(%o5) x = %e^{-\beta t} (\%k1 sin(\omega t) + \%k2 cos(\omega t))
\end{verbatim}

Then we impose more explicit initial conditions with the statement

\begin{verbatim}
(%i6) ic2( %, t=0, x=x[0], 'diff(x,t)=0 );
(%o6) x = %e^{-\beta t} \left( \frac{x_0 \beta \sin(\omega t)}{\omega} + x_0 \cos(\omega t) \right)
\end{verbatim}

We leave it to the reader to use MAXIMA to verify that this solution in fact satisfies the original equation and the imposed initial conditions.

Like the function `solve`, `ode2` actually invokes a succession of methods in turn until it finds one that works. The system variable `method`, which can be displayed with the statement

\begin{verbatim}
(%i7) method;
(%o7) constcoeff
\end{verbatim}
is set by `ode2` to the method actually used.

The difference between the function `diff` and the function `'diff` deserves more explicit explanation. As we said above, the first of these, when invoked, will result in immediate evaluation of any derivatives that can be evaluated while the second suppresses evaluation of derivatives until a later time when that evaluation is explicitly requested. To illustrate the difference, suppose we define the function

\begin{verbatim}
(%i8) f(y) := a * sin( omega * y );
(%o8) f(y) := a sin(\omega y)
\end{verbatim}

Then, the two statements

\begin{verbatim}
(%i9) diff( f(y), y );
(%o9) a\omega \cos(\omega y)
\end{verbatim}
reveal the difference. In the first, the derivative is evaluated; in the second, it is left unevaluated. If, at some later time, we wish to evaluate the derivative in \( \%o10 \), we could again use the function \( \text{ev} \), supplying an optional argument to convey the function in \( \%o10 \) that we wish \( \text{ev} \) to evaluate. The statement

\[
\text{(\%11) ev( \%o10, \text{diff} );}
\]

accomplishes this objective.

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, right after line \( \%o2 \) we might have submitted the statements

\[
\text{(\%i12) subst( omega[0]^2*m, k, \%o2 );}
\]

Remembering that \( \omega_0^2 = k/m \), where \( \omega_0 \) is the natural frequency of the oscillator, forsake \( k \) in favor of \( \omega_0 \) and \( m \).

\[
\text{(\%i13) subst(omega[0], t, \%);}
\]

Change the independent variable to \( \tau = \omega_0 t \).

\[
\text{(\%i14) subst(omega[0]*'diff(x,tau), 'diff(x,tau/omega[0]), \%);}
\]

\[
\text{(\%i15) subst(omega[0]^2*'diff(x,tau,2), 'diff(x,tau/omega[0],2), \%);}
\]

\[
\text{(\%i16) expand( \%/(m*omega[0]^2) );}
\]

Divide entire equation by \( m\omega_0^2 \) and cancel common factors.

\[
\text{(\%i17) subst(beta*m*omega[0], b, \% );}
\]

Introduce a single symbol \( \beta \) for \( b/m\omega_0 \).

This process is admittedly tedious.20 Still, at this point, we have swallowed all of the dimensional constants either into a rescaling of the time \( \tau = \omega_0 t \) or into a single dimensionless parameter \( \beta = b/m\omega_0 \). We could, of course, rescale \( x \) to some chosen dimensionless quantity, but the equation is linear in \( x \) and we can simply interpret the equation as it stands in those terms. We would then solve the equation as before with the statement21

\[
\text{21Here, the question we answered with 'n;' is equivalent to the previous question whether the motion we seek is underdamped, critically damped, or overdamped. The answer \( \beta < 2 \) is equivalent to the answer \( b/m\omega_0 = b/(m\sqrt{k/m}) < 2 \) or \( b^2/mk < 4 \) or \( b^2 < 4mk \). Thus, the dimensionless inequality \( \beta < 2 \) here is exactly equivalent to the dimensional inequality \( b^2 < 4km \) that appeared earlier in this section.}
\]

---

20The parent program MACSYMA had a function \( \text{ode_indeptran} \), which facilitated the transformation of the independent variable in ordinary differential equations. MAXIMA has apparently not implemented that feature.
6.7. USING MAXIMA

(%i18) ode2( %, x, tau );
Is \((\beta - 2)(\beta + 2)\) positive, negative, or zero? n;

(%o18) \(x = \%e^{-\beta \tau/2} \left( \%k1 \sin \frac{\sqrt{4 - \beta^2 \tau^2}}{2} + \%k2 \cos \frac{\sqrt{4 - \beta^2 \tau^2}}{2} \right)\)

Finally, we impose the initial conditions with the statement

(%i19) ic2( %, tau = 0, x = x[0], 'diff(x,tau) = 0 );

(%o19) \(x = \%e^{-\beta \tau/2} \left( x_0 \cos(\sqrt{4 - \beta^2 \tau^2}/2) - \frac{x_0 \beta \sqrt{4 - \beta^2 \tau^2} \sin(\sqrt{4 - \beta^2 \tau^2}/2)}{\beta^2 - 4} \right)\)

which—as we leave to the reader to show—is a dimensionless form of the dimensional result obtained at line (%o6) above.

(%i20) reset()$ kill(all)$

See item 6 in Section 6.1.

We have, of course, only mentioned a few (depends, ode2, ic2, and method) of the functions and system variables MAXIMA makes available for addressing first and second order ordinary differential equations. Important additional functions having a role in this context include atvalue, odelin, ic1, bc2, and ode_check. Beyond method, MAXIMA provides some information about what it has done in the variables intfactor, odeindex, and yp. Further, the package contrib_ode can be loaded to provide additional resources for solving ODEs. You should make it a point to read about these functions, option variables, system variables, and keywords in the MAXIMA manuals.

6.7.9 . . . for Lists, Matrices, and Arrays

In MAXIMA, a list is “an ordered set of elements, separated by commas and enclosed in square brackets”; a matrix is “a two-dimensional, ordered set of elements”; and an array—the exposition of whose creation and manipulation we will leave to the MAXIMA manuals—is “an n-dimensional data structure”. A list can be created by statements like

(%i11) lst : [ sin(x), cos(x), tan(x) ];
(%o11) [sin(x), cos(x), tan(x)]

and its individual elements can be accessed with the notation lst[n], where n—the index of the element—is enclosed in square brackets and counts upwards from 1. The elements of a list can themselves be lists, as for example in

(%i12) lstlst : [ [1,2], [3,4] ];
(%o12) [[1, 2], [3, 4]]

and, as we have seen already, many of MAXIMA’s functions return their responses as lists or lists of lists.

A matrix, on the other hand, is not a list or a list of lists. Matrices are created, examined, and edited with statements like

(%i13) mat1 : matrix( [1,2], [3,4] );

(%o13) \[
\begin{bmatrix}
1 & 2 \\
3 & 4
\end{bmatrix}
\]
Examine (1,1) and (2,1) elements. Row and column indices start at 1 (not 0) and are enclosed in square brackets. Further, the (1,1) and (1,2) elements can alternatively be referred to as \texttt{mat1[1][1]} and \texttt{mat1[1][2]}. The first (second) index identifies the row (column).

\begin{verbatim}
(%i4) mat1[1,1];
(%o4) 1
(%i5) mat1[2,1];
(%o5) 3

(%i6) mat1[1,2] : 5$
(%i7) mat1;
(%o7) 
[1 5
3 4]

(%i8) mat1[1,2] : 2$
\end{verbatim}

Change (1,2) element. Verify change. Change it back.

Assorted quantities related to a particular matrix can also be created by invoking other MAXIMA functions as in the following examples:

\begin{verbatim}
(%i9) mat2 : transpose( mat1 );
(%o9) 
[1 3
2 4]

(%i10) invert( mat1 );
(%o10) 
[-2 1
3 1
2 -2]

(%i11) load(functs)$
\end{verbatim}

Load package to add function \texttt{tracematrix}.

\begin{verbatim}
(%i12) val1 : tracematrix( mat1 );
(%o12) 5

(%i13) val2 : determinant( mat1 );
(%o13) -2
\end{verbatim}

Evaluate trace (sum of diagonal elements) of \texttt{mat1}. Evaluate determinant of \texttt{mat1}.

Other common manipulations include evaluating the standard matrix product and an element-by-element product, achieved with statements like

\begin{verbatim}
(%i14) mat1.mat2;
(%o14) 
[5 11
11 25]

(%i15) mat1*mat2;
(%o15) 
[1 6
6 16]
\end{verbatim}

Evaluate the result of ordinary matrix multiplication. Evaluate the result of element-by-element multiplication.

Finally (for this section anyway), MAXIMA includes functions for finding eigenvalues and eigenvectors, which we illustrate with the statements

\begin{verbatim}
(%i16) eval1 : eigenvalues( mat1 );
(%o16) 
[5
-2]

(%i17) evl2 : eigenvectors( mat1 );
(%o17) 
[[5, [1, 2], 1], [-2, [1, 2], 1]]
\end{verbatim}

Evaluate eigenvalues of \texttt{mat1}. Evaluate eigenvalues and eigenvectors of \texttt{mat1}.

6.7. USING MAXIMA

(%i16) spinx : matrix( [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.
Multiline statements are achieved by typing
the carriage return without a ';' or by typ-
ing (SHIFT/ENTER) at the end of all lines
except the last. (See item 2 in Section 6.1.)

(%o16)

\[
\begin{bmatrix}
0 & \frac{1}{\sqrt{2}} & 0 \\
\frac{1}{\sqrt{2}} & 0 & \frac{1}{\sqrt{2}} \\
0 & \frac{1}{\sqrt{2}} & 0 \\
\end{bmatrix}
\]

(%i17) eigenvectors( spinx );

Find eigenvectors and eigenvalues. Note
that the function eivects is a synonym for
eigenvectors.

(%o17)

\[
\begin{bmatrix}
[-1,1,0], [1,1,1] \\
[1,-\sqrt{2},1] \\
[1,\sqrt{2},1] \\
[1,0,-1] \\
\end{bmatrix}
\]

Show eigenvalues ([−1, 1, 0]) and
multiplicities ([1, 1, 1])
Show first eigenvector
Show second eigenvector
Show third eigenvector

At the top level, the output of eigenvectors (eivects) is a list with two (sub)lists. The first
(sub)list is itself a list of two (subsub)lists, the first of which lists the distinct eigenvalues—here
[−1,1,0]—and the second of which lists the multiplicity (degeneracy) of each eigenvalue—here
[1,1,1]. The second (sub)list contains a (subsub)list for each distinct eigenvalue. For an eigenvalue
with multiplicity (degeneracy) \( n \), the corresponding (subsub)list contains \( n \) (subsubsub)lists, whose
elements are the components of the \( n \) eigenvectors belonging to the corresponding eigenvalue. Here,
since each eigenvalue is non-degenerate, each (subsubsub)list contains only one (three-component)
list, which explains the apparently redundant double brackets around that one eigenvector. Note
that the eigenvectors as returned by eigenvectors are not (necessarily) normalized; the alternative
function uniteigenvectors (or, more simply, ueivects) is basically identical to eigenvectors
but it returns normalized eigenvectors.\(^{22}\)

(%i18) eigenvalues( spinx );

Find eigenvalues and multiplicities alone.
Note that the function eivals is a synonym
for eigenvalues.

(%o18) 

\[
[-1,1,0], [1,1,1]
\]

(%i19) reset();$ kill(all)$

We have, of course, only mentioned a few (matrix, transpose, invert, tracematrix (in
the package functs), determinant, eigenvectors, eivects, uniteigenvectors, eigenvalues,
eivals) of the functions MAXIMA makes available for manipulating matrices. Important additional
functions include addcol, addrow, adjoint, charpoly, coefmatrix, copymatrix, diagmatrix,
echelon, genmatrix, ident, minor, rank, row, submatrix, and zeromatrix. You should make
it a point to read about these functions and any associated option variables in the MAXIMA manu-
als.

6.7.10 ... for Vector Calculus

MAXIMA does not appear to have built in features or a suitable package for evaluating the standard
differential operators usually symbolized by \( \nabla \) (gradient) and \( \nabla^2 \) (Laplacian) applied to scalar

\(^{22}\) You may have to study the output on line (%o16) and the explanatory paragraph for awhile before the structure
of this output becomes clear.
Table 6.2: The MAXIMA command file MAXvectcart.mac. Here, the structure /*...*/ brackets comments in the file (See Section 6.9.)

/* MAXvectcart.MAXIMA (UNIX) or MAXvectcart.mac (Windows)

Command file to define functions for evaluating the gradient and Laplacian of a scalar function and the divergence and curl of a vector function of three cartesian variables.

*/

gradcart(f, x,y,z ) := [diff(f,x), diff(f,y), diff(f,z)]$
lapcart(f, x,y,z ) := diff(f,x,2) + diff(f,y,2) + diff(f,z,2)$
divcart(V, x,y,z ) := diff(V[1],x) +
diff(V[2],y) + diff(V[3],z)$
curlcart(V, x,y,z ) := [ diff(V[3], y) - diff(V[2],z),
diff(V[1], z) - diff(V[3],x),
diff(V[1], y) - diff(V[2],x) ]$

functions and \( \nabla \cdot \) (divergence) and \( \nabla \times \) (curl) applied to vector functions. We can, however, quickly define functions that will evaluate these quantities in, for example, Cartesian coordinates. More specifically, with \( f \) and \( V \) representing scalar and vector functions of three variables, the statements

\[
\begin{align*}
\text{gradcart}(f, x,y,z ) & := [\text{diff}(f,x), \text{diff}(f,y), \text{diff}(f,z)] \\
\text{lapcart}(f, x,y,z ) & := \text{diff}(f,x,2) + \text{diff}(f,y,2) + \text{diff}(f,z,2) \\
\text{divcart}(V, x,y,z ) & := \text{diff}(V[1],x) + \text{diff}(V[2],y) + \text{diff}(V[3],z) \\
\text{curlcart}(V, x,y,z ) & := [ \text{diff}(V[3], y) - \text{diff}(V[2],z), \\
& \quad \text{diff}(V[1], z) - \text{diff}(V[3],x), \\
& \quad \text{diff}(V[1], y) - \text{diff}(V[2],x) ] \\
\end{align*}
\]

define these four operators in Cartesian coordinates and reflect the definitions

\[
\begin{align*}
\nabla f(x,y,z) &= \frac{\partial f}{\partial x} i + \frac{\partial f}{\partial y} j + \frac{\partial f}{\partial z} k \\
\nabla^2 f(x,y,z) &= \frac{\partial^2 f}{\partial x^2} + \frac{\partial^2 f}{\partial y^2} + \frac{\partial^2 f}{\partial z^2} \\
\nabla \cdot V(x,y,z) &= \frac{\partial V_1}{\partial x} + \frac{\partial V_2}{\partial y} + \frac{\partial V_3}{\partial z} \\
\nabla \times V(x,y,z) &= \left( \frac{\partial V_3}{\partial y} - \frac{\partial V_2}{\partial z} \right) i + \left( \frac{\partial V_1}{\partial z} - \frac{\partial V_3}{\partial x} \right) j + \left( \frac{\partial V_1}{\partial y} - \frac{\partial V_2}{\partial x} \right) k \\
\end{align*}
\]

Because \( f, V, x, y, \) and \( z \) in these definitions are variables, these definitions do not mandate the use of particular names for the variables involved.

For convenience, these definitions are combined in the command file—see Section 6.9—MAXvectcart.mac—see Table 6.2—available in the directory $HEAD/maxima.\textsuperscript{23} If this file is copied to your current default search directory,\textsuperscript{24} the statement

\texttt{load( MAXvectcart.mac )}

\texttt{gradcart(f, x,y,z ) := [diff(f,x), diff(f,y), diff(f,z)]}$

\texttt{lapcart(f, x,y,z ) := diff(f,x,2) + diff(f,y,2) + diff(f,z,2)$}

\texttt{divcart(V, x,y,z ) := diff(V[1],x) +
\texttt{diff(V[2],y) + diff(V[3],z)$}

\texttt{curlcart(V, x,y,z ) := [ diff(V[3], y) - diff(V[2],z),
\texttt{diff(V[1], z) - diff(V[3],x),
\texttt{diff(V[1], y) - diff(V[2],x) ]$}

\texttt{See the Local Guide for the value of $HEAD$ at your site.}

\texttt{See Section 6.16.3 for how to place this directory at the head of the reserved variable file_search_maxima so that the load command will, in fact, find the file.}
6.8. LOOPS, LOGICAL EXPRESSIONS, AND CONDITIONALS

(%i1) batchload( "MAXvectcart.mac" )$

will load these definitions. Then, various vector derivatives can be evaluated with statements like

(%i2) gradcart( x^2 + y^2 + z^2, x, y, z);
(%o2) [2x, 2y, 2z]
(%i3) q : a^2 + b^2 + c^2$
(%i4) lapcart( q, a, b, c);
(%o5) 6
(%i5) divcart( [x^3*y^2*z, x*y^3*z^2, x^2*y*z^3], x, y, z);
(%o6) [3x^2y^2z^2 + 3x^2yz^2 + 3x^2y^2z, 3x^3y^2z^2 + 3x^2y^2z^2 + 3x^2y^2z^2]

Here, the statements at %i3 and %i4 demonstrate that the first argument of these functions can be a variable defined elsewhere and the variables need not be the x, y, and z used in the definitions. Note also that MAXIMA uses its own order for the terms at %o5: the connection to the first argument in divcart would be more transparent if the order of the output were $3x^2y^2z + 3xy^2z^2 + 3x^2yz^2$.

These definitions are even more flexible and can function properly even if the first argument is an unspecified function of the stipulated variables. For example, the statement

(%i7) lapcart( f(x, y, z), x, y, z);
(%o7) \frac{d^2 f(x, y, z)}{dz^2} + \frac{d^2 f(x, y, z)}{dy^2} + \frac{d^2 f(x, y, z)}{dx^2}

returns implicit derivatives (though, again, the displayed order of the terms is a bit unconventional.

The creation of command files MAXvectcyl.mac and MAXvectsph.mac to provide commands for evaluating these vector derivatives in cylindrical and spherical coordinates is left to the exercises.

6.8 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 MAXIMA. The simplest is the for/thru/step/do loop, which we have already met in Sections 6.7.2, 6.7.4, and 6.7.5. Briefly, the statements

(%i11) x : [ 0, 0, 0, 0, 0, 0, 0, 0, 0, 0 ];
(x) [0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
(%i12) for i from 1 thru 10 step 2 do x[i] : i^2;
(%o12) done
(%i13) x;
(x) [1, 0, 9, 0, 25, 0, 49, 0, 81, 0]

create a 10-element list of zeros and then assign the squares of the odd integers to the odd elements of that list. In this structure, the component step can be omitted, in which case the index—here i—will be incremented by 1 at each pass through the loop.

In more recent versions of MAXIMA, not only can a colon : be substituted for from in this statement but the step can alternatively appear before the thru, i.e., for i from 1 step 2 thru 10 ... . We will adhere to the older syntax, which still works in these more recent versions.
Other available loop structures include the `while` loop and the `unless` loop, each of which requires a condition whose constant updating with each pass through the loop will ultimately change the condition to a value that will stop the loop. The general syntax of these loops is\footnote{The keyword `from` can replace the colon `:` in these statements.}

\begin{verbatim}
for (variable) : (start) step (increment) while (condition) do (block of statements)
\end{verbatim}

and

\begin{verbatim}
for (variable) : (start) step (increment) unless (condition) do (block of statements)
\end{verbatim}

More specifically, these loops are illustrated in the statements

\begin{verbatim}
(%i4) x : [ 0, 0, 0, 0, 0, 0, 0, 0, 0, 0 ]$
(%i5) for i : 1 step 2 while i < 11 do x[i] : i^2$
(%i6) x;
(%o6) [1, 0, 9, 0, 25, 0, 49, 0, 81, 0]
(%i7) for i : 1 step 2 unless i > 10 do x[i] : i^3$
(%o7) [1, 0, 27, 0, 125, 0, 343, 0, 729, 0]
\end{verbatim}

The specification of `step` is optional. If `step` is omitted, then the index—here `i`—will be incremented by 1. Further, if the block of statements includes a statement that adjusts `i`, that adjustment will take place on top of the automatic incrementing of `i`, e.g.,

\begin{verbatim}
(%i8) x : [ 0, 0, 0, 0, 0, 0, 0, 0, 0, 0 ]$
(%i9) for i : 1 unless i > 9 do block( x[i] : i^3, i : i+2 )$
(%i10) x;
(%o10) [1, 0, 0, 64, 0, 343, 0, 0, 0, 0]
\end{verbatim}

Note that (1) the index has been incremented by 3—not 2—so we adjust only `x[1]`, `x[4]`, and `x[7]` and (2) groups of statements to be seen as a single composite statement are provided as arguments to the function `block`. The statements in a block are separated with commas.

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

\begin{verbatim}
for i : 1 thru 10 do if x[i] < 0 then x[i] : -x[i];
\end{verbatim}

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

\begin{verbatim}
for i : 1 thru 10 do if x[i]> 10 then x[i] : 10;
\end{verbatim}

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

\begin{verbatim}
if a >= 0 then b : a else b : -a;
\end{verbatim}

will set `b` equal to the absolute value of `a` (though the function `abs` will do so more easily). Multiple statements in either the `then` clause or the `else` clause will be separated by commas and enclosed within parentheses, as in

\begin{verbatim}
if condition then ( statement, statement, ... )
else ( statement, statement, ... );
\end{verbatim}
For selection among more than two options, the statement would have the form

\[
\text{if } \text{condition then block of statements}
\]
\[
\text{elseif } \text{condition then block of statements}
\]
\[
\text{elseif } \ldots
\]
\[
\text{else block of statements;}
\]

All conditions may be single or composite.

Finally, a word about composite conditions: The illustrations above made use of (single) conditions like \(i > 9\), which will evaluate to \text{true} or \text{false} depending on the value of \(i\). Indeed, the MAXIMA function \text{is}, for example,

\[
\begin{align*}
\%i10) & \quad i : 5$
\%i11) [ \text{is}(i > 9), \text{is}(i < 10) ];
\%o11) & \quad [\text{false}, \text{true}]
\end{align*}
\]

allows the testing of a logical condition. The six logical operators =, <, >, <=, >=, and <> can be used to test for equal, less than, greater than, less than or equal, greater than, and unequal.

In some situations, one may need to combine two or more conditions into a composite condition or negate a condition. MAXIMA makes available the logical operators \text{and}, \text{or}, and \text{not} to facilitate combining conditions and negating conditions. More specifically, with \(C1\) and \(C2\) individual conditions

- \((C1\ \text{and}\ C2)\) will be \text{true} if \(C1\) and \(C2\) are both \text{true} and \text{false} otherwise,
- \((C1\ \text{or}\ C2)\) will be \text{true} if either \(C1\) or \(C2\) is true or both \(C1\ \text{and}\ C2\) are true and \text{false} otherwise, and
- \(\text{not}\ (C1)\) will be true if \(C1\) is false and false if \(C1\) is true.

For example, the coding

\[
\begin{align*}
\%i12) & \quad x : [0,0,0,0,0,0,0,0,0]$
\%i13) & \quad i : 1$
\%i14) \quad \text{while}(i > 0 \text{ and } i < 11) \text{ do } (x[i] : i^3, i : i+2)$
\%i15) \quad x;
\%o15) & \quad [1,0,27,0,125,0,343,0,729,0]
\end{align*}
\]

not only illustrates a (somewhat silly) composite condition controlling the \text{while} statement, but also recognizes that the \text{for} component can also be omitted \text{provided} a suitable incrementing statement is included in the block of statements within the body of the loop. Pure \text{while} loops and, for that matter, pure \text{unless} loops are legitimate. 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.

As is the case in most languages, loops and conditional structures can be nested, though constructing the syntax correctly can sometimes be challenging.

### 6.9 Command Files

In addition to responding to statements supplied interactively in i-lines, MAXIMA can read statements from a variety of command files. Many such files are included in the standard distribution as
examples or demonstrations and are executed by invoking the command `example` or the command `demo` with appropriate arguments.\footnote{See items 3 and 4 in Section 6.2.} Other built-in command files, including packages, can be read and executed by one or another of the commands:

- **`load( filename );`** This command will load files containing MAXIMA code and also files containing LISP code. Default file types for MAXIMA code (e.g., `.mac`) are listed in the option variable `file_type_maxima`; default file types for LISP code (e.g., `.lisp`) are listed in the option variable `file_type_lisp`. The search paths used are listed in `file_search_maxima` and `file_search_lisp`, respectively.\footnote{See Section 6.16.3 for a discussion of these two option variables and of ways to change them.} The command `load` reads and executes the expressions in the file *silently*, i.e., does not display i- or o-lines and does not assign i and o labels along the way. Thus, a file to be loaded with `load` can contain no i- or o-labels to refer to previous lines and only those expressions that have been explicitly bound to mnemonic variables will be available after the command file has been executed.

- **`batch( filename );`** This command will load files containing MAXIMA code. Default file types (e.g., `.mac`) are listed in the option variable `file_type_maxima`. The search path used is listed in `file_search_maxima`.\footnote{See Section 6.16.3 for a discussion of this option variable and of ways to change it.} Without stopping (unless the file includes requests for user input), `batch` reads and displays each i-line, computes and displays each o-line, and returns control to the MAXIMA prompt, having assigned *all* the labels (i- and o-line labels as well as explicitly assigned mnemonic labels) that would have been assigned had the statements been executed interactively. The MAXIMA environment available after the command file has been executed by `batch` is identical to the environment that would have existed had the statements been entered one at a time interactively.

- **`batchload( filename );`** This command will load files containing MAXIMA code. Default file types (e.g., `.mac`) are listed in the option variable `file_type_maxima`. The search path used is listed in `file_search_maxima`.\footnote{See Section 6.16.3 for a discussion of this option variable and of ways to change it.} The command `batchload` reads and executes the expressions in the file *silently*, i.e., does not display i- or o-lines and does not assign i and o labels along the way. Thus, a file to be loaded with `batchload` can contain no `%`-signs to refer to previous lines and only those expressions that have been explicitly bound to mnemonic variables will be available after the command file has been executed.

For files having a default file type and stored in a directory in the applicable search path, neither the file type nor the path need be specified in the argument of the command and filenames can be supplied without quotation marks. Otherwise, the full path must be specified and the entire argument of the command must be enclosed in (double) quotation marks.

User-defined command files can be added by creating a text file containing the desired statements just as they would be typed interactively. In particular, no i-labels will appear in the file. For files having the appropriate default type and located in a directory in the search list defined by the option variable listed above, the file name *without* the file type and *without* quotation marks is sufficient as an argument. For other file types and/or directories, the full path and file name must be supplied and enclosed in double quotation marks.\footnote{Note that MAXIMA in Windows knows how to translate ‘/’ into ‘\’ for transmission to the operating system, so single (forward) slashes can be used to separate components of a path, even in the Windows operating system.}

Sometimes, creation of a command file facilitates repeated use of quantities not defined in standard MAXIMA. 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 MAXIMA; repeated typing of the correct parts of the sequence is not then necessary. We could, for example, create the file `testmultiply.mac` containing the statements listed in Table 6.3. Then, assuming that this file has been stored in a directory in MAXIMA’s search path, perhaps adjusted
Table 6.3: The MAXIMA command file testmultiply.mac. The structure /*...*/ allows us to include comments in the file, though the comments will not be displayed when the file is read and executed by MAXIMA.

/* Command file testmultiply.mac */

/* Enter matrix */
mat1 : matrix([1,2], [3,4]);
/* Evaluate transpose */
mat2 : transpose(mat1);
/* Evaluate matrix product */
mat1.mat2;
/* Multiply element by element */
mat1*mat2;

as described in the previous paragraph, we would execute the statements in this file by submitting to MAXIMA one of the statements

- **batch( testmultiply );** which executes the statements in the file non-stop, displaying both the statements and the responses along the way and assigning all labels (i, o, and user-defined).
- **batchload( testmultiply );** which executes the statements in the file non-stop without displaying i-lines or o-lines and assigning only user-defined labels.
- **demo( "PathToFile/testmultiply.mac" );** which executes the statement in the file, pausing after each statement, displaying both the statements and the responses, and assigning all labels (i, o, and user-defined).\(^{32}\)

Were the directory containing the file not in the appropriate search path, specification of the full path would have to precede the file name.\(^{33}\)

Suppose, as a second example, that we frequently had two three-component lists defining two three-component vectors and that we wanted to be able easily to evaluate their cross and dot products. We might create a command file containing the lines listed in Table 6.4. Here, beyond the comments, the first three lines—actually a single statement—define the function **cross**, which takes two three-component lists as arguments and returns a three component list whose components are those of the cross product of the two vectors. The last line defines the function **dot**, which takes the same arguments but returns a scalar value equal to the dot product of those two lists treated as three-component vectors. This file is named **crossdot.mac** and can be accessed in the directory $\text{HEAD}/\text{maxima}$. Thus, the statement

batchload("$\text{HEAD}/\text{maxima/crossdot.mac}");

will load this file (silently), defining the two functions **dot** and **cross** for subsequent use.

Especially when statements or sets of statements are to be executed repeatedly, it may even be valuable to **translate** from MAXIMA’s language into LISP,\(^{34}\) since execution will then be faster. This

---

\(^{32}\)The option variable **file_search_demo** provides the search path for files submitted to **demo**. If modified by statements similar to those described above for modifying **file_search_maxima**, **PathToFile** can be omitted in the argument of **demo**.

\(^{33}\)See the note at the beginning of this chapter.

\(^{34}\)Behind the scenes, MAXIMA is written in LISP, so LISP is essentially the native language for the tool.
translation can be accomplished in two ways. We might, for example, load the file `crossdot.mac` with the above statement and translate each individual function with the statements

```lisp
translate( cross );
translate( dot );
```

Subsequent invocations of these functions in the same session will then make use of the LISP code rather than the original MAXIMA code. Alternatively, we could create a file containing the LISP code for both functions by submitting the statement\(^{35}\)

```lisp
translate_file( "Fullpath/crossdot.mac" );
```

thereby creating the file `crossdot.LISP` in the directory specified by `Fullpath`.\(^{36}\) The translator also creates a file named `crossdot.UNLISP`, which records pertinent information that may be useful if the translation is to be debugged. Once the file `crossdot.lisp` resides in the default directory and the system variable `file_search_lisp` has been edited to include the directory containing the LISP file, the statement\(^{37}\)

```lisp
load( "crossdot.lisp" );
```

will load `crossdot.lisp`, since the loader will find it before finding `crossdot.mac`, and the translated versions of the two functions will be available for use for the remainder of the session. Because the translated versions have this time been stored in a file, they can be loaded directly in later sessions as well; the translation itself need be done only once.

Even greater run-time efficiencies can be achieved if statements or sets of statements are compiled to convert MAXIMA code into the binary code directly executed by the computer. We might, for example, load the file `crossdot.mac` as above and compile each individual function with the statements

```lisp
compile( cross );
compile( dot );
```

Subsequent invocations of these functions in the same session will then make use of the compiled code rather than the original MAXIMA code. Alternatively, we could create a file containing the compiled code for `crossdot.mac` by submitting the statement\(^{38}\)

```lisp
compile_file( "Fullpath/crossdot.mac" );
```

thereby creating the file `crossdot.fasl` in the directory specified by `Fullpath`. The statement

```lisp
load( "Fullpath/crossdot.fasl" );
```

will then load `crossdot.fasl`, since the loader will find it before finding `crossdot.mac`) or `crossdot.lisp`, and the compiled versions of the two functions will be available for use for the remainder of the session. As with the translated versions, the compiled versions can be loaded directly in later sessions; the compilation itself need be done only once.

\(^{35}\)In this context, there does not appear to be a default path or a default file type.

\(^{36}\)The `.LISP` file is an ASCII file, so we can examine the LISP code by opening the file in a text editor.

\(^{37}\)The function `load` could have been included in the list earlier in this section but was not because it is a more general function. It searches first for a binary file, then for a LISP file, and finally for a file containing MAXIMA code, loading whichever it finds first. In fact, when it finds only MAXIMA code, it reverts to `batchload`.

\(^{38}\)In this context, there does not appear to be a default path or a default file type.
/* crossdot.MAXIMA (UNIX) or crossdot.mac (Windows)

Command file to define functions to evaluate the cross
and dot products of two three-component lists. */

cross(a,b) := [ a[2]*b[3] - a[3]*b[2],
               a[3]*b[1] - a[1]*b[3],


We conclude this section with a pointer to a way to achieve yet further efficiencies in execution.
In translating or compiling the functions above, MAXIMA has had to assume that the variables
might have any possible data type or structure. If only certain limited types are possible, then the
number of cases with which the translated or compiled code would have to contend could be reduced.
As a last refinement, we might use MAXIMA’s function mode declare, described in detail in the
MAXIMA manuals, to limit the cases MAXIMA should expect.

6.10 High-Resolution Plotting

High-resolution plotting in MAXIMA is provided by an external plotting package, which must be
installed separately from MAXIMA. Several such packages are available, each providing a different
plot format. We limit the discussion in this book to the default plot format, specifically gnuplot
on Windows platforms and gnuplot.pipes on non-Windows platforms. MAXIMA plotting commands
calculate the points to be plotted, write those points and appropriate commands to a properly
constructed plot file created in the directory specified by the variable maxima_tempdir and by de-
fault named maxout.gnuplot (Windows) or maxout.gnuplot.pipes (non-Windows), and submit
that file for execution by the plotting package.

We illustrate the production of high resolution graphics by returning to the damped harmonic
oscillator. First, we enter the function generated by solving the damped harmonic oscillator equation
at line (%o6) in Section 6.7.8 with the statement

(%i1) soln : x[0] * exp(-beta*t) * (beta*sin(omega*t)/omega + cos(omega*t));

Then, recognizing that β and ω are not independent (ω^2 = k/m − β^2) and hence that assigning
values to β and ω in effect fixes k/m, we use the statements

(%i2) ( beta:0.1, omega:1.0, x[0]:1.0 )$

---

39 Other packages provide plot formats mgnuplot and xmaxima, though not all are available on all platforms. The MAXIMA command get_plot_option( plot_format ); will return the current plot format and the command set_plot_option( [plot_format, DesiredFormat ] ); will set the plot format to the specified desired format.
40 See Section 6.16.2 for the default identity of this directory and for information about how to change it.
41 As described in Section 6.11.2, an option specified in the plotting command can direct the description of the graph
to a different file in an alternative format. Produced in this way, the graph will not appear on the screen.
to assign specific values to $\beta$, $\omega$, and $x_0$. Finally, we create the graph with the statement

\[
\text{(\%i3) plot2d( soln, [t, 0.0, 20.0] )$
\]

This statement allows MAXIMA to select a range appropriate to the dependent variable for the vertical axis, plots $\text{soln}$ as a function of $t$ over the interval $0.0 \leq t \leq 20.0$,\(^{42}\) and labels the axes with default values that may or may not be what would be preferred.

By default either in the GUI or the CLI,\(^{43}\) graphs are displayed in a separate window, and display of a graph will prevent execution of further MAXIMA commands until that graph window has been closed. These graphs can be resized on the screen in the usual way by dragging the borders with the mouse.

Alternatively, but only in the GUI, graphical displays can be embedded in the wxMaxima window by prepending the characters ‘wx’ to the plotting function, e.g. \text{plot2d $\rightarrow$ wxplot2d}, etc. In this case, once the graph has been created, additional commands can be executed without deleting the graph. Once created, an embedded graph cannot be resized, though setting the system variable \text{wxplot_size} to a two-component list, e.g.,

\[
\text{wxplot_size : [500, 250]}$
\]

will specify in pixels the size of the region containing all subsequent plots. (The default value is [600, 400].) If the size of only one plot is to be changed from the global value of \text{wxplot_size}, a command of the form

\[
\text{wxplot2d( ... ), wxplot_size=[500, 250]$
\]

will do the job.

Assorted \textit{plot options} allow more detailed customization of the resulting graph. For example, the statement

\[
\text{(\%i4) plot2d( soln, [t, 0.0, 20.0], [y, -2.0, 2.0] )$
\]

exploits the plot option $y$ to override the default range on the vertical axis,\(^ {44}\) and the statement

\[
\text{(\%i5) plot2d( soln, [t, 0.0, 20.0], [y, -1.0, 1.0],
[title, "Damped Oscillator"], [xlabel, "time"],
[ylabel, "x"], [color, black], [grid2d, true],
[label, ["beta=0.1, omega=1.0", 10.0, 0.5]] )$
\]

exploits the plot options $y$, \text{title}, \text{xlabel}, \text{ylabel}, \text{color}, \text{grid2d}, and \text{label} and generates a plot in which the range on both axes, the title, the labels on both axes, the color of the resulting graph, the drawing of full grid lines, and the addition of the indicated label starting horizontally at the point 10.0 and centered vertically at the point 0.5 are all specified by assigning appropriate values to several plot options.

MAXIMA also allows the presentation of more than one graph on a single set of axes. For example, we might add the velocity $v$ to the position $x$ in the graph of the previous paragraph with the statements

\(^{42}\)The argument specifying the range of the independent variable is mandatory. In that argument, the variable name will be whatever name identifies the independent variable.

\(^{43}\)or, for that matter, in the third interface identified in the second paragraph of this chapter.

\(^{44}\)In this context, the variable $y$ is always used for that axis, regardless of the actual name of the function being plotted.
Figure 6.1: Graph of position and velocity for a damped oscillator with $\beta = 0.1$, $\omega = 1.0$, and $x(0) = x_0 = 1.0$, and $v(0) = 0.0$. 

which exploit the additional plot options `style` and `legend` and the (sub)plot option `lines` to arrange for different line weights for the two graphs and the inclusion of an explanatory legend. The resulting graph is shown in Fig. 6.1.

By default, the two-dimensional graphs produced above are drawn with an adaptive routine which adjusts the values at which the functions are evaluated to optimize the smoothness of the graph. The plot option `nticks`, set by including a component like `[nticks, 50]` in the `plot2d` command, sets the initial number of points used in the adaptive algorithm that decides ultimately how many points actually to use. The default value of `nticks` is 29.

We could explore other features of this motion by, for example, changing the value of $\beta$ and replotting the graph with the statements

or, returning to the original value of $\beta$, we could produce a graph of the trajectory in the phase plane ($\text{solndot}$ versus $\text{soln}$) with the statements

or, returning to the original value of $\beta$, we could produce a graph of the trajectory in the phase plane ($\text{solndot}$ versus $\text{soln}$) with the statements
Figure 6.2: Graph of velocity versus position for a damped oscillator with \( \beta = 0.1, \omega = 1.0, \) and \( x(0) = 1.0, \) and \( v(0) = 0.0. \)

\[
\begin{align*}
\text{\small Damped Oscillator} \\
\text{\small \hspace{1cm} Damped Oscillator} \\
\text{\small \hspace{1cm} Damped Oscillator} \\
\end{align*}
\]

\[
\begin{align*}
[y, -1.0, 1.0], &[title, "Damped Oscillator"], [xlabel, "x"], \\
[ylabel, "v"], &[color, black], [style, [lines,4]], [same_xy, true] \\
\end{align*}
\]

In particular, this last statement sets the plot option \texttt{same\_xy} to \texttt{true} to stipulate that the units on horizontal and vertical scales shall be the same for the display. The resulting graph is shown in Fig. 6.2.

Note that, in the \texttt{Gnuplot} window, you can generate an expanded view of a selected portion of the display by moving the cursor to one corner of the rectangle bounding the area to be expanded, clicking the right mouse button, moving the cursor to the diagonally opposite corner, and clicking the left mouse button. Clicking the left mouse button on the icon pictured as a circle with a left-pointing arrow in the window’s tool bar restores the previous display.

MAXIMA can, of course, also produce surface plots and contour plots in several different ways. The simplest and quickest surface plot is produced by statements like

\[
\begin{align*}
\text{(\texttt{\%i12) fct : sin( 2.0*\%pi*x ) * sin( 3.0*\%pi*y )}} \\
\text{(\texttt{\%i13) plot3d( fct, [x, 0.0, 1.0], [y, 0.0, 1.0], [title, "Mode 2,3"], \\
grid, 20, 20], [legend, \"\"] )}} \\
\end{align*}
\]

Several plot options for \texttt{plot3d} (and \texttt{wxplot3d} in the GUI), some not here illustrated, include

- \texttt{grid}, which specifies the number of grid points to be used along the \( x \) and \( y \) axis, respectively. The default value is 30, 30.
- \texttt{legend}, which specifies a label added to the entire graph. The default value produces the expression conveying the plotted function \texttt{fct} as the label. Setting the plot option to \"\" (as here) or to \texttt{false} suppresses the legend altogether.
- \texttt{palette}, which specifies the spectrum of colors to be used in shading each element of the surface to convey the value of \texttt{fct} on that element. The statement at line \texttt{\%i13} exploits the
6.10. HIGH-RESOLUTION PLOTTING

default value of this option. The default value, however, can be overridden to change the choice of colors. Alternatively, including the component \texttt{[palette, false]} in the arguments of \texttt{plot3d} will suppress the shading altogether, producing a wire mesh diagram. Including the component \texttt{[palette, [value, 0,0,0,1]]} will use a gray scale to convey the values of \texttt{fct}.

- \texttt{mesh_lines_color}, which controls the color of the lines defining the mesh when a palette is being used. The default value is \texttt{black} but other colors can be substituted. The lines can be suppressed altogether by including the component \texttt{[mesh_lines_color, false]} among the arguments of \texttt{plot3d}. This option is not effective when \texttt{palette} is false.

- \texttt{color}, which specifies the color of the mesh lines when \texttt{[palette, false]} is operative.

- \texttt{color_bar}, with default value \texttt{false} for \texttt{plot3d}, which, when set to \texttt{true}, produces a bar conveying the colors associated with values of the \texttt{z} coordinate.

Thus, the statement\textsuperscript{46}

\begin{verbatim}
(%i14) plot3d( fct, [x, 0.0, 1.0], [y, 0.0, 1.0], [title, "Mode 2,3"],
             [grid, 20, 20], [legend, ""], [palette, false], [color, black] )$
\end{verbatim}

and the statement

\begin{verbatim}
(%i15) plot3d( fct, [x, 0.0, 1.0], [y, 0.0, 1.0], [title, "Mode 2,3"],
             [grid, 20, 20], [legend, ""], [palette, [value, 0, 0, 0, 1]] )$
\end{verbatim}

will produce the graphs in Fig. 6.3. Pressing and holding ML with the cursor somewhere on the graph produced by \texttt{plot3d} (but not \texttt{wxplot3d}) and moving the cursor will rotate the diagram to facilitate fathoming its features.

An alternative display replaces the (three-dimensional) mesh surface with contour lines in a plane and is produced for the function \texttt{fct} by the statement

\begin{verbatim}
(%i16) contour_plot( fct, [x, 0.0, 1.0], [y, 0.0, 1.0], [legend, false],
               [title, "Mode 2, 3"], [gnuplot_preamble, "set cntrparam levels 10"] )$
\end{verbatim}

Here, we

- suppress display of the function \texttt{fct} and of a chart showing the correspondence of values with colors with \texttt{[legend, false]}. The legend but not the chart would be suppressed if we had included the component \texttt{[legend, ""]} in the argument list, but the chart lies on top of—and obscures—the plot.

- override the default number of levels (3) by including the component \texttt{[gnuplot_preamble, "set cntrparam levels 10"]}. This statement, which exploits the MAXIMA plot option \texttt{gnuplot_preamble} communicates the \texttt{gnuplot} command \texttt{set ...} to the external plotting package, positioning that command at the beginning of the file created by \texttt{contour_plot} so that \texttt{gnuplot} will execute that instruction before generating the plot from that file. The illustrated statement specifies that 10 contour levels be drawn; the default is 5. Alternative options include

\textsuperscript{45}For details on constructing alternative values for \texttt{palette}, see the MAXIMA manuals.

\textsuperscript{46}To produce the graph on the left in Fig. 6.3, this statement was executed on a Linux computer by xmaxima. Executed on a Windows computer with either xmaxima or wxmaxima and on a Linux computer with wxmaxima, the statement appears to ignore the stipulation \texttt{[color, black]}. Note that Section 12.4 of the Version 5.38.1 MAXIMA manual asserts that, with \texttt{plot3d}, if \texttt{palette} is given the value \texttt{false}, the surfaces will not be shaded but represented with a mesh of curves only and the colors of the lines will be determined by the option \texttt{color}. Note also that replacing the option \texttt{[color, black]} with \texttt{[gnuplot_preamble, "set lt 1 lc rgb 'black' lw 2; set lt cycle 1"]} achieves the desired coloring. (The option \texttt{gnuplot_preamble} is described in the penultimate paragraph of this section (Section 6.10.).)
Figure 6.3: The upper graph shows a mesh surface plot of the function $z(x, y) = \sin(2\pi x) \sin(3\pi y)$ and the lower graph shows a shaded surface plot of the function $z(x, y) = \sin(2\pi x) \sin(3\pi y)$.

- set cntrparam levels discrete $x_1$, $x_2$, $x_3$, ..., which specifies the levels be drawn at the values $x_1$, $x_2$, $x_3$, ..., and
- set cntrparam levels incremental start, incr, end, which specifies that levels be drawn at values starting with start, ending with end, and stepped in increments incr between start and end.

The contour plot produced by the above statement has several deficiencies, which we correct as follows:

- The resulting plot is displayed in a rectangular area, which distorts the shape of the contour lines. A better plot would use equal increments on both axes and, because the range on the two axes is the same, yield a square display. Achieving that adjustment involves invoking another option within the set command to gnuplot, expanding the above statement to be

```plaintext
(%i17) contour_plot( fct, [x, 0.0, 1.0], [y, 0.0, 1.0], [legend, false],
               [title, "Mode 2, 3"], [gnuplot_preamble, "set cntrparam levels 10;"
```
Note that several gnuplot commands can be incorporated within the same pair of quotation marks in the gnuplot_preamble option if the commands are separated from one another with a semicolon.

- Even with this first adjustment, however, the contour lines are still cycled through a sequence of colors and the resulting display is still not particularly suited to a black and white presentation. We can force the display to use only the first color in the set through which it by default cycles by further embellishing the statement to use the keywords linetype (which can be abbreviated lt) and cycle, specifically

(%i18) contour_plot( fct, [x, 0.0, 1.0], [y, 0.0, 1.0], [legend, false],
   [title, "Mode 2, 3"], [gnuplot_preamble, "set cntrparam levels 10;
   set size ratio -1; set linetype cycle 1"] )$

This modification instructs gnuplot to use only the first color as it cycles through the contour lines.

- Unfortunately, the first color happens to be red. To make the contour lines black, make them a bit heavier, and make the border the same weight as the contour lines, we could add an invocation of the keywords linecolor (which can be abbreviated lc) and linewidth (which can be abbreviated lw; default 1) in the statement

(%i19) contour_plot( fct, [x, 0.0, 1.0], [y, 0.0, 1.0], [legend, false],
   [title, "Mode 2, 3"], [gnuplot_preamble, "set cntrparam levels 10;
   set size ratio -1; set lt 1 lc rgb 'black' lw 2;
   set lt cycle 1"] )$

Here, apparently, we must enclose the color black in single quotation marks to avoid confusion with the double quotation marks enclosing the value of the option gnuplot_preamble.

- Since the border of the graph is a contour line at the value 0, we can make that border the same weight as the contour lines by adding a set command with the keyword border with the statement

(%i20) contour_plot( fct, [x, 0.0, 1.0], [y, 0.0, 1.0], [legend, false],
   [title, "Mode 2, 3"], [gnuplot_preamble, "set cntrparam levels 10;
   set size ratio -1; set lt 1 lc rgb 'black' lw 2;
   set border 15 lw 2; set lt cycle 1"] )$

The integer after the word border uses the coding 1, 2, 4, and 8 for the bottom, left, top, and right borders, respectively (i.e., 1, 10, 100, and 1000 in binary), specify which border is to be set to the specified linewidth. To specify all four borders, the integer is set to $1 + 2 + 4 + 8 = 15$, i.e., to 1111 in a binary notation.\footnote{Interestingly, while the above specifications of linewidths yield the contour lines and borders of the same width on screen, the translation to a PostScript file does not preserve that identity.}

- Finally, we note that some of the contour lines—those in the lower and upper left and middle right—correspond to positive values of the function and others—those in the middle left and lower and upper right—correspond to negative values. To convey these differences, we invoke one more MAXIMA plot option—label—to place plus and minus signs in the appropriate regions of the contour map. The resulting statement

\begin{verbatim}
set size ratio -1")
\end{verbatim}
Figure 6.4: Contour map of the function $z(x, y) = \sin(2\pi x) \sin(3\pi y)$.

achieves this objective. Note that it took a bit of experimentation to place the plus and minus signs in the center of each region.

The final graph is shown in Fig. 6.4.

(\%i21) reset()$ kill(all)$

See item 6 in Section 6.1.

We have, of course, only mentioned a few of the functions and plot options MAXIMA makes available for generating graphical displays. Additional plot options include axes, azimuth, box, elevation, label, logx, logy, pdf_file, same_xyz, xtics, xy_scale, ytics, and yx_ratio. You should make it a point to read about these plot options in the MAXIMA manuals.

6.11 Making Hard Copy …

The method used to make hard copy of MAXIMA output depends both on whether that output is textual or graphical and on whether a Windows or non-Windows UNIX version is in use.

6.11.1 … of Text

Though each option may work well in only some cases, text in a MAXIMA display can be printed by several routes:
6.11. MAKING HARD COPY

- In the GUI (WX-MAXIMA) on all platforms, selecting ‘Print’ from the File menu will direct the contents of the MAXIMA workbook (text and special symbols, including Greek letters) to the printer selected in the resulting window. Embedded graphics will also be correctly printed via this procedure.

- In the CLI (X-MAXIMA) on all platforms, highlighting the text to be printed, selecting ‘copy’ from the Edit menu, pasting the text into an available text editor, and saving the result will provide a file that can then be sent to a printer by using the features of the text editor or the operating system. Greek letters cannot be displayed in or printed from the CLI. Printing graphics from this interface requires use of the commands described in Section 6.11.2.

- In the in-shell interface on some platforms, text can be cut from the MAXIMA dialog and pasted into an available text editor to create a file that can subsequently be printed. Greek letters cannot be displayed in or printed from this interface. Printing graphics from this interface requires use of the commands described in Section 6.11.2.

- In the GUI, the CLI and the in-shell interface on all platforms, all or part of the textual dialog carried out in any conversation with MAXIMA can be captured in a text file that can subsequently be directed to a printer or incorporated in other documents, perhaps with editing, though in some cases with some text editors the resulting file may be poorly formatted. We begin by opening an output file with the statement

```maxima
writefile( "FullFileName" );  or  appendfile( "FullFileName" );
```

where `FullFileName` must contain the full path to the intended destination of the file to be written. Two system-defined option variables—`maxima_tempdir` and `maxima_userdir`—store paths to possibly useful directories and may be useful in this context.\(^{48}\) Rather than explicitly typing a full path, one can construct the path within the argument of `writefile` with statements like

```maxima
writefile( concat( maxima_tempdir, "/FileName" ) );
writefile( concat( maxima_userdir, "/FileName" ) );
```

Here, `FileName` must include the appropriate file type.\(^{49}\) Once this statement has been executed, all subsequent dialog will appear on the screen and will also be written into the specified file, with `writefile` creating a new file that overwrites any existing file of the specified name or with `appendfile` adding the new dialog at the end of an existing file.\(^{50}\) The statement

```maxima
playback();
```

which redisplay the entire dialog or, with appropriate arguments (see the MAXIMA manuals), selected parts of the dialog, can be used to avoid reentering statements already executed if we decide after the fact that we should have captured them and their output into the file. To turn off capturing into the file, use the statement

```maxima
closefile();
```

While the resulting file can be edited, copied into other files, and/or printed, it cannot be reloaded into MAXIMA.\(^{51}\)

---

48 See Section 6.16.2 for information about the default values of these two variable and how to change those values.

49 Of course, the user must have write access to the identified directory. Also look again at the syntactical issue discussed in footnote 31.

50 Actually, whether `writefile` overwrites or appends to an existing file may depend on the particular implementation of the user’s operating system.

51 To save MAXIMA dialog in a way that can be reloaded, see the `save` and `stringout` commands in the MAXIMA manuals.
• MAXIMA's capability to write \LaTeX\ descriptions of equations on the screen and into an output file is described in Section 6.12.

6.11.2 ... of Graphs

While embedded graphs in the GUI can be sent to the printer as described in the first bullet in Section 6.11.1, obtaining hard copy of a graph produced in a separate pop-up window on any platforms and with `plot_format gnuplot` in Windows and `plot_format gnuplot.pipes` in non-Windows requires a different approach. In these contexts, a description of the graph can be written into a file by re-executing the plot command (\texttt{plot2d}, \texttt{plot3d}, or \texttt{contour_plot}), adding to the optional arguments one of the following components:

\begin{itemize}
  \item \texttt{[pdf\_file, "FileName"]} Produces output in PDF format
  \item \texttt{[png\_file, "FileName"]} Produces output in PNG format
  \item \texttt{[ps\_file, "FileName"]} Produces output in (encapsulated) PostScript format
  \item \texttt{[svg\_file, "FileName"]} Produces output in SVG format
\end{itemize}

where \texttt{FileName}, which should include the file type and must be enclosed in (double) quotation marks, will be specified in one of two ways:

1. If it does not contain one or more (forward) slashes, the file will be written into the directory specified by the option variable \texttt{maxima\_tempdir}.\footnote{See Section 6.16.2 for information about \texttt{maxima\_tempdir}.}

2. If it does contain one or more (forward) slashes, the full path, including the filename itself, must be specified, and the output file will be placed in the specified location.

In either case, the user must have write access to the specified directory.

6.12 Output in \LaTeX\ Format

A particularly useful feature of MAXIMA is its ability to write \LaTeX\ versions of expressions to the screen or into a file. Full details are laid out in the MAXIMA manuals. In brief, the statement

\begin{verbatim}
tex( expr ); or tex( label );
\end{verbatim}

converts the specified expression or the expression assigned to the specified label into \LaTeX\ format and displays the result in the \texttt{MAXIMA} window. If, however, invocation of \texttt{tex} includes a destination, e.g.,

\begin{verbatim}
tex( expr, "FullFileName.tex" ); or tex( label, "FullFileName.tex" );
\end{verbatim}

where \texttt{FullFileName} includes the full path and the file type, the output will instead be written into the specified ASCII file. If the specified file does not exist, it will be created; if the file does exist, the output from this statement will be appended to the existing contents. Once created, this file can, of course, be edited or incorporated into other \LaTeX\ documents.

Two system-defined option variables—\texttt{maxima\_tempdir} and \texttt{maxima\_userdir}—store paths to possibly useful directories and may be useful in this context.\footnote{See Section 6.16.2 for information about the default values of these two variable and how to change those values.} Rather than explicitly typing a full path, one can construct the path within the argument of \texttt{tex} with statements like...
Here, FileName must include the appropriate file type.\footnote{Of course, the user must have write access to the identified directory. Also look again at the syntactical issue discussed in footnote 31.}

Automatic translation of arbitrary expressions into \TeX{} is a daunting task. While fairly sophisticated, the MAXIMA translator is not perfect. Edits will often be necessary to “correct” the translator’s glitches. In particular, for expressions entered explicitly and expressions identified either by a user-defined label or an o-line label, the translator uses the \TeX{} characters ‘$$’ to begin and end displayed equations, and we may wish to replace those with the \LaTeX{} forms \begin{equation} and \end{equation}. We may also wish to delete the translator’s provision of an explicit equation number equal to the MAXIMA label on the line. Provided we use names like \texttt{omega} and \texttt{Omega} for variables in MAXIMA, the translator will convert them into \texttt{\omega} and \texttt{\Omega}. For i-line labels, the translator simply encloses the string of ASCII characters on the i-line in a \verbatim{} environment.

### 6.13 Animation

Beyond the capacity for three-dimensional diagrams produced by MAXIMA to be dragged to different orientations with the cursor (see the discussion of \texttt{plot3d} towards the end of Section 6.10), a detailed discussion of MAXIMA’s capabilities for producing animated displays is beyond the scope of this book. Those interested in animations should read about the dynamics package in the MAXIMA manuals.

### 6.14 Using the Workbook

Within the GUI, the window in which MAXIMA commands are entered has full capabilities as a workbook.\footnote{The counterpart in MAXIMA competitors is more commonly called a notebook.} We have so far met only \texttt{Input} cells, which contain executable MAXIMA statements (and possibly the associated output). These cells are bracketed along the left edge of the window with a bracket that has at its top a line sloping from the southwest to the northeast and, when output is also present, a short horizontal line separating input from output. For purposes of organizing larger documents, \texttt{Text} cells, \texttt{Title} cells, \texttt{Section} cells, \texttt{Subsection} cells, and \texttt{Subsubsection} cells can contain text in various styles.

Suppose we already have on-screen a MAXIMA workbook containing a few MAXIMA statements and their output. Suppose, further, that we want to precede each statement with an explanatory paragraph. We could achieve that end for one of the statements by

1. Placing the cursor—a horizontal blinking line—at the point in the document where you wish to insert some text;
2. Selecting ’Insert \texttt{(desired cell type)} Cell’ from the Cell menu;\footnote{Space will be opened at the point of the cursor and marked at the left edge with a symbol to convey the type of cell created. Most of the time, a \texttt{Text} cell will be appropriate, but different levels of heading can be inserted with the other types of cell.} and finally
3. Placing the cursor—now a vertical blinking line—in the newly created space (if it isn’t already there) and typing the desired text.
This process can, of course, be repeated to provide a textual description for each of the remaining statements. Similarly, new \textit{Input} cells can be created by selecting ‘Insert Input Cell’ from the \textsc{Cell} menu.

The \textsc{Cell} menu also provides tools to hide and recreate output lines. Selecting ‘Remove All Output’ shortens the workbook by deleting all output lines. Selecting ‘Evaluate All Cells’ recreates the output. Conveniendy, \textsc{MAXIMA} workbooks can be saved by selecting ‘Save’ or ‘Save As’ from the \textsc{File} menu and migrating in the resulting browser to select the destination of the saved file, which will by default be given the file type \texttt{.wxmx}. In a subsequent session, the file can be reloaded most easily by selecting ‘Open’ from the \textsc{File} menu and then browsing to find and click ML on the desired file. Generally, output lines will be deleted before saving, since that output can be easily regenerated after the workbook has been reloaded into \textsc{MAXIMA}.

Beyond saving the workbook as a \texttt{.wxmx} file, \textsc{MAXIMA} can also export the workbook in a variety of formats. Selecting ‘Export …’ from the \textsc{File} menu brings up a browser in which one can select \texttt{.html}, \texttt{.mac}, or \texttt{.tex} as the type of file to be output, browse to the desired directory, and specify a suitable file name. As with most automatic creation of these files, the result is likely to be a bit cumbersome and some editing, particularly of a \texttt{.tex} file, may be warranted.

This short section can, of course, not do full justice to the wide spectrum of features of \textsc{MAXIMA}’s workbook interface. You must work with it, explore the menus in the GUI, and read the manual.

\section{Pattern Matching}

\textsc{MAXIMA} offers pattern matching as a powerful type of search and replace, as well as a means to construct our own commands. Essentially, this capability provides a means to tell \textsc{MAXIMA} how to recognize certain structures regardless of the specific symbols that appear in the structures, and to take some action with the recognized patterns. Pattern matching is quite complicated, so we limit the discussion here to a few simple examples. Full detail can be found in the \textsc{MAXIMA} manuals.

\subsection{Search and Replace: Small Angle Approximations}

A human can be told something like “replace each sine term with its argument” and know what to do. Careful instruction with pattern matching can convey this same concept to \textsc{MAXIMA}. In this section, we develop the pattern matching commands to perform small angle approximations for sine and cosine terms.

First, we need to define a \textit{pattern variable} to match “the argument” of a term. The argument can be anything, so we need a variable which will match anything. To this end, we use the command \texttt{matchdeclare}, whose syntax is

\begin{verbatim}
matchdeclare( PatternVariable, Test )
\end{verbatim}

Once this statement has been executed, \textit{PatternVariable} will match anything for which \textit{Test} is true.\footnote{See the \textsc{MAXIMA} manuals for an enumeration and description of the available tests.}

For example, the statement

\begin{verbatim}
matchdeclare( x, freeof("+", "-")
\end{verbatim}

declares that the variable \texttt{x} will match any expression that does not contain either a “+” sign or a “-” sign, and the statement
matchdeclare( arg, true )

declares that the variable $\text{arg}$ will match anything at all. After a test is performed, if the match is
successful, the pattern variable has a value equal to whatever it matched.

Second, we need to use the command \texttt{defrule} to define a rule for carrying out replacements. The general syntax of a statement using this command is

\texttt{defrule( \textit{RuleName}, \textit{PatternVariable}, \textit{Replacement} )}

Since the value of a pattern variable is whatever it matched, a pattern variable can be used in the
replacement to mean the same thing that it matched in the pattern. For example, the statement

\texttt{defrule( sine, sin(arg), arg )}

defines a rule named \texttt{sine} that, when applied to an expression, will replace every appearance of
\texttt{sin(arg)} with \texttt{arg}, regardless of what the specific argument of the sine function happens to be.

Third (and finally), we need the commands \texttt{apply1}, \texttt{apply2}, and \texttt{applyb1} to apply the defined
rules in various ways to an expression. All of these commands use a common syntax of the form

\texttt{apply*( \textit{Expression}, \textit{Rule1}, \textit{Rule2}, ..., \textit{Rulen} )}

They differ in the order in which the various rules are applied. \texttt{apply1} applies the first rule over
and over to the expression until it effects no further changes, then applies it to every subexpression
until it effects no further changes, continuing to the lowest level subexpressions. The command then
starts over with the second rule and follows the same process, and so on with the rest of the rules.
In contrast, \texttt{apply2} applies all of the rules to the expression, then moves to the subexpressions
and applies all of the rules to each of them one at a time, and so on down to the lowest level
subexpressions. The \texttt{applyb1} works the same as the \texttt{apply1}, but works from the bottom up, starting
with the lowest subexpression.

As a specific example, we construct rules which will do small angle approximations. We begin
by defining a suitable expression with the statement

\begin{verbatim}
(%i1) terms : sin(x^2) + cos(2*x) + sin(x+3);
(%o1) sin(x^2) + sin(x+3) + cos(2x)
\end{verbatim}

Then, we invoke the statements

\begin{verbatim}
(%i2) matchdeclare(arg, true)$

(%i3) defrule(sine, sin(arg), arg);

(%o3) sine : sin(arg) → arg

(%i4) defrule(cosine, cos(arg), 1);

(%o4) cosine : cos(arg) → 1

(%i5) apply1(terms, sine, cosine);

(%o5) x^2 + x + 4

(%i6) reset()$ kill(all)$
\end{verbatim}

Define pattern variable \texttt{arg} to match anything.

Define rule to replace \texttt{sin(arg)} by \texttt{arg}.

Define rule to replace \texttt{cos(arg)} by \texttt{1}.

Apply rules to expression.

See item 6 in Section 6.1.

to complete the process. This result is, in fact, correct, considering what we told MAXIMA to do. Here $x + 3$ from the sine was combined with the 1 from the cosine to yield $x + 4$.

This result is \textit{not}, however, correct mathematically, since $x + 3$ is certainly not small if $x$ is.
We need to modify our pattern variable so that it does not match expressions with addition or
subtraction in them (or, equivalently, to match only terms containing no addition or subtraction).
To this end, we modify the items in the previous script to be
%i1) terms : sin(x^2) + cos(2*x) + sin(x+3)$
%i2) matchdeclare(arg, freeof("+", "-"))$
%i3) defrule(sine, sin(arg), arg)$
%i4) defrule(cosine, cos(arg), 1)$
%i5) apply1(terms, sine, cosine);

(%o5) sin(x + 3) + x^2 + 1

%i16) reset()$ kill(all)$

See item 6 in Section 6.1.

This time, we have defined the pattern variable arg to match only those expressions that are themselves free of plus and minus signs and the result on line (%o5) is now mathematically correct when x is small (except that we haven’t recognized the wisdom in this case of replacing cos 2x with 1 – (2x)^2/2). (Modification of the rule cosine would address that issue.)

6.15.2 Simplification Rules

MAXIMA has a set of rules which it applies to every term it sees in order to simplify them. For example, x * x is automatically changed to x^2, 3 + 2 to 5, 4/6 to 2/3, etc. With pattern matching, we can define additional rules to be added to MAXIMA’s automatic simplification routine. The process is similar to the definition of a rule above, except that the substitution is done without the use of an apply command. The commands to make this automatic substitution rule are tellsimp and tellsimpafter, the former defining a new rule that will be applied before the existing ones and the latter defining a new rule that will be applied after the existing ones. The syntax for both is of the form

    tellsimp( Pattern, Replacement )  or  tellsimpafter( Pattern, Replacement )

Names for the simplification rules are generated automatically by the system; we will never need to use them explicitly.

Suppose, for example, that we wanted small angle approximations to be applied to every term we enter into MAXIMA. We might achieve this objective with the statements

%i1) matchdeclare(arg, freeof("+", "-"))$
%i2) tellsimpafter(sin(arg), arg);
%o2) [sinrule2, simp-%sin]
%i3) tellsimpafter(cos(arg), 1);
%o3) [cosrule1, simp-%cos]

Once these rules have been defined, their application is automatic. Entering the expression

%i4) sin(x^2) + cos(2*x) + sin(x+3);

for example, yields the immediate output

(%o4) sin(x + 3) + x^2 + 1

%i5) reset()$ kill(all)$

See item 6 in Section 6.1.
6.15.3 Pattern Matching Functions

Finally, we can use pattern matching to define a function to test an expression against a given pattern. This feature uses the command `defmatch` to define the new function. The syntax is

\[\text{defmatch}(\text{FunctionName, Pattern, Parameter1, ..., Parametern})\]

The pattern must be an expression containing only pattern variables and parameters. When the function is used, it is called with the syntax

\[\text{FunctionName}(\text{Expression, Parameter1, ..., Parametern})\]

If the given expression can be made to match the pattern, the function returns the values that each pattern variable and parameter matched. If MAXIMA cannot make the expression match the pattern, the function returns `false`.

As an example, we use `matchdeclare` and `defmatch` to construct a function that tests whether an expression is quadratic in a given variable. The statements

\[
\begin{align*}
\text{(\%i1)} & \quad \text{matchdeclare([a, b, c], freeof(x));} \\
\text{(\%o1)} & \quad \text{done} \\
\text{(\%i2)} & \quad \text{defmatch(quadratic, a*x}^2+b*x+c, x); \\
\text{(\%o2)} & \quad \text{quadratic}
\end{align*}
\]

define that function, and the statements

\[
\begin{align*}
\text{(\%i3)} & \quad \text{quadratic((y+2)*(y-1), y);} \\
\text{(\%o3)} & \quad [c = -2, a = 1, b = 1, x = y] \\
\text{(\%i4)} & \quad \text{quadratic(z}^3+2*z-3, z); \\
\text{(\%o4)} & \quad \text{false} \\
\text{(\%i5)} & \quad \text{quadratic((x+y)*(x+z), x);} \\
\text{(\%o5)} & \quad [c = yz, a = 1, b = z + y, x = x]
\end{align*}
\]

test that function with several examples.

6.16 Miscellaneous Occasionally Useful Tidbits

6.16.1 Specifying directories

Different operating systems use different forms to specify the path that identifies a particular directory, for example, to set the default directory or to identify the location of a file to be read or written. The character separating directories in a path depends on the operating system. In UNIX, that character is a forward slash; in Windows, it is a backslash, though a single forward slash will be properly interpreted even in Windows. Indeed, if you wish to use backslashes, each must be `doubled` because a single backslash will not be properly interpreted.

6.16.2 MAXIMA’s Default Directory

Special option variables, which are defined by default when MAXIMA is launched but which can be changed by the user, are available to facilitate specifying paths for reading and writing files. Unless local adjustments have made changes (see Local Guide),
• the option variable \texttt{maxima\_tempdir} will point by default to a subdirectory of the user's home directory, $\texttt{UserHome/}...$\textsuperscript{58} In particular, MAXIMA uses this directory, called the default directory or the current directory, for the storage of temporary files when plots are created.

• the option variable \texttt{maxima\_userdir} will point by default to the directory $\texttt{UserHome/maxima}$. The default value of this directory is placed first in MAXIMA's search path.$^59$

(The directory $\texttt{UserHome}$ is defined in the \textit{Local Guide}.) The current values of these two option variables can be \textit{displayed} with the statements

\begin{verbatim}
maxima\_tempdir and maxima\_userdir
\end{verbatim}

and \textit{changed} with statements like

\begin{verbatim}
maxima\_tempdir : "FullPathToDirectory"
maxima\_userdir : "FullPathToDirectory"
\end{verbatim}

Note, however, that changing \texttt{maxima\_userdir} during a session will not automatically replace the default value at the beginning of MAXIMA's search path; the search path must also be explicitly changed.

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

\section*{6.16.3 MAXIMA's Search Path}

Whenever an explicit path is not included in the file name of a file that MAXIMA must read with \texttt{load}, \texttt{demo}, and some other functions, the program searches for the specified file in the sequences of directories stored in the option variables \texttt{file\_search\_maxima} (for MAXIMA files) and \texttt{file\_search\_lisp} (for lisp files). The values of these variables are set by default when MAXIMA is launched and can be \textit{displayed} with the statements

\begin{verbatim}
file\_search\_maxima and file\_search\_lisp
\end{verbatim}

The default value of the directory$\textsuperscript{60}$ \texttt{maxima\_userdir} appears at the beginning of each of these search paths but that component in these variables must be changed \textit{explicitly} if \texttt{maxima\_userdir} is changed during the session. The values of these two variables can be \textit{changed} in several ways. The statement

\begin{verbatim}
VariableName : [ "FilePath1", "FilePath2", ... ]
\end{verbatim}

will redefine the variable to include \textit{only} the specified list. Alternatively, the statement

\begin{verbatim}
VariableName : append( VariableName, [ "FilePath1", "FilePath2", ... ] )
\end{verbatim}

will add the specified list to the \textit{end} of the current list, and the statement

\begin{verbatim}
VariableName : append( [ "FilePath1", "FilePath2", ... ], VariableName )
\end{verbatim}

\textsuperscript{58}On the author’s Windows machine, \texttt{maxima\_tempdir} points to $\texttt{UserHome/AppData/Local/Temp}$.

\textsuperscript{59}See Section 6.16.3.

\textsuperscript{60}See Section 6.16.2.
will add the specified list at the *beginning* of the current list. Note that, in the syntax of all of these statements, the square brackets are necessary, even if the enclosed list contains only one element.

In specifying *FilePath*, the wild card ***###*** can be used to stand for the file name that is the search target and enclosing components in {...} will allow abbreviating the path. For example, if we execute the statement

```
file_search_maxima : ["c:/users/cookd/###.{mac,mc}"];  
```

and then execute the statement

```
load( testfile )
```

the files *c:/users/cookd/testfile.mac* and *c:/users/cookd/testfile.mc*, should they exist, will both satisfy the search.

### 6.16.4 Customizing MAXIMA

When MAXIMA is started up, its configuration is defined by a number of default specifications built into the program. MAXIMA also responds to a user-specified initialization file, which provides a means for further automatic customization of MAXIMA in ways different from the vendor-supplied defaults. This file is, in the first instance, written initially in MAXIMA code and is named *maxima-init.mac* but—see Section 6.9—can be translated or compiled to speed execution. Whether in MAXIMA code, translated, or compiled, this file resides in a subdirectory of the user’s home directory, which will be reported by MAXIMA in response to the statement

```
maxima_userdir;
```

typically *$UserHome/maxima*. The file *maxima-init.mac* may contain any MAXIMA statements and is executed automatically immediately when MAXIMA is launched.

### 6.16.5 Restoring MAXIMA’s Initial State

See item 6 in Section 6.1.

### 6.16.6 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

```
(%i48) plot2d( sin(x), [x, 0.0, 10.0], [label, ["New Graph",5.0,0.0]] );
```

This statement invokes the plot option *label* to specify that the text ‘New Graph’ be added to the graph with the lower left corner of the text located at the point [5.0,0.0] and the text displayed horizontally. The command *plot2d* does not appear to provide control over the positioning of the text relative to the specified point or the orientation of the text, though the command *draw2d* in the *draw* package does offer that additional control through the options *label_alignment* and *label_orientation*. Full detail on these commands and their extension to three-dimensional graphs can be found in the MAXIMA manuals.
CHAPTER 6. INTRODUCTION TO MAXIMA

Figure 6.5: Space curve of a charged particle moving in a constant magnetic field along the z axis.

6.16.7 Space Curves

Sometimes it is desirable to view the trajectory of a particle in three-dimensional space. MAXIMA has the ability to plot space curves from knowledge of the trajectory defined parametrically by the functions $x(t)$, $y(t)$, and $z(t)$ giving the coordinates of points on a three-dimensional path. With that information, the command `plot3d` 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

$$
\begin{align*}
x(t) &= \cos t \\
y(t) &= \sin t \\
z(t) &= \alpha t 
\end{align*}
$$

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

```maxima
(%i1) alpha : 0.5$
(%i2) plot3d( [cos(t), sin(t), alpha*t], [t, 0, 30], [y, -1, 1], [z, 0, 20], [same_xy, true], [xtics, -1, 0.5, 1], [ytics, -1, 0.5, 1], [grid, 500, 1], [legend, false] );
```

will produce the graph in Fig. 6.5. Here, in particular, the plot option `grid` defines the number of points at which the trajectory is evaluated in the interval on the parameter $t$. (The role of the second integer in that option is unclear.) and the options `xtics` and `ytics` stipulate that tics are to be drawn and labeled at increments of 0.5 in the interval $-1 \leq x, y \leq 1.0$.

6.17 Miscellaneous Commands

Several miscellaneous commands to MAXIMA are also of general utility. We should in particular be aware of at least the following commands:

- `apply();`, which facilitates application of a function to every element in a list, e.g. `apply("*", [1, 2, 3])`, which returns the value 6.
• `apropos();`, which returns a list of valid arguments for `describe();`, where those arguments all contain the string in the argument of `apropos();`. The statement `apropos("sin");`, for example, returns a list of all functions, option variables, and other entities containing the string `sin` for which `describe();`—see Section 6.2—will give additional information.

• `assume();`, which attaches properties to symbolic variables, e.g., `assume( b > 0 );`. Assumed properties can be removed with `forget();`.

• `declare();`, which permits attaching a variety of features to variables. The statement `declare(n, integer);`, for example, tells MAXIMA to consider only integer values for the variable `n`. Declared features can be removed with `remove();`. The statement `features;` will result in a display of the system variable `features`, which contains a list of all the features that can be declared.

• `dispfun( FunctionName );`, which displays the definition of a user-declared function.

• `forget();`, which instructs MAXIMA to “forget” properties assigned with `assume`.

• `functions;`, which displays the names of all user-declared functions.

• `infolists;`, which displays a list of the names of the system variables that store lists of information maintained by the system.

• `labels( symbol );`, which displays a list of the i-line, o-line, and intermediate expression labels to which values have been bound.

• `loadfile( FileName );`, which restores MAXIMA to the state it had when the “loaded” file was created with the command `save`.

• `map();`, which specifies application of a given function separately to each term of an expression.

• `product();`, which multiplies given values over a specified range.

• `remfunction( FunctionName );`, which “un”defines a previously defined function.

• `remove();`, which causes MAXIMA to “forget” a feature attached to a variable by `declare`.

• `remvalue( VariableName );`, which removes MAXIMA’s knowledge of the value previously bound to a variable.

• `save( FileName , all );`, which saves MAXIMA’s current state in a file that can subsequently be read into MAXIMA to restore that state. (See `loadfile`.)

• `stringout();`, which outputs selected quantities to a specified file.

• `sum();`, which sums given values over a specified range of indices.

• `values;`, which displays the names of variables to which values have been bound.

Additional information about these commands can be displayed by entering the command name in the search box of the index in the Maxima manual.

6.18 References

In this chapter, we have introduced only the most important features of MAXIMA. A full description of all commands and features is contained in the MAXIMA manuals, including

• several documents and resources as described in Section 6.2.
a number of documents, links to which can be found by entering “Maxima help” in the search window of your browser. In particular, you should find links to


and

- many other official and unofficial documents.

more specific documents on particular MAXIMA commands, links to which can be found by entering “CommandName in MAXIMA”, e.g., “integrate in MAXIMA” in the search window of your browser.

The user is urged to browse in these manuals, being alert not only to the available functions but also to the assorted option variables and keywords that modify their behavior.

6.19 Exercises

6.1. Read the MAXIMA manuals to find out how the commands combine, distrib, expand, multthru, and xthru 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

(a) \( \frac{x}{a+b} + \frac{y}{a+b} \)
(b) \( (a+b)(c+d) + (a+b)c \)
(c) \( (x+y)(z+w) \)
(d) \( a(b+c(d+e)) \)
(e) \( \frac{x}{a+b} + \frac{y}{c+d} \)
(f) \( \frac{(a-x)^2}{(a^2-2ax+x^2)^{3/2}} \)

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.

6.2. Use MAXIMA 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) \( \frac{(a-x)^2}{(a^2-2ax+x^2)^{3/2}} \) \( \Rightarrow \) \( \frac{1}{|x-a|} \)
(b) \( \sinh( \ln(x + \sqrt{x^2 + a^2}) - \ln(a) ) \) \( \Rightarrow \) \( \frac{x}{a} \)
(c) \( \frac{1}{x + \sqrt{y}} \) \( \Rightarrow \) \( \frac{x - \sqrt{y}}{x^2 - y} \)

These samples are chosen to illustrate particularly the use of radcan and ratsimp but other commands will surely also be needed. Write two or three paragraphs in which you describe your efforts, including some indication of approaches that were not successful. Don’t be overly concerned about the order of terms within various sets of parentheses; that order is particularly difficult to control. Focus instead on creating the general form of each desired result. Hint: For part (c), you might find it useful to look up the option variable algebraic.

6.3. Use MAXIMA to convert each of the expressions in the left-hand column in the table below into the expression in the associated right-hand column:
6.19. EXERCISES

(a) \( \frac{d}{dx} \left( x^2 e^{-x^2} \right) \implies -2x(x^2 - 1)e^{-x^2} \)

(b) \( \sin(\sqrt{a^2(x + 3\lambda) + x^2(3a + x) + y}) \implies \sin((a + x)^{3/2} + y) \)

(c) \( cg + cf + b^2d + 2abd + a^2d + b^2c + 2abc + a^2c \implies (a + b)^2(c + d) + c(f + g) \)

(d) \( a e^{(-b+i\omega)t} + a e^{(-b-i\omega)t} \implies a e^{-bt}(e^{-i\omega t} + e^{i\omega t}) \)

(e) \( a e^{(-b+i\omega)t} + a e^{(-b-i\omega)t} \implies 2a e^{-bt}\cos(\omega t) \)

(f) \( x^2 + y^2 + z^2 - 2a(x + y) + 2a^2 \implies (x - a)^2 + (y - a)^2 + z^2 \)

In several cases, you may need to invoke part, pickapart, subst, and/or substpart but other commands will surely also be needed. Invoke microscopic dissection of the expressions only as a last resort. Write two or three paragraphs in which you describe your efforts, including some indication of approaches that were not successful. Don’t be overly concerned about the order of terms within various sets of parentheses in the final form; that order is particularly difficult to control. Focus instead on creating the general form of each desired result.

6.4. In Section 6.7.8, we set up the differential equation for a damped harmonic oscillator but then pursued the solution only for the underdamped case. Starting with the dimensionless form\(^{61}\)

\[
\frac{d^2x}{dt^2} + 2\alpha \frac{dx}{dt} + x = 0 \quad ; \quad x(0) = 1 \quad ; \quad \frac{dx}{dt} \bigg|_{t=0} = 0
\]

of the equation deduced towards the end of Section 6.7.8, work out the solution for the underdamped case and then find also the solutions for the critically damped and overdamped cases defined, respectively by

\[ \alpha > 0 \quad \text{and} \quad \alpha < -\gamma, > 1 \]

Describe the differences in the physical behavior for the three cases. For simplicity, take the initial conditions to be \( x(0) = 1 \) and \( v(0) = 0 \).

6.5. 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

\[
\begin{align*}
P_0(x) &= 1 \\
P_1(x) &= x \\
P_2(x) &= \frac{1}{2} (3x^2 - 1)
\end{align*}
\]

\[
\begin{align*}
P_3(x) &= \frac{1}{2}(5x^3 - 3x) \\
P_4(x) &= \frac{1}{8}(35x^4 - 30x^2 + 3) \\
P_5(x) &= \frac{1}{8}(63x^5 - 70x^3 + 15x)
\end{align*}
\]

a. Use the generating function and MAXIMA’s capabilities for evaluating Taylor series to find the first half-dozen Legendre polynomials, extracting each as an expression bound to a variable.

\(^{61}\)The forms of the solution work out more easily if, in contrast to the choice in the text, we express the coefficient of the second term as \textit{twice} a constant, so \( \alpha = b/2m\omega_0 \).
b. Start by binding the value 1 to \( P[0] \) and the value \( x \) to \( P[1] \) and then, using the recursion relationship, find the next several Legendre polynomials. \textit{Hint}: The MAXIMA statements would be

\begin{verbatim}
(%i1) P[0] : 1;
(%i2) P[1] : x;
\end{verbatim}

and then

\begin{verbatim}
(%i3) P[n] := (2*n-1)*x*P[n-1]/n - (n-1)*P[n-2]/n;
\end{verbatim}

(Verify the expression on the right by using MAXIMA 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 a function \( P[n] \) involving \( n \) that can be evaluated at any \( n \). Once \( P_0 \) and \( P_1 \) have been defined, you can find \( P_2 \) and then \( P_3 \) and then \( \ldots \) with statements like

\begin{verbatim}
(%i4) P[2];
(%i5) P[3];
\end{verbatim}

(Some simplifications may be necessary to cast the results in the standard form of the above table.)

c. Find the first half-dozen Legendre polynomials by using the command \texttt{diff} 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 MAXIMA (see manuals for the structure of a \texttt{for} loop), 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.)

e. Within MAXIMA, 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. MAXIMA actually knows quite a bit about many of the important special functions of mathematical physics. In particular, the package \texttt{orthopoly}, which can be loaded with the command \texttt{load(orthopoly)} and which is described fully in the MAXIMA manuals, provides two different ways to evaluate a very large number of these special functions. Take a look at that documentation and use the function \texttt{legendre_p} (syntax \texttt{legendre_p(n, x)}) in that package to determine the first several Legendre polynomials.

6.6. Figure 6.6 shows the circuit diagram for a Wheatstone bridge. Using Kirchoff’s laws, set up the equations from which you could determine the currents in each branch of the circuit. Then, using MAXIMA, (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.)
6.7. 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

\[
\psi(x) = \begin{cases} 
A e^{ikx} + B e^{-ikx} & x < 0 \\
D \cosh(\kappa x) + F \sinh(\kappa x) & 0 < x < w \\
C e^{ikx} & x > w 
\end{cases}
\]

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

\[
\psi(0^-) = \psi(0^+) \quad ; \quad \psi(w^-) = \psi(w^+) \quad ; \quad \left| \frac{d\psi}{dx} \right|_{0^-} = \left| \frac{d\psi}{dx} \right|_{0^+} \quad ; \quad \left| \frac{d\psi}{dx} \right|_{w^-} = \left| \frac{d\psi}{dx} \right|_{w^+}
\]

where superscript plus and minus signs identify points slightly below and slightly above the indicated value of \( x \), respectively. (While \( k \) and \( \kappa \) can be taken to be real for this barrier, the constants \( A, B, C, D, \) and \( F \) may—and probably will—be complex.) Use MAXIMA’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

\[
R = \left| \frac{B}{A} \right|^2 = \frac{(\kappa^2 + k^2)^2 \sinh^2(\kappa w)}{4\kappa^2 k^2 + (\kappa^2 + k^2)^2 \sinh^2(\kappa w)} \quad ; \quad T = \left| \frac{C}{A} \right|^2 = \frac{4k^2 \kappa^2}{4\kappa^2 k^2 + (\kappa^2 + k^2)^2 \sinh^2(\kappa w)}
\]

Here, the vertical bars symbolize the absolute value of the complex number enclosed by them.
(d) Verify that \( R + T = 1 \).

6.8. The distribution of wavelengths \( \lambda \) in the blackbody spectrum at (absolute) temperature \( T \) is given by

\[
u(\lambda, T) = \frac{8 \pi c h}{\lambda^5} \frac{1}{e^{c h / \lambda kT} - 1}
\]

where \( c \) is the speed of light, \( h \) is Planck’s constant, and \( k \) is Boltzmann’s constant. In terms of the variable \( y = c h / \lambda kT \), this function has the alternative expression

\[
\frac{(c h)^4 \nu(\lambda, T)}{8 \pi (kT)^5} = f(y) = \frac{y^5}{e^y - 1}
\]
6.10. The Laguerre polynomials

(a) Verify this transformed form and then, using MAXIMA, (b) obtain a graph of \( f(y) \) versus \( y \), making sure to extend the graph over an interval that includes its peak and estimate the value of \( y \) at which that peak occurs; (c) show that the peak occurs for values of \( y \) satisfying

\[(y - 5)e^y + 5 = 0\]

(d) obtain a graph of this function versus \( y \); (e) make another estimate of the value of \( y \) at which the original function has its maximum; and (f) show that the wavelength \( \lambda_m \) at which this maximum occurs satisfies

\[\lambda_m T = 0.28978 \times 10^{-2} \text{ m K}\]

Hints: (1) Remember that maxima in a function occur where the derivative of that function with respect to the appropriate variable is zero. (2) Because \( e^y \) varies rapidly with \( y \), you may have to play a bit to find a suitable range of values of \( y \) over which to plot these graphs.

6.9. Use MAXIMA to verify (a) that the solution presented as d6 in Section 6.7.8 satisfies the differential equation and the initial conditions presented in c2 and c6 of that section and (b) that the solutions given in d6 and d17 agree with each other.

6.10. 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 \( n \geq 0 \) be a polynomial of order \( n \) that is orthogonal to all previous polynomials, i.e., that

\[
\int_0^\infty e^{-x} L_n(x)L_m(x) \, dx = 0 \quad \text{for all } m < n
\]

and that the arbitrary overall sign remaining be resolved by requiring in addition that \( L_n(0) = 1 \). Alternatively, they can be determined from the recursion relationship

\[(n + 1)L_{n+1}(x) = (2n + 1 - x)L_n(x) - nL_{n-1}(x)\]

provided we include the first two \( L_0(x) = 1 \) and \( L_1(x) = 1 - x \) to get the recursion started. Yet again, they can be found from the application of multiple differentiation as implied by Rodrigues’ formula

\[L_n(x) = \frac{e^x}{n!} \frac{d^n}{dx^n} (x^n e^{-x})\]

However they are determined, the first half dozen of these polynomials will turn out to be

\[
\begin{align*}
L_0(x) &= 1 \\
L_1(x) &= 1 - x \\
L_2(x) &= 1 - 2x + \frac{1}{2}x^2 \\
L_3(x) &= 1 - 3x + \frac{3}{2}x^2 - \frac{1}{6}x^3 \\
L_4(x) &= 1 - 4x + 3x^2 - \frac{3}{2}x^3 + \frac{1}{24}x^4 \\
L_5(x) &= 1 - 5x + 5x^2 - \frac{5}{2}x^3 + \frac{5}{24}x^4 - \frac{1}{120}x^5
\end{align*}
\]

a. Determine the first half dozen Laguerre polynomials by constructing them one at a time to satisfy the requirements of orthogonality and normalization.

b. Start by binding the value 1 to \( L[0] \) and the value \( 1 - x \) to \( L[1] \). Then, using the recursion relationship, find the next several Laguerre polynomials. Hint: The MAXIMA statements might be

\[
\begin{align*}
\texttt{(\%i1) \ L[0] : 1;} \\
\texttt{(\%i2) \ L[1] : 1-x;} \\
\texttt{and then} \\
\texttt{(\%i3) \ L[n] := (2*n-1-x)*L[n-1]/n - (n-1)*L[n-2]/n;} \\
\texttt{(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 a function \( L[n] \) 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} \\
\texttt{(\%i4) \ L[2];} \\
\texttt{(\%i5) \ L[3];} \\
\texttt{\ldots}
\end{align*}
\]
6.19. **EXERCISES**

(Some simplifications may be necessary to cast the results in the standard form of the above table.)

c. Find the first half-dozen Laguerre polynomials by using the command `diff` to evaluate Rodrigues’ formula.

d. Be clever and, using either matrices or loops constructed in MAXIMA (see manuals for the structure of a `for` loop), 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.)

e. MAXIMA actually knows quite a bit about many of the important special functions of mathematical physics. In particular, the package `orthopoly`, which can be loaded with the command `load(orthopoly)` and which is described fully in the MAXIMA manuals, provides two different ways to evaluate a very large number of these special functions. Take a look at that documentation and use the function `laguerre` (syntax `laguerre(n, x)`) in that package to determine the first several Laguerre polynomials.

6.11. Find the eigenvalues and eigenvectors of the matrix

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

6.12. (a) Find the eigenvalues and eigenvectors of the matrix

\[
\begin{pmatrix}
2 & -1 & 0 & 0 \\
-1 & 2 & -1 & 0 \\
0 & -1 & 2 & -1 \\
0 & 0 & -1 & 2
\end{pmatrix}
\]

(b) Use the function `part` to explore the structure of the entity returned by the function `eigenvectors` used in part (a).

6.13. When a (weak) constant external electric field of magnitude \( F \)—We reserve \( E \) for energy in this exercise.—is imposed on a hydrogen atom, the energy of the states with principal quantum number \( n \) shift from the energy given by the Bohr model by amounts determined by the eigenvalues of the matrix whose elements are \( \langle nlm|eF|nl'm' \rangle \), where \( l, m, l', m' \) range over all possible values of those quantum numbers allowed by the particular value of \( n \). If the states by which the rows and columns are labeled are ordered \(|2, 0, 0 \rangle, |2, 1, -1 \rangle, |2, 1, 0 \rangle, \) and \(|2, 1, 1 \rangle\), then the matrix for the state \( n = 2 \) is

\[
3e a_0 F \begin{pmatrix}
0 & 0 & -1 & 0 \\
0 & 0 & 0 & 0 \\
-1 & 0 & 0 & 0 \\
0 & 0 & 0 & 0
\end{pmatrix}
\]

where \( e \) is the magnitude of the charge on the electron and \( a_0 \) is the Bohr radius. Similarly, if the states by which the rows and columns are labeled are ordered \(|3, 2, 2 \rangle, |3, 1, 1 \rangle, |3, 2, 1 \rangle, |3, 0, 0 \rangle, |3, 1, 0 \rangle, |3, 2, 0 \rangle, |3, 1, -1 \rangle, |3, 2, -1 \rangle, \) and \(|3, 2, -2 \rangle\), then the matrix for the state \( n = 3 \) is

\[
3e a_0 F \begin{pmatrix}
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & -9/2 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & -3\sqrt{6} & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & -9/\sqrt{3} & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & -9/\sqrt{3} & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & -9/2 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & -9/2 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0
\end{pmatrix}
\]
6.14. Develop a procedure using pattern matching that will convert a polynomial in \( x^2 \) to a polynomial in \( y \), where \( y = x^2 \).

6.15. As shown in Fig. 6.7, 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

\[
KE = \frac{1}{2} m \dot{x}^2 + \frac{1}{2} m \dot{y}^2; \quad PE = \frac{1}{2} k \left[ \sqrt{(\ell - x)^2 + y^2} - \ell_0 \right] + \ldots
\]

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 MAXIMA 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 \( \text{diff}( L, \text{diff}(x,t) ) \) are perfectly acceptable to MAXIMA, provided the dependence of \( x \) on \( t \) is indicated, either explicitly \((x(t))\) or through a prior execution of the statement \( \text{depends}(x,t) \).

6.16. In the Cartesian coordinate system illustrated in Fig. 6.8, 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

\[
KE = \frac{1}{2} m_1 \dot{x}_1^2 + \frac{1}{2} m_1 \dot{y}_1^2 + \frac{1}{2} m_2 \dot{x}_2^2 + \frac{1}{2} m_2 \dot{y}_2^2; \quad PE = m_1 g y_1 + m_2 g y_2
\]
use MAXIMA 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.

6.17. The complete elliptic integrals of the first and second kinds are given by
\[
K(k) = \int_{\theta_0}^{\pi/2} \frac{d\phi}{(1 - k^2 \sin^2 \phi)^{1/2}}; \quad E(k) = \int_{\theta_0}^{\pi/2} (1 - k^2 \sin^2 \phi)^{1/2} d\phi
\]
Use MAXIMA to evaluate these integrals to $O(k^6)$ by expanding the integrands in Taylor series before evaluating the integrals.

6.18. Find the Laplace transform of each of the functions

(a) $f(t) = t^n$  (b) $f(t) = e^{-at}$  (c) $f(t) = \cosh(at)$  (d) $\frac{df(t)}{dt}$

and the inverse Laplace transform of each of the functions

(e) $\tilde{f}(s) = \frac{a + bs}{s^2 + \omega^2}$  (f) $\tilde{f}(s) = \frac{a}{(s^2 + 9\omega^2)(s^2 + 4\omega^2)(s^2 + \omega^2)}$

In (c), you may have to express the hyperbolic cosine in exponential form while in (f) you may have to help MAXIMA’s `ilt` routine by first invoking the `partfrac` routine to expand the desired function in partial fractions.

6.19. 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, 0.7259383, and 0.7258887, respectively.

6.20. Find symbolic expressions for the coefficients $a$, $b$, and $c$ that will cause the parabola $y = ax^2 + bx + c$ to pass through the three points $(x_1 = x_2 - \delta x, y_1)$, $(x_2, y_2)$, and $x_3 = x_2 + \delta x, y_3)$. Then integrate the parabola over the interval $x_1 \leq x \leq x_3$ and deduce Simpson’s rule
\[
\int_{x_1}^{x_3} y(x) \, dx = \frac{\delta x}{3} \left( y_1 + 4y_2 + y_3 \right)
\]
for (approximate) numerical integration.
6.21. The midpoint rule $M$ and Simpson’s rule $S$ for evaluating integrals numerically start with the assumptions that
\[
\int_a^b f(x) \, dx \approx M = (b - a) f \left( \frac{a + b}{2} \right) \quad ; \quad \int_a^b f(x) \, dx \approx S = \frac{b - a}{6} \left( f(a) + 4 f \left( \frac{a + b}{2} \right) + f(b) \right)
\]
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 $f(x) = cx^3 + dx^2 + ex + f!$ Optional. Try to construct geometric arguments that would provide insight into the correctness of these analytic results.

6.22. Using MAXIMA, obtain graphs of the potential
\[
U(x) = -\frac{U_0 a^2 (a^2 + x^2)}{8 a^4 + x^4}
\]
and the associated force $F = -dU/dx$. 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 $\tau = x/a$, i.e., rewrite the function in the form
\[
\frac{U(x)}{U_0} = \frac{1 + \tau^2}{8 + \tau^4}
\]
You should be able to find a similar dimensionless version of the expression giving the force.

6.23. Using MAXIMA, find complete (symbolic) solutions to each of the following problems, and use MAXIMA 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. \[ m \frac{d^2 x}{dt^2} = -eE_0 \cos(\omega t + \theta), \quad x(0) = x_0, \quad v(0) = v_0, \]
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. \[ m \frac{d^2 x}{dt^2} = -mg + \frac{b (dx/dt)^2}{2}, \quad x(0) = 0, \quad v(0) = 0, \]
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 $v = v_{i_0} \hat{i} + v_{i_0} \hat{k}$.

6.24. Using the functions \texttt{gradcart}, \texttt{lapcart}, \texttt{divcart}, and \texttt{curlcart} defined in the file \texttt{MAXvectcart.mac} described in Section 6.7.10.

a. Evaluate the gradient of (i.e., the negative of the force field associated with) each of the functions
\[
V_1(x, y, z) = \frac{1}{(x^2 + y^2 + z^2)^{1/2}} \quad V_2(x, y, z) = \frac{z}{(x^2 + y^2 + z^2)^{3/2}}
\]
Paralleling your development after Section 6.7.10, develop the command file MAXvectcyl.mac to provide functions gradcyl, lapcyl, divcyl, and curlcyl to return the derivatives

\[
\nabla f(r, \phi, z) = \frac{\partial f}{\partial r} \hat{r} + \frac{1}{r} \frac{\partial f}{\partial \phi} \hat{\phi} + \frac{\partial f}{\partial z} \hat{k}
\]

\[
\nabla^2 f(r, \phi, z) = \frac{1}{r} \frac{\partial}{\partial r} \left( r \frac{\partial f}{\partial r} \right) + \frac{1}{r^2} \frac{\partial^2 f}{\partial \phi^2} + \frac{\partial^2 f}{\partial z^2}
\]

\[
\nabla \cdot \mathbf{V}(r, \phi, z) = -\frac{1}{r} \frac{\partial}{\partial r} \left( r \frac{\partial V_r}{\partial r} \right) + \frac{1}{r \sin \theta} \frac{\partial}{\partial \phi} \left( \frac{\partial Q_\phi}{\partial \phi} \right) + \frac{1}{r} \left( \frac{\partial (rQ_\phi)}{\partial r} - \frac{\partial Q_r}{\partial \phi} \right) \hat{k}
\]

for evaluating these entities in cylindrical coordinates. Here, \( r, \phi, \) and \( z \) are the (cylindrical) radial, azimuthal, and axial coordinates. Then,

a. Evaluate the gradient of (i.e., the negative of the force field associated with) each of the functions

\[
V_1(r, \phi, z) = \frac{1}{(r^2 + z^2)^{1/2}} \quad V_2(r, \phi, z) = \frac{r}{(r^2 + z^2)^{3/2}}
\]

\[
V_3(r, \phi, z) = \frac{2z^2 - r^2}{(r^2 + z^2)^{5/2}} \quad V_4(r, \phi, z) = \frac{e^{-a(r^2 + z^2)^{1/2}}}{(r^2 + z^2)^{1/2}}
\]

of the cylindrical variables.

b. Verify that each field obtained in part (a) is conservative by showing that the curl of each is zero.

c. With \( \mathbf{r} = r \hat{r} + z \hat{k} \), show (1) that \( \nabla \times \mathbf{r} = 0 \) and (2) that \( \nabla \cdot \mathbf{r} = 3 \).

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 \( V_4(r, \phi, z) \), \( \nabla^2 V_4(r, \phi, z) \).

6.25. Paralleling your development after Section 6.7.10, develop the command file MAXvecl sph.mac to provide functions gradsph, lapssph, divsph, and curlsph to return the derivatives

\[
\nabla f(r, \theta, \phi) = \frac{\partial f}{\partial r} \hat{r} + \frac{1}{r} \frac{\partial f}{\partial \theta} \hat{\theta} + \frac{1}{r \sin \theta} \frac{\partial f}{\partial \phi} \hat{\phi}
\]

\[
\nabla^2 f(r, \theta, \phi) = \frac{1}{r^2} \frac{\partial}{\partial r} \left( r^2 \frac{\partial f}{\partial r} \right) + \frac{1}{r^2 \sin \theta} \frac{\partial}{\partial \theta} \left( \sin \theta \frac{\partial f}{\partial \theta} \right) + \frac{1}{r^2 \sin^2 \theta} \frac{\partial^2 f}{\partial \phi^2}
\]

\[
\nabla \cdot \mathbf{V}(r, \theta, \phi) = \frac{1}{r^2} \frac{\partial}{\partial r} \left( r^2 \frac{\partial V_r}{\partial r} \right) + \frac{1}{r \sin \theta} \frac{\partial}{\partial \phi} \left( \sin \theta \frac{\partial Q_\phi}{\partial \phi} \right) + \frac{1}{r} \left( \frac{\partial (rQ_\phi)}{\partial r} - \frac{\partial Q_r}{\partial \phi} \right)
\]

\[
\nabla \times \mathbf{Q}(r, \theta, \phi) = \frac{1}{r \sin \theta} \left( \frac{\partial (\sin \theta Q_\theta)}{\partial \phi} - \frac{\partial Q_\phi}{\partial \theta} \right) \hat{r} + \left( \frac{1}{r \sin \theta} \frac{\partial Q_\phi}{\partial \phi} - \frac{1}{r} \frac{\partial (rQ_\phi)}{\partial r} \right) \hat{\theta}
\]

for evaluating the entities in spherical coordinates. Here, \( r, \theta, \) and \( \phi \) are the (spherical) radial, polar, and azimuthal angles. Then,

a. Evaluate the gradient of (i.e., the negative of the force field associated with) each of the functions

\[
V_1(r, \theta, \phi) = \frac{2z^2 - r^2}{(r^2 + z^2)^{1/2}} \quad V_2(r, \theta, \phi) = \frac{e^{-a(r^2 + z^2)^{1/2}}}{(r^2 + z^2)^{1/2}}
\]

\[ \]
\[ +\frac{1}{r} \left( \frac{\partial(rQ\theta)}{\partial r} - \frac{\partial Q}{\partial \theta} \right) \hat{\phi} \]

for evaluating these entities in spherical coordinates. Here, \( r, \theta, \) and \( \phi \) are the (spherical) radial, polar, and azimuthal coordinates. Then,

a. Evaluate the gradient of (i.e., the negative of the force field associated with) each of the functions
\[ V_1(r, \theta, \phi) = \frac{1}{r} \quad V_2(r, \theta, \phi) = \frac{z}{r^2} \quad V_3(r, \theta, \phi) = \frac{r^2(3\cos^2\theta - 1)}{r^3} \quad V_4(r, \theta, \phi) = \frac{e^{-ar^2/2}}{r} \]

of the spherical variables.

b. Verify that each field obtained in part (a) is conservative by showing that the curl of each is zero.

c. With \( r = r \hat{r} \), show (1) that \( \nabla \times r = 0 \) and (2) that \( \nabla \cdot r = 3 \).

d. With \( V_5(r, \theta, \phi) = r^2 \), show that \( \nabla^2 V_5(r, \theta, \phi) = \nabla \cdot \nabla V_5(r, \phi, z) = 6 \).

e. Show that \( \nabla^2 V_1(r, \theta, \phi) = 0 \). (Note: Except at \( r = z = 0 \).)

f. Evaluate the Laplacian of \( V_4(r, \theta, \phi) \), \( \nabla^2 V_4(r, \theta, \phi) \).

6.27. Create four three-component vectors \( \mathbf{A} \), \( \mathbf{B} \), and \( \mathbf{C} \) with statements like
\[
\mathbf{A} : [\mathbf{A}[1], \mathbf{A}[2], \mathbf{A}[3]] \quad \mathbf{B} : [\mathbf{B}[1], \mathbf{B}[2], \mathbf{B}[3]] \\
\mathbf{C} : [\mathbf{C}[1], \mathbf{C}[2], \mathbf{C}[3]] \quad \mathbf{D} : [\mathbf{D}[1], \mathbf{D}[2], \mathbf{D}[3]]
\]

Then, using the file \texttt{crossdot.mac} as described in Section 6.9, show that

a. \( \mathbf{A} \times \mathbf{B} = -\mathbf{B} \times \mathbf{A} \), or, equivalently, that \( \mathbf{A} \times \mathbf{B} + \mathbf{B} \times \mathbf{A} = 0 \).

b. \( \mathbf{A} \times (\mathbf{B} \times \mathbf{C}) = (\mathbf{A} \cdot \mathbf{C})\mathbf{B} - (\mathbf{A} \cdot \mathbf{B})\mathbf{C} \)

c. \( (\mathbf{A} \times \mathbf{B}) \times \mathbf{C} = (\mathbf{A} \cdot \mathbf{C})\mathbf{B} - (\mathbf{B} \cdot \mathbf{C})\mathbf{A} \)

d. \( \mathbf{A} \cdot (\mathbf{B} \times \mathbf{C}) = (\mathbf{A} \times \mathbf{B}) \cdot \mathbf{C} \)

e. \( (\mathbf{A} \times \mathbf{B}) \cdot (\mathbf{C} \times \mathbf{D}) = (\mathbf{A} \cdot \mathbf{C})(\mathbf{B} \cdot \mathbf{D}) - (\mathbf{A} \cdot \mathbf{D})(\mathbf{B} \cdot \mathbf{C}) \)
Chapter 9

Introduction to Programming

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

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

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

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

---

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

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

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

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

9.1.1 Variables, Variable Names, and Use of Memory

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

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

- the first character must be a letter\(^2\) (a–z; A–Z),
- subsequent characters can be chosen from the letters (a–z; A–Z), the digits (0–9), and the underscore character (_), and
- the several special words (keywords, e.g., IF, DO, UNTIL, ...) that we shall introduce in subsequent sections are reserved (or protected) and cannot be used as variable names.

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

\(^2\)Some languages are case-sensitive and others are not. For our general discussion, we shall assume case insensitivity, so a and A are indistinguishable in variable names.
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 \(\times\) 3 matrix, ...). To that end, the array is among the structured variables available in all scientific programming languages. An array may be one-dimensional (a vector), two-dimensional (an \(m \times n\) matrix), or even higher dimensional. As a structured entity, an array is identified by a single name, e.g., \(data\), \(values\%\), or \(names\$\), depending on the data type of the elements of the array. Individual elements are identified by attaching an integer index or indices to the array name, e.g., \(data(I\%, J\%)\), \(values\%(4\%)\), 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

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

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

In C and in FORTRAN 90 and perhaps in a few other languages, it is possible to change the dimensions of an array dynamically, i.e., to bypass the need for the size of arrays to be explicitly specified in a DIMENSION statement and, instead, arrange for the size of an array to be set, for example, in response to values input when the program is executed. The coding to accomplish this dynamic allocation of memory will be introduced if and when we find the need for it in the remainder of this book.
CHAPTER 9. INTRODUCTION TO PROGRAMMING

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:

- Assign a value to a named memory location:

  \(\langle \text{variable} \rangle \leftarrow \langle \text{expression determining value to be assigned} \rangle\)

e.g.,

  \text{val} \leftarrow \sin(\pi \times x) \\
  \text{count}\% \leftarrow \text{count}\% + \text{1}\% \\
  \text{first\_name}$ \leftarrow "\text{David}" \\

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

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

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

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

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

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

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

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

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

\[
\begin{align*}
\langle \text{condition} \rangle & \quad \text{true if} \\
A@ = B@ & \quad A@ \text{ is the same as } B@ (A@ \text{ equals } B@) \\
A@ < B@ & \quad A@ \text{ occurs before } B@ \text{ in some collating sequence (e.g., numerical order, alphabetic order) } (A@ \text{ less than } B@) \\
A@ > B@ & \quad A@ \text{ occurs after } B@ \text{ in that collating sequence (A@ greater than } B@) \\
A@ <> B@ & \quad A@ \text{ differs from } B@ (A@ \text{ not equal to } B@) \\
A@ <= B@ & \quad A@ \text{ occurs before or is the same as } B@ (A@ \text{ less than or equal to } B@) \\
A@ >= B@ & \quad A@ \text{ occurs after or is the same as } B@ (A@ \text{ greater than or equal to } B@)
\end{align*}
\]

Table 9.2: Truth tables for OR and AND.

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

9.1.3 Logical Expressions and Conditions

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

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

\[\text{NOT } \langle \text{condition} \rangle\]

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

### 9.1.4 Syntactic Wrinkles

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

- to place more than one statement on a single physical line. In our generic code, we shall use the character ‘;’ to separate individual statements on a single line. The three lines of code early in Section 9.1.2 might be combined in one line with the coding

\[
\text{val} \leftarrow \sin(\pi x); \quad \text{count}\% \leftarrow \text{count}\% + 1\%; \quad \text{first name}\$ \leftarrow \text{"David"}
\]

- to spread a single statement over more than one physical line. In our generic code, we shall use the symbol \(\rightarrow\) at the end of a line to indicate that the statement continues on the next line. Thus, for example, the two-line statement

\[
\text{U}(I\%, J\%) \leftarrow 0.25 \times ( \text{U}(I\%+1\%, J\%) + \text{U}(I\%-1\%, J\%) \rightarrow + \text{U}(I\%, J\%+1\%) + \text{U}(I\%, J\%-1\%) )
\]

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 \texttt{BEGIN BLOCK} and \texttt{END BLOCK} to “bracket” the several statements that we wish the compiler or interpreter to see as a single (compound) statement. For example, the coding

\[
\text{BEGIN BLOCK}
\begin{align*}
\text{val} & \leftarrow \sin(\pi x) \\
\text{count}\% & \leftarrow \text{count}\% + 1\% \\
\text{first name}\$ & \leftarrow \text{"David"}
\end{align*}
\text{END BLOCK}
\]

would group the three statements as a unit.

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

### 9.1.5 Essential Control Structures

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

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

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

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

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

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

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

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

- Multi-way selection\(^10\)

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

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

- Repetition

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

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

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

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

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

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

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

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

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

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

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

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

Y ← sin( X )

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

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

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

\textbf{WHILE} (condition) \textbf{DO} (block of statements)

and the block is not executed at all if (condition) is \texttt{false} when the loop is entered. In the second
version, the test is at the end of the loop, e.g.,

\textbf{REPEAT} (block of statements) \textbf{UNTIL} (condition)

and the block will be executed at least once, whether (condition) is \texttt{true} or \texttt{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.,

\textbf{FOR} I% ← IMIN% \textbf{THRU} IMAX% \textbf{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

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

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

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

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

\texttt{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@:\[^{12}\]

   \[
   \text{TEMP@} \leftarrow \text{A@} \quad \text{! Save value in A@}
   \]
   \[
   \text{A@} \leftarrow \text{B@} \quad \text{! Copy B@ to A@}
   \]
   \[
   \text{B@} \leftarrow \text{TEMP@} \quad \text{! Copy original A@ from TEMP@}
   \]
   
   The temporary memory cell TEMP@ saves the original value of A@ so that it is still retrievable after the value of B@ has been copied into cell A@, overwriting the original contents of cell A@.

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

   \[
   \text{SENTINEL@} \leftarrow \text{⟨agreed-upon special value⟩}
   \]
   \[
   \text{[COUNT%} \leftarrow \text{0%]}
   \]
   \[
   \text{LOOP}
   \]
   \[
   \text{READ ITEM@}
   \]
   \[
   \text{EXIT_LOOP WHEN ITEM@ = SENTINEL@}
   \]
   \[
   \text{⟨Statements processing ITEM@⟩}
   \]
   \[
   \text{[COUNT%} \leftarrow \text{COUNT%} + 1%]
   \]
   \[
   \text{END LOOP}
   \]
   
   If the statements enclosed in [...] are included, then when the loop is completed the variable COUNT% will have as its value the number of values of ITEM@ processed. Note that the position in the sequence at which COUNT% is incremented is critical. As a general rule, counters should be started at the value zero before anything happens and incremented by one immediately after each of the events to be counted. More often than not, the end result of thoughtless or unsystematic positioning of the incrementation will be a final count that is off by one, one way or the other.

3. Stopping a loop by counting up:

   \[
   \text{NUMBER_OF_TIMES%} \leftarrow \text{⟨desired number of executions⟩}
   \]
   \[
   \text{COUNTER%} \leftarrow \text{0%}
   \]
   \[
   \text{LOOP}
   \]
   \[
   \text{EXIT_LOOP WHEN COUNTER% = NUMBER_OF_TIMES%}
   \]
   \[
   \text{⟨block of statements⟩}
   \]
   \[
   \text{COUNTER%} \leftarrow \text{COUNTER%} + 1%
   \]
   \[
   \text{END LOOP}
   \]
   
   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:

[^{12}]: Note again the use of the character ! to flag comments. See footnote 9.
TIMES_REMAINING% ← (desired number of executions)

LOOP
    EXIT_LOOP WHEN TIMES_REMAINING% = 0%
    (block of statements)
    TIMES_REMAINING% ←− TIMES_REMAINING% - 1%
END_LOOP

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

5. Summing [and counting]:

SENTINEL ← (agreed-upon special value)
SUM ←− 0.0
[COUNT% ←− 0%]

LOOP
    READ ITEM
    EXIT_LOOP WHEN ITEM = SENTINEL
    SUM ←− SUM + ITEM
    [COUNT% ←− COUNT% + 1%]
END_LOOP
WRITE "The sum is "; SUM

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

6. Finding extreme, stopping with a sentinel:

SENTINEL@ ←− (agreed-upon special value)
READ (first) ITEM@ from list
EXTREME@ ←− ITEM@

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

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

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

7. Finding extreme, stopping by counting:

```
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:

```
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

$$\frac{\partial^2 U}{\partial x^2} + \frac{\partial^2 U}{\partial y^2} = 0$$  \hspace{1cm} (9.1)
for a function $U(x,y)$, which may be interpreted as an electrostatic potential, a temperature distribution, a velocity potential describing the incompressible, steady-state flow of a fluid, ... Beyond the partial differential equation itself, a complete problem requires the statement of boundary 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 $0^\circ$ and the fourth edge is maintained at $100^\circ$ 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)

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

$$U_{i,j} \approx \frac{1}{4} \left( U_{i+1,j} + U_{i-1,j} + U_{i,j+1} + U_{i,j-1} \right)$$  \hspace{1cm} (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—includes 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.\textsuperscript{14} Supposing that we seek a solution on a $15 \times 15$ grid of points, we make a first pass at constructing an algorithm to implement this overall strategy by listing the statements

\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 \textit{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
  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 \textit{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.

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

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

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

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

\[
\text{FOR } K\% \leftarrow 1\% \text{ THRU } NZ\% \text{ DO} \\
\quad \text{FOR } J\% \leftarrow 1\% \text{ THRU } NY\% \text{ DO} \\
\quad \quad \text{FOR } I\% \leftarrow 1\% \text{ THRU } NX\% \text{ DO WRITE } A(I\%,J\%,K\%) \\
\]

—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.

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

```
DIMENSION ARRAY(n_x,n_y) ! Prepare array for values
NARR% ←− 1% ! Set number of arrays
NX% ←− n_x ! Set number of grid points in each coordinate
NY% ←− n_y
NZ% ←− 1%
DX ←− (x_{\text{max}} - x_{\text{min}})/(n_x - 1) ! Set increments
DY ←− (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

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

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

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

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

---

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.

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.

### 9.11 Solving Laplace’s Equation with PYTHON

Note: All PYTHON programs (*.py) and PYTHON-created data files (*.py.dat or *.python.dat) 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.

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 9.4. Note the following about this program:

- The for loop was introduced in Section 5.4. Remember, in particular, that the first line in a loop ends with a 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 print statement in this innermost statement suppresses the automatic line feed at the end of each execution of that statement. The simpler statement print($U$) would also display the solution on the screen, though the resulting output (especially if we had used a much large array for $U$) might be quite uninterpretable.

- 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.
Table 9.4: 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()

• 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

    xx = np.linspace(0.0,1.0,15); yy = np.linspace(0.0,1.0,15)
x, y = np.meshgrid(xx, yy)
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D
fig = plt.figure()
ax = fig.axes(projection='3d')
```

18 These commands were first described in Section 5.11.
9.12. CREATING AND STORING TWO-DIMENSIONAL SCALAR ARRAYS

ax.plot_wireframe(x,y,U, color='black')
plt.show()

This program is stored in a file named \texttt{laplace.py} in the directory $\texttt{HEAD/python}$. Once that file has been copied to the default directory, it can be run with a command as described in the \textit{Local Guide}. In particular, in some installations, the command

\begin{verbatim}
python laplace.py
\end{verbatim}

issued at an MS-DOS or UNIX/LINUX command window, the statement

\begin{verbatim}
execfile('laplace.py') or exec(open('laplace.py').read() )
\end{verbatim}

issued at the PYTHON prompt \texttt{>>>}, or selecting ‘Run Module’ from the \textbf{Run} menu or pressing F5 in the PYTHON Edit Window will execute the program and create the file \texttt{laplace.py.dat} in the default directory.\(^{19}\)

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

\begin{equation}
I(x,y) = I_0 \left( \frac{\sin x}{x} \right)^2 \left( \frac{\sin y}{y} \right)^2
\end{equation}

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.\(^{20}\)

9.12.6 \ldots 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 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\(^{21}\)

\begin{verbatim}
19Incidentally, remember that, if necessary, execution of a PYTHON program can usually be aborted by typing (CONTROL/C).
20Note 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.
21In 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.
\end{verbatim}
Table 9.5: 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()
```

```
# 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

for j in np.arange(iy):
    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
```

Next, we evaluate the irradiance given by Eq. (9.4) with the lines\footnote{Program notes: (1) With \texttt{ix=50}, the loop controlled by the statement \texttt{for i in np.arange(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 \texttt{ixth} element. (2) With \texttt{i} and \texttt{j} ranging from 0 to 49 and \texttt{dx} and \texttt{dy} defined as \texttt{6\pi/49}, \texttt{xf} and \texttt{yf} range from \texttt{-3\pi} to \texttt{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 outer loop.}

```
for j in np.arange(iy):
    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
```
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

\[
p(x, y, z) = \frac{1}{2\pi(27)^{3/2}} e^{-2\rho/3} \left( \frac{3z^2}{\rho^2} - 1 \right)^2 = \frac{1}{2\pi(27)^{3/2}} e^{-2\rho/3} \left( 9z^4 - 6z^2\rho^2 + \rho^4 \right) \tag{9.5}
\]

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

\[
\rho = \sqrt{x^2 + y^2 + z^2} \tag{9.6}
\]

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

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

9.13.6 ... 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.J, is named \texttt{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 \( xs, ys, \) and \( zs \) for the \textit{squares} of the coordinates and \( rho \) and \( rhos \) for the radial coordinate and its square.

- Evaluating the probability density given by Eq. (9.5). Basically, we need three loops, one ranging over each coordinate from its initial value \([x,y,z]0\) 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 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, \texttt{pdens.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.\footnote{For instructions on how to compile and run the program, see Section 9.12.6 and the \textit{Local Guide}.} Once the program has been executed, the file \texttt{pdens\_python.dat} will exist in the default directory.
9.14 Creating and Storing Two-Dimensional Vector Arrays

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

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

$$E(r,t) = A \sin \frac{n\pi y}{b} e^{i(k_x x - \omega t)} \hat{k}$$

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

where $n = 1, 2, 3, \ldots$ and, with $c$ standing for the speed of light,

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

If we focus on the physical fields (real parts of $E$ and $H$) specifically at $t = 0$, we find that

$$E(r) = A \cos \kappa_x x \sin \frac{n\pi y}{b} \hat{k}$$

$$H(r) = \frac{An\pi}{\omega \mu_0 b} \sin \kappa_x x \cos \frac{n\pi y}{b} \hat{i} - \frac{A\kappa_x}{\omega \mu_0} \cos \kappa_x x \sin \frac{n\pi y}{b} \hat{j}$$

\[\text{For a derivation of the magnetic field for a wave propagating in the z direction and a discussion on wave guides in general, see The Theory of the Electromagnetic Field by David M. Cook (Prentice-Hall, Englewood Cliffs, NJ, 1975) or Introduction to Electrodynamics by David J. Griffiths (Prentice-Hall, Upper Saddle River, NJ, 1999), Third Edition. The first of these books, out of print since the early 1990’s, became available in a Dover reprint in January, 2003, but is now (January 2017) also out of print. It may still be available in your local library.}\]
To simplify these equations further, we divide $E$ by $A$ and $H$ by $A\pi/\omega_0 b$, and we elect to measure lengths in units of $b$ by introducing the variables $\bar{y} = y/b$ and $\bar{x} = x/b$. Equation (9.10) then becomes

$$
\frac{E(r)}{A} = \cos \kappa_x b \bar{x} \sin n\pi \bar{y} \hat{k}
$$

$$
\frac{H(r)}{A\pi/\omega_0 b} = \sin \kappa_x b \bar{x} \cos n\pi \bar{y} \hat{i} - \frac{\kappa_x b}{n\pi} \cos \kappa_x b \bar{x} \sin n\pi \bar{y} \hat{j}
$$

(9.11)

and we find that the field of interest depends on two parameters $n$ and $\kappa_x b$. Further, Eq. (9.9) imposes the constraint

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

(9.12)

on these parameters. The only non-zero components of the fields in Eq. (9.11) are

$$
\begin{align*}
E_z &= \cos 2\pi \bar{x} \sin 2\pi \bar{y} \\
\bar{H}_x &= \sin 2\pi \bar{x} \cos 2\pi \bar{y} \\
\bar{H}_y &= -\cos 2\pi \bar{x} \sin 2\pi \bar{y}
\end{align*}
$$

(9.13)

where $\bar{x}$ can range over any values—we choose $0 \leq \bar{x} \leq 1$—but, to be inside the guide, $\bar{y}$ is confined to the region $0 \leq \bar{y} \leq 1$. Each component of these fields can now be represented by a two-dimensional array. The $H$ 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 $\bar{H}_x(\bar{x}, \bar{y})$, $\bar{H}_y(\bar{x}, \bar{y})$, and $\bar{E}_z(\bar{x}, \bar{y})$ over the region $0 \leq \bar{x}, \bar{y} \leq 1$, dividing each axis into 30 segments of length $1.0/29.0$, which will entail evaluating the field components at 30 values of $\bar{x}$ and 30 values of $\bar{y}$—a total of $30 \times 30 = 900$ values. Normally, these field components would be stored in three $30 \times 30$ two-dimensional arrays. Because of our declared file format, however, we must view ourselves as needing storage for three $30 \times 30 \times 1$ three-dimensional arrays. We shall, however, view the structure as a single $30 \times 30 \times 1 \times 3$ array $\bar{H}$, with $\bar{H}(*,*,1,1)$ storing $\bar{H}_x$, $\bar{H}(*,*,1,2)$ storing $\bar{H}_y$, and $\bar{H}(*,*,1,3)$ storing $\bar{E}_z$.\(^{25}\)

\section*{9.14.6 \ldots 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

\footnote{\(^{25}\)We have supposed indices starting at 1. In languages where indices start at 0, the associations would, of course, be $\bar{H}(*,*,0,0)$ storing $\bar{H}_x$, $\bar{H}(*,*,0,1)$ storing $\bar{H}_y$, and $\bar{H}(*,*,0,2)$ storing $\bar{E}_z$.}
Table 9.6: 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( 'Program described in CPSUP
' );
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()
```

# wavegd.py

```python
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

we evaluate the field components in Eq. (9.14) with the lines26,27

```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]
```
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

\[
\frac{[E_x(x, y, z), E_y(x, y, z), E_z(x, y, z)]}{q/4\pi \varepsilon_0 a^2} = \left[ \begin{array}{c} \frac{[x - 1, y - 1, z]}{[(x - 1)^2 + (y - 1)^2 + z^2]^{3/2}} - \frac{[x + 1, y - 1, z]}{[(x + 1)^2 + (y - 1)^2 + z^2]^{3/2}} \\ + \frac{[x + 1, y + 1, z]}{[(x + 1)^2 + (y + 1)^2 + z^2]^{3/2}} - \frac{[x - 1, y + 1, z]}{[(x - 1)^2 + (y + 1)^2 + z^2]^{3/2}} \end{array} \right] \tag{9.15}
\]

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

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

9.15.6 ... 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.P, 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 \(xfm, xfp, yfm\) and \(yfp\) for \(x - 1, x + 1, y - 1,\) and \(y + 1\), respectively; variables \(xfms, xfps, yfms, yfps,\) and \(zf\) 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.

\(\text{We have supposed indices starting at } 1.\text{ In languages where indices start at } 0,\text{ the associations would, of course, be } E(*,*,0,0) \text{ storing } E_x, E(*,*,0,1) \text{ storing } E_y, \text{ and } E(*,*,0,2) \text{ storing } E_z.\)
• 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, \texttt{quadpole.py} can be copied from the directory \$\texttt{HEAD/python} into your own directory, executed or—should this be necessary for your application—edited before being executed to create the desired file.\footnote{For instructions on how to run the program, see Section 9.12.6 and the \textit{Local Guide}.} Once the program has been executed, the file \texttt{quadpole\_python.dat} will exist in the default directory.

### 9.16 Reading Files

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

#### 9.16.6 ... 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 \texttt{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

\[
f = \text{open('irrad\_python.dat', 'r')}\]

Then we must read past the five lines of comments with statements like\footnote{Remember that the character \texttt{\textbackslash n} will at the end of each line will be included in the values assigned to the variables \texttt{ln?}.}

\[
\text{ln1=f.readline(); ln2=f.readline(); ln3=f.readline(); ln4=f.readline(); ln5=f.readline()}
\]

Next, we read the four dimensions from the file with the statements

\[
narr=int(f.readline()); \text{ix=int(f.readline()); iy=int(f.readline()); iz=int(f.readline())}
\]

Here, the action of the command \texttt{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
Table 9.7: Algorithm for Exercise 9.1.

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

```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.\footnote{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.}

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.V.

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 9.7 for determining the parking fee for a particular patron more explicit by expanding the two statements in the IF-THEN-ELSE structure. \textbf{Hint:} Work through several numeric examples by hand, noting particularly all decisions that you must make in order to know what arithmetic to do. \textbf{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 \textit{CASE} structures and (b) only \textit{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 9.8 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 \textit{first} if the list were to be alphabetized.
9.18. EXERCISES

Table 9.8: Algorithm for Exercise 9.5.

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

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%*A% + B%*B% = C%*C%, subject to the restriction that A%, B%, and C% shall all be
smaller than some value MAXNUM% supplied as input. Hint: Systematically examine all possibilities,
but do so thoughtfully. For example, there is no point in examining cases for which C%<A% or
C%<B%. Express your loops so that these cases (and any others that you can reject a priori) will
not even be considered. Optional: For MAXNUM% = 20%, determine the number of executions of your
innermost loop.

9.9. Suppose you have N% cards laid out in a row on a table. On each card is a single word. Determine
the end result of applying the mystery procedure laid out in Table 9.9 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 9.10 and choose a suitable name for the 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.

9.15. 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
Table 9.9: Procedure for Exercise 9.9.

PROCEDURE ???????
SCANEND% ← N%
LOOP
    CARD% ← 1%
    Obtain word on card CARD% and store in WORD$
    LATEST_WORDS$ ← WORD$
    LATEST_CARD% ← CARD%
LOOP
    CARD% ← CARD% + 1%
    Obtain word on card CARD% and store in WORD$
    IF WORD$ occurs after LATEST_WORDS$
        THEN BEGIN_BLOCK
        LATEST_WORDS$ ← WORD$
        LATEST_CARD% ← CARD%
        END_BLOCK
    END_IF
    EXIT_LOOP WHEN CARD% = SCANEND%
END_LOOP
Exchange card LATEST_CARD% with card SCANEND%
SCANEND% ← SCANEND% - 1%
EXIT_LOOP WHEN SCANEND% = 1%
END_LOOP
END_PROCEDURE

given by the equation

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

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

9.16. 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
Table 9.10: Procedure for Exercise 9.10.

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

language of your choice, testing your program(s) with the trajectory given by

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

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

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

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

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

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

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

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.

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

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

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 \(E\) at the point \(r = x \hat{i} + y \hat{j}\) in the \(xy\) plane is given in mks units by

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

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

9.20. The velocity field of an object rotating about the \(z\) axis with angular momentum \(\omega\) is given in terms of the angular velocity and the position vector \(r\) by the expression

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

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 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 \(E\) at the point \(r = x \hat{i} + y \hat{j} + z \hat{k}\) is given in mks units by

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

Expressing coordinates in terms of the dimensionless variables \(\bar{x} = x/a\), \(\bar{y} = y/a\), and \(\bar{z} = z/a\) and measuring \(E\) in the unit \(q/(4\pi \epsilon_0 a^2)\), create a file conforming to the structure described in Section 9.3.2 and containing the \(x\), \(y\), and \(z\) components of this field over the interval \(-2a \leq
9.18. EXERCISES

$\leq 2a$. Divide the interval in each coordinate direction into about 25 segments, but choose the precise number carefully so as to avoid division by zero in evaluating the field at any point. Optional: Read your file into a suitable graphical display program and produce graphs showing the character of this field.

9.22. Write and test a program to ask for the latitude and longitude of both a point of departure D and a point of arrival A on the surface of the earth and then calculate and print out the “crow-flies” distance along a great circle route from D to A. Make sure your program prints the shorter of the two distances, regardless of the location of the points, and make sure your program doesn’t run into difficulties if the two points happen to be at opposite ends of a diameter. Take the earth to be a perfect sphere with a circumference of 24900 miles (radius 3963 miles). For purposes of testing, note that Albany, NY, is at [43°40′ N, 73°45′ W]; Grand Junction, CO, is at [39°5′ N, 108°33′ W]; Los Angeles, CA, is at [34°3′ N, 118°15′ W]; Appleton, WI, is at [44°16′ N, 88°25′ W]; Calcutta India, is at [22°32′ N, 88°20′ E]; Sydney, Australia, is at [33°52′ S, 151°12′ E]; Paris, France, is at [48°49′ N, 2°29′ E]; and Stockholm, Sweden, is at [59°21′ N, 18°4′ E].
# 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]) + '
' )

# ***** Close file *****

f.close()
# 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\n' )
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)+'
' )

# ***** Write probability density to file; close file *****

f.close()
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.M  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
H=np.zeros((xdim,ydim,zdim,narr)) # Prepare for values

def wavegd_python.dat', 'w')

# ***** 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])+'\n' )

f.close()
# quadpole.py

import numpy as np  # Import needed packages
narr = 3  # Set number of arrays
xdim=30; ydim=30; zdim=30  # Set dimensions: 30x30x3
xrange=4.0  # Set length of intervals
yrange=4.0
zrange=4.0
x0=-2.0; y0=-2.0; z0=-2.0  # Set starting values of x,y,z
E=np.zeros((xdim,ydim,zdim,narr))

dx = xrange/(xdim - 1.0);  # Set increments
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
            rmp = (xfms + yfps + 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/rmp) + xfp*(1.0/rmm - 1.0/rmp)
            E[x,y,z,1] = yfm*(1.0/rmm - 1.0/rmp) + yfp*(1.0/rmm - 1.0/rmp)
            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'
)
f.write( 'The electric field from 4 point charges positioned
')
f.write( 'on the corners of a square.'
)
f.write( 'Program described in CPSUP
')
f.write( '**
')

f.write( str(narr)+'
' )
f.write( str(zdim)+'
' )
f.write( str(ydim)+'
' )
f.write( str(xdim)+'
' )
# ***** 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])+'\n' )

f.close()
9.V 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 11

Solving Ordinary Differential Equations

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

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

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

11.1 Sample Problems

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

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

11.1.1 Projectile in a Viscous Medium

The projectile shown in Fig. 11.1 moves in a viscous medium near the surface of the earth. It experiences two forces, the gravitational attraction of the earth\(^1\) \(-mg \mathbf{k}\) and the viscous force \(\mathbf{F}_v\) 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.
11.1. SAMPLE PROBLEMS

opposite to the velocity \( \mathbf{v} \). Usually, the \textit{magnitude} of the viscous force is a function of the \textit{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

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

(11.1)

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

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

(11.2)

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

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

(11.3)

or, in component form, to

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

(11.4)

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

When the speed is too large for a linear approximation to the viscous damping, we can sometimes take the viscous force to be proportional instead to the \textit{square} of the \textit{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

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

(11.5)

or, in component form, to

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

(11.6)

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

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

(11.7)

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

To cast these equations in a dimensionless form, we would start by choosing a unit of length, say \( a \)—which might be chosen arbitrarily, chosen to be one of the initial positions, or merely symbolized initially and chosen later to simplify the equations as we converge on their dimensionless form. Then, dividing by \( m \) and rescaling the dimensional coordinates by introducing the dimensionless
coordinates \( \bar{x} = x/a, \bar{y} = y/a, \) and \( \bar{z} = z/a \) (i.e., \( \bar{r} = r/a \)), we find that Eqs. (11.3) and (11.5) become
\[
d\bar{r}^2 dt^2 = -\frac{g}{a} k - \frac{b}{m} \frac{d\bar{r}}{dt}
\]
and
\[
d\bar{r}^2 dt^2 = -\frac{g}{a} k - \frac{ba}{m} \frac{d\bar{r}}{dt} \sqrt{\left(\frac{d\bar{x}}{dt}\right)^2 + \left(\frac{d\bar{y}}{dt}\right)^2 + \left(\frac{d\bar{z}}{dt}\right)^2}
\]
respectively. Next, we recognize that \( \sqrt{g/a} \) is dimensionally a frequency.\(^2\) Hence, the variable \( \bar{t} = t/\sqrt{g/a} \) provides a suitable rescaling of the independent variable. In terms of \( \bar{t} \), Eqs. (11.8) and (11.9) become
\[
d\bar{r}^2 \bar{t}^2 = -\dot{k} - \frac{b}{m} \sqrt{\frac{\bar{a}}{\bar{g}}} \frac{d\bar{r}}{d\bar{t}}
\]
and
\[
d\bar{r}^2 \bar{t}^2 = -\dot{k} - \frac{ba}{m} \frac{d\bar{r}}{d\bar{t}} \sqrt{\left(\frac{d\bar{x}}{d\bar{t}}\right)^2 + \left(\frac{d\bar{y}}{d\bar{t}}\right)^2 + \left(\frac{d\bar{z}}{d\bar{t}}\right)^2}
\]
Finally, we introduce the symbol \( \bar{\beta} \) for the dimensionless quantity \( b\sqrt{a/g}/m \) in the first of these equations and for the dimensionless quantity \( ba/m \) in the second.\(^3\) In the end, the equations in dimensionless form are
\[
d\bar{r}^2 \bar{t}^2 = -\dot{k} - \bar{\beta} \frac{d\bar{r}}{d\bar{t}}
\]
and
\[
d\bar{r}^2 \bar{t}^2 = -\dot{k} - \bar{\beta} \frac{d\bar{r}}{d\bar{t}} \sqrt{\left(\frac{d\bar{x}}{d\bar{t}}\right)^2 + \left(\frac{d\bar{y}}{d\bar{t}}\right)^2 + \left(\frac{d\bar{z}}{d\bar{t}}\right)^2}
\]
These equations are, of course, to be solved subject to the dimensionless initial conditions
\[
\bar{r}(0) = \frac{r_0}{a} \quad ; \quad \bar{v}(0) = \frac{d\bar{r}}{d\bar{t}}(0) = \frac{d(r/a)}{d(t/\sqrt{g/a})}(0) = \frac{v_0}{\sqrt{g/a}}
\]
from which we conclude that dimensionless velocities are measured in units of \( \sqrt{g/a} \).

In truth, however, these particular equations are not really second-order equations. They can be reduced to first-order equations by focusing attention on the components \( \{v_x, v_y, v_z\} \) of the velocity as the dependent variables. In those terms, the three members of Eq. (11.4) become
\[
m \frac{dv_x}{dt} = -bv_x \quad ; \quad m \frac{dv_y}{dt} = -bv_y \quad ; \quad m \frac{dv_z}{dt} = -mg - bv_z
\]
the three members of Eq. (11.6) become
\[
m \begin{bmatrix} \frac{dv_x}{dt} & \frac{dv_y}{dt} & \frac{dv_z}{dt} \end{bmatrix} = [0, 0, -mg] - b [v_x, v_y, v_z] \sqrt{v_x^2 + v_y^2 + v_z^2}
\]
and Eq. (11.7) becomes
\[
m \frac{dv_z}{dt} = -mg - bv_z |v_z|
\]
Once these first-order equations have been solved, the components of the position vector can then be found by solving the differential equations
\[
\frac{dx}{dt} = v_x \quad ; \quad \frac{dy}{dt} = v_y \quad ; \quad \frac{dz}{dt} = v_z
\]
\(^2\)Using \([\ldots]\) to indicate “the dimensions of \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 Chain Radioactive Decay

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

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

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

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

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

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

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

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

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

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

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

(11.22)

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

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

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

$$\frac{d\overline{N}}{d\overline{t}} = \overline{N}(1 - \overline{N}) \; ; \; \overline{N}(0) = \frac{N_0}{N_c}$$

(11.23)

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

11.1.4 Forced, Driven, Damped Harmonic Oscillation

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

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

(11.24)

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

$$x(0) = x_0 \; ; \; v(0) = v_0$$

(11.25)

before the problem is fully stated.

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.
To cast this equation in dimensionless form, we introduce a unit of length, say \( \ell \), rescale position with the expression \( x = x/\ell \), introduce a unit of time, say \( \tau \), rescale the physical time with the expressions \( t = t/\tau \), and thereby transform Eq. (11.24) to

\[
\frac{d^2 \bar{x}}{d\tau^2} = -\frac{\tau^2 k}{m} \bar{x} - \frac{\tau b}{m} \frac{d\bar{x}}{d\tau} + \frac{\tau^2 F(\tau \bar{t})}{m\ell} \tag{11.26}
\]

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 \( \tau \) \textit{any way we like}. Clearly, the choice \( \tau^2 k/m = 1 \) or \( \tau = \sqrt{m/k} \) is judicious. With this choice, the equation of motion becomes

\[
\frac{d^2 \bar{x}}{d\tau^2} = -\bar{x} - \frac{b}{\sqrt{mk}} \frac{d\bar{x}}{d\tau} + \frac{F(\bar{t})}{k\ell} \tag{11.27}
\]

where, in the final form, we have set \( \beta = b/\sqrt{mk} \) and \( \bar{F}(\bar{t}) = F(\tau \bar{t})/k\ell \). The single, dimensionless parameter \( \beta \) contains the essential influence of the three parameters \( m, b, \) and \( k \) once differences attributable to scaling have been removed. The dimensionless force \( \bar{F} \) 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

\[
\bar{x}(0) = \frac{x_0}{\ell} ; \quad \bar{v}(0) = \frac{d\bar{x}}{d\tau}(0) = \frac{d(x/\ell)}{d(t/\tau)} = \frac{v_0}{\ell/\tau} \tag{11.28}
\]

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

\[
\frac{d\bar{x}}{d\tau} = \bar{v} ; \quad \frac{d\bar{v}}{d\tau} = -\bar{x} - \beta \bar{v} + \bar{F}(\bar{t}) \tag{11.30}
\]

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

$$V(t) - L \frac{di}{dt} - \frac{q}{C} - iR = 0 \implies L \frac{di}{dt} + iR + \frac{q}{C} = V(t) \quad (11.31)$$

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

$$i = \frac{dq}{dt} \quad (11.32)$$

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

$$q(0) = q_0 ; \quad \frac{dq}{dt}(0) = i(0) = i_0 \quad (11.33)$$

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

$$L \frac{d^2q}{dt^2} + R \frac{dq}{dt} + \frac{q}{C} = V(t) \quad (11.34)$$

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

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

11.1.6 Coupled Oscillators

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

\(^7\)Years ago, when digital computers were not as fast as they have come to be and smooth graphical output from a digital computer was unheard of, correspondences such as this one were the basis of the analog computer, 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.
11.1. SAMPLE PROBLEMS

Figure 11.5: A system of two coupled objects.

Figure 11.6: An object moving in a central force.

of constant $k'$. Let the (horizontal) surface on which these objects slide be frictionless, and let $x_1(t)$ and $x_2(t)$ be the displacement of each object from its equilibrium point. Then, Newton’s second law yields

$$m \frac{d^2 x_1}{dt^2} = -k x_1 + k'(x_2 - x_1) \quad ; \quad m \frac{d^2 x_2}{dt^2} = -k x_2 - k'(x_2 - x_1)$$  \hfill (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 $\kappa = k'/k$. The equations then become

$$\frac{d^2 \bar{x}_1}{d\bar{t}^2} = -\bar{x}_1 + \kappa (\bar{x}_2 - \bar{x}_1) \quad ; \quad \frac{d^2 \bar{x}_2}{d\bar{t}^2} = -\bar{x}_2 - \kappa (\bar{x}_2 - \bar{x}_1)$$  \hfill (11.36)

Initial conditions such as

$$\bar{x}_1(0) = \bar{x}_{10} \quad ; \quad \frac{d\bar{x}_1}{d\bar{t}}(0) = \bar{v}_{10} \quad ; \quad \bar{x}_2(0) = \bar{x}_{20} \quad ; \quad \frac{d\bar{x}_2}{d\bar{t}}(0) = \bar{v}_{20}$$  \hfill (11.37)

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

11.1.7 Motion under Central Forces

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

\(^8\)A central force is one whose direction is always away from or towards a fixed point—the force center—and whose magnitude depends only on the distance from that point.
the differential equation
\[ m \frac{d^2 \mathbf{r}}{dt^2} = \mathbf{F} = f(r) \mathbf{\hat{r}} \] (11.38)

where \( f(r) > 0 \) corresponds to a repulsive force and \( f(r) < 0 \) corresponds to an attractive force. Two articulations of this equation are in order. A more specific expression in polar coordinates is treated in an exercise. Here, we extract its components in Cartesian coordinates \((x, y)\). Since \( r^2 = x^2 + y^2 \), \( r = x \mathbf{i} + y \mathbf{j} \), and
\[ \mathbf{\hat{r}} = \frac{r}{r} = \frac{x \mathbf{i} + y \mathbf{j}}{\sqrt{x^2 + y^2}} \] (11.39)

the Cartesian components of Eq. (11.38) are
\[
\begin{align*}
m \frac{d^2 x}{dt^2} &= f \left( \sqrt{x^2 + y^2} \right) \frac{x}{\sqrt{x^2 + y^2}}; \\
m \frac{d^2 y}{dt^2} &= f \left( \sqrt{x^2 + y^2} \right) \frac{y}{\sqrt{x^2 + y^2}}
\end{align*}
\] (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
\[
\begin{align*}
x(0) &= x_0; & \frac{dx}{dt}(0) &= v_x(0) = v_{x0}; \\
y(0) &= y_0; & \frac{dy}{dt}(0) &= v_y(0) = v_{y0}
\end{align*}
\] (11.41)

Everything discussed so far in this subsection applies to all central forces regardless of the specific form or sign of \( f(r) \). To set a more specific problem, let us narrow our purview to the planetary problem,\(^9\) in which a planet of mass \( m \) orbits a central sun of mass \( M \). When \( M \gg m \), as is often the case, the sun does not move appreciably under the action of the gravitational force exerted on it by the planet. Thus, we can treat the gravitational force on the planet as originating in a fixed force center, in which case
\[ f(r) = -G \frac{mM}{r^2} = -G \frac{mM}{x^2 + y^2} \] (11.42)

where \( G \) is the universal gravitational constant. With this specific force, the members of Eq. (11.40) become
\[
\begin{align*}
\frac{d^2 x}{dt^2} &= -G \frac{mM}{(x^2 + y^2)^{3/2}}; & \frac{d^2 y}{dt^2} &= -G \frac{mM}{(x^2 + y^2)^{3/2}}
\end{align*}
\] (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
\[ \hat{x} = \frac{x}{\ell}; \quad \hat{y} = \frac{y}{\ell}; \quad \hat{r} = \frac{r}{\ell} \] (11.44)

With this change, Eq. (11.43) becomes
\[
\begin{align*}
\frac{d^2 \hat{x}}{dt^2} &= -G \frac{mM}{\ell^3 \left( \hat{x}^2 + \hat{y}^2 \right)^{3/2}}; & \frac{d^2 \hat{y}}{dt^2} &= -G \frac{mM}{\ell^3 \left( \hat{x}^2 + \hat{y}^2 \right)^{3/2}}
\end{align*}
\] (11.45)

Finally, we introduce the dimensionless time variable \( \tau = t \sqrt{GM/\ell^3} \) to find that
\[
\begin{align*}
\frac{d^2 \hat{x}}{d\tau^2} &= -\frac{\hat{x}}{\left( \hat{x}^2 + \hat{y}^2 \right)^{3/2}}; & \frac{d^2 \hat{y}}{d\tau^2} &= -\frac{\hat{y}}{\left( \hat{x}^2 + \hat{y}^2 \right)^{3/2}}
\end{align*}
\] (11.46)

Interestingly, in Cartesian coordinates, all parameters have disappeared from the equations.

---

\(^9\)Remember, too, that the problem of a charged particle moving in the field of another charged particle is mathematically identical to the planetary problem. (See exercises.)
Translation of the initial conditions into dimensionless form is easier than translation of the differential equations. The positions, of course, become

\[ x(0) = \frac{x_0}{\ell} ; \quad y(0) = \frac{y_0}{\ell} \]  

(11.47)

To translate the velocities, we argue that

\[ \frac{dx}{dt}(0) = \frac{d(x/\ell)}{d(t\sqrt{GM/\ell^3})}(0) = \frac{v_x}{\sqrt{GM/\ell}} ; \quad \frac{dy}{dt}(0) = \frac{d(y/\ell)}{d(t\sqrt{GM/\ell^3})}(0) = \frac{v_y}{\sqrt{GM/\ell}} \]  

(11.48)

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

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

\[ \frac{mv_{\text{circ}}^2}{r_{\text{circ}}} = \frac{GMm}{r_{\text{circ}}^2} \Rightarrow v_{\text{circ}}^2 = \frac{GM}{r_{\text{circ}}} \]  

(11.49)

Translated into dimensionless form, this special relationship becomes

\[ \left( \sqrt{\frac{GM}{\ell}} v_{\text{circ}} \right)^2 = \frac{GM}{\ell r_{\text{circ}}} \Rightarrow v_{\text{circ}}^2 = \frac{1}{r_{\text{circ}}} \]  

(11.50)

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

Second, the planetary problem admits two conservation laws, each of which may be valuable in assessing the accuracy of numerically generated solutions. In a dimensional presentation, conservation of energy yields that

\[ E = \frac{1}{2} m \left( v_x^2 + v_y^2 \right) - \frac{GMm}{\sqrt{x^2 + y^2}} = \text{constant} \]  

(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

\[ \frac{1}{2} (\bar{v}_x^2 + \bar{v}_y^2) - \frac{1}{\sqrt{\bar{x}^2 + \bar{y}^2}} = \text{constant} ; \quad \bar{x}\bar{v}_y - \bar{y}\bar{v}_x = \text{constant} \]  

(11.53)

### 11.1.8 Standing Waves in a String

When a wave propagates in a flexible string, the (transverse) displacement \( u(x,t) \) at time \( t \) of the element of the string nominally at coordinate \( x \) satisfies the wave equation

\[ \frac{\partial^2 u}{\partial t^2} = c^2 \frac{\partial^2 u}{\partial x^2} \]  

(11.54)
where \( c \) is the speed of propagation of the wave along the string. In the special case that the motion of each element is sinusoidal with frequency \( \omega \), we can write
\[
u(x, t) = f(x) \cos(\omega t)
\] (11.55)
and find, on substitution into the wave equation, that \( f(x) \), which gives the amplitude of the sinusoidal motion of the element nominally at \( x \), satisfies
\[
\frac{d^2 f}{dx^2} + k^2 f = 0
\] (11.56)
where \( k^2 = \frac{\omega^2}{c^2} \). If, finally, the string is firmly tied down at two points, say \( x = 0 \) and \( x = \ell \), then this second-order, homogeneous, constant-coefficient, differential equation must be solved subject to the boundary conditions
\[
f(0) = 0 \quad ; \quad f(\ell) = 0
\] (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
\[
-\frac{\hbar^2}{2m} \frac{d^2 \psi}{dx^2} + V(x) \psi = E \psi \quad \Rightarrow \quad \frac{d^2 \psi}{dx^2} + \frac{2m}{h^2} \left( E - V(x) \right) \psi = 0
\] (11.58)
where \( \hbar \) is Planck’s constant divided by \( 2\pi \). We anticipate subsequent exploration of two specific cases. Suppose, for example, that the potential energy is infinite except in the interval \(-\ell \leq x \leq +\ell\), in which interval it is zero, i.e., suppose the particle is confined in an infinitely deep potential well that extends over the specified interval. Then, the wave function describing this particle must satisfy the Schrödinger equation with \( V = 0 \) inside this interval, and it must go to zero at each end of the interval. We seek solutions to the boundary value problem
\[
\frac{d^2 \psi}{dx^2} + \frac{2mE}{\hbar^2} \psi = 0 \quad ; \quad \psi(0) = \psi(\ell) = 0
\] (11.59)
Especially if we substitute the shorthand \( k^2 = \frac{2mE}{\hbar^2} \) (or \( E = \hbar^2k^2/2m \)), we recognize that this quantum mechanical problem is mathematically identical to the classical standing wave problem described in Section 11.1.8. This problem admits solutions only for a discrete set of special values for \( k \), i.e., only for special energies.

Suppose, alternatively, that we take the potential energy to be \( V(x) = \frac{1}{2}kx^2 \), which describes the quantum analog to the classical harmonic oscillator with spring constant \( k \). In this case, the time-independent Schrödinger equation becomes
\[
\frac{d^2 \psi}{dx^2} + \frac{2m}{\hbar^2} \left( E - \frac{1}{2}kx^2 \right) \psi = 0
\] (11.60)
or, in dimensionless form,
\[
\frac{d^2 \psi}{d\xi^2} + \left( 2\epsilon - \xi^2 \right) \psi = 0
\] (11.61)
where \( \omega = \sqrt{k/m} \), \( \epsilon = E/\hbar\omega \) and \( \xi = x\sqrt{\hbar/m\omega} \). In contrast to the infinitely deep well, the domain for the quantum oscillator extends over the interval \(-\infty < x < +\infty\), and we must require that \( \psi(x) \) approach zero as \( x \) approaches either end of this interval. Numerically, infinite domains
are complicated. Note, however, that the equation in this case is quadratic in \( x \). Thus, the solutions to the equation can be divided into two sets, one of which contains functions that are even in \( x \) (even parity) and the other of which contains functions that are odd in \( x \) (odd parity). Even functions, however, necessarily have zero derivative and non-zero value at \( x = 0 \) while odd functions, in contrast, have non-zero derivative and zero value at \( x = 0 \). These properties mean that we can replace the original boundary value problem involving Eq. (11.60) over an infinite interval with an initial value problem for the solution over half of the infinite interval. When we come later to address the actual solution of this problem, we will therefore focus on two sub-problems, one defined by the equation and initial values

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

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

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

and yielding odd solutions. The interesting aspect of these problems is that we will be able to find solutions satisfying the boundary requirement at \( \pi = +\infty \) 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

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

(11.64)

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

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

(11.65)

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

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

(11.66)

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

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

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

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

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

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

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

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

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

### 11.3 Solving ODEs Symbolically with MAXIMA

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 of the commands available in MAXIMA for performing these operations were discussed briefly in Section 6.7.8. The full spectrum of commonly used commands is conveniently divided into two categories. Those commands that are useful in addressing a single equation of whatever order include

```
depends( dvar1, ivar1, dvar2, ivar2, ... );
```

This command, in which each argument may be a single variable or a list, informs MAXIMA that certain quantities depend on other quantities—a specification necessary for the routines that solve single ODEs. The syntax is illustrated more explicitly in the examples below. The command `depends` must be invoked before `ode2`.

```
ode2( eqn, depvar, indvar );
```

which (despite the 2 in its name) requests the solution of the first- or second-order ordinary differential equation `eqn` for the dependent variable `depvar` as a function of the independent variable `indvar`. `ode2` actually invokes a succession of increasingly more complicated methods until it finds one that works. Available system variables, which provide information about the process once a solution has been obtained, include

- `method`, which is set to the method ultimately used.
- `intfactor`, which is set to any integrating factor used and will have the value `false` if no integrating factor is used.
- `yp`, which is set to the particular solution used if variation of parameters is the method and will not be set at all if that method is not used.

Each of these variables can be displayed to reveal its value. Further, solutions returned by `ode2` use the variable `%c` for the integration constant in the solution to a first-order equation and the variables `%k1` and `%k2` for the integration constants in the solution to a second-order equation.
ic1( sol, xval, yval )

which imposes an initial condition on the solution sol obtained by ode2 for a first-order
equation, i.e., determines a specific (symbolic) value for the constant %c. xval is a statement
of the form x = x0 specifying a value of the independent variable, and yval is a statement
of the form y = y0 specifying the value of the dependent variable at the given value of the
independent variable.

ic2( sol, xval, yval, ydrvval )

which imposes initial conditions on the solution sol obtained by ode2 for a second-order
equation, i.e., determines specific (symbolic) values for the constants %k1 and %k2. xval is
a statement of the form x = x0 specifying a value of the independent variable, yval and
ydrvval are statements of the form y = y0 and diff(y,x) = ydrv0 specifying the values of
the dependent variable and its first derivative at the given value of the independent variable.

bc2( sol, xval1, yval1, xval2, yval2 )

which imposes boundary conditions on the solution sol obtained by ode for a second-order
equation, i.e., determines specific (symbolic) values for the constants %k1 and %k2. Here, xval1
and xval2 are statements of the form x = x1 and x = x2 specifying two different values of
the independent variable; and yval1 and yval2 are statements of the form y = y1 and y =
y2 specifying the values of the dependent variable at the two given values of the independent
variable.

MAXIMA does not appear to have the capacity to solve ODEs when one or both boundary conditions
impose constraints on the derivatives of the functions.

To address systems of linear equations, MAXIMA uses the Laplace transform (see Sections 1.5.2,
6.7.7 and 11.2) and, thus, incorporates the initial conditions as the solutions are found. Behind the
scenes, MAXIMA invokes the already-described\(^\text{14}\) commands laplace and ilt (and, occasionally,
we may have to do the same). To support a more direct attack, MAXIMA provides two commands
that are specifically built for solving systems of linear, ordinary differential equations:

\begin{verbatim}
desolve([list of equations], [list of dependent variables]);
\end{verbatim}

which requests the solution of the system of linear equations provided in the first list for the
variables provided in the second list.

\begin{verbatim}
atvalue(function, indval, functionval);
\end{verbatim}

which provides the value functionval to be imposed by desolve on the function function
at the value of the independent variable specified by indval. Here, function must include
the independent variable as an explicit argument and may be either the dependent variable
itself or some derivative of that dependent variable, indval is a statement of the form x = 0
specifying a value of the independent variable, and functionval is an expression stipulating
the value to be assumed by function at the specified point. The syntax of this command
is more clearly illustrated in the specific examples below. The command atvalue must be
invoked before desolve.

The command depends plays no role in dealing with systems of equations; all dependences must be
explicitly displayed, both in desolve and in atvalue.

\(^{14}\)See Section 6.7.7.
The MAXIMA distribution also includes the command `contrib_ode`, which can be loaded into the MAXIMA workspace with the command `load('contrib_ode');`. The additional features, which include a capacity to apply other methods to solve linear and non-linear first-order ODEs and linear homogeneous second-order ODEs, are described in the MAXIMA manuals. The syntax is the same as that for `ode2`. Be aware, however, that (as of May 2016) this package has not yet stabilized so it should be used with caution. Further, once it has stabilized, it may be moved to be available without invoking the `load` command.

11.3.1 Projectile in a Viscous Medium

For the sake of a comparison with the result obtained in Section 11.2, we begin by using MAXIMA to solve the third member of Eq. (11.4). The necessary commands are\(^{15,16}\)

```maxima
(%i1) depends( z, t )$
(%i2) eq : m*'diff(z,t,2)=-m*g-b*'diff(z,t)$
(%i3) soln : ode2( eq, z, t );

Is b zero or nonzero? n;

(%o3) z = %k2 e^{-bt/m} - \frac{bgmt - gm^2}{b^2} + %k1

(%i4) method;
(%o4) variationofparameters

(%i5) ic2( %o3, t=0, z=z0, 'diff(z,t)=vz0 );

(%o5) z = \frac{bz_0 + mvz_0}{b} - \frac{e^{-bt/m}(bmvz_0 + gm^2)}{b^2} - \frac{bgmt - gm^2}{b^2}
```

Unfortunately, this form of the solution is not particularly transparent. Cleaning up the expression can be tricky, and success is likely to depend on having an \textit{a priori} sense of the desired end result. The route to adopt will be clearer if we invoke the statement

```maxima
(%i6) soln : expand( rhs(%) );

(%o6) z_0 - \frac{me^{-bt/m}vz_0}{b} + \frac{mvz_0}{b} - \frac{gm^2e^{-bt/m}}{b^2} - \frac{gmt}{b} + \frac{gm^2}{b^2}
```

to extract the right-hand side of the solution at %o5 and expand the result so that the individual terms will be clearer. In this form, we note that two terms have a common factor of \(gm^2/b^2\) and two other terms have a common factor of \(mvz_0/b\). Because the two terms containing the factor \(e^{-bt/m}\) are not stored internally in a way that allows isolation of the factors of interest, a statement like

```maxima
collectterms(soln, gm^2/b^2);
```

will not achieve the desired rearrangement. Instead, we must (temporarily) replace \(gm^2/b^2\) with, say, \(\alpha\) and \(mvz_0/b\) with, say, \(\beta\) before \texttt{collectterms} can be invoked successfully. Even those temporary replacements, however, must be done in a way that makes sure MAXIMA can, in fact, identify the quantities to be replaced in its internal storage. Thus, substituting \(\alpha\) as desired requires that we invoke the statements

\(^{15}\)As in Chapter 6, we abbreviate the presentation of MAXIMA dialogs by making liberal use of terminating dollar signs to suppress intermediate output. You are urged to duplicate the dialog in an actual session with MAXIMA, replacing all terminating dollar signs with semi-colons.

\(^{16}\)See Section 6.7.8 for a discussion of the use of the tic in the present context.
11.3. SOLVING ODES SYMBOLICALLY WITH MAXIMA

(%i7) subst( alpha*b^2/g, m^2, % );
(%o7) z_0 - \frac{m e^{-bt/m} v_{z0}}{b} + \frac{mv_{z0}}{b} - \alpha e^{-bt/m} - \frac{gmt}{b} + \alpha

(%i8) subst( beta*b/m, vz0, % );
(%o8) z_0 - \beta e^{-bt/m} + \alpha \left(1 - e^{-bt/m}\right) - \frac{gmt}{b} + \beta

(%i9) collectterms(%, alpha, beta);
(%o9) z_0 + \beta \left(1 - e^{-bt/m}\right) + \alpha \left(1 - e^{-bt/m}\right) - \frac{gmt}{b}

(%i10) collectterms( %, 1-%e^(-b*t/m) );
(%o10) z_0 + (\beta + \alpha) \left(1 - e^{-bt/m}\right) - \frac{gmt}{b}

(%i11) soln : ev( %, alpha = g*m^2/b^2, beta = m*vz0/b );
(%o11) z_0 + \left(1 - e^{-bt/m}\right) \left(\frac{mv_{z0}}{b} + \frac{gm^2}{b^2}\right) - \frac{mg}{b} t

We can now also see the common factor of \(m/b\) in the second parentheses, so an even cleaner form of the solution would be

\[ z(t) = z_0 - \frac{mg}{b} t + m \left(v_{z0} + \frac{mg}{b}\right) \left(1 - e^{-bt/m}\right) \quad (11.69) \]

but we will not struggle to force MAXIMA to achieve this last rearrangement. This result agrees fully with the result obtained in Eq. (11.68).

Wisdom suggests that we should whenever possible actually verify any solution that MAXIMA generates. We can substitute the solution into the original equation and evaluate the derivatives explicitly with the statement

(%i12) expand( ev( eq, z=soln, diff ) );
(%o12) -be^{-bt/m}v_{z0} - mge^{-bt/m} = -be^{-bt/m}v_{z0} - mge^{-bt/m}

The left and right sides of this result are clearly equal, though we could make that more explicit either by using the statement

(%i13) lhs(%) - rhs(%) ;
(%o13) 0

or by invoking the test

(%i14) is( %o12 ) ;
(%o14) true

where, with %o12 having the form \(a = b\), the command is simply returns \(true\) or \(false\) depending on whether the specified expression is, in fact, true or false. To complete the verification, we check satisfaction of the initial conditions with the statement

(%i15) vel : diff( soln, t )$
(%i16) expand( ev( [ soln, vel ], t=0 ) );
(%o16) \left[ z_0, v_{z0} \right]
CHAPTER 11. SOLVING ORDINARY DIFFERENTIAL EQUATIONS

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\(^\text{17}\)

\[(\%i17) \ \text{expand( taylor( soln, b, 0, 1 ) );} \]
\[(\%o17) \ z_0 + v_0 z_0 t - \frac{1}{2} g t^2 - \frac{b t^2}{2m} \left(v_0 - \frac{1}{3} g t\right) \]
\[(\%o17) \ v_0 - g t - \frac{b t}{m} \left(v_0 - \frac{1}{2} g t\right) \]

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.

11.3.2 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 MAXIMA statements

\[(\%i11) \ \text{depends( N, t )$} \]
\[(\%i12) \ \text{eq : 'diff( N, t ) = k*N*(1-N/NC)$} \]
\[(\%i13) \ \text{ode2( %, N, t );} \]
\[(\%o13) \ \frac{-[\log(N - N_c) - \log(N)]}{k} = t + \%c \]
\[(\%o14) \ \text{ic1( %, t = 0, N = N0 );} \]
\[(\%o14) \ \frac{-\log(N - N_c) - \log(N)}{k} = -\log(N_0 - N_c) + \log(N_0) \]

\[(\%i15) \ \text{method;} \]
\[(\%o15) \ \text{separable} \]

Strictly, of course, we want $N$ expressed as an explicit function of $t$. The \texttt{solve} command, however, will not by itself solve \%o4 for $N$. We must guide MAXIMA with the statements

\[(\%i16) \ \text{part( k*solve( %o4, t ), 1 );} \]

\[(\%o16) \ kt = \log(N_0 - N_c) - \log(N - N_c) - \log(N_0) + \log(N) \]

\(^{17}\)Again, we have taken the liberty to rearrange the terms in MAXIMA’s output to facilitate recognition of the limits.
11.3. SOLVING ODES SYMBOLICALLY WITH MAXIMA

(%i7) logcontract( % );

Compress logarithms.

(%i7) \( kt = \log \left( \frac{N(N_c - N_0)}{N_0(N_c - N)} \right) \)

Remove logarithms by using the map function to apply exp separately to each side of %i7.

(%i8) \( e^{kt} = \frac{N(N_c - N_0)}{N_0(N_c - N)} \)

(%i9) part( solve( %, N ), 1 );

Solve (finally) for N.

(%i9) \( N = \frac{N_0N_c e^{kt}}{N_0 e^{kt} + N_c - N_0} \)

(%i10) ev( %, t = 0 );

Verify satisfaction of initial condition.

(%i10) \( N = N_0 \)

(%i11) lhs( eq ) - rhs( eq ) = 0$

Rearrange equation ...

(%i11) ev( %, %o9, diff, radcan );

... and verify that solution satisfies it.

Note: Here, the versatility of the function ev is again demonstrated. On line \%i12, we have told MAXIMA to evaluate %o11 by substituting the solution from line %o9, explicitly evaluating any occurrences of the operator diff, and finally applying the operator radcan to the result. Thus, in one statement, we have told MAXIMA to accomplish several operations.

(%o12) 0 = 0

(%i13) limit( %o9, t, inf);

Examine long-term solution.

Is k positive, negative, or zero? p;

(%o13) \( N = N_c \)

When k > 0, solution approaches carrying capacity, regardless of initial population.

(%i14) limit( %o9, t, inf);

Is k positive, negative, or zero? n;

(%o14) \( N = 0 \)

When k < 0, population becomes extinct.

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 extract the function to be plotted and recast it with the statements

(%i15) rhs( %o9)/NC;

Recast to measure population in units of \( N_c \).

(%i15) \( \frac{N_0 e^{kt}}{N_0 e^{kt} + N_c - N_0} \)

(%i16) factor( subst( alpha*NC, N0, % ) );

Recast to measure initial population in units of \( N_c \).

(%i16) \( \frac{\alpha e^{kt}}{\alpha e^{kt} - \alpha + 1} \)

(%i17) soln : subst( tau/k, t, % );

Finally, recast to measure time in units of \( 1/k \).

(%i17) \( \frac{\alpha e^{\tau}}{\alpha e^{\tau} - \alpha + 1} \)

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

(%i18) soln1: ev( soln, alpha=0.25 )$

(%i19) soln2: ev( soln, alpha=0.5 )$

(%i20) soln3: ev( soln, alpha=1.0 )$
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, which—because \( N(0)/N_c = \alpha \), is also the value of \( \alpha \) for the graph.

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

(\%i24) plot2d( ev( soln, alpha=0.05 ), [tau, 0.0, 10.0], [y, 0.0, 1.0],
   [legend, false], [style, [lines, 4]], [color, black],
   [xlabel, "tau"], [ylabel, "N(tau) / 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.

11.3.3 Damped Harmonic Oscillator

A third example of the use of MAXIMA to solve a single second-order, linear, constant-coefficient, homogeneous, ordinary differential equation was presented as an example in Chapter 6. You are invited to review the discussion of the damped harmonic oscillator in Sections 6.7.8 and 11.1.4.

11.3.4 Chain Radioactive Decay

As a first example of the use of MAXIMA 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
11.3. SOLVING ODES SYMBOLICALLY WITH MAXIMA

Figure 11.8: Logistic growth when the population is initially much smaller than the carrying capacity. This curve is called the *sigmoid* curve.

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 MAXIMA statements

\[
\begin{align*}
(\%i1) \ & \text{eqa} : \ '\text{diff}( A(t), t ) = -k_A A(t) \\
(\%i2) \ & \text{eqb} : \ '\text{diff}( B(t), t ) = k_A A(t) - k_B B(t) \\
(\%i3) \ & \text{eqc} : \ '\text{diff}( C(t), t ) = k_B B(t) \\
(\%i4) \ & \text{atvalue}( A(t), t = 0, A_0 ) \\
(\%i5) \ & \text{atvalue}( B(t), t = 0, 0.0 ) \\
(\%i6) \ & \text{atvalue}( C(t), t = 0, 0.0 ) \\
(\%i7) \ & \text{desolve( [eqa,eqb,eqc], [A(t),B(t),C(t)] );} \\
\end{align*}
\]

Evidently, in this chain decay, we see a linear combination of two exponential decays, each with its own distinct half-life.

We conclude the discussion of this example by generating a quick graph of \( A(t) \), \( B(t) \), and \( C(t) \) for a specific set of values with the statements

\[
\begin{align*}
(\%i8) \ & A : \ \text{rhs( part(\%o7,1) )} \\
(\%i9) \ & B : \ \text{limit( rhs(part(\%o7,2) ), k_A, k_B )} \\
(\%i10) \ & C : \ \text{limit( rhs(part(\%o7,3) ), k_A, k_B )} \\
(\%i11) \ & [ A0:1000, kA:0.1, kB:0.1 ] \\
\end{align*}
\]

and plot the graph with the statement

\[
(\%i12) \ \text{plot2d( [A,B,C], [t,0.0,50.0], [xlabel, "t"], [ylabel, "A, B, C"], [legend, false], [color, black], [style, [lines, 4]] );}
\]
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.

The resulting graph is shown in Fig. 11.9.

### 11.3.5 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.\(^1\)

First, suppressing the display of MAXIMA’s responses, we enter the equations with the statements\(^2\)

```plaintext
(%i1) eq1 : 'diff(x1(t),t,2) + x1(t) - kappa*(x2(t)-x1(t)) = 0
(%i2) eq2 : 'diff(x2(t),t,2) + x2(t) + kappa*(x2(t)-x1(t)) = 0
```

Then we specify the initial values (again suppressing the display) and invoke `desolve` with the statements\(^3\)

```plaintext
(%i3) atvalue( x1(t), t = 0, x10 )
(%i4) atvalue( x2(t), t = 0, x20 )
(%i5) atvalue( 'diff(x1(t), t), t = 0, 0 )
(%i6) atvalue( 'diff(x2(t), t), t = 0, 0 )
(%i7) desolve( [eq1,eq2], [x1(t),x2(t)] );
```

Is $2\kappa + 1$ positive, negative, or zero? p;

---

\(^1\)You are urged to explore other initial conditions.

\(^2\)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.
11.3. SOLVING ODES SYMBOLICALLY WITH MAXIMA

(%o7) \[
\begin{align*}
    x_1(t) &= \frac{1}{2} \cos(t)(x_{20} + x_{10}) - \frac{1}{2} \cos(\sqrt{2\kappa + 1} t)(x_{20} - x_{10}), \\
    x_2(t) &= \frac{1}{2} \cos(t)(x_{20} + x_{10}) + \frac{1}{2} \cos(\sqrt{2\kappa + 1} t)(x_{20} - x_{10})
\end{align*}
\]

Clearly, the solution is a superposition of different sinusoidal oscillations, one at frequency 1 (in units of $\omega = \sqrt{\kappa/m}$) 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).

11.3.6 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 would begin by finding the solution to the ODE with the statements

(%i1) depends( f, x ) $ \\
(%i2) eq : '\text{diff}( f, x, 2 ) + k^2 * f = 0 $ \\
(%i3) soln : ode2( eq, f, x );

Is $k$ zero or nonzero? n;

(%o3) $f = %k1 e^{ikx} + %k2 e^{-ikx}$

Then (naively, as it turns out), we might attempt to impose the boundary conditions with the statement

(%i4) bc2(soln, x=0, f=0, x=el, f=0 );

but we would quickly find that this statement doesn’t work.^{20} (MAXIMA returns $f = 0$, having concluded that only the trivial solution to the ODE can be made to satisfy both boundary conditions.) MAXIMA’s routine bc2 doesn’t recognize that we can find non-trivial solutions, but only if we are prepared to constrain $k$ to a limited set of values. To help MAXIMA recognize that the problem is a bit more subtle, we issue the statements

(%i5) eq1 : ev( rhs(soln), x=0 ) = 0; \quad \text{Impose condition at } x = 0. \\
(%o5) %k2 + %k1 = 0 \\
(%i6) eq2 : ev( rhs(soln), x=el ) = 0; \quad \text{Impose condition at } x = \ell. \\
(%o6) %k1 e^{ik\ell} + %k2 e^{-ik\ell} = 0

to impose the boundary conditions explicitly. At this point, we recognize that we have a pair of equations to be solved simultaneously for $%k1$ and $%k2$. We also recognize, however, that these equations are homogeneous and will in fact have non-trivial solutions only if the determinant of the coefficient matrix happens to be zero. To complete the problem, then, we must extract that coefficient matrix, set its determinant to zero, and solve the resulting equation to find any values of $k$ that may admit non-trivial solutions to the equations themselves. We accomplish that end with the statements

^{20}Had we imposed different values on the solution at the two ends of the interval, invocation of bc2 would have been more successful. Try it, say, with $f(0) = 0$ and $f(\ell) = x_1$. 

The statement at line \texttt{\%i9} also produces a warning, specifically “solve: using arc-trig functions to get a solution. Some solutions will be lost.” \textsc{maxima} fails to recognize that, formally, the solution to $\sin(k\ell) = 0$ is $k\ell = \sin^{-1} 0$, which admits the solution $k = n\pi/\ell$, where $n$ is any positive or negative integer or zero.

Having now found acceptable values of $k$, we find the corresponding values of $\%k1$ and $\%k2$ by returning to equations eq1 and eq2. For the allowed values of $k$,

\begin{verbatim}
(%i10) declare( n, integer )$
(%i11) eq1a : ev( eq1, k = n*%pi/el )$
(%o11) $\%k2 + \%k1 = 0$
(%i12) ev( eq2, k = n*%pi/el )$
(%i13) eq2a : factor( demoivre( \% ) )$
(%o13) $(\%k2 + \%k1)(-1)^n = 0$
\end{verbatim}

As expected, the two equations at \texttt{\%o11} and \texttt{\%o12} are the same. \textit{Both} will be satisfied if we require that $\%k1 = -\%k2$, but these equations impose no further constraints. We learn nothing about the remaining constant, which evidently remains arbitrary insofar as the conditions of the problem are concerned. To complete the solution, then, we invoke the statements

\begin{verbatim}
(%i14) [\%k1 : -\%k2, k : n*%pi/el]$  
(%i15) demoivre( ev( rhs(soln ) ))$
(%i16) solna : expand( \% )$
(%i17) ev(solna, \%k2 = -A/(2*%i) )$
(%o17)  $A \sin \frac{n\pi x}{\ell}$
\end{verbatim}

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 $n\pi/\ell$ 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 as simply an alternative choice of $\%k2$). Further, since $\omega = kc$, we would associate with the solution \texttt{\%o15} the frequency

$$\omega_n = k_n c = \frac{n\pi c}{\ell} = n\omega_1$$

(11.70)

where $\omega_1$ is the fundamental frequency. We find, in particular, that all allowed frequencies are integer multiples of the fundamental frequency.

### 11.3.7 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
11.3. SOLVING ODES SYMBOLICALLY WITH MAXIMA

in Section 11.3.6. Since the allowed values of \( k \) have turned out to be \( n\pi/\ell \), we then conclude that the allowed energies in the quantum problem would be

\[
E_n = \frac{\hbar^2 k_n^2}{2m} = \frac{n^2\hbar^2 \pi^2}{2m\ell^2}
\]

Further, we would associate the (unnormalized) wave function

\[
\psi_n(x) = N_n \sin \frac{n\pi x}{\ell}
\]

with the energy \( E_n \). In the quantum problem, the allowed energies are not integer multiples of the lowest energy; rather, the allowed energies increase in proportion to the squares of the integers.

11.3.8 Finding Solutions by Series Methods

Particularly when analytic methods appear to be yielding little result, we are sometimes tempted to invoke the method of Frobenius, representing the solution to the ODE of interest as a power series of the form

\[
f(x) = \sum_{n=0}^{\infty} a_n x^{n+\alpha}
\]

and seeking an indicial equation determining \( \gamma \) and recursion relationships determining \( a_n \) from a few of the early coefficients. In the absence of a simple MAXIMA 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 MAXIMA systematically through the process. For the sake of concreteness, suppose we seek a solution to the equation

\[
d^2y/dx^2 + y = 0
\]

We might then define a (truncated) series for the solution with the statement

(\%i1) y : sum( a[i]*x^(i+alpha), i, 0, 8 );

(\%o1) \[a_8 x^{\alpha+8} + a_7 x^{\alpha+7} + a_6 x^{\alpha+6} + a_5 x^{\alpha+5} + a_4 x^{\alpha+4} + a_3 x^{\alpha+3} + a_2 x^{\alpha+2} + a_1 x^{\alpha+1} + a_0 x^{\alpha}\]

(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

(\%i2) eq : diff(y,x,2) + y;

(\%o2) \[a_8 x^{\alpha+8} + a_7 x^{\alpha+7} + a_6 x^{\alpha+6} + a_5 x^{\alpha+5} + a_4 x^{\alpha+4} + a_3 x^{\alpha+3} + a_2 x^{\alpha+2} + a_1 x^{\alpha+1} + a_0 x^{\alpha}\]

where we have omitted several of the terms in the middle of this long expression. Not surprisingly, each term has a factor of \( x^{\alpha} \). Since we ultimately will set this expression equal to zero, we can divide the expression by \( x^{\alpha} \). To clear the negative powers of \( x \), we also multiply by \( x^2 \). The necessary MAXIMA statements and their output are

(\%i13) eq1 : distrib(x^2*eq/x^alpha)$

Multiply by \( x^2 \), divide by \( x^{\alpha} \), and distribute the product over all terms.

(\%i14) eq2 : collectterms( eq1, x^10, x^9, \ldots, x^2, x );

Bring together terms with the same power of \( x \). Here \ldots is to be replaced by all powers of \( x \) before executing the statement.

(\%o4) \[a_8 \alpha(\alpha-1) + a_1 \alpha(\alpha+1)x + [a_2(\alpha+1)(\alpha+2) + a_0] x^2 + [a_3(\alpha+2)(\alpha+3) + a_1] x^3 + \ldots\]
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 elements of an array and display them with the statement

\[
\text{(%i5) for i:0 thru 10 do coef[i] : coeff(eq2, x, i ) ;}
\]

\[
\text{(%o5) done}
\]

\[
\text{(%o6) for i:0 thru 10 do print("coef[",i,"] =", coef[i] )}
\]

\[
\begin{align*}
\text{coef[0]} &= a_0(\alpha - 1) \\
\text{coef[1]} &= a_1\alpha(\alpha+1) \\
\text{coef[2]} &= a_2(\alpha+1)(\alpha+2) + a_0 \\
\text{coef[3]} &= a_3(\alpha+2)(\alpha+3) + a_1 \\
\text{coef[4]} &= a_4(\alpha+3)(\alpha+4) + a_2 \\
\text{coef[5]} &= a_5(\alpha+4)(\alpha+5) + a_3 \\
\text{coef[6]} &= a_6(\alpha+5)(\alpha+6) + a_4 \\
\text{coef[7]} &= a_7(\alpha+6)(\alpha+7) + a_5 \\
\text{coef[8]} &= a_8(\alpha+7)(\alpha+8) + a_6 \\
\text{coef[9]} &= a_7 \\
\text{coef[10]} &= a_8
\end{align*}
\]

though we must ignore coef9 and coef10, which are incomplete because of the truncation of the initial series at \( %e1 \). We then infer that, in the range \( 2 \leq n \leq 8 \) (and presumably beyond 8), the general coefficient satisfies the condition

\[
\text{coef}_n = a_n(\alpha + n - 1)(\alpha + n) + a_{n-2}; \quad n \geq 2
\]

(11.74)

The statement

\[
\text{(%i7) coef[n] : a[n]*(alpha+n-1)*(alpha+n) + a[n-2];}
\]

\[
\text{(%o7) (n + \alpha - 1)(n + \alpha)a_n + a_{n-2}}
\]

will enter this relationship into MAXIMA’s workspace.

Pursuing the solution from this point by the method of Frobenius, we must require (1) \( a_0 \neq 0 \) and (2) the coefficient of each power of \( x \) separately to be zero. For \( \text{coef}_0 \) to be zero, we find from the statements

\[
\text{(%i7) coef[0]/a[0];}
\]

\[
\text{(%o7) \alpha(\alpha - 1)}
\]

\[
\text{(%i8) alp : solve( %=0, alpha );}
\]

\[
\text{(%o8) [\alpha = 0, \alpha = 1]}
\]

that \( \alpha \) can be either 0 or 1. Then, choosing first \( \alpha = 0 \), we find that \( \text{coef}_1 = 0 \) is already satisfied, so we conclude that, with \( \alpha = 0 \), both \( a_0 \) and \( a_1 \) remain undetermined. Finally, to set \( \alpha = 0 \) in Eq. (11.74) and solve for \( a_n, n \geq 2 \), we invoke the MAXIMA statement

\[
\text{(%i9) part( factor( solve( ev( coef[n], alpha=0 ) = 0, a[n] ), 1 ) );}
\]

\[
\text{(%o9) \quad a_n = -\frac{a_{n-2}}{(n-1)n}}
\]

to find a recursion relationship that determines all of the coefficients from the first two. Here, \textbf{factor} replaces \( n^2 - n \) with \( (n-1)n \) and \textbf{part} extracts the one element of the one-element list generated by \textbf{solve}. 

\[
\text{(%i5) for i:0 thru 10 do coef[i] : coeff(eq2, x, i ) ;}
\]

\[
\text{(%o5) done}
\]

\[
\text{(%o6) for i:0 thru 10 do print("coef[",i,"] =", coef[i] )}
\]

\[
\begin{align*}
\text{coef[0]} &= a_0(\alpha - 1) \\
\text{coef[1]} &= a_1\alpha(\alpha+1) \\
\text{coef[2]} &= a_2(\alpha+1)(\alpha+2) + a_0 \\
\text{coef[3]} &= a_3(\alpha+2)(\alpha+3) + a_1 \\
\text{coef[4]} &= a_4(\alpha+3)(\alpha+4) + a_2 \\
\text{coef[5]} &= a_5(\alpha+4)(\alpha+5) + a_3 \\
\text{coef[6]} &= a_6(\alpha+5)(\alpha+6) + a_4 \\
\text{coef[7]} &= a_7(\alpha+6)(\alpha+7) + a_5 \\
\text{coef[8]} &= a_8(\alpha+7)(\alpha+8) + a_6 \\
\text{coef[9]} &= a_7 \\
\text{coef[10]} &= a_8
\end{align*}
\]

\[
\text{(%i7) coef[n] : a[n]*(alpha+n-1)*(alpha+n) + a[n-2];}
\]

\[
\text{(%o7) (n + \alpha - 1)(n + \alpha)a_n + a_{n-2}}
\]

\[
\text{(%i8) alp : solve( %=0, alpha );}
\]

\[
\text{(%o8) [\alpha = 0, \alpha = 1]}
\]

\[
\text{(%i9) part( factor( solve( ev( coef[n], alpha=0 ) = 0, a[n] ), 1 ) );}
\]

\[
\text{(%o9) \quad a_n = -\frac{a_{n-2}}{(n-1)n}}
\]
We, of course, would like to have explicit values for each of the first several coefficients. We proceed as follows

\begin{verbatim}
(%i10) a[n] : -a[n-2]/((n-1)*n)
(%i11) for i:2 thru 10 do a[i] : ev( a[n], alpha=0, n=i )$
(%i12) y : collectterms( ev( y, alpha=0 ), a[0], a[1] );
\end{verbatim}

Bind solution for \(a_n\).
Evaluate coefficients and bind to variables.
Substitute coefficients into original function with \(\alpha = 0\).

\[a_0 \left(\frac{x^8}{40320} - \frac{x^6}{720} + \frac{x^4}{24} + \frac{x^2}{2} + 1\right) + a_1 \left(-\frac{x^7}{5040} + \frac{x^5}{120} - \frac{x^3}{6} + x\right)\]

Recognizing that \(2! = 2, 3! = 6, 4! = 24, 5! = 120, 6! = 720, 7! = 5040,\) and \(8! = 40320,\) we then also recognize that the series in the first parentheses is in fact the beginnings of the series for \(\cos(x)\) and the series in the second parentheses is the beginnings of the series for \(\sin(x)\), so the solution to the original equation is, evidently,

\[y(x) = a_0 \cos x + a_1 \sin x\] \hspace{1cm} (11.75)

We can confirm the series expansions with the MAXIMA statements

\begin{verbatim}
(%i13) taylor( cos(x), [x, 0, 8] );
(%o13) \text{T/} \frac{x^8}{40320} - \frac{x^6}{720} + \frac{x^4}{24} - \frac{x^2}{2} + 1 + \ldots
(%i13) taylor( sin(x), [x, 0, 7] );
(%o13) \text{T/} -\frac{x^7}{5040} + \frac{x^5}{120} - \frac{x^3}{6} + x + \ldots
\end{verbatim}

The solution when \(\alpha = 1,\) which (along with \(\alpha = 0\)) makes \(\text{coef}_0 = 0\) when \(a_0 \neq 0,\) can be found in much the same way. In that case, \(\text{coef}_1 = a_1\alpha(\alpha + 1) = 0\) leads us to conclude that \(a_1 = 0.\) When you pursue this alternative to its conclusion, you will find that you have generated only one series, which will turn out to be the series for \(\sin(x)\). We chose \(\alpha = 0\) first because it leads to the full general solution to the original equation, including the two arbitrary constants we know must exist in the general solution to an ordinary, linear, second order ODE. Choosing \(\alpha = 0\) leads only to half of the general solution.

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 \textit{algorithm}—for calculating \textit{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 \textit{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,\(^{21} \) we can write the approximation

\[
\frac{dx}{dt} = f(x, t) \implies \frac{x(t + \Delta t) - x(t)}{\Delta t} \approx f(x, t) \implies x(t + \Delta t) \approx x(t) + f(x, t) \Delta t \quad (11.77)
\]

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

\[
f_{\text{old}} = f(x_{\text{old}}, t_{\text{old}}) ; \quad x_{\text{new}} = x_{\text{old}} + f_{\text{old}} \Delta t ; \quad t_{\text{new}} = t_{\text{old}} + \Delta t \quad (11.78)
\]

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 \( x(2\Delta t) \) 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.\(^{22} \) As this example shows, we can generate a complete—though approximate—solution from initial knowledge of only

1. The differential equation, which determines the rate of change of the dependent variable from the dependent and independent variables themselves,

2. The initial condition, which starts the process by providing the first row in a table containing \( t, x, \) and \( f \),

3. Specific values for any parameters—here there happen to be none—in the differential equation, and

4. A choice of time step \( \Delta t \).

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

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

\[
A(t + \Delta t) = A(t) + \frac{dA}{dt}(t) \Delta t = A(t) - k_A A(t) \Delta t \quad (11.79)
\]

\[
B(t + \Delta t) = B(t) + \left[ k_A A(t) - k_B B(t) \right] \Delta t \quad (11.80)
\]

\[
C(t + \Delta t) = C(t) + k_B B(t) \Delta t \quad (11.81)
\]

\(^{21}\)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.

\(^{22}\)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.

```
dt ← Δt

told ← 0.0
xold ← x0
loop
    fold ← f(xold, told)
    print, told, xold, fold
exit_loop when done
xnew ← xold + fold*dt

tnew ← told + dt
xold ← xnew
told ← tnew
end_loop
```

For definiteness, we take \( A(0) = 1000.000 \), \( B(0) = C(0) = 0.000 \), and \( k_A = k_B = 0.100 \), and we select a time step of \( Δt = 0.250 \). Equations (11.79), (11.80), and (11.81) at \( t = 0.0 \) then become

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

Continuing the algorithm further by applying Eqs. (11.79), (11.80), and (11.81) for \( t = 0.25 \), we find that

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

Going yet one more step, Eqs. (11.79), (11.80), and (11.81) for \( t = 0.50 \) yield that

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

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.\(^{25}\)

---

23One 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.

24This 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 \( Δt \). We shall return to assess the suitability of this choice in later sections.

25Actually, 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 \( A(t + Δt) + B(t + Δt) + C(t + Δt) = A(t) + B(t) + C(t) \). Solutions generated by Euler’s method applied to this problem automatically satisfy the conservation law, regardless of \( Δt \).
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>1000.00</td>
</tr>
<tr>
<td>0.25</td>
<td>975.000</td>
<td>25.000</td>
<td>0.000</td>
<td>1000.00</td>
</tr>
<tr>
<td>0.50</td>
<td>950.625</td>
<td>48.750</td>
<td>0.625</td>
<td>1000.00</td>
</tr>
<tr>
<td>0.75</td>
<td>926.859</td>
<td>71.297</td>
<td>1.844</td>
<td>1000.00</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

11.6.2 Improved Euler Method

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

1. calculate $f_{\text{old}}$,
2. calculate the predicted values
   $$x_{\text{pred}} = x_{\text{old}} + f_{\text{old}} \Delta t \quad ; \quad t_{\text{new}} = t_{\text{old}} + \Delta t$$
3. calculate $f_{\text{pred}} = f(x_{\text{pred}}, t_{\text{new}})$, and
4. calculate the final (corrected) value
   $$x_{\text{new}} = x_{\text{old}} + \frac{1}{2} \left( f_{\text{old}} + f_{\text{pred}} \right) \Delta t$$

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.\(^{26}\)

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.

---

\(^{26}\)Phrased as we have described it, the improved Euler method entails first calculating a *predicted* value at the end point of an interval and then uses that value to determine a *corrected* final value at that point before going on to the next point. Such methods are often called *predictor-corrector* methods.
Table 11.3: Improved Euler algorithm.

\[\begin{align*}
\text{dt} & \leftarrow \Delta t \\
\text{told} & \leftarrow 0.0 \\
xold & \leftarrow x_0 \\
\text{loop} & \\
\quad \text{fold} & \leftarrow f(xold, told) \\
\quad \text{print, told, xold, fold} & \quad \text{Evaluate } f_{\text{old}}. \\
\quad \exit_{\text{loop when done}} & \quad \text{Display results.} \\
\quad xpred & \leftarrow xold + \text{fold} \times dt \\
\quad tnew & \leftarrow \text{told} + \text{dt} \\
\text{fpred} & \leftarrow f(xpred, tnew) \\
\text{xnew} & \leftarrow xold + \frac{1}{2} (\text{fold} + \text{fpred}) \times dt \\
xold & \leftarrow \text{xnew} \\
told & \leftarrow \text{tnew} \\
\end{align*}\]

end_loop

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>\vdots</td>
<td>\vdots</td>
<td>\vdots</td>
<td>\vdots</td>
<td>\vdots</td>
</tr>
</tbody>
</table>

11.6.3 Runge-Kutta Methods

A popular alternative viewpoint, which leads to the deduction of what are called \textit{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

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

\[
x(t) + f(x, t) \Delta t + \frac{1}{2} \frac{df(x(t), t)}{dt} \Delta t^2 + O(\Delta t^3)
\]

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

\[
x(t) + f(x, t) \Delta t + \frac{1}{2} \left( f(x, t) \frac{\partial f(x(t), t)}{\partial x} + \frac{\partial f(x(t), t)}{\partial t} \right) \Delta t^2 + O(\Delta t^3) \quad (11.87)
\]

Motivated by the improved Euler method on the other hand, we are tempted to suppose that we might build a stepping algorithm by\textsuperscript{27}

1. introducing a judicious time \(t' = t + a \Delta t\), where \(a\), constrained by \(0 \leq a \leq 1\), is yet to be specified,

\textsuperscript{27}Note that the steps laid out here reduce to the improved Euler method if we take \(a = b = 1\) and \(w_1 = w_2 = 1/2\).
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

\[
f(x', t') = f(x + b \Delta t f(x, t), t + a \Delta t) = f(x, t) + \frac{\partial f(x, t)}{\partial x} b \Delta t f(x, t) + \frac{\partial f(x, t)}{\partial t} a \Delta t + O(\Delta t^2)
\]

(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

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

(11.89)

which will match Eq. (11.87) through terms of order \( O(\Delta t^2) \) if we choose

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

(11.90)

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 27—values that reveal that the improved Euler method is a member of this Runge-Kutta family. A second common choice takes \( w_1 = 0, w_2 = 1, \) and \( a = b = \frac{1}{2} \), in which case the stepping equations become

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

(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

---

28 Actually, we choose only \( a \). Then \( b = a, w_2 = 1/2a, \) and \( w_1 = 1 - w_2 \) are fixed.
Table 11.5: A Runge-Kutta Method.

<table>
<thead>
<tr>
<th>Step</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>( dt \leftarrow \Delta t )</td>
<td>Choose time step.</td>
</tr>
<tr>
<td>( t_{old} \leftarrow 0.0 )</td>
<td>Initialize ( t_{old} ).</td>
</tr>
<tr>
<td>( x_{old} \leftarrow x_0 )</td>
<td>Initialize ( x_{old} ).</td>
</tr>
<tr>
<td>loop</td>
<td>Evaluate ( f_{old} ).</td>
</tr>
<tr>
<td>( f_{old} \leftarrow f(x_{old}, t_{old}) )</td>
<td>Display results.</td>
</tr>
<tr>
<td>print, ( t_{old} ), ( x_{old} ), ( f_{old} )</td>
<td>Find change using rate at ( t_{old} ).</td>
</tr>
<tr>
<td>exit_loop when done</td>
<td>Find change using rate at ( t_{new} ).</td>
</tr>
<tr>
<td>( k_1 \leftarrow f_{old} \cdot dt )</td>
<td>Calculate ( t_{new} ).</td>
</tr>
<tr>
<td>( k_2 \leftarrow f(x_{old}+k_1/2, t_{old}+dt/2) \cdot dt )</td>
<td>Calculate ( x_{new} ).</td>
</tr>
<tr>
<td>( t_{new} \leftarrow t_{old} + dt )</td>
<td>Replace old values with new.</td>
</tr>
<tr>
<td>( x_{new} \leftarrow x_{old} + (k_1+k_2)/2 )</td>
<td></td>
</tr>
<tr>
<td>( x_{old} \leftarrow x_{new} )</td>
<td></td>
</tr>
<tr>
<td>( t_{old} \leftarrow t_{new} )</td>
<td></td>
</tr>
<tr>
<td>end_loop</td>
<td></td>
</tr>
</tbody>
</table>

elaborate expression, expand it to include higher-order terms in \( \Delta t \), and insist on agreement with a higher-order Taylor expansion deduced from the differential equation. The calculational labor becomes increasingly complicated. Partly because of its popularity, we present without derivation the essence of a fourth-order Runge-Kutta algorithm, limiting ourselves only to the steps that would replace the ones calculating \( k_1 \), \( k_2 \), and \( x_{new} \) in the above algorithm:

\[
\begin{align*}
  k_1 & \leftarrow f(x_{old}, t_{old}) \cdot dt \\
  k_2 & \leftarrow f(x_{old}+k_1/2, t_{old}+dt/2) \cdot dt \\
  k_3 & \leftarrow f(x_{old}+k_2/2, t_{old}+dt/2) \cdot dt \\
  k_4 & \leftarrow f(x_{old}+k_3, t_{old}+dt) \cdot dt \\
  x_{new} & \leftarrow x_{old} + (k_1+2k_2+2k_3+k_4)/6
\end{align*}
\]

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 given time step. Equivalently, a lower-order method requires a smaller time step to give the same degree of accuracy as a higher-order method.

### 11.6.4 Assessing Accuracy

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

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

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

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

(11.92)

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

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

(11.93)

or that

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

(11.94)

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)

$$= x_{\text{RK2}}(t + \Delta t) + O(\Delta t^4)$$

(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 $N O(\Delta t^p)$ when the error per step is of order $O(\Delta t^p)$. For a fixed interval, $N$ is itself of order $1/\Delta t$. Thus, we infer that the global truncation error is of order $O(\Delta t^{p-1})$. This result does not help us much in determining the global error. It does, however, support the conclusion that, for the four methods in the previous paragraph, halving the step size will reduce the global error by a factor of two, four, four, and sixteen, respectively. In particular, halving the step size with the fourth-order Runge-Kutta method will add at least one more decimal digit to the accuracy of the solution overall (and may well do much better).

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

### 11.6.5 Adaptive Methods

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

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

---

33 Sometimes the assessment of accuracy will be based not on the generated solution itself but on some property, e.g., the energy, computed from that solution.

34 In contrast, Euler’s method, for example, would require using throughout a time step small enough to generate
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

$$x_{i+1} = x_i + \frac{\Delta t}{24} \left( 55f_i - 59f_{i-1} + 37f_{i-2} - 9f_{i-3} \right) \quad (11.99)$$

(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.

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

$$x_{i+1} = x_i + \frac{\Delta t}{24} \left( 9f_{i+1} + 19f_i - 5f_{i-1} + f_{i-2} \right) \quad (11.100)$$

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 $\texttt{\$HEAD/python}$, where (as defined in the Local Guide) $\texttt{\$HEAD}$ 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 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. SOLVING ODES NUMERICALLY WITH PYTHON

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 $0 \leq t \leq 50.0$ 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

```python
import numpy as np

# Set initial values.
t = [0.0]; A = [1000.0]
B = [0.0]; C = [0.0]
kA = 0.1; kB = 0.1
dt = 0.25

# Set parameters.

# Set time step.

The solution is then calculated, time instant by time instant, with the loop

```python
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 $t$.

```python
# Display the resulting solution.

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

```python
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()
```

```python
```
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>531.44</td>
<td>354.29</td>
<td>114.26</td>
<td></td>
</tr>
<tr>
<td>0.500</td>
<td>540.36</td>
<td>341.28</td>
<td>118.36</td>
<td></td>
</tr>
<tr>
<td>0.250</td>
<td>544.64</td>
<td>335.16</td>
<td>120.19</td>
<td></td>
</tr>
<tr>
<td>0.125</td>
<td>546.74</td>
<td>332.20</td>
<td>121.06</td>
<td></td>
</tr>
<tr>
<td>0.0625</td>
<td>547.78</td>
<td>330.73</td>
<td>121.49</td>
<td></td>
</tr>
</tbody>
</table>

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

---

36For simplicity, we have calculated each new time by adding the time step to the previous time. If we were more concerned about minimizing roundoff error, we would be better off replacing the last statement in the loop with the statement $t[i] = (i-1) \times \Delta t$, though doing so would increase the execution time. (Multiplication takes more time than addition.)
Figure 11.11: Solution of chain radioactive decay via Euler’s method. The solid, dotted, and dashed lines show the solution for $\Delta t = 0.0625$, 0.5, and 2.0, respectively. While reducing $\Delta t$ from 2.0 to 0.5 clearly makes a difference on the scale of the graph, the further reduction from 0.5 to 0.0625 is hardly noticeable over the entire range of the independent variable.

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>

A similar demonstration using the improved Euler method is left to an exercise. In particular, the values presented in Table 11.7 reflect the result of that exercise when the procedures leading to Table 11.6 are repeated with the improved Euler method. Clearly, the results in Table 11.7 are converging more quickly on stable values to more decimal places than we observed in Table 11.6. Remember that, with Euler’s method, halving $\Delta t$ reduces the global truncation error by a factor of two while, with the improved Euler method, halving $\Delta t$ reduces the global truncation error by a factor of four. Said another way, we would have to reduce $\Delta t$ by a factor of ten to gain one decimal digit in a solution by Euler’s method; we would have to reduce $\Delta t$ by only a factor of $\sqrt{10} = 3.2$ to gain a decimal digit in a solution by the improved Euler method. The values in these two tables reflect this difference.
11.10.2 The PYTHON Command odeint

The simpler of PYTHON’s commands for solving ODEs is \texttt{scipy.integrate.odeint},\textsuperscript{37} which is patterned after the \texttt{lsoda} component in a large FORTRAN solver (the Livermore Solver for Ordinary Differential Equations), written by Alan Hindmarsh and first appearing in 1980. The simplest statement invoking \texttt{odeint} has the general form\textsuperscript{38}

\[
soln = \texttt{odeint}( \texttt{FunctionName}, \texttt{InitialConds}, \texttt{IndepVar} )
\]

where

- \texttt{soln}—any legal variable name will do—is the name of the (scalar or list) variable in which the solution is returned.
- \texttt{FunctionName} is the name of the function defining the equations to be solved; see Section 11.10.3 for the structure of this definition.
- \texttt{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 \texttt{odeint}.
- \texttt{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 \texttt{odeint}.

The procedure is adaptive, which means that, in generating the solution, \texttt{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 \texttt{IndVar}, but, behind the scenes, \texttt{odeint} will be taking steps of various sizes depending on how rapidly the solution changes and the tolerances—default or otherwise—specified when \texttt{odeint} is invoked. Because the varying step sizes may not generate solutions at the user-specified values of the independent variable, \texttt{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 \texttt{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, \texttt{odeint} admits the keyword \texttt{args} that provides for setting parameters that \texttt{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 \texttt{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.\textsuperscript{39}

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\textsuperscript{40} defining a function whose execution by PYTHON returns

\textsuperscript{37}The commands \texttt{scipy.integrate.ode} and \texttt{scipy.integrate.solve_ivp} are a bit more versatile. See Section 11.10.11 for details on these alternate integrators.

\textsuperscript{38}We assume that the statement \texttt{from scipy.integrate import odeint} has previously been executed.

\textsuperscript{39}The URL \texttt{docs.scipy.org/doc/scipy/reference/generated/scipy.integrate.odeint.html} links to a very detailed discussion of \texttt{odeint}.

\textsuperscript{40}See Section 5.6.1.
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.
- `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, `decay`, then the file storing the function will be named `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 `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 `n`—and the dependent variables. Since we have three dependent variables (the number of atoms of species `A`, `B`, and `C`), `n` will have three elements, which PYTHON refers to as `n[0]`, `n[1]`, and `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 ↦ n[0]`, `B ↦ n[1]`, and `C ↦ n[2]`. Rewriting the differential equations Eq. (11.19) in terms of the elements of `n`, we have the system of equations

\[
\frac{d}{dt} n[0] = -k_A n[0] \quad ; \quad \frac{d}{dt} n[1] = k_A n[0] - k_B n[1] \quad ; \quad \frac{d}{dt} n[2] = k_B n[1]
\]  

(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
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 decay, stipulates that the values of the dependent variables will be supplied in the list n, and stipulates that the value of the the independent variable will be supplied in the scalar t.\(^{42}\) Respectively, the two executable lines define the values of the two parameters and assign to the value of the function a 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 decay.py in the default directory or in a directory in PYTHON’s search path,\(^{43}\) we would produce a solution for the differential equations it defines with the statements

\[
\text{import numpy as np}
\]
\[
\text{import matplotlib.pyplot as plt}
\]
\[
\text{from scipy.integrate import odeint}
\]
\[
\text{execfile( ‘decay.py’ )}
\]

or
\[
\text{exec(open( ‘decay.py’ ).read() )}
\]

\[
\text{t = np.linspace(0.0, 50.0, 401)}
\]
\[
\text{ic = [1000.0, 0.0, 0.0 ]}
\]
\[
\text{n = odeint( decay, ic, t )}
\]

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, it is 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., kA and kB in the example of this section. Such explorations would be much easier if the parameters could be set before odeint is called and be known inside the function that odeint

\(^{42}\)For this first example, we hard code the values of the parameters kA and kB in the function definition. Later in this section we will explain how to set these parameters in other ways.

\(^{43}\)See Sections 5.16.1 and 5.16.2 for information about the default directory and the search path.
calls to determine the derivatives. Conveniently, `odeint` provides the keyword `args` for just this purpose. To modify the above example, we would start by editing the function `decay` to be

```python
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:

```python
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) )
```

---

44We 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.
We have invoked the keyword \texttt{args}—a tuple—to \texttt{odeint} to provide the parameters that \texttt{odeint} simply passes on to the function it calls (here, \texttt{decaymod.py}). To explore other values for the parameters, we simply reset the value of $k_A$ and $k_B$ and re-execute the following statements in the sequence immediately above.

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

$$\frac{d}{dt} y[0] = y[1] \quad \frac{d}{dt} y[1] = -y[0] - \beta y[1]$$

(11.102)

Then, using the symbol $\beta$ for $\beta$, we create the py-file
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. bb is a
    # dimensionless parameter and has a value of bb=b/sqrt(m*k), where b
    # is the damping, m is the mass and k is the spring constant.
    # bb will be passed to damposc through odeint with the keyword args.
    derivs = [ 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

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, ) )

bb=0.25
y2 = odeint( damposc, ic, t1, args=(bb, ) )

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

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.

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

\[
\frac{d\pi}{dt} = \upsilon_x ; \quad \frac{d\upsilon_x}{dt} = -\frac{\pi}{(x^2 + y^2)^{b/2}} ; \quad \frac{d\upsilon_y}{dt} = \upsilon_y ; \quad \frac{d\upsilon_y}{dt} = -\frac{\upsilon_y}{(x^2 + y^2)^{b/2}}
\]  

(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 $\pi$, $\upsilon_x$, $\upsilon_y$, and $\upsilon_y$ into correspondence with the elements $x[0]$, $x[1]$, $x[2]$, and $x[3]$ of the four-element vector $x$. Then, one possibility for the 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.
11.10. SOLVING ODES NUMERICALLY WITH PYTHON

Table 11.8: The PYTHON function 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
```

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
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
Figure 11.14: Circular and elliptical orbits for a planet orbiting a sun. The orbits are shown in the \( xy \) plane and in two different phase planes.

\[
E_1 = 0.5 \times (x_1[1] x_1[3] + x_1[3] x_1[1]) - 1.0/\sqrt{x_1[0]^2 + x_1[2]^2}
\]

\[
L_1 = x_1[0] x_1[3] - x_1[2] x_1[1]
\]

\[
E_2 = 0.5 \times (x_2[1] x_2[3] + x_2[3] x_2[1]) - 1.0/\sqrt{x_2[0]^2 + x_2[2]^2}
\]

\[
L_2 = x_2[0] x_2[3] - x_2[2] x_2[1]
\]

will calculate these quantities. Then, statements like

\[
\text{print}(E1); \text{print}(L1); \text{print}(E2), \text{print}(L2)
\]
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 inssofar 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

```python
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 \( (y_0 = 0) \), the
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

\[
\frac{df}{dx} = g \quad ; \quad \frac{dg}{dx} = -k^2 f \quad ; \quad f(0) = 0 \quad ; \quad f(\ell) = 0
\]  

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
Figure 11.15: Open orbits of a satellite moving under the influence of a sun at the origin.

```python
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)
```
11.10. SOLVING ODES NUMERICALLY WITH PYTHON

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.

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

\[
\begin{align*}
\text{k} &= 2.0 \\
\text{f} &= \text{odeint( stdwaves, ic, x, args=(k,))} \\
\text{plt.plot( x, f[:,0], linewidth=2.0, color='black', linestyle=':' )} \\
\text{k} &= 3.0 \\
\text{f} &= \text{odeint( stdwaves, ic, x, args=(k,))} \\
\text{plt.plot( x, f[:,0], linewidth=2.0, color='black', linestyle='--' )} \\
\text{k} &= 4.0 \\
\text{f} &= \text{odeint( stdwaves, ic, x, args=(k,))}; \\
\text{plt.plot( x, f[:,0], linewidth=2.0, color='black' )} \\
\text{plt.grid()} \\
\end{align*}
\]

All four tentative solutions are displayed in Fig. 11.16. Evidently, somewhere between \( k = 3.0 \) and \( k = 4.0 \), the solution will assume the proper value at \( x = 1 \), and we are now in a position via manual trial and error to seek more refined estimates of that specific value of \( k \).

Having now concluded that we should be adjusting \( k \) as we seek acceptable solutions and having recognized that the value of the solution at \( x = 1 \) is critical, we imagine that a graph showing \( f(1) \)
as a function of $k$ might help us in deciding where we should look in an effort to find additional solutions. Suppose we were to seek solutions for values of $k$ ranging from 0 to 20 (say) in steps of 0.2. We might then solve the problem for each $k$ but save only the value $f(1)$ for each solution. To generate that information, we would have to create a loop that migrated through the values of $k$, calculated the solution at each value, and saved the value $f(1)$. We might use the statements

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

Here, we

- Establish in $k$ values that will be passed to $k$, one at a time.
- 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 $\text{soln}[i]$ 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 tractible 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\footnote{We here use the symbol $y$ for what we earlier called $x$.} $0 \leq y < \infty$. To set up the problem, we must first regard the second-order equation as

\begin{verbatim}
We here use the symbol $y$ for what we earlier called $x$.
\end{verbatim}
a pair of first-order equations in which we associate \( \psi[0] \) with \( \psi \) and \( \psi[1] \) with \( d\psi/dy \). The equations we must solve then become

\[
\frac{d}{dy} \psi[0] = \psi[1] \quad ; \quad \frac{d}{dy} \psi[1] = -(2\epsilon - y^2)\psi[0]
\]

(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

```python
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] )
```
def qmshm( psi, y, ee ):
    # QMESH: returns the derivatives for the quantum harmonic oscillator
    # QMESH defines the dimensionless Schroedinger equation for the quantum
    # simple harmonic oscillator, which is
    # 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

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

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, -1.6, '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

$$e = [ 1.0, 1.5, 2.0, 2.5, 3.0, 3.5 ]$$

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.text( 3.0, 1.2, '3.0', fontsize=16 )
plt.text( 2.5, 1.7, '3.5', fontsize=16 )
```

to place the labels appropriately, produces the graph shown in Fig. 11.19. From this graph, we conclude that $\epsilon = 2.5$ yields another solution that goes to zero at infinity, and we argue that another allowed energy corresponds to the value $\epsilon = 2.5$. We might even be tempted to speculate—correctly as it turns out—that additional allowed energies for even states will correspond to the values $\epsilon = 4.5, 6.5, 8.5, \ldots$.

We turn next to the odd states, for which we set $\psi(0) = 0.0$ and $d\psi(0)/dy = 1.0$ before exploring solutions for various values of $\epsilon$. Indeed, the previous paragraphs suggest that we might expect to find the lowest odd state corresponding to $\epsilon = 1.5$. To test that expectation, we set the initial conditions and energies with the statements
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.

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, 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 $E = \epsilon \hbar \omega$, we infer that the energies of the quantum harmonic oscillator are given by

$$E_n = \left(n + \frac{1}{2}\right) \hbar \omega \quad (11.106)$$

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 \texttt{odeint} are not appropriate to the task at hand, these keywords can be given different values. The available keywords with the default values are

- \texttt{atol} and \texttt{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 lists, in which case each component specifies the tolerances for the corresponding dependent variable. Both default to $1.49012 \times 10^{-8}$.

- \texttt{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.

- \texttt{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 \( e/\text{ewt} \leq 1.0 \) 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

\[ n = \text{odeint}(\text{decay}, \text{ic}, \text{t}, \text{atol}=5.0, \text{rtol}=0.1) \]

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.\(^{47}\)

### 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\(^{48}\)

\[ \text{print(open('decay.py').read())} \]

or by printing the entire file to PYTHON or with whatever command to your operating system lists the file on the screen.\(^{49}\) 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.s

\(^{47}\)See the URL [docs.scipy.org/doc/scipy/reference/generated/scipy.integrate.odeint.html](https://docs.scipy.org/doc/scipy/reference/generated/scipy.integrate.odeint.html).

\(^{48}\)Specification of a full path will be necessary if the file is not in a directory in the default directory of PYTHON’s search path.

\(^{49}\)In UNIX and windows, one possible command would be `more $HEAD/python/decay.py`; see the Local Guide.
• **apollo.py**: describes a satellite in the gravitational attraction of two fixed suns of equal mass.

• **damp_harm.py**: describes the one-dimensional damped, unforced harmonic oscillator using dimensional variables.

• **damp_osc.py**: describes the one-dimensional, damped, unforced harmonic oscillator using dimensionless variables.

• **decay.py**: describes three-species radioactive decay.

• **decay_mod.py**: describes three-species radioactive decay using global variables to communicate parameters to the function.

• **drvn_osc.py**: describes the one-dimensional, damped, forced harmonic oscillator using dimensional variables.

• **drvn_pend.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.

• **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.

• **qmsshm.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**,\(^{50}\) 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 (\( \text{IndVar}, \text{DepVars} \)) from the order (\( \text{DepVars}, \text{IndVar} \)) proper for **odeint**. Consequently functions used with **odeint** must be recast to work with **ode**.

\(^{50}\)See [docs.scipy.org/doc/scipy/reference/generated/scipy.integrate.ode.html](http://docs.scipy.org/doc/scipy/reference/generated/scipy.integrate.ode.html) for details.
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`\textsuperscript{51} 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.

### 11.11 Solving ODEs Numerically with MAXIMA

Numerical approaches to solving ODEs most commonly expect that the equation(s) to be solved will be presented as a system of first-order equations of the form\textsuperscript{52}

$$\frac{dx_i}{dt} = f(x_1, x_2, \ldots, x_n, t), \quad 1 \leq i \leq n$$

(11.107)

The one command available in standard MAXIMA for solving ordinary differential equations numerically is `rk`, which uses the fourth-order Runge-Kutta method and has the general syntax

\[
rk(Deriv(s), DepVar(s), InitVal(s), [ IndepVar, Start, Stop, Incr ]);
\]

where `Deriv(s)` is the derivative of the single dependent variable or a list of such derivatives, `DepVar(s)` is the dependent variable or a list of those variables, `InitVal(s)` is the initial condition or a list of those conditions, and—in a list with four components—`IndepVar` is the (single) independent variable, `Start` and `Stop` are the starting and stopping values of the independent variable, and `Incr` specifies the increment by which the independent variable is to be stepped from `Start` to `Stop`. In this case, both the differential equations and the initial conditions must be specified; MAXIMA will not introduce undetermined integration constants. Further, any parameters in the equations must, of course, be given explicit numerical values before `rk` is executed. When invoked, `rk` returns a solution expressed as a list of lists, with each of the interior lists being a list whose first element is the value of the independent variable and whose remaining elements are the solution(s) to the differential equation(s) in the order in which the dependent variable(s) is (are) presented in the second argument of `rk`. MAXIMA’s `part` command can be used to extract the solution at a single value of the independent variable.

Thus, for example, to address the problem of chain radioactive decay that we have already solved symbolically in Section 11.3.4, we would first define the equations and initial conditions as lists with the statements

\[
(\text{i1}) \quad \text{eqs} : \quad [-kA*A, kA*A -kB*B, kB*B]
\]

\[
(\text{i2}) \quad \text{ics} : \quad [1000, 0, 0]
\]

Then we would give explicit numerical values to the parameters with the statements

\textsuperscript{51}See docs.scipy.org/doc/scipy/reference/generated/scipy.integrate.solve_ivp.html for details.

\textsuperscript{52}Higher-order equations must be recast as a system of first-order equations.
Next, we would invoke the solver with the statement

```
%i5) soln : rk( eqs, [A,B,C], ics, [t, 0, 50, 1 ] )$
```

Here, we have suppressed the output because it is a bit of a mess, consisting of a list of 51 four-component sublists with most components being double-precision values. Each of the four-component sublists has the structure \([t, a(t), b(t), c(t)]\). To illustrate more fully, we extract the first, second, and last four-component sublists with the statements

```
%i6) part( soln, 1 );

%i6) [0.0, 1000.0, 0.0, 0.0]

%i7) part( soln, 2 );

%i7) [1.0, 904.8375, 90.48333333333332, 4.679166666666667]

%i8) part( soln, 51 );

%i8) [50.0, 6.737977516754986, 33.68973244593903, 959.5722900373056]
```

A graphical display of this solution is most useful. To exploit `plot2d`, however, we need separate lists of the pairs `AA : [t, A(t)]`, `BB : [t, B(t)]`, and `CC : [t, C(t)]`, which we extract from `soln` with the statements

```
%i9) AA : makelist( [p[1],p[2] ], p, soln );

%i9) [ [0.0, 1000.0], [1.0, 904.83...], ..., [50.0, 6.73...]]

%i10) BB : makelist( [p[1],p[3] ], p, soln );

%i10) [ [0.0, 0.0], [1.0, 90.48...], ..., [50.0, 33.68...]]

%i11) CC : makelist( [p[1],p[4] ], p, soln );

%i11) [ [0.0, 0.0], [1.0, 4.67...], ..., [50.0, 959.57...]]
```

Here, `p` is a dummy variable (and can be anything). `makelist` has three arguments, and the first statement, for example, assigns to the variable `AA` a list of two-component (sub)lists, each of which contains the first and second components of the corresponding four-component (sub)list in `soln`. With these variables in hand, we then create the desired graph with the statement

```
plot2d( [ [discrete,AA], [discrete,BB], [discrete,CC] ], [legend, false],
        [color, black], [xlabel, "t"], [ylabel, "A, B, C"],
        [style, [lines, 4]] )$
```

where the plot option `discrete` informs `plot2d` that data points rather than a function is submitted as the argument. The resulting graph is similar to Fig. 11.9.

There remains only the question of accuracy. The MAXIMA manuals say only that `rk` uses a fourth-order Runge-Kutta method and do not identify option variables that might give the user control over that accuracy. Comparison of the results generated by MAXIMA in the above example with those generated for the same problem by other methods of known accuracy, however, supports the conclusion that the results obtained in the above example are characterized by an absolute accuracy of no less than \(\pm 1.0\) and perhaps of \(\pm 0.1\).

\footnote{See Section 11.6.3 for a description of the method and a brief discussion of its accuracy.}
CHAPTER 11. SOLVING ORDINARY DIFFERENTIAL EQUATIONS

11.16 Exercises

11.16.1 ... using Symbolic Methods

11.1. Find the motion of a driven, damped oscillator satisfying the differential equation

\[ m \frac{d^2 x}{dt^2} + b \frac{dx}{dt} + kx = mf \cos \omega t \]

subject to the general initial conditions

\[ x(0) = x_0 ; \quad \frac{dx}{dt}(0) = v_0 \]

and then determine the particular initial values that might be imposed so that the transient part of the solution is wiped out from the beginning, i.e., so that the motion is identically the steady-state motion from the moment the oscillator is set into motion. Assume that the oscillator is underdamped. (Note that, so that \( m \) will ultimately appear only in conjunction with \( b \) and \( k \), we have chosen to define \( f \) as a force per unit mass rather than simply a force.)

11.2. Take away the walls and the two springs connecting the blocks to the walls in the system of Fig. 11.5, let the masses be different (\( m_1 \) and \( m_2 \), say) and denote the constant of the one spring by \( k \). Suppose the blocks are constrained to move along a straight line. Measuring from an arbitrarily selected origin on that line, let the coordinates of the particles be \( x_1 \) and \( x_2 \), respectively. The equations of motion for this system are

\[ m_1 \frac{d^2 x_1}{dt^2} = k(x_2 - x_1) ; \quad m_2 \frac{d^2 x_2}{dt^2} = -k(x_2 - x_1) \]

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

\[ X(t) = \frac{m_1 x_1(t) + m_2 x_2(t)}{m_1 + m_2} \quad \text{and} \quad Y(t) = x_2(t) - x_1(t) \]

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

\[ \frac{As + B s^3}{\alpha s^4 + \beta s^2 + \gamma} = s \left( \frac{A + Bs^2}{\alpha s^2 + \beta s^2 + \gamma} \right) \]

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
\[
\frac{s}{\alpha s^4 + \beta s^2 + \gamma} = \frac{R}{s^2 + R'} + \frac{Q}{s^2 + Q'}
\]
whose inverse Laplace transform is easier to find. (Indeed, even Table 1.2 is adequate to the task.) Finding the constants \( R, A \) and \( R', Q' \) 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
\[
\begin{align*}
\frac{d^2 x_1}{dt^2} &= -kx_1 + k(x_2 - x_1) \\
\frac{d^2 x_2}{dt^2} &= -k(x_2 - x_1) + k(x_3 - x_2) \\
\frac{d^2 x_3}{dt^2} &= -k(x_3 - x_2) - kx_3
\end{align*}
\]

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
\[
\begin{align*}
\frac{d^2 \theta}{dt^2} + \frac{m_2}{m_1 + m_2} \frac{d^2 \phi}{dt^2} + \frac{g}{l} \theta &= 0 \\
\frac{d^2 \phi}{dt^2} + \frac{d^2 \theta}{dt^2} + \frac{g}{l} \phi &= 0
\end{align*}
\]
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.
CHAPTER 11. SOLVING ORDINARY DIFFERENTIAL EQUATIONS

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
\[ m \frac{d^2 r}{dt^2} = \frac{qQ}{4\pi\varepsilon_0} \frac{r}{r^3} \]
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—not dimensional and dimensionless—of the planetary problem in Section 11.1.7 from Cartesian coordinates \((x, y)\) to polar coordinates \((r, \phi)\), where \( x = r \cos \phi \) and \( y = r \sin \phi \). That is

(a) Show that the polar components of the equations of motion in the first instance are
\[ m \left[ \frac{d^2 r}{dt^2} - r \left( \frac{d\phi}{dt} \right)^2 \right] = f(r) ; \quad m \left( \frac{d^2 \phi}{dt^2} + 2 \frac{d\phi}{dt} \frac{dr}{dt} \right) = 0 \]
(b) Recognizing that
\[ \frac{d}{dt} \left( mr^2 \frac{d\phi}{dt} \right) = m \left( r^2 \frac{d^2 \phi}{dt^2} + 2r \frac{dr}{dt} \frac{d\phi}{dt} \right) \]
show that
\[ m r^2 \frac{d\phi}{dt} = \text{constant} = L \quad \Rightarrow \quad \frac{d\phi}{dt} = \frac{L}{m r^2} \]
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 \( \frac{d\phi}{dt} = \frac{L}{m r^2} \)—see Chapter 13—to find \( \phi(t) \).

(c) Translate the initial conditions to polar coordinates, finding that
\[
\begin{align*}
r(0) &= \sqrt{x_0^2 + y_0^2} \\
\phi(0) &= \arctan \frac{y_0}{x_0} \\
dr dt(0) &= \frac{x_0 v_{x0} + y_0 v_{y0}}{\sqrt{x_0^2 + y_0^2}} \\
d\phi dt(0) &= \frac{x_0 v_{y0} - y_0 v_{x0}}{x_0^2 + y_0^2}
\end{align*}
\]

(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
\[
\begin{align*}
d\frac{\tau}{dt^2} &= -\frac{1}{r^2} + \frac{\beta}{r^3} \\
\frac{d\phi}{dt} &= \pm \frac{\sqrt{\beta}}{r^2}
\end{align*}
\]
\[
\tau(0) = \frac{r_0}{\ell} = \frac{\sqrt{x_0^2 + y_0^2}}{\ell} \\
\phi(0) = \arctan \frac{y_0}{x_0}
\]
\[
\begin{align*}
\frac{d\tau}{dt}(0) &= \frac{x_0 v_{x0} + y_0 v_{y0}}{\sqrt{x_0^2 + y_0^2} \sqrt{GM/\ell}} \\
\frac{d\phi}{dt}(0) &= \frac{x_0 v_{y0} - y_0 v_{x0}}{(x_0^2 + y_0^2)^{3/2} \sqrt{GM/\ell}}
\end{align*}
\]
where \( \beta = \frac{L^2}{(GMm^2)\ell} \) is a constant, and—in the second equation—the upper sign applies when \( L > 0 \) and the lower sign when \( L < 0 \). While all parameters disappeared from the equations in Cartesian coordinates, the parameter \( \beta \) remains in the equations in polar coordinates.

(f) Recast the Cartesian expressions in the last paragraph of Section 11.1.7 for both the dimensional and the dimensionless statements of conservation of energy and angular momentum in the planetary problem into polar coordinates.

**Hint:** Remember (or take as given if you haven’t met them yet) that the radial and azimuthal components of the acceleration of a particle in polar coordinates are given by \( a_r = \ddot{r} - r\dot{\phi}^2 \) and \( a_\phi = r\ddot{\phi} + 2\dot{r}\dot{\phi} \).

### 11.16.2 . . . using Numerical Methods

#### 11.12. In an appropriate dimensionless presentation, standing waves in a string must satisfy the boundary value problem
\[
\frac{d^2 y}{dx^2} + k^2 y = 0 \quad ; \quad y(0) = y(1) = 0
\]
Suppose that the interval \( 0 \leq x \leq 1 \) is divided into \( n \) equal segments of length \( \Delta x = 1/n \), let \( x_i = i \Delta x \) (with \( i = 0, 1, 2, \ldots, n \)), and let \( y_i = y(x_i) \). Evaluate the ODE at \( x = x_i \), approximate the second derivative with the difference formula
\[
\left. \frac{d^2 y}{dx^2} \right|_{x=x_i} \approx \frac{y_{i+1} - 2y_i + y_{i-1}}{\Delta x^2} \quad ; \quad i = 1, 2, 3, \ldots n - 1
\]
and note that \( y_0 = y_n = 0 \). Show that the values \( y_i \) for \( i = 0, 1, 2, \ldots, n \) satisfy a system of \( n + 1 \) linear algebraic equations of the form \( MY = \alpha Y \), where \( Y \) is an \( (n + 1) \)-component vector whose elements are the values of \( y_i \) and \( \alpha \) is determined from \( k^2 \) and \( \Delta x \). Then argue that the allowed values of \( k^2 \) can be determined from the eigenvalues of the matrix \( M \). That is, show that this transformation turns a boundary value problem involving a differential equation into an approximately equivalent matrix eigenvalue problem.

11.14. Some radioisotopes exhibit a branched decay sequence such as

\[
\begin{array}{c}
A \rightarrow B \\
k_1 \\
C \rightarrow D \\
k_2 \\
k_3 \\
k_4
\end{array}
\]

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 \( \mathbf{\bar{p}}(0) = 1 \) and \( \mathbf{\bar{v}}(0) = 0 \), produce graphs of \( \mathbf{\bar{p}} \) versus \( \mathbf{\bar{v}} \) for several values of \( \beta \) on the interval \( 0.0 < \beta < 2.0 \). Then using the knowledge that \( \beta = \frac{b}{m \omega} = \frac{b}{\sqrt{mk}} \), and that \( \Omega = \omega t \), 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 \( (\beta = 2.0) \) and overdamping \( (\beta > 2.0) \).

11.17. Explore the behavior of the Van der Pol oscillator described in dimensionless form by the equation

\[
\frac{d^2 \mathbf{\bar{p}}}{dt^2} = \frac{d \mathbf{\bar{p}}}{dt} (1 - \mathbf{\bar{p}}^2) - \mathbf{\bar{p}}
\]

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 \( t \), \( \theta/dt \) versus \( t \), and \( \theta/dt \) 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 \( \bar{t} = \sqrt{g/l} t \) so that the equation becomes \( d^2 \theta/d\bar{t}^2 + \sin \theta = 0 \).

11.19. Suppose that the “gravitational” force were not inverse square but instead depended on some other (negative) power of the radial coordinate. The dimensionless equations of motion then would be

\[
\frac{d^2 \mathbf{\bar{p}}}{dt^2} = -\frac{\mathbf{\bar{p}}}{(\mathbf{\bar{p}}^2 + \mathbf{\bar{y}}^2)^{b/2}}; \quad \frac{d^2 \mathbf{\bar{y}}}{dt^2} = -\frac{\mathbf{\bar{y}}}{(\mathbf{\bar{p}}^2 + \mathbf{\bar{y}}^2)^{b/2}}
\]

[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
11.16. EXERCISES

\[ b = 1.45, \quad b = 1.55, \] or anything else you can think of, and write a paragraph or two describing the nature of the changes in some detail. **Make sure your solutions are generated to an adequate accuracy to support your conclusions.**

11.20. Explore the scattering orbits that occur when an object moves under the action of a repulsive inverse square force, and compare your results with those for an attractive force.

11.21. Deduce the equations of motion for a space ship of mass \( m \) coasting freely in the \( xy \) plane under the gravitational influence of two suns, each of mass \( M \) and located respectively at \((R, 0)\) and \((-R, 0)\). Then express the equations in dimensionless form and, creating all necessary files, thoroughly explore the motion of this space ship. In particular, you might search for an orbit that 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 \( E = E_x \hat{i} \) and \( B = B_z \hat{k} \). In vector form, the equation of motion for this particle is

\[
m \frac{d^2 \mathbf{r}}{dt^2} = q \mathbf{E} + q \frac{d \mathbf{r}}{dt} \times \mathbf{B}
\]

Verify the equations of motion

\[
m \frac{d^2 x}{dt^2} = qE_x + qB_z \frac{dy}{dt} ; \quad m \frac{d^2 y}{dt^2} = -qB_z \frac{dx}{dt} ; \quad m \frac{d^2 z}{dt^2} = 0
\]

for the specific fields of this exercise, express them in dimensionless form (note that \( \omega = qB_z/m \) is a frequency and \( E_x/B_z \) is a velocity), and thoroughly explore the behavior of the particle in this situation. Try to understand the motion intuitively. **Hint:** You should find that, in terms of an arbitrarily selected unit of length \( \ell \), the equations involve a single parameter \( qE_x/(m\omega^2\ell) \), which can alternatively be written as \( (E_x/B_z)/(\omega\ell) \) — the ratio of the velocity \( E_x/B_z \) determined by the fields to the characteristic velocity implied by your choice of a length unit and the frequency \( \omega \). Note that this exercise actually has more than one parameter, since the initial components of the velocity—probably expressed in units of \( \omega\ell \)—also influence the solution.

11.23. A particle having mass \( m \) and carrying charge \( q \) moves in the \( xy \)-plane while experiencing an electric field given by \( \mathbf{E}(x, y) = \alpha y \hat{j} \), where \( \alpha \) is a constant. Assume that \( \alpha \) and \( q \) are both positive. With \( \mathbf{r} = x \hat{i} + y \hat{j} \), the vector equation of motion for this particle then is

\[
m \frac{d^2 \mathbf{r}}{dt^2} = -q\alpha y \hat{j}
\]

(a) Show that, in component form, the equations of motion for this particle are

\[
\frac{d^2 x}{dt^2} = 0 ; \quad \frac{d^2 y}{dt^2} = -\frac{\alpha q}{m} y = -by
\]

(b) With \( b \) a global parameter, create an appropriate file to define these equations for the ODE-solver in an available numerical/graphical tool. (c) Use that tool to obtain graphs of the trajectories in the \( xy \)-plane of several particles projected from the origin at different angles and with different speeds. (d) Speculate on a use for this field.

11.24. An important system in the early study of chaos is described by the Lorenz equations

\[
\begin{align*}
\frac{dx}{dt} &= a(y - x) \\
\frac{dy}{dt} &= -xz + bx - y \\
\frac{dz}{dt} &= xy - cz
\end{align*}
\]

Create an appropriate file defining these equations and then thoroughly explore the behavior of this system. Graphs of \( y \) versus \( x \), \( z \) versus \( x \), and \( z \) versus \( y \) when \( a = 10.0, \ b = 28.0 \) and \( c = 8.0/3.0 \)
under the initial conditions \( x_0 = 1.0, \ y_0 = 0.0, \) and \( z_0 = 0.0 \) are particularly interesting. While
graphs of \( y \) versus \( x \), \( z \) versus \( x \), and \( z \) versus \( y \) are interesting, the true beauty of the trajectory is
best seen using a three-dimensional space curve. Be sure to examine the path from several different
vantage points in the space around the path, an objective most easily accomplished if the display
of the path allows rotation of the path on the screen.

11.27. The dynamics of the chemical reaction

\[
A + B \rightleftharpoons C + D
\]

is governed by the equations

\[
\begin{align*}
\frac{dA}{dt} &= -k_f AB + k_r CD \\
\frac{dB}{dt} &= -k_f AB + k_r CD \\
\frac{dC}{dt} &= k_f AB - k_r CD \\
\frac{dD}{dt} &= k_f AB - k_r CD
\end{align*}
\]

where \( A(t), \ B(t), \ C(t), \) and \( D(t) \) are the concentrations of each molecule in the reaction vessel,
and \( k_f \) and \( k_r \) are the forward and reverse rate constants, respectively. Suppose that the reaction
is started with \( A(0) = A_0, \ B(0) = B_0, \) and \( C(0) = D(0) = 0. \) Cast the equations in dimensionless
form, using \( A_0 \) as the unit of concentration and \( k_f A_0 t \) as the dimensionless time. Then explore
the behavior of the system as a function of the initial concentration of \( B, \) measured in units of
\( A_0 \) and the reverse rate constant, measured in units of \( k_f. \) Look particularly at the dependence
of the ultimate equilibrium on these parameters. Make sure your results are generated to adequate
accuracy.

11.28. In classical ecology, the interaction between a predator and a prey, with populations \( x(t) \) and \( y(t), \)
respectively, is modeled with the equations

\[
\frac{dx}{dt} = -k_1 x + k_2 xy ; \quad \frac{dy}{dt} = k_3 y - k_4 xy
\]

where \( k_1 \) and \( k_3 \) are parameters describing the way each population would evolve in the absence
of the other and \( k_2 \) and \( k_4 \) are parameters describing strength of the interaction between the two
species, which we take to be proportional to the likelihood of an encounter between a member of one
species and a member of the other species. Depending on the parameters and the initial populations
\( x(0) = x_0, \ y(0) = y_0, \) the system may approach a stable equilibrium or, alternatively, one or the
other of the populations may become extinct. Explore this system to determine conditions under
which each of these circumstances occurs and write a paragraph or two describing your findings.
Make sure your results are generated to an accuracy adequate to support your conclusions.

11.29. Examine the undamped, unforced harmonic oscillator carefully, using at least two different methods
and several time steps. Monitor the accuracy of your solution by monitoring the constancy of the
energy of the oscillator given (in dimensional form) by \( E = \frac{1}{2} mv^2 + \frac{1}{2} kx^2. \)

11.30. In the text, we illustrated the use of one or more numeric processing programs to solve the chain
radioactive decay problem of Section 11.1.2 using Euler’s method. Using an available numeric
processing program like IDL, MATLAB, OCTAVE, or PYTHON, repeat that development using
the improved Euler method and comment on what the resulting graphs reveal about the adequacy
of the several time steps used.

11.31. The type of child’s swing shown in Fig. 11.22 is hung with elastic ropes. Suppose the ropes are long
enough so that the child/swing can be represented by a point mass at the end of a spring, assume
the spring obeys Hooke’s law with constant \( k \) and has an unstretched length \( a, \) and let the motion
of the mass be confined to a single vertical plane. Show that, in the coordinate system illustrated,
the equations of motion are

\[
\begin{align*}
m \frac{d^2 x}{dt^2} &= -kx + \frac{kax}{\sqrt{x^2 + y^2}} \quad ; \quad m \frac{d^2 y}{dt^2} = mg - ky + \frac{kay}{\sqrt{x^2 + y^2}}
\end{align*}
\]
Then, introducing \( \omega_0^2 = k/m, \ T = \omega_0 t, \ \tau = x/a, \) and \( \eta = y/a, \) cast the equations in dimensionless form. After the suggested rescalings, only one parameter—\( g/a\omega_0^2 \)—remains and this parameter is the square of the ratio of the frequency \( \sqrt{g/a} \) of a simple pendulum of length \( a \)—call it the "swing" frequency—to the frequency \( \sqrt{k/m} \) of an object of a mass \( m \) bobbing on a spring of stiffness \( k \)—call it the "bounce" frequency. Using an available numeric processing program like IDL, MATLAB, OCTAVE, or PYTHON, explore the motions for several values of this one parameter, including values larger than, equal to, and smaller than 1. Write several paragraphs describing and presenting evidence for your discoveries.
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

\[
\frac{dp}{dt} = f \quad \text{and} \quad \frac{dx}{dt} = v
\]

(13.1)

which are to be solved subject to the prescribed initial conditions \( x(0) = x_0, v(0) = v_0 \), and \( p(0) = p_0 \). Here, \( x, v, \) and \( p \) are the position, velocity, and momentum of the particle, respectively, and, with \( c \) standing for the speed of light, the relationship between \( p \) and \( v \) assumes one of the forms

\[
p = \frac{mv}{\sqrt{1 - v^2/c^2}} \quad \text{or} \quad p = mv
\]

(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
\[
\frac{d^2 x}{dt^2} = \frac{dv}{dx} \quad \Rightarrow \quad m \frac{dv}{dx} = f(x) \quad \Rightarrow \quad m v \, dv = f(x) \, dx
\]  
(13.4)
Then, Newton’s second law becomes
\[
m \frac{d^2 x}{dt^2} = f(x) \quad \Rightarrow \quad m v \frac{dv}{dx} = f(x) \quad \Rightarrow \quad m v \, dv = f(x) \, dx
\]  
(13.5)
Finally, by integrating this last expression from initial values to general values at some other time, we find that
\[
m \int_{v_0}^v v' \, dv' = \frac{1}{2} m v^2 - \frac{1}{2} m v_0^2 = \int_{x_0}^x f(x') \, dx'
\]  
(13.6)
and the task of finding the velocity (as a function of \( x \)) is reduced to the straight-forward evaluation of an integral. Once that integral has been evaluated, we can then predict the position by exploiting the relationship
\[
\frac{dx}{dt} = v(x) \quad \Rightarrow \quad dt = \frac{dx}{v(x)} \quad \Rightarrow \quad \int_{t_0}^t dt' = t - t_0 = \int_{x_0}^x \frac{dx'}{v(x')}
\]  
(13.7)
and we have reduced the second step in the solution to the evaluation of an integral as well. Both integrals deduced in this paragraph may have internal parameters, so they might fall into either the second or the third of our three categories.

13.1.2 Center of Mass

The location \( \mathbf{r}_{\text{cm}} \) of the center of mass of an object is given by
\[
\mathbf{r}_{\text{cm}} = \frac{1}{M} \int \mathbf{r} \, dm
\]  
(13.8)
where \( \mathbf{r} \) locates a representative element of the object, \( dm \) is the mass of that element, \( M = \int dm \) is the total mass of the object, and the integral extends over the region of space (volume, surface, or line) occupied by the object. More specifically, if the object lies in a plane and polar coordinates \((r, \phi)\) are appropriate, we would write this integral more explicitly in the form
\[
\mathbf{r}_{\text{cm}} = \frac{1}{M} \int \sigma(r, \phi) \mathbf{r} \, r \, dr \, d\phi = \frac{1}{M} \int \sigma(r, \phi) \left( r \cos \phi \, \mathbf{i} + r \sin \phi \, \mathbf{j} \right) \, r \, dr \, d\phi
\]  
(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

\[
\mathbf{r}_{cm} = \frac{1}{M} \int_0^a \int_0^{\pi} \frac{M}{2\pi a^2} \left( r \cos \phi \ \hat{i} + r \sin \phi \ \hat{j} \right) r \, dr \, d\phi = \frac{2}{\pi a^2} \int_0^a \int_0^{\pi} r^2 \left( \cos \phi \ \hat{i} + \sin \phi \ \hat{j} \right) \, dr \, d\phi
\]

\[ (13.10) \]

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

\[
\frac{r_{cm}}{a} = \frac{2}{\pi} \int_0^1 \int_0^{\pi} \lambda^2 \left( \cos \phi \ \hat{i} + \sin \phi \ \hat{j} \right) \, d\phi \, d\lambda
\]

\[ (13.11) \]

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

\[
I = \int r^2 \, dm
\]

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

\[
dm = \sigma \, dA = \frac{4M}{\pi a^2} \sqrt{a^2 - y^2} \, dy
\]

(13.15)

Finally, the moment of inertia of the entire object is given by the integral

\[
I = \frac{4M}{\pi a^2} \int_0^a y^2 \sqrt{a^2 - y^2} \, dy = \frac{4}{\pi} \frac{M a^2}{2} \int_0^1 \lambda^2 \sqrt{1 - \lambda^2} \, d\lambda
\]

(13.16)

where we have introduced the dimensionless variable \(\lambda = y/a\) and expressed the moment of inertia in units of \(Ma^2\), which is the moment of inertia of a point mass \(M\) a distance \(a\) from the axis. This integral is simply a number, and it therefore falls into the first of our three categories.

13.1.4 The Large Amplitude Simple Pendulum

Suppose we seek the period \(T\) of a simple pendulum of length \(l\) and mass \(m\) as shown in Fig. 13.2, but we do not wish to make the conventional small amplitude approximation, under which the period \(T_0\) is given by \(2\pi \sqrt{l/g}\), where \(g\) is the acceleration of gravity. With \(y\) standing for the vertical coordinate of the pendulum (measured upward from its point of support), \(I = ml^2\) for its moment of inertia about the point of support, and \(\omega\) for its angular velocity, we start by noting that the total energy (kinetic plus potential) of the pendulum in a general position is given by

\[
\text{energy} = \frac{1}{2} I \omega^2 + mgy = \frac{1}{2} ml^2 \left( \frac{d\theta}{dt} \right)^2 - mgl \cos \theta
\]

(13.17)

If, with \(\theta_0\) symbolizing the amplitude of the motion, we invoke conservation of energy, equating the energy at a general point to the energy at the highest point in the swing (where \(d\theta/dt = 0\) and \(\theta = \theta_0\)), we find that

\[
\frac{1}{2} ml^2 \left( \frac{d\theta}{dt} \right)^2 - mgl \cos \theta = -mgl \cos \theta_0 \quad \Rightarrow \quad \frac{d\theta}{dt} = \pm \sqrt{\frac{2g}{l}} \sqrt{\cos \theta - \cos \theta_0}
\]

(13.18)

(We assume that the pendulum does not have sufficient energy to swing over the top.) This last relationship then leads to the conclusion that

\[
dt = \pm \sqrt{\frac{l}{2g}} \frac{d\theta}{\sqrt{\cos \theta - \cos \theta_0}} \quad \Rightarrow \quad \int_0^{T/4} dt = \frac{T}{4} = \sqrt{\frac{l}{2g}} \int_0^{\theta_0} \frac{d\theta}{\sqrt{\cos \theta - \cos \theta_0}}
\]

(13.19)
13.1. SAMPLE PROBLEMS

where we have integrated over one-quarter of the period \((0 < t < T/4)\) in time and over one-quarter of the swing \((0 < \theta < \theta_0)\) in angle, and we have taken the positive sign because, in its motion over this interval, the pendulum indeed has positive angular velocity. Further, we have assumed from symmetry that the full period \(T\) is four times the time required for the pendulum to swing from its lowest point to its highest point.

This integral can be recast in numerous ways. Anticipating an existing standard function, we invoke the half angle identity \(\cos \theta = 1 - 2 \sin^2(\theta/2)\), finding that

\[
T = 2 \sqrt{\frac{l}{g}} \int_0^{\theta_0} \frac{d\theta}{\sqrt{\sin^2(\theta_0/2) - \sin^2(\theta/2)}}
\] (13.20)

We then introduce the variable \(\phi\) defined by \(\sin(\theta/2) = \sin(\theta_0/2) \sin \phi\) to find that

\[
T(k) = 4 \sqrt{\frac{l}{g}} \int_0^{\pi/2} \frac{d\phi}{\sqrt{1 - k^2 \sin^2 \phi}}
\] (13.21)

where the (dimensionless) parameter \(k = \sin(\theta_0/2)\) is determined by the amplitude of the pendulum’s motion. By writing \(T(k)\), we have drawn attention in the notation to the dependence of the period on \(k\) (and hence on the amplitude). Note also that, when the amplitude is small, \(k \approx 0\) and the integral can be readily evaluated to yield that \(T_0 = 2\pi \sqrt{l/g}\), 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

\[
\frac{T(k)}{T_0} = 2 \pi \int_0^{\pi/2} \frac{d\phi}{\sqrt{1 - k^2 \sin^2 \phi}} = 2 \pi K(k)
\] (13.22)

Here, for purposes of notation alone, we have introduced the integral

\[
K(k) = \int_0^{\pi/2} \frac{d\phi}{\sqrt{1 - k^2 \sin^2 \phi}}
\] (13.23)

which defines a standard, tabulated function known as the complete elliptic integral of the first kind. Note that the value of this integral depends on a parameter not in a limit but in the very structure of the integrand. This integral falls into the third of our three categories.

13.1.5 Statistical Data Analysis

When repeated measurements of a single quantity are subject to a large number of individually small, random fluctuations, the distribution of those measurements about their mean follows the normal or Gaussian distribution function given by

\[
G(x) = \frac{1}{\sqrt{2\pi}\sigma^2} e^{-(x-\mu)^2/(2\sigma^2)}
\] (13.24)

where \(\mu\) and \(\sigma\) are the mean and standard deviation of the distribution, respectively, and \(G(x)\) is normalized so that \(\int_{-\infty}^{\infty} G(x) \, dx = 1\). The probability that a single measurement will lie between \(\mu - \alpha \sigma\) and \(\mu + \alpha \sigma\) is then given by

\[
P(\mu - \alpha \sigma < x < \mu + \alpha \sigma) = \int_{\mu - \alpha \sigma}^{\mu + \alpha \sigma} G(x) \, dx = \frac{2}{\sqrt{\pi}} \int_0^{\alpha/\sqrt{2}} e^{-s^2} \, ds = \text{erf} \left( \frac{\alpha}{\sqrt{2}} \right)
\] (13.25)

where \(s = (x - \mu)/(\sigma \sqrt{2})\) and, for purposes of notation alone, we have introduced the integral

\[
\text{erf}(x) = \frac{2}{\sqrt{\pi}} \int_0^x e^{-s^2} \, ds
\] (13.26)
which defines a standard, tabulated function known as the error function. Thus, determining the probability given by Eq. (13.25) boils down to evaluating an integral that depends on a parameter appearing in the upper limit, i.e., to evaluating an integral that falls into the second of our three categories.

13.1.6 The Cornu Spiral

The two integrals
\[ C(u) = \int_0^u \cos \left( \frac{\pi t^2}{2} \right) \, dt \quad ; \quad S(u) = \int_0^u \sin \left( \frac{\pi t^2}{2} \right) \, dt \] (13.27)
which are called the Fresnel integrals, appear in the study of Fresnel diffraction. Together, they define the Cornu spiral, which is a graph of \( S(u) \) versus \( C(u) \). Each is a function of its upper limit as a parameter and falls into the second of our three categories.

13.1.7 Electric and Magnetic Fields and Potentials

Among the richest sources of important—and frequently analytically intractable—integrals are the relationships in electromagnetic theory that determine fields and potentials from prescribed sources. For a distribution of static charge, for example, we identify an element of that charge \( dq' \) 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
\[ E(r) = \frac{1}{4\pi\varepsilon_0} \int \frac{r - r'}{|r - r'|^3} \, dq' \quad \text{and} \quad V(r) = \frac{1}{4\pi\varepsilon_0} \int \frac{dq'}{|r - r'|} \] (13.28)
which extend over all charges in the source. Similarly, for a source consisting of a steady current \( I' \) in a wire, we identify an element \( dr' \) of the wire located at \( r' \) and, in mks units, find the magnetic field \( B(r) \) and the magnetic vector potential \( A(r) \) at the point \( r \) by evaluating the integrals
\[ B(r) = \frac{\mu_0}{4\pi} \int \frac{I' \, dr' \times (r - r')}{|r - r'|^3} \quad \text{and} \quad A(r) = \frac{\mu_0}{4\pi} \int \frac{I' \, dr'}{|r - r'|} \] (13.29)
which extend over the path followed by the current.

More specifically, suppose we seek the electrostatic potential in the plane midway between two identical uniformly charged circular rings of radius \( a \) with their planes parallel, their axes coincident, and their centers separated by \( 2a \). The envisioned situation is shown in Fig. 13.3. Each ring carries a total charge \( Q \) with (linear) charge density \( \lambda \). Using cylindrical coordinates \((r, \phi, z)\), we first find the potential at the general point \( r = r \cos \phi \hat{i} + r \sin \phi \hat{j} + z \hat{k} \), produced by one such ring positioned in the \( xy \) plane with its center at the origin. Let \( r' = a \cos \phi' \hat{i} + a \sin \phi' \hat{j} \) locate an element on that (single) ring. In this notation,
\[ r - r' = (a \cos \phi - a \cos \phi') \hat{i} + (a \sin \phi - a \sin \phi') \hat{j} + z \hat{k} \] (13.30)
so
\[ |r - r'| = \left[ (r \cos \phi - a \cos \phi')^2 + (r \sin \phi - a \sin \phi')^2 + z^2 \right]^{1/2} = \left[ r^2 + a^2 - 2ar \cos(\phi' - \phi) + z^2 \right]^{1/2} \] (13.31)
Placing these results into the second member of Eq. (13.28) and recognizing that \( dq' = \lambda a \, d\phi' \), we at last find that
\[ V_{\text{one}}(r, \phi, z) = \frac{1}{4\pi\varepsilon_0} \int_0^{2\pi} \frac{\lambda a \, d\phi'}{r^2 + a^2 - 2ar \cos(\phi' - \phi) + z^2} \] (13.32)
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

\[
V_{\text{one}}(r, \phi, z) = \frac{\lambda a}{4\pi\epsilon_0} \int_0^{2\pi} \frac{d\alpha}{[r^2 + a^2 - 2ar\cos\alpha + z^2]^{1/2}} \tag{13.33}
\]

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 \( 2a \). The observation point in the midplane is thus a distance \( a \) “above” one of the rings and the same distance \( a \) “below” the other. We find the potential produced by these two rings by adding a contribution from each ring, concluding that

\[
V_{\text{midplane}}(r) = V_{\text{one}}(r, \phi, -a) + V_{\text{one}}(r, \phi, a) = \frac{2\lambda a}{4\pi\epsilon_0} \int_0^{2\pi} \frac{d\alpha}{[r^2 + 2a^2 - 2ar\cos\alpha]^{1/2}} \tag{13.34}
\]

Finally, introducing the variable \( s \) defined by \( r = sa \) to express the radial coordinate in dimensionless terms, we find the expression

\[
V_{\text{two}}(s) = V_{\text{midplane}}(sa) = \frac{2\lambda a}{4\pi\epsilon_0 a} \int_0^{2\pi} \frac{d\alpha}{[s^2 - 2s\cos\alpha + 2]^{1/2}} \tag{13.35}
\]

or, even better, the expression

\[
V(s) = \frac{\sqrt{2}}{2\pi} \int_0^{2\pi} \frac{d\alpha}{[s^2 - 2s\cos\alpha + 2]^{1/2}} \tag{13.36}
\]

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 \( 2a \). This integral falls into the third of our three categories, since its value is a function of the parameter \( s \) in the integrand.
13.1.8 Quantum Probabilities

In the standard interpretation, the wave function \( \psi(x) \) for a one-dimensional quantum system is a probability amplitude, and the quantity \( |\psi(x)|^2 \) is a probability density. Further the wave function by convention is normalized so that \( \int_{-\infty}^{\infty} |\psi(x)|^2 \, dx = 1 \). Thus, the integral

\[
P(x_1 < x < x_2) = \int_{x_1}^{x_2} |\psi(x)|^2 \, dx \tag{13.37}
\]
gives the probability of finding the quantum system with its coordinate somewhere between \( x = x_1 \) and \( x = x_2 \).

More specifically, we remember that the wave function for a quantum harmonic oscillator in its ground state is given by

\[
\psi(x) = \left( \frac{m\omega}{\pi \hbar} \right)^{1/4} e^{-y^2/2} \tag{13.38}
\]
where \( \hbar \) is Planck’s constant, \( m \) is the mass of the oscillator and, with \( k \) the spring constant and \( a = \sqrt{\hbar \omega/k} \), \( \omega = \sqrt{k/m} \) is the frequency of the oscillator; \( y = x/a \) is a dimensionless coordinate.

Since the classical turning point of the oscillator occurs when its energy (\( \frac{1}{2} \hbar \omega \) for the ground state) is equal to the potential energy \( \frac{1}{2} k x_{\text{turn}}^2 \), the turning point of this oscillator is given by

\[
\frac{1}{2} \hbar \omega = \frac{1}{2} k x_{\text{turn}}^2 \implies x_{\text{turn}} = \sqrt{\frac{\hbar \omega}{k}} = a \tag{13.39}
\]
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

\[
f(x) = \sum_{n} a_n \phi_n(x) \tag{13.43}
\]
Though the argument we will here present is not mathematically rigorous, the expansion coefficients $a_n$ can be quickly determined by multiplying Eq. (13.43) by $w(x) \phi_j(x)$, integrating over the interval $a \leq x \leq b$, and exchanging the order of integration and summation to find that

$$\int_a^b w(x) \phi_j(x) f(x) \, dx = \sum_n a_n \int_a^b w(x) \phi_j(x) \phi_n(x) \, dx = \sum_n a_n N_n \delta_{nj} = N_j a_j$$

We conclude that, if we know either of $f(x)$ or $a_n$, we can determine the other, i.e., that

$$f(x) = \sum_n a_n \phi_n(x) \iff a_n = \frac{1}{N_n} \int_a^b w(x) \phi_n(x) f(x) \, dx$$

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

$$\{1, \sin \frac{n\pi x}{l}, \cos \frac{m\pi x}{l}; \ n, m = 1, 2, 3, \ldots\}$$

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

$$\int_{-l}^{l} 1 \times 1 \, dx = 2l \ ; \ \int_{-l}^{l} 1 \times \sin \frac{n\pi x}{l} \, dx = 0 \ ; \ \int_{-l}^{l} 1 \times \cos \frac{n\pi x}{l} \, dx = 0$$

$$\int_{-l}^{l} \sin \frac{n\pi x}{l} \sin \frac{m\pi x}{l} \, dx = l \delta_{nm} \ ; \ \int_{-l}^{l} \sin \frac{n\pi x}{l} \cos \frac{m\pi x}{l} \, dx = 0$$

$$\int_{-l}^{l} \cos \frac{n\pi x}{l} \cos \frac{m\pi x}{l} \, dx = l \delta_{nm} \ ; \ \int_{-l}^{l} \cos \frac{n\pi x}{l} \sin \frac{m\pi x}{l} \, dx = 0$$

The expansion

$$f(x) = \frac{a_0}{2} + \sum_{n=1}^{\infty} \left( a_n \cos \frac{n\pi x}{l} + b_n \sin \frac{n\pi x}{l} \right)$$

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

$$a_n = \frac{1}{l} \int_{-l}^{l} f(x) \cos \frac{n\pi x}{l} \, dx \ ; \ b_n = \frac{1}{l} \int_{-l}^{l} f(x) \sin \frac{n\pi x}{l} \, dx$$

### 13.2 Evaluating Integrals Symbolically with MAXIMA

The MAXIMA command for symbolic integration is `integrate`. It takes either two or four arguments, the first of which supplies the expression to be integrated, the second of which identifies the variable of integration, and—when present—the third and fourth of which specify the lower and upper limits, respectively. Thus, for example, the statement

```maxima
integrate( sin(k*x), x );
```

1To put this set into the form of the previous paragraph, we would identify $\phi_1(x) = 1$, $\phi_2(x) = \sin(\pi x/l)$, $\phi_3(x) = \cos(\pi x/l)$, $\phi_4(x) = \sin(2\pi x/l)$, etc.

2The first coefficient is written $a_0/2$ rather than $a_0$ to simplify the integrals determining the coefficients. With the choice we have made, the integral for $a_0$ turns out to be obtainable by setting $n = 0$ in the expression for $a_n$; we do not need a third—and different—expression for that one coefficient.

3Remember that, in some versions of MAXIMA, the terminating semicolon is unnecessary.
evaluates the indefinite integral \( \int \sin(kx) \, dx \) while the statement

\[
\text{integrate( sin(k*x), x, 0, %pi );}
\]

evaluates the definite integral \( \int_{0}^{\pi} \sin(kx) \, dx \). If MAXIMA 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. MAXIMA also evaluates improper integrals, recognizing the symbols \( \text{inf} \) for \( +\infty \) and \( \text{minf} \) 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.

In this section, we illustrate the use of the command \text{integrate} to evaluate some of the integrals deduced in Section 13.1. Beyond integration per se, we also show how, in many cases, MAXIMA can be used to set up the integral as well. To abbreviate the presentation of MAXIMA dialogs, we make liberal use of terminating dollar signs to suppress intermediate output. The reader is therefore urged to duplicate the dialogs in an actual session with MAXIMA, either replacing the dollar signs with semicolons or eliminating them (depending on the version of MAXIMA in use).

### 13.2.1 Relativistic Motion Under a Constant Force

Suppose the particle to which Eq. (13.3) applies moves relativistically, starting from rest at the origin \( [x(0) = 0, \, v(0) = 0, \, p(0) = 0] \), and experiences a constant force \( f \). Then, the basic relationships from which we would determine \( x(t) \) and \( v(t) \) are

\[
\begin{align*}
p(t) &= \int_{0}^{t} f \, dt', \quad x(t) = \int_{0}^{t} v(t') \, dt', \quad \text{and} \quad p(t) = \frac{mv(t)}{\sqrt{1 - v(t)^2/c^2}} \\
\end{align*}
\]  

(13.52)

The following “conversation” with MAXIMA will find the desired quantities and explore a few of their properties:\footnote{MAXIMA statements are shown on the left; comments describing the statements are shown on the right.}

\[
\begin{align*}
\text{(i1) assume( c>0, f>0, m>0, t>0 )} & \quad \text{Declare } c, f, m \text{ and } t \text{ to be positive.} \\
\text{(i2) p : integrate(f, tp, 0, t);} & \quad \text{Integrate (constant) } f \text{ to find } p. \\
\text{(i3) p = m * v / sqrt(1 - v^2/c^2);} & \quad \text{Relate momentum to velocity.} \\
\text{(i4) f^2 t^2 = m^2 v^2/(1 - v^2/c^2)} & \quad \text{Prepare to solve for } v. \\
\text{(i5) solve( %, v );} & \quad \text{Extract second solution. (Physically, } v \text{ and } f \text{ must have the same sign.)} \\
\text{(i6) v : part( %, 2, 2 );} & \quad \text{Find limit of } v \text{ as } t \to \infty. \\
\text{(i7) limit( v, t, inf );} & \quad \text{Substitute } tp \text{ for } t \text{ in } v. \\
\text{(i8) vp : subst( tp, t, v )$} & \quad \text{Integrate } vp \text{ to find position.} \\
\text{(i9) x : integrate( vp, tp, 0, t );} & \quad \text{Integrate } vp \text{ to find position.} \\
\text{(i9) c f \left( \frac{\sqrt{f^2 t^2 + c^2 m^2}}{f^2} - \frac{cm}{f^2} \right)}
\end{align*}
\]
13.2. EVALUATING INTEGRALS SYMBOLICALLY WITH MAXIMA

(%i10) vclassical : taylor( v, t, 0, 4 );
   Find limit of v for small t.
   \[\frac{ft}{m} - \frac{f^3 t^3}{2c^2 m^3} + \ldots\]

(%i11) xclassical : taylor( x, t, 0, 4 );
   Find limit of x for small t.
   \[\frac{ft^2}{2m} - \frac{f^3 t^4}{8c^2 m^3} + \ldots\]

Reassuringly, the velocity approaches \(c\) as \(t \to \infty\) and, with the acceleration \(a\) identified as \(f/m\), \(v\) and \(x\) for small \(t\) (classical limit) are \(at\) and \(\frac{1}{2}at^2\) respectively.

13.2.2 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

(%i1) rr : [r*cos(phi), r*sin(phi), 0];
   Assign \(r\).

(%i2) sigma : m / (%pi * a^2 / 2);  
   Evaluate \(\sigma\), which is constant.

(%i3) r_cm : integrate( integrate(sigma * rr * r, phi, 0, %pi), r, 0, a )/m;

(%o3) \[0, \frac{4a^3}{3\pi}, 0\]

(%i4) float(%);

(%o4) \[0.0, 0.4244131815783876 a, 0.0\]

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, incidentally, that MAXIMA has at (%i3) effected an item-by-item integration of the elements of a list with a single statement.

Note also that, in line (%i3), we have nested one integrate command within another. 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.

13.2.3 Moment of Inertia; Radius of Gyration

The evaluation by MAXIMA of the moment of inertia of a semicircular plate as given by Eq. (13.16) is quick and straightforward. We need only the statements

(%i11) assume( a>0, m>0 );

(%i12) integrate( lambda^2*sqrt(1-lambda^2), lambda, 0, 1);
   \[\pi/16\]

(%i13) I : m*a^2*(4/%pi)*%;

(%o3) \(ma^2/4\)

(%i14) k : sqrt(I/m);

(%o4) \(a/2\)

to evaluate the necessary integral and find both the moment of inertia \(I = ma^2/4\) and the radius of gyration \(k = a/2\).
13.2.4 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

\[
V(x, y, z) = \frac{\lambda}{4\pi\epsilon_0} \int_{-a}^{+a} \frac{dz'}{|r-r'|} \tag{13.53}
\]

E lecting 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

\[
\begin{align*}
%i1 & \text{ rr : [r*cos(phi), r*sin(phi), z]} \$ \\
%i2 & \text{ rp : [0, 0, zp]} \$ \\
%i3 & \text{ sep : rr - rp} \\
%i4 & \text{ sep * sep} \$ \\
%i5 & \text{ trigsimp(%)$} \\
%i6 & \text{ denom : sqrt(%)$} \\
%i7 & \text{ cnst : 1/(4*pi*epsilon[0])$} \\
%i8 & \text{ integ : lambda * cnst / denom;}
\end{align*}
\]

\[
\lambda = \frac{4\pi\epsilon_0 \sqrt{x'^2 - 2xz' + z^2 + r^2}}
\]

\[
%i9 \text{ assume( r>0, z>0, a>0 )$}
\]

\[
%i10 \text{ v : integrate( integ, zp, -a, a );}
\]

\[
\begin{align*}
%i10 & \text{ lambda$} \\
& \text{ asinh[(z+a)/r] - asinh[(z-a)/r]} \\
& \text{ 4*pi*epsilon[0]}
\end{align*}
\]

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
Evaluate potential in $xy$ plane.

\[ v_0 : v, z = 0; \]

\[ \frac{\lambda}{2\pi\epsilon_0} \text{asinh} \left( \frac{a}{r} \right) \]

\[ \text{Let } a/r \to s. \]

\[ \text{Expand near } s = 0 \ (a/r \to 0). \]

\[ \text{Return to original variables.} \]

\[ \text{Express in terms of total charge.} \]

\[ \frac{q}{4\pi\epsilon_0 r} \]

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 would expand the potential with the statement

\[ \text{Expand potential at } z = 0 \text{ for small } r, \text{ keeping only first non-zero term.} \]

or, equivalently

\[ -\frac{\lambda}{2\pi\epsilon_0} \log \frac{r}{a} + \frac{\lambda \log 2}{2\pi\epsilon_0} \]

This 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.

### 13.2.5 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 $\mathbf{B}(\mathbf{r})$ is given by the Biot-Savart law, which is the first member of Eq. (13.29). If the (single) loop has radius $a$ and lies in the $xy$ plane with its center at the origin, then, for an observation point on the $z$ axis, the various vectors in the expression are given by $\mathbf{r} = sa \mathbf{k}$, where (to facilitate expressing things in dimensionless form) the $z$ coordinate is written as a multiple $s$ of the radius $a$ of the loop, $\mathbf{r'} = a \cos \phi \mathbf{i} + a \sin \phi \mathbf{j}$, and $d\mathbf{r'} = (−a \sin \phi \mathbf{i} + a \cos \phi \mathbf{j}) d\phi$. Evaluation of the integrand and the integral for a single loop then proceeds with the statements

---

\text{The file $\$HEAD/maxima/crossdot.mac$ was described in Section 6.9 and exists in the public directory structure. (The symbol $\$HEAD$ is defined in the Local Guide.) It defines the two functions $\text{cross}(\mathbf{v}_1, \mathbf{v}_2)$ and $\text{dot}(\mathbf{v}_1, \mathbf{v}_2)$ for evaluating the cross and dot products of two three-component vectors.}
Define cross and dot.

Declare a positive.

Assign r.

Assign r′.

Assign dr′.

Evaluate r − r′.

Evaluate |r − r′|².

Recognize cos² φ + sin² φ = 1.

mag2 = |r − r′|².

Recognize cos² φ + sin² φ = 1.

Finally, we bring various factors together to construct the integrand, evaluate the integral, and recast the result in a dimensionless form with the statements

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, objectives accomplished with the statements

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
13.2. EVALUATING INTEGRALS SYMBOLICALLY WITH MAXIMA

Figure 13.5: On-axis magnetic field of a pair of loops. The highest of the three graphs shows the field when the loop separation is \( c = 1.0 \) (separation of the two loops equal to twice the radius of each loop). The middle graph corresponds to \( c = 0.5 \) (the Helmholtz separation) and the lower graph corresponds to \( c = 0.25 \). Here, the default labels on the \( x \) and \( y \) axes stand for \( z/a \) and \( B_{\text{pair}} / B_{\text{pairc}} \), respectively.

\[
\begin{align*}
(\%i20) & \quad c100 : \text{bpair, } c=1.0 \\
(\%i21) & \quad c050 : \text{bpair, } c=0.5 \\
(\%i22) & \quad c025 : \text{bpair, } c=0.25 \\
(\%i23) & \quad \text{plot2d( } [ \text{c100, c050, c025 } ] , \ [s, 0.0, 3.0], \ [y, 0.0, 2.0], \\
& \quad \quad \text{[style, [lines, 4]], [color, black], [legend, false],} \\
& \quad \quad \text{[xlabel, "s = z/a"], [ylabel, "B(z) / B(0)"] } ) ;
\end{align*}
\]

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

\[
(\%i24) \quad \text{taylor(bpair, s, 0, 4);}
\]

\[
(\%o24) \quad \frac{1 + (12c^2 - 3)s^2 + (120c^4 - 180c^2 + 15)s^4}{2c^4 + 4c^2 + 2} + \frac{8c^8 + 32c^6 + 48c^4 + 32c^2 + 8}{8c^8 + 32c^6 + 48c^4 + 32c^2 + 8} + \ldots
\]

Clearly there is a particular value of \( c \) 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

\[
(\%i25) \quad \text{solve( } \frac{1 + (12c^2 - 3)s^2 + (120c^4 - 180c^2 + 15)s^4}{2c^4 + 4c^2 + 2} = 0 , c \text{ );}
\]

\[
(\%o25) \quad [c = 0.5, c = 0.5]
\]

\[
(\%i26) \quad \text{plot2d( } [ \text{solve( } \frac{1 + (12c^2 - 3)s^2 + (120c^4 - 180c^2 + 15)s^4}{2c^4 + 4c^2 + 2} = 0 , c \text{ ), c = 0.5 } ] , \ [s, 0.0, 3.0], \ [y, 0.0, 2.0], \\
& \quad \quad \text{[style, [lines, 4]], [color, black], [legend, false],} \\
& \quad \quad \text{[xlabel, "s = z/a"], [ylabel, "B(z) / B(0)"] } ) ;
\]

\[
(\%i27) \quad \text{plot2d( } [ \text{solve( } \frac{1 + (12c^2 - 3)s^2 + (120c^4 - 180c^2 + 15)s^4}{2c^4 + 4c^2 + 2} = 0 , c \text{ ), c = 0.25 } ] , \ [s, 0.0, 3.0], \ [y, 0.0, 2.0], \\
& \quad \quad \text{[style, [lines, 4]], [color, black], [legend, false],} \\
& \quad \quad \text{[xlabel, "s = z/a"], [ylabel, "B(z) / B(0)"] } ) ;
\]

\[
(\%i28) \quad \text{plot2d( } [ \text{solve( } \frac{1 + (12c^2 - 3)s^2 + (120c^4 - 180c^2 + 15)s^4}{2c^4 + 4c^2 + 2} = 0 , c \text{ ), c = 0.0 } ] , \ [s, 0.0, 3.0], \ [y, 0.0, 2.0], \\
& \quad \quad \text{[style, [lines, 4]], [color, black], [legend, false],} \\
& \quad \quad \text{[xlabel, "s = z/a"], [ylabel, "B(z) / B(0)"] } ) ;
\]

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

\[
(\%i24) \quad \text{taylor(bpair, s, 0, 4);}
\]

\[
(\%o24) \quad \frac{1 + (12c^2 - 3)s^2 + (120c^4 - 180c^2 + 15)s^4}{2c^4 + 4c^2 + 2} + \frac{8c^8 + 32c^6 + 48c^4 + 32c^2 + 8}{8c^8 + 32c^6 + 48c^4 + 32c^2 + 8} + \ldots
\]

Clearly there is a particular value of \( c \) 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
CHAPTER 13. EVALUATING INTEGRALS

\( (\%i25) \) part( %, 2, 1, 1 );
\( (\%o25) \) 12c^2 - 3

Extract first factor of numerator of second term of Taylor series. Note that the numbers in the argument of `part` will depend on how `\%o24` is arranged by your system.

\( (\%i26) \) solve( %, c );
\( (\%o26) \) \([c = -\frac{1}{2}, c = \frac{1}{2}]\)

Find c to make numerator zero.

The two solutions are equivalent. Both reveal that the physical arrangement in which the field has no \( s^2 \) term has one loop positioned one-half of the loop radius above the midplane and the other positioned one-half of the loop radius below the midplane. That is, the loops are separated by their common radius—the Helmholtz separation.

When the loops have that special separation, we determine how the field varies with \( s \) (when \( s \) is small) by using the statements

\( (\%i27) \) \%o24, c = 1/2;
\( (\%o27) \) \(-\frac{144s^4 - 125}{125}\)

\( (\%i28) \) float( expand(%) );
\( (\%o28) \) 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

\( (\%i29) \) float( solve( %o28=0.99, s) );
\( (\%o29) \) \(0.3052367917903921, \ldots\)
\( (\%i30) \) float( solve( %o28=0.95, s ) );
\( (\%o30) \) \(0.4564354645876384, \ldots\)

MAXIMA, of course, returns a list containing all four of the solutions to the quartic equation. We have displayed the only solution that is both real and positive.

Unfortunately, these solutions are only approximate, since we have used a truncated expansion of the field, an expansion that becomes increasingly incorrect as \( s \) increases. To find more accurately how far from the center we can stray before the field has fallen to 99% or 95% of its value at the center, we need return to the exact expression of `bpair` but restrict it to the Helmholtz case by setting \( c = 1/2 \) with the statement

\( (\%i31) \) bhelm : bpair, c = 1/2$

Then, we seek the values of \( s \) at which this expression assumes the value 0.99 (1% fall off) or 0.95 (5% fall off). Equivalently, we seek the roots of the expressions `bhelm - 0.99 = 0` and `bhelm - 0.95 = 0`. The dependence of these expression on \( s \), however, is complicated, and the command `solve` is not up to the task. Thus, we draw (prematurely—see Section 14.11) on MAXIMA’s command `find_root`, which uses a binary search to find a root in a specified range. The above results provide us with sensible initial guesses, so we invoke the statements\(^6\)

\[^6\text{In `find_root`, the first argument is the expression whose root is sought, the second is the variable to be adjusted in finding the root, and the third and fourth provide the range within which a root is sought. By default, `find_root` strives to find the root to absolute and relative tolerances of 0.0 and 0.0.}\]
13.2. EVALUATING INTEGRALS SYMBOLICALLY WITH MAXIMA

(\%i32) \text{find_root(bhelm-0.99, s, 0.3, 0.35);} \quad \text{Find } s \text{ when field is down 1%}.

(\%o32) 0.313746025198168

(\%i33) \text{find_root(bhelm-0.95, s, 0.4, 0.5)} \quad \text{Find } s \text{ when field is down 5%}.

(\%o33) 0.48845550123516077

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

(\%i34) \text{float( ev(bhelm, s=0.5) );}

(\%o34) 0.9458241851693391

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 results at \( \%o32 \)–\( \%o34 \) 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.

13.2.6 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. MAXIMA does, however, make it easy to examine the way in which the period departs from its limiting value as the amplitude moves away from the very small. We simply determine a Taylor expansion of the integrand with the statements

(\%i11) \text{integrand : 1/sqrt(1-k^2*sin(phi)^2);} \\

(\%i1) \frac{1}{\sqrt{1 - k^2 \sin^2 \phi}}

(\%i12) \text{taylor( \%, k, 0, 4 );}

(\%o2) /T/ 1 + \frac{(\sin^2 \phi)k^2}{2} + \frac{(3\sin^4 \phi)k^4}{8} + \ldots

Then we insert the pre-multiplying factor \( 2/\pi \) and integrate the series on \( \phi \) with the statement

(\%i13)\text{expand( 2*integrate( \%, phi, 0, \%pi/2 ) / \%pi );}

(\%o3) 1 + \frac{k^2}{4} + \frac{9k^4}{64} + \ldots

Finally, we substitute \( \sin(\theta_0/2) \) for \( k \) with the statement

(\%i14)\text{subst(sin(theta[0]/2),k,%);} \\

(\%o4) 1 + \frac{1}{4} \sin^2(\theta_0/2) + \frac{9}{64} \sin^4(\theta_0/2) + \ldots

and expand this result as a power series in \( \theta_0 \).
CHAPTER 13. EVALUATING INTEGRALS

(%i5) taylor( %, theta[0], 0, 4 );

(%o5) /T/ \(1 + \frac{1}{16} \theta^2 + \frac{11}{3072} \theta^4 + \ldots\)

Note that \(\theta_0\) must here be expressed in radians.

To see how significantly the period departs from the small-amplitude value when \(\theta_0\) is not quite zero, let us evaluate this expression for \(\theta_0 = 5^\circ, 10^\circ, 15^\circ, \text{ and } 20^\circ\). The MAXIMA statements

(%i6) [ 5, 10, 15, 20 ] $%\text{ Set angles in degrees.}$

(%i7) ampl : (%/180)*%pi $%\text{ Convert angles to radians.}$

(%i8) per1 : float( 1+ampl^2/16 ) $%\text{ Calculate first correction. Values are rounded to six digits after the decimal point.}$

(%o8) [ 1.000476, 1.001904, 1.004284, 1.007615 ]

(%i9) per2 : per1 + float(11*ampl^4/3072) $%\text{ Add second correction.}$

(%o9) [ 1.000476, 1.001907, 1.004301, 1.007669 ]

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 \(2\pi\sqrt{\frac{l}{g}}\).

13.2.7 Fourier Coefficients for Half-Rectified Signal

Suppose we sought to express the half-rectified sine wave

\[ f(x) = \begin{cases} 0 & -l \leq x < 0 \\ \sin \left(\frac{\pi x}{l}\right) & 0 \leq x < l \end{cases} \quad (13.54) \]

in a Fourier series. In accordance with Eq. (13.51), the coefficients would then be given by the integrals

\[ a_0 = \frac{1}{l} \int_{0}^{l} \sin \left(\frac{\pi x}{l}\right) \, dx \quad ; \quad a_n = \frac{1}{l} \int_{0}^{l} \sin \left(\frac{\pi x}{l}\right) \cos \left(\frac{n\pi x}{l}\right) \, dx \quad ; \quad b_n = \frac{1}{l} \int_{0}^{l} \sin \left(\frac{\pi x}{l}\right) \sin \left(\frac{n\pi x}{l}\right) \, dx \quad (13.55) \]

where \(n = 1, 2, 3, \ldots\). We find these coefficients quickly by submitting to MAXIMA the statements

(%i10) declare( n, integer )$

(%i11) assume( l>0 )$

(%i12) integrate(sin( %+pi*x/l), x, 0, l) / l$

(%o12) \frac{2}{\pi}$

(%i13) integrate(sin( %+pi*x/l)*cos(n %+pi*x/l), x, 0, l) / l$

(%o13) 0$

(%i14) integrate(sin( %+pi*x/l)*sin( n %+pi*x/l), x, 0, l) / l$

(%o14) \frac{1}{2}$

We find these coefficients quickly by submitting to MAXIMA the statements

For \(n = 1\), the expression for \(a_1\) in \(%o5\) is indeterminate, and find alternative evaluation.

Find \(a_n, n > 0\).

Find \(a_n, n = 1\).
Here, the first statement limits the nature of \( n \) so that MAXIMA can automatically replace expressions like \( \sin n\pi \) and \( \cos n\pi \) with explicit evaluations, the second statement assures MAXIMA that \( l \) is positive, and the remaining statements evaluate the desired integrals. Thus, we find from the output on lines \((\%o3), (\%o5), (\%o6), (\%o7),\) and \((\%o8)\) that

\[
\begin{align*}
a_0 &= \frac{2}{\pi}; & a_1 &= 0; & a_n &= -\frac{1+(-1)^n}{\pi(n^2-1)}, n > 1; & b_1 &= \frac{1}{2}; & b_n &= 0, n > 1 \\
(13.56)
\end{align*}
\]

With these values, we assemble the Fourier series in accordance with Eq. (13.50) to find that

\[
\begin{align*}
f(x) &= \frac{1}{\pi} - \sum_{n=2}^{\infty} \frac{1}{\pi(n^2-1)} \cos \frac{n\pi x}{l} + \frac{1}{2} \sin \frac{\pi x}{l} \\
&= \frac{1}{\pi} + \frac{1}{2} \sin \frac{\pi x}{l} - \frac{2}{\pi} \sum_{m=1}^{\infty} \frac{1}{4m^2-1} \cos \frac{2m\pi x}{l} \\
&= \frac{1}{\pi} + \frac{1}{2} \sin \frac{\pi x}{l} - \frac{2}{\pi} \left( \frac{1}{3} \cos \frac{2\pi x}{l} + \frac{1}{15} \cos \frac{4\pi x}{l} + \frac{1}{35} \cos \frac{6\pi x}{l} + \cdots \right) \\
(13.57)
\end{align*}
\]

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

\[
f_n(x) = \frac{1}{\pi} + \frac{1}{2} \sin \frac{\pi x}{l} - \frac{2}{\pi} \sum_{m=1}^{m} \frac{1}{4m^2-1} \cos \frac{2m\pi x}{l}
(13.58)
\]

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 \( n \) with the statements listed in Table 13.1. Here, we force \( m \) to be an integer, recast the general coefficient in \((\%o4)\) by replacing \( n \) with \( 2m \), construct a MAXIMA function defining the truncated series, and plot the truncated series for each of four different truncation points. Shown in Fig. 13.6, the resulting graphs clearly reveal the convergence of the sum to the half-rectified wave as more terms are included.

### 13.5 Algorithms for Numerical Integration

Unfortunately, very many integrals of great interest have no closed form, analytic evaluation. To address these integrals, numerical analysts have developed many formulae—often called quadrature formulæ—for numerical integration. In this section, we describe several of these formulæ.

#### 13.5.1 Newton-Cotes Quadrature

One family of quadrature formulæ can be deduced by starting with the recognition that the definite integral

\[
A = \int_{a}^{b} f(x) \, dx
(13.59)
\]

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
Table 13.1: Graphing several Fourier series with \textsc{MAXIMA}.

\begin{verbatim}
(%i9) declare( m, integer )
(%i10) factor( subst( 2*m, n, %o5 ) );
(%i11) a(m) := -2/(%pi*(2*m-1)*(2*m+1))
(%i12) f(x,n) := 1/%pi + \sin(%pi*x)/2 + \sum( a(m)*\cos(2*m*%pi*x), m, 1, n )
(%i13) plot2d( f(x,1), [x, -3.0, 3.0], [y, -0.5, 1.5], [xlabel, "x"], [ylabel, "f(x,1)"], [style, [lines, 4]], [legend,false], [color, black] )
(%i14) plot2d( f(x,2), [x, -3.0, 3.0], [y, -0.5, 1.5], [xlabel, "x"], [ylabel, "f(x,2)"], [style, [lines, 4]], [legend,false], [color, black] )
(%i15) plot2d( f(x,4), [x, -3.0, 3.0], [y, -0.5, 1.5], [xlabel, "x"], [ylabel, "f(x,4)"], [style, [lines, 4]], [legend,false], [color, black] )
(%i16) plot2d( f(x,8), [x, -3.0, 3.0], [y, -0.5, 1.5], [xlabel, "x"], [ylabel, "f(x,8)"], [style, [lines, 4]], [legend,false], [color, black] )
\end{verbatim}

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 $n = 1, 2, 4, \text{ and } 8$, respectively. These graphs were created with \textsc{MAXIMA}.

\begin{center}
\begin{tabular}{ccc}
\includegraphics[width=0.4\textwidth]{figure13.6a} & \includegraphics[width=0.4\textwidth]{figure13.6b} \\
\includegraphics[width=0.4\textwidth]{figure13.6c} & \includegraphics[width=0.4\textwidth]{figure13.6d}
\end{tabular}
\end{center}
Figure 13.7: Division of interval $a < x < b$ into $N$ segments.

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).

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

$$
\int_a^b f(x) \, dx \approx f_0 \Delta x + f_1 \Delta x + \cdots + f_i \Delta x + \cdots + f_{N-1} \Delta x
$$

$$
\approx (f_0 + f_1 + \cdots + f_i + \cdots + f_{N-1}) \Delta x \quad (13.60)
$$

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

$$
\int_a^b f(x) \, dx \approx M_N = \left( f_{1/2} + f_{3/2} + f_{5/2} + \cdots + f_{N-3/2} + f_{N-1/2} \right) \Delta x \quad (13.61)
$$

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,

$$
\int_a^b f(x) \, dx \approx T_N = \frac{1}{2} (f_0 + f_1) \Delta x + \frac{1}{2} (f_1 + f_2) \Delta x + \cdots + \frac{1}{2} (f_{N-1} + f_N) \Delta x
$$

$$
= \left( \frac{1}{2} f_0 + f_1 + f_2 + \cdots + f_i + \cdots + f_{N-1} + \frac{1}{2} f_N \right) \Delta x
$$
which, as with the mid-point rule, is 100% accurate when \( f(x) \) is linear, i.e., when \( f(x) \) is a polynomial of the same order as the one used to approximate the function.

A final (for here) and still better approximation is obtained if we pair the strips—which then requires \( N \) to be even—and approximate the area of each pair by the area under the parabola fitted to the values of \( f(x) \) at the three points defining the pair. The result,

\[
\int_{a}^{b} f(x) \, dx \approx S_N = \frac{1}{3} (f_0 + 4f_1 + 2f_2 + 4f_3 + \cdots + 2f_{N-2} + 4f_{N-1} + f_N) \Delta x
\]  

(13.63)

is called Simpson's rule. (See the first exercise in Section 13.15.2.) For reasons similar to those that apply for the mid-point 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

\[
T_1 = T_{2^0} = \frac{f(a) + f(b)}{2} (b - a)
\]

(13.64)

Then, halving the step size and introducing \( x_1 = a + (b - a)/2 \), the midpoint of the interval (see Fig. 13.9, in which—for present convenience—we label the points differently than we did in Fig. 13.7), we find that

\[
T_2 = T_{2^1} = \left[ \frac{f(a)}{2} + f(x_1) + \frac{f(b)}{2} \right] \frac{(b - a)}{2}
= \frac{1}{2} T_1 + f(x_1) \frac{(b - a)}{2}
\]

(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

\[ x_4 = a + \frac{1}{8}(b-a) \quad x_5 = a + \frac{3}{8}(b-a) \]
\[ x_6 = a + \frac{5}{8}(b-a) \quad x_7 = a + \frac{7}{8}(b-a) \]  \hspace{1cm} (13.67)

and continuing one more step, we find that

\[
T_8 = T_{2^3} = \frac{1}{2} T_4 + \left[ f(x_4) + f(x_5) + f(x_6) + f(x_7) \right] \frac{(b-a)}{8}
\]
\[
= \frac{1}{2} T_4 + \left[ f \left( a + \frac{b-a}{8} \right) + f \left( a + \frac{3(b-a)}{8} \right) + f \left( a + \frac{5(b-a)}{8} \right) \right.
\]
\[
+ f \left( a + \frac{7(b-a)}{8} \right) \right] \frac{(b-a)}{8}  \hspace{1cm} (13.68)
\]

and, in general, with

\[ x_{2N+j} = a + \frac{(2j+1)(b-a)}{2^{N+1}} \; ; \; \; 0 \leq j < 2^{N+1} \]  \hspace{1cm} (13.69)

that

\[
T_{2^{N+1}} = \frac{1}{2} T_{2^N} + \left[ f \left( a + \frac{b-a}{2^{N+1}} \right) + f \left( a + \frac{3(b-a)}{2^{N+1}} \right) + \ldots \right.
\]
\[
+ f \left( a + \frac{(2^{N}-1)(b-a)}{2^{N+1}} \right) \right] \frac{(b-a)}{2^{N+1}}  \hspace{1cm} (13.70)
\]

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

\[
T_4 = \left[ \frac{1}{2} f(a) + f(x_2) + f(x_1) + f(x_3) + \frac{1}{2} f(b) \right] \frac{b-a}{4}
\]
\[
= \left[ f(a) + 2f(x_2) + 2f(x_1) + 4f(x_3) + f(b) \right] \frac{b-a}{8}  \hspace{1cm} (13.71)
\]

and that

\[
T_8 = \left[ \frac{1}{2} f(a) + f(x_4) + f(x_2) + f(x_3) \right.
\]
\[
\left. + f(x_1) + f(x_6) + f(x_3) + f(x_7) + \frac{1}{2} f(b) \right] \frac{b-a}{8}  \hspace{1cm} (13.72)
\]

Note, in particular, the combination

\[
\frac{4T_8 - T_4}{3} = \left[ f(a) + 4f(x_4) + 2f(x_2) + 4f(x_5) + 2f(x_1) \right.
\]
\[
\left. + 4f(x_6) + 2f(x_3) + 4f(x_7) + f(b) \right] \left( \frac{b-a}{8 \times 3} \right)  \hspace{1cm} (13.73)
\]

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

\[
\frac{4T_8 - T_4}{3} = \left[ f(a) + 4f(x_4) + 2f(x_2) + 4f(x_5) + 2f(x_1) \right.
\]
\[
\left. + 4f(x_6) + 2f(x_3) + 4f(x_7) + f(b) \right] \left( \frac{\Delta x}{3} \right)  \hspace{1cm} (13.74)
\]
which—mirabile dictu—we recognize as Simpson's rule for evaluating the integral with 8 divisions of the interval! We conclude that

\[ S_8 = \frac{4T_8 - T_4}{3} \]  

(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 four values \( T_n, T_{2n}, T_{4n}, \) and \( T_{8n} \). We could then use Eq. (13.76) to generate the three values \( S_{2n}, S_{4n}, \) and \( S_{8n} \). In Romberg integration, we next generate a pair of still more accurate values from the formulae

\[ X_{4n} = \frac{(16S_{4n} - S_{2n})}{15} \]  
\[ X_{8n} = \frac{(16S_{8n} - S_{4n})}{15} \]

and then we generate a further improved value from the formula \( Y_{8n} = \frac{(64X_{8n} - X_{4n})}{63} \). This process can, of course, be continued indefinitely—though we rarely have to go even as far as we have described.

### 13.5.3 Assessing Error

Numerical evaluations, of course, only approximate the integral. Two distinctly different sorts of errors can occur. **Truncation errors** arise because the integral has been approximated by a finite sum; **roundoff errors** arise because computers do not store non-integers to 100% precision and, in the evaluation of a sum, the imprecision with which each component is represented within the computer can accumulate as the number of arithmetic operations increases. Truncation errors become smaller as the width of strips is reduced. Roundoff errors, unfortunately, become more significant as the width of strips is reduced (because, with narrower strips, more arithmetic must be done). Usually, roundoff errors are negligible, the more so as the sophistication of the algorithm increases (and, hence, the amount of arithmetic decreases). Provided we do not strive for accuracy greater than about 1 part in \( 10^5 \) or \( 10^6 \) (with single precision floating point arithmetic), we can usually ignore roundoff errors. Thus, provided the function being integrated is such that the algorithm converges fairly rapidly with decreasing strip width, the quickest way to obtain a reasonably reliable estimate of truncation error is to evaluate the integral with two different step widths, the second being half of the first, and compare the two results. Presuming that roundoff error has not begun to be important, we can be confident that the second result is more accurate than the first. Thus, if the two agree to 1 part in \( 10^3 \), say, we can with reasonable confidence, assume that the second value is good to one part in \( 10^3 \). Indeed, the second value is probably better than that, but assessing its accuracy by this method would entail obtaining a third value by using a strip width half of that used to determine the second value.footnote{7} 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).footnote{8}

footnote{7}We shall make this criterion a bit more explicit in the next paragraphs.

footnote{8}We shall see in later sections how we might decide when roundoff has started to be 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 \( \frac{d^2 f}{dx^2} = 0 \) or in the third case that \( \frac{d^4 f}{dx^4} = 0 \) 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.

2. If \( D_{\text{max}}(i) \) is the maximum value of \( \frac{d^i f}{dx^i} \) in the interval \( a < x < b \), then the above expressions support the inequalities

\[ \left| \int_a^b f(x) \, dx - M_N \right| \leq \frac{(b - a)^3}{24N^2} D_{\text{max}}(2) \] (13.80)

\[ \left| \int_a^b f(x) \, dx - T_N \right| \leq \frac{(b - a)^3}{12N^2} D_{\text{max}}(2) \] (13.81)

\[ \left| \int_a^b f(x) \, dx - S_N \right| \leq \frac{(b - a)^5}{180N^4} D_{\text{max}}(4) \] (13.82)

(though we must keep in mind that these approximations will frequently be extremely crude, so these upper bounds may well be very conservative). Provided that problems with computer roundoff do not begin to appear, we conclude from these results that an upper bound on the error in the midpoint and trapezoidal rules falls off like \( 1/N^2 \) while that bound in Simpson’s rule falls off like \( 1/N^4 \). Doubling \( N \) therefore reduces the error in the midpoint and trapezoidal rules by a factor of four, while doubling \( N \) reduces the error in Simpson’s rule by a factor of sixteen. With Simpson’s rule, every doubling of \( N \) should gain at least one more decimal point in accuracy, so the convergence criterion described in the first paragraph in this subsection is particularly apt when Simpson’s rule is used.

13.5.4 Iterative and Adaptive Algorithms

In the previous subsections, we assumed that the user of a particular algorithm would actually view the value obtained for a succession of values of \( N \) and decide personally when to stop by examining the changes that occur as \( N \) is successively doubled. We can, of course, program a computer to make those decisions. One extremely common approach exploits Simpson’s rule (probably via the trapezoidal rule and Romberg integration) to obtain \( S_2, S_4, S_8, \ldots \), compares each new value with its predecessor and stops when the absolute value of the difference is smaller than a tolerance—either absolute or relative—prescribed in advance. As a guard against an infinite loop, these algorithms should also stop if the desired tolerance has not been achieved in some maximum number of refinements and should print a warning when the desired tolerance has not been achieved. This method is said to be *iterative*, because it generates a succession of results, examines each new result in turn, and repeats the process until the new result meets or exceeds the prescribed tolerance. The points at which the function is evaluated, however, are determined ahead of time and are not influenced at all by the nature of the specific integrand to which the algorithm is applied.

Another family of algorithms (which may be iterative or noniterative) aims to minimize computational labor by estimating—though the methods for doing so are often crude—the accuracy obtained with each strip as the evaluation unfolds and shrinking or enlarging that strip to achieve a particular tolerance before going on to the next strip. In these *adaptive* methods, the points at which the function is evaluated are adjusted in response to the particular function being integrated. Because the assessment of accuracy at a particular strip can result either in shrinking or enlarging the width of that—or the next—strip, adaptive methods focus the computational effort in regions where the function varies rapidly and give less attention to regions in which the function varies slowly.

13.5.5 Gaussian Quadrature

The approach of *Gaussian quadrature* to numerical integration is more complicated than the Newton-Cotes approach but significantly better in some respects. In the Gaussian approach, both the points at which the function is to be evaluated and the weights to be applied to each value are adjusted to achieve maximum accuracy when the function is approximated by a polynomial of a given order.

The development of a formula for Gaussian quadrature is simplified if we begin by introducing a set of \( m + 1 \) points \( t_i, (i = 0, 1, 2, \ldots, m) \) that divide the interval \( t_0 = a \leq t \leq b = t_m \) into \( m \) segments, the \( i \)-th of which extends over the interval \( t_{i-1} \leq t \leq t_i \). The values \( t_i \) may—but need not—be equally spaced. In this notation, we write the integral of interest as a sum of integrals over each segment, i.e., we write

\[
\int_a^b g(t) \, dt = \sum_{i=1}^m \int_{t_i}^{t_{i+1}} g(t) \, dt
\]

(13.83)

To facilitate the discussion, however, we rescale and translate the variable in the \( i \)-th segment by introducing the variable \( x \) defined by

\[
x = 2t - (t_{i+1} + t_i) \quad \text{or} \quad t = \frac{t_{i+1} + t_i}{2} + \frac{t_{i+1} - t_i}{2} x = t_i^{\text{mid}} + \frac{\Delta t_i}{2} x
\]

(13.84)

where \( t_i^{\text{mid}} \) is the coordinate at the midpoint of the \( i \)-th segment and \( \Delta t_i \) is the width of the \( i \)-th segment. With this change, \( x \) ranges from \(-1\) to \(+1\) as \( t \) ranges from \( t_i \) to \( t_{i+1} \), so the integrals of interest now assume the form

\[
\int_a^b g(t) \, dt = \sum_{i=1}^m \int_{t_i}^{t_{i+1}} g(t) \, dt = \sum_{i=1}^m \frac{\Delta t_i}{2} \int_{-1}^{1} g \left( t_i^{\text{mid}} + \frac{\Delta t_i}{2} x \right) dx = \sum_{i=1}^m \frac{\Delta t_i}{2} \int_{-1}^{1} f_i(x) \, dx
\]

(13.85)
where \( f_i(x) = g(t_i^{\text{mid}} + \Delta t_i x/2) \). In essence, then, we must evaluate an integral of the form

\[
\int_{-1}^{1} f(x) \, dx
\]

(13.86)

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

\[
\int_{-1}^{1} f(x) \, dx = \sum_{k=1}^{N} w_k f(x_k)
\]

(13.87)

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

\[
\int_{-1}^{1} f(x) \, dx = w_1 f(x_1) + w_2 f(x_2)
\]

(13.88)

We choose \( w_1 \), \( w_2 \), \( x_1 \) and \( x_2 \) so that the formula gives the correct answer for the special cases

\[
f(x) = 1, \text{ yielding that } \int_{-1}^{1} dx = 2 = w_1 + w_2
\]

(13.89)

\[
f(x) = x, \text{ yielding that } \int_{-1}^{1} x \, dx = 0 = w_1 x_1 + w_2 x_2
\]

(13.90)

\[
f(x) = x^2, \text{ yielding that } \int_{-1}^{1} x^2 \, dx = \frac{2}{3} = w_1 x_1^2 + w_2 x_2^2
\]

(13.91)

\[
f(x) = x^3, \text{ yielding that } \int_{-1}^{1} x^3 \, dx = 0 = w_1 x_1^3 + w_2 x_2^3
\]

(13.92)

Because the integral is a linear function of its integrand, a formula that yields the correct answer in these four cases will also yield the correct answer for any linear combination of these special cases, i.e., for any cubic polynomial. These four equations determine the four unknowns. Eqs. (13.90) and (13.92) imply that

\[
w_1 x_1 = -w_2 x_2 \quad \text{and} \quad w_1 x_1^3 = -w_2 x_2^3
\]

(13.93)

which, when we divide the second by the first, yields \( x_1^2 = x_2^2 \), implying that \( x_1 = -x_2 \). (We reject the plus sign so the two values will be distinct.) Then Eqs. (13.91) and (13.89) yield that

\[
\frac{2}{3} = (w_1 + w_2) x_2^2 = 2x_2^2 \quad \implies \quad x_2^2 = \frac{1}{3}
\]

(13.94)

from which we also conclude that \( x_2^2 = 1/3 \). Thus,

\[
x_1 = -\frac{1}{\sqrt{3}} \quad \text{and} \quad x_2 = \frac{1}{\sqrt{3}}
\]

(13.95)

Next, Eq. (13.90) implies that \( w_1 = w_2 \) and then Eq. (13.89) implies that \( w_1 = w_2 = 1 \). We conclude that

\[
\int_{-1}^{1} f(x) \, dx = f \left( -\frac{1}{\sqrt{3}} \right) + f \left( \frac{1}{\sqrt{3}} \right)
\]

(13.96)
The result is exact for \textit{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.\(^{10}\)

Returning to the original function and variables as laid out in Eq. (13.85), we finally find that Eq. (13.96) supports the expression

\[
\int_a^b g(t) \, dt = \sum_{i=1}^{m} \frac{\Delta t_i}{2} \left[ g \left( t_i^{\text{mid}} - \frac{\Delta t_i}{2} \right) + g \left( t_i^{\text{mid}} + \frac{\Delta t_i}{2} \right) \right]
\]

\[
= \sum_{i=1}^{m} \frac{\Delta t_i}{2} \sum_{j=1}^{2} w_j g \left( t_i^{\text{mid}} + \frac{\Delta t_i}{2} x_j \right) \tag{13.97}
\]

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

\[
\int_{-1}^{1} f(x) \, dx = 0.23692689 f(-0.90617985) + 0.47862867 f(-0.53846931)
\]

\[
+ 0.56888889 f(0.00000000) \tag{13.98}
\]

\[
+ 0.47862867 f(0.53846931) + 0.23692689 f(0.90617985)
\]

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

\[
w_1 = 0.23692689 \quad x_1 = -0.90617985 \]

\[
w_2 = 0.47862867 \quad x_2 = -0.53846931 \]

\[
w_3 = 0.56888889 \quad x_3 = +0.0000000 \tag{13.99}
\]

\[
w_4 = 0.47862867 \quad x_4 = +0.53846931 \]

\[
w_5 = 0.23692689 \quad x_5 = +0.90617985 \]

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 \textit{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\(^{11}\)

\[
L_N(x_i) = 0 \quad \text{and} \quad w_i = \frac{2}{(1 - x_i^2)(dL_N(x_i)/dx)^2} \tag{13.101}
\]

and that

\[
\left| \int_{-1}^{1} f(x) \, dx - \sum_{k=1}^{N} w_k f(x_k) \right| = \frac{2^{2N+1}(N!)^4}{(2N+1)(2N)!^3} \left| \frac{d^{2N}f}{dx^{2N}} \right|_{x=\xi} \tag{13.102}
\]

\(^{10}\)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.

where $-1 < \xi < 1$. This result shows that the $N$-point formula of this type will be 100\% accurate for polynomials of degree $2N - 1$ or lower—a property which we have already inferred informally. Because of the role played by the Legendre polynomials in these formulae, they are sometimes referred to as Gauss-Legendre formulae.\(^\text{12}\)

### 13.9 Evaluating Integrals Numerically with PYTHON

**Note:** All PYTHON program (.py) files referred to in this chapter are available in the directory `$\text{HEAD/python}$`, where (as defined in the 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

$$I = \text{erf}(1) = \frac{2}{\sqrt{\pi}} \int_0^1 e^{-x^2} \, dx$$

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\(^\text{13}\)

```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 )
```

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 13.2. 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.

\(^\text{12}\)A more general integral that can be approached with the techniques of this subsection has the form

$$\int_a^b f(x) \, w(x) \, dx,$$

where $w(x)$ is a weighting function. In the case we dealt with, $w(x) = 1$ and, in our rescaling, the interval became the interval from $-1$ to $1$. That the Legendre polynomials $L_n(x)$ are orthogonal on the interval $-1 < x < 1$ with weight $w(x) = 1$ is part of the reason that the roots of these polynomials ultimately emerged as important. For other weight functions and other intervals, a different set of polynomials would have played the role of the Legendre polynomials. Thus, there are several different types of Gaussian quadrature, each specific to a particular weight and basic interval.

\(^\text{13}\)Alternatively, as long as $n \geq 2$, the fifth through eighth lines in this code can be replaced with the single statement

$$y = -0.5*f[0] + \text{sum}(f) - 0.5*f[n].$$
Table 13.2: 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>0.82526295597</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>0.842700726970</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>

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

\[
S_2 = \frac{4 \times T_2 - T_1}{3} = \frac{4 \times 0.82526295597 - 0.77174332258}{3} = 0.843102830043333 (13.104)
\]

and

\[
S_4 = \frac{4 \times T_4 - T_2}{3} = \frac{4 \times 0.838367777441 - 0.82526295597}{3} = 0.842736051389 (13.105)
\]

Continuing this process of Richardson extrapolation, we find the values \( S_8 = 0.84270035846 \), \( S_{16} = 0.842700933572 \), and \( S_{32} = 0.842700801745 \). 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

\[
y = f \cdot \text{transpose}(w).
\]
13.9. EVALUATING INTEGRALS NUMERICALLY WITH PYTHON

\[ n = 2 \]
\[ x = \text{np.linspace}(0.0, 1.0, n+1) \]
\[ f = 2.0*\text{np.exp}(-x**2)/\text{np.sqrt}(\text{np.pi}) \]
\[ w = \text{np.zeros}(n+1) \]
\[ w[0] = 1.0; w[n] = 1.0 \]
\[ \text{for } i \text{ in np.arange}(1,n,2): w[i]=4.0 \]
\[ \text{for } i \text{ in np.arange}(2,n-1,2): w[i]=2.0 \]
\[ y = 0.0; \]
\[ \text{for } i \text{ in np.arange}(0,n): y = y + w[i]*f[i] \]
\[ \text{int} = (1.0/n)*y/3.0 \]
\[ \text{print( int )} \]
\[ 0.843102830043 \]

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 \( g(t) \) with \( 2e^{-x^2}/\sqrt{\pi} \) and we divide the interval \( 0 \leq x \leq 1 \) into \( m = 4 \) segments of equal width, we might invoke the PYTHON statements

\[ \text{pts} = [-1.0/\text{np.sqrt}(3.0), 1.0/\text{np.sqrt}(3.0)] \]
\[ w = [1.0, 1.0] \]
\[ a = 0.0; b = 1.0 \]
\[ \text{m} = 4 \]
\[ x = \text{np.linspace}(a, b, m+1) \]
\[ \text{xmid} = \text{np.zeros}(m) \]
\[ \text{for } i \text{ in np.arange}(0,m): \]
\[ \text{xmid}[i]=(x[i]+x[i+1])/2.0 \]
\[ \text{dx} = (b-a)/m \]
\[ \text{int} = 0.0 \]
\[ \text{for } i \text{ in np.arange}(0,m): \]
\[ \text{fcts} = 0.0 \]
\[ \text{for } j \text{ in [0,1]:} \]
\[ \text{fcts} = \text{fcts} + w[j]*\text{np.exp}(-\text{xmid}[i]+\text{dx}*\text{pts}[j])/2.0)**2 \]
\[ \text{int} = \text{int} + \text{fcts} \]
\[ \text{int} = \text{dx*int/np.sqrt(np.pi)} \]
\[ \text{print( int )} \]
\[ 0.842699298102 \]

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

\[ \text{m} = 5 \]
\[ x = \text{np.linspace}(a, b, m+1) \]
\[ \text{xmid} = \text{np.zeros}(m) \]
\[ \text{for } i \text{ in np.arange}(0,m): \]
\[ \text{xmid}[i]=(x[i]+x[i+1])/2.0 \]
\[ \text{dx} = (b-a)/m \]
\[ \text{int} = 0.0 \]
\[ \text{for } i \text{ in np.arange}(0,m): \]
\[ \text{fcts} = 0.0 \]
\[ \text{for } j \text{ in [0,1]:} \]
\[ \text{fcts} = \text{fcts} + w[j]*\text{np.exp}(-\text{xmid}[i]+\text{dx}*\text{pts}[j])/2.0)**2 \]
\[ \text{int} = \text{int} + \text{fcts} \]
\[ \text{int} = \text{dx*int/np.sqrt(np.pi)} \]
\[ \text{print( int )} \]
\[ 0.842699298102 \]
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

\[
\begin{align*}
m &= 4 & \text{Set number of segments.} \\
dx &= (b-a)/m & \text{Set width of each segment.} \\
x &= \text{np.linspace}(a, b, m+1) & \text{Set values dividing segments.} \\
xmid &= \text{np.zeros}(m) & \text{Prepare list for midpoint values.} \\
\text{for } i \text{ in np.arange}(0, m): & \quad \text{Evaluate midpoints.} \\
& \quad xmid[i] = (x[i] + x[i+1])/2.0
\end{align*}
\]

Finally, we evaluate the sum, multiply with an overall factor whose inclusion was postponed, and display the result with the statements

\[
\begin{align*}
\text{int} &= 0.0 \\
\text{for } i \text{ in np.arange}(0, m): & \quad \text{fcts} = 0.0 \\
& \quad \text{for } j \text{ in } [0, 1, 2, 3, 4]: \\
& \quad \quad \text{fcts} = \text{fcts} + w[j] \times \text{np.exp}(-(xmid[i]+dx*pts[j]/2.0)**2) \\
\text{int} &= \text{int} + \text{fcts} \\
\text{int} &= 0.842700797138
\end{align*}
\]

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 `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

- `scipy.integrate.quad` (PYTHON 2 and 3), which uses a general purpose technique from the FORTRAN library QUADPACK,
- `scipy.integrate.fixed_quad` (PYTHON 2 and 3), which uses Gaussian quadrature of a fixed order (default 5) specified with the keyword \( n \),
- `scipy.integrate.quadrature` (PYTHON 2 and 3), which uses Gaussian quadrature but adaptively adjusts the order until a fixed tolerance has achieved, and
- `scipy.integrate.romberg` (PYTHON 2 and 3), which is adaptive and uses Romberg extrapolation to speed convergence

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,\(^{15}\) \( \text{Function} \) is a properly defined function returning the

\(^{15}\)The routine must, of course, have been appropriately imported before invoking it.
13.9. EVALUATING INTEGRALS NUMERICALLY WITH PYTHON

integrand, and LowLim and 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 scipy.integrate provides also the routines

- scipy.integrate.trapz (PYTHON 2 and 3), which use the trapezoidal rule,
- scipy.integrate.simps (PYTHON 2 and 3) and scipy.integrate.simpson (PYTHON 3), which use 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
- scipy.integrate.romb (PYTHON 2 and 3), which uses Romberg integration

to evaluate one-dimensional integrals when the function is supplied by a table of values. The routines trapz, simps, and simpson are invoked with a statement like

\[ \text{VarName} = \text{RoutineName}( y, x ) \]

while the routine romb is invoked with a statement like

\[ \text{VarName} = \text{romb}( y, \text{dx=Space} ) \]

Here VarName is the variable in which the value returned will be stored, RoutineName is the name of the integration routine to be invoked,\(^{16}\) 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 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.\(^{17}\)

The simplest of these routines—\texttt{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
```

\(^{16}\)The routine must, of course, have been appropriately imported before invoking it.

\(^{17}\)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.
As invoked here, the routine \texttt{trapz} has, of course, yielded $T_{16}$—and this result agrees with previous evaluations of the integral by this method with 16 divisions. The result from \texttt{trapz} is accurate only to the third digit; those from \texttt{simps} and \texttt{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).\footnote{Inclusion of parameters in the definition of the function will be discussed in Section 13.9.8.} 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

\begin{verbatim}
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
\end{verbatim}

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 \texttt{gausint.py}) and using defaults for all keywords, we would evaluate the integral and display its value with a statements\footnote{\texttt{quad} returns a tuple containing the value of the integral and an estimate of the error in that value; \texttt{fixed_quad} returns a tuple containing the value of the integral and a fixed value of None; \texttt{quadrature} returns a tuple containing the value of the integral and the difference between the last two estimates of the integral; and \texttt{romberg} returns only the value of the integral.}

\begin{verbatim}
  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)
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
\end{verbatim}

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 \texttt{quad} claims an accuracy much better than that and \texttt{romberg} implies an accuracy a bit better than that. For example, if we believe the change reported by \texttt{quadrature}, we might presume the value of the integral to be $0.842700793 \pm 0.000000007$.

\subsection{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
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

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, 1.428952489682337e-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

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 = sp.trapz( y, x )
print(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

\[
\begin{array}{cc}
 n & I \\
100 & 0.24962670804382944 \\
1000 & 0.2499881660155479 \\
10000 & 0.24999962568360487 \\
100000 & 0.2499999881627824 \\
1000000 & 0.2499999962567337 \\
\end{array}
\]

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.\(^{20}\)

\(^{20}\)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.
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

\[ P(|x| > |x_{\text{turn}}|) = 1.0 - \text{erf}(1.0) = 1.0 - 0.842701 = 0.157299 \]  

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

\[ g(x) = \int_{0}^{x} f(t) \, dt \]  

(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

\[ \text{N} = \langle \text{appropriate value} \rangle \]  
\[ x = \text{np.linspace}(a, b, \text{N+1}) \]  
\[ g = \text{np.zeros}(\text{N+1}) \]  
\[ \text{for } i = \text{in np.arange}(0, \text{N+1}): \]  
\[ g[i] = \text{sp.quad}(\text{integ}, 0, x[i])[0] \]

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 \text{quad} and accepting the default tolerance.

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 \( x \).

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

\[ \text{N} = \langle \text{appropriate value} \rangle \]  
\[ x = \text{np.linspace}(0, 3, \text{N+1}) \]  
\[ g = \text{np.zeros}(\text{N+1}) \]  
\[ \text{for } i = \text{in np.arange}(0, \text{N+1}): \]  
\[ g[i] = \text{sp.quad}(\text{integ}, 0, x[i])[0] \]

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 \text{quad} and accepting the default tolerance.

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 \( x \).

---

21 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.

22 We assume that `numpy` and `scipy.integrate` have been imported as `np` and `sp`, respectively.

23 This 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.
13.9. EVALUATING INTEGRALS NUMERICALLY WITH PYTHON

Figure 13.10: The error function.

```
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] )[0]
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()
```

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

```
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
```
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

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 $C$.
Add title.
Label axes.
Turn on grid.
Set size of tick labels.
Set range on axis.
Make axes square.
The resulting graph is shown in Fig. 13.11.

13.9.8 Integrals as Functions of an Internal Parameter

The situation in which an integral of interest is a function of a parameter in the integrand is more
difficult because we must somehow sneak the parameters in the integrand 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 vari-
ables will work with all of PYTHON's routines for integrating specified functions. Conven-
iently, these routines offer an alternative. Calls to these routines can take advantage of the keyword args.
The full structure of statements involving these commands is

\[
\text{VarName} = \text{RoutineName}( \text{Function, LowLim, UpLim, args=... } )
\]

\footnote{This sequence of statements actually has two glitches. First, it is computationally inefficient in the way described in footnote 23. Second, and less significantly, the values of $S(0)$ and $C(0)$ appear twice in the vectors finally plotted.
\footnote{Passing parameters is not an issue when tabulated data are to be integrated.}}
Here—beyond the quantities already defined—the keyword 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 args has only one value, that value must be followed by a seemingly irrelevant comma, e.g., args = (P1,).

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 \texttt{quad} (or some other integration routine), including a use of the keyword \texttt{args}.

Note that the \textit{names} of the parameters need \textit{not} be the same at command level as they are in the function \texttt{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 \texttt{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 \texttt{rings.py}, then invoke PYTHON to evaluate and display the desired integral by using \texttt{quad}, say, with the statement

```python
q = sp.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
13.10 Evaluating Integrals Numerically with MAXIMA

In addition to its routines for addressing integrals symbolically, MAXIMA contains several commands for numerical evaluation (sometimes called quadrature) of one-dimensional definite integrals. Among the available commands are the following:

- **quad_qag**( \( f(x), x, a, b, key, epsrel=\ldots, epsabs=\ldots, limit=\ldots \)), which integrates a general function over a finite interval by globally adaptive Gaussian integration.
- **quad_qags**( \( f(x), x, a, b, epsrel=\ldots, epsabs=\ldots, limit=\ldots \)), which adds extrapolation to the procedure invoked by **quad_qag**.

Table 13.3 conveys which of these keywords are recognized by which integration routines.

---

Figure 13.12: The potential in the midplane of a charged ring.
CHAPTER 13. EVALUATING INTEGRALS

Table 13.3: 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>

- **quad_qagi** (f(x), x, a, b, epsrel=..., epsabs=..., limit=...), which integrates a general function over an infinite or semi-infinite interval by mapping the interval into a finite interval and then invoking **quad_qags**.

The arguments in these functions are

- **f(x)** the function to be integrated, which can be either the full function, e.g., sqrt(1+x^2) or the label to which the function has been bound
- **x** the integration variable
- **a** the lower limit
- **b** the upper limit
- **key** (only in **quad_qag**) an integer between 1 and 6, inclusive, specifying the order of the Gaussian integration rule to be invoked
- **epsrel** desired relative error; default is 10^{-8} in double precision
- **epsabs** desired absolute error; default is 0
- **limit** maximum number of subintervals into which the adaptive procedure will divide the interval a ≤ x ≤ b; default is 200

At least one of a and b must be infinite in **quad_qagi**. Note also that the three keyword arguments **epsrel**, **epsabs**, and **limit** are optional, can be presented in any order, and take the form of **keyword = value**. Descriptions of additional routines for evaluating Fourier transforms, weighted integrals, and Cauchy principal values of integrals can be found in the MAXIMA manuals.

These commands return a four-component list consisting of, in order, an approximation to the integral, the estimated absolute error of the approximation, the number of times the integrand was evaluated, and an error code. The error code will have one of the values

0  if no problems were detected in the evaluation
1  if the maximum number of subintervals was exceeded
2  if excessive roundoff error was detected
3  if very bad behavior occurs
6  if the input function is invalid

With these features of MAXIMA, we can evaluate our standard example in many ways. We might, for example use the statement
13.10. EVALUATING INTEGRALS NUMERICALLY WITH MAXIMA

(%i1) float( integrate( 2*exp(-t^2)/sqrt(%pi), t, 0, 1 ) );
(%o1)    0.8427007929497148

which will evaluate the integral analytically [obtaining erf(1)], evaluate the error function of argument 1, and finally display the result, presumably to double-precision accuracy. Alternatively, we might accept defaults for all parameters and use the statement

(%i2) quad_qag( 2*exp(-t^2)/sqrt(%pi), t, 0, 1, 2 );
(%o2) [0.842700792949715, 9.35585... × 10^{-15}, 21, 0]

which yields the value 0.842700... to a claimed absolute error of 9 × 10^{-15} with 21 evaluations of the integrand and no problems detected in the evaluation. If we increase the order of the integration but still accept the defaults for the parameters, we find that

(%i3) quad_qag( 2*exp(-t^2)/sqrt(%pi), t, 0, 1, 4 );
(%o3) [0.8427007929497151, 9.35585... × 10^{-15}, 41, 0]

These results are all in agreement to the 15th decimal place.

13.10.1 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

\[ P(|x| > |x_{\text{turn}}|) = 1.0 - \text{erf}(1.0) = 1.0 - 0.8427008 = 0.1572992 \] (13.108)

which has been rounded to seven digits after the decimal point. We conclude that in a bit over 15\% of the measurements, the quantum oscillator will be found in the classically forbidden region!

13.10.2 The Error Function

To define and integrate the function \texttt{gaussian} and then generate successive numerical values of the function \texttt{erf(x)} defined by Eq. (13.26) as \texttt{x} varies from 0.0 to 3.0 in steps of 0.1, we might use the statements

(%i1) gaussian : 2.0 * exp(-t^2) / sqrt(%pi)$
(%i2) for j : 0 thru 30 do (q[j]:quad_qag(gaussian, t, 0, j/10, 2), i[j]:j/10)$

(%i3) [i[0], q[0]] ;
(%o3) [0, [0.0, 0.0, 21, 0]]

... Check evaluation for upper limit = 0

(%i4) [i[10], q[10]] ;
(%o4) [1, [0.8427..., 9.355... × 10^{-15}, 21, 0]]

The arrays \texttt{q} and \texttt{i} created by the statement on line \texttt{%i2} must then be converted into lists with the statements
before the graph of Fig. 13.13 can be created. Here, the notation \( q[v][1] \) in \( \%i6 \) extracts the \( v \)-th four-component list from \( q \) and then the first component of that four-component list to generate in \( q1 \) a list whose \( i \)-th component is solely the evaluation of the integral with an upper limit equal to the \( i \)-th component of \( i1 \). The resulting integral as a function of its upper limit can be graphed with the statement

\[
(\%i9) \text{plot2d( [discrete, i1, q1], [xlabel, "x"], [ylabel, "erf(x)"], [title, "Error Function"], [style, [lines, 4]], [color, black], [grid2d, true] )$}
\]

Though it will make little difference in the present example, integration will proceed more quickly if the function to be integrated is compiled before the integrals are evaluated and the above statements generating the graph are executed. To that end, we would define the function and compile it with the statements

\[
(\%i11) \text{gaussian(t) := 2.0 * exp(-t^2) / sqrt(%pi)$}
(\%i12) \text{compile(gaussian)$}
(\%i12) \text{for j : 0 thru 30 do (q[j]:quad_qag(gaussian, t, 0, j/10, 2), i[j]:j/10)$}
\]

13.10.3 The Off-Axis Electrostatic Potential of Two Rings

Using MAXIMA, we could evaluate and plot the integral in Eq. (13.36) to find the electrostatic potential at the radial coordinate \( sa \) in the midplane between two identical, uniformly charged parallel rings of radius \( a \). The statements
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 $x(0) = x_0 > 0$ and $v(0) = 0$.

13.4. Suppose an object of mass $m$ moves non-relativistically in one dimension under the action of the force $f(t) = f_0 e^{-bt}$, where both $b$ and $f_0$ are positive. Let $x(0) = x_0$ and $v(0) = v_0$. Use symbolic integration to find $x(t)$ and $v(t)$ by evaluating the integrals in Eq. (13.3). Then, find and interpret both the limits of these two results as $t \to \infty$ and the Taylor expansion of these two results for small $t$.

13.5. The normalized Lorentz distribution function is given by

$$p(x) = \frac{1}{\pi} \frac{a/2}{x^2 + (a/2)^2}$$

Using symbolic integration, (a) verify that $\int_{-\infty}^{+\infty} p(x) \, dx = 1$, (b) evaluate—as best you can—the average $\bar{x}$ 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

$$P(r) = \frac{4}{a^3} \int_0^r e^{-2r'/a} \, r' \, dr' = \frac{4}{a^3} \int_0^{r/a} e^{-2\rho} \, \rho^2 \, d\rho$$

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
13.10. Consider a surface in the $xy$ plane having uniform mass density $\sigma$ and having the shape of a cardioid given in polar coordinates by the function $r(\phi) = a(1 - \cos \phi)$. Using symbolic integration, find (a) the center of mass of this object, (b) the moment of inertia tensor of this object about the $x$, $y$, and $z$ axes, and (c) the radius of gyration about the $z$ axis. Hints: The center of mass is defined in Section 13.1.2; the moment of inertia tensor is a $3 \times 3$ tensor whose $ij$ element is given by

$$I_{ij} = \int [(x_1^2 + x_2^2 + x_3^2)\delta_{ij} - x_i x_j] \, dm$$

where $x_1, x_2,$ and $x_3$ symbolize $x, y,$ and $z$, respectively; $\delta_{ij}$ is the Kronecker delta, which has the value 1 when $i = j$ and the value 0 otherwise; and the radius of gyration is defined in Section 13.1.3.

13.11. In quantum mechanics, the two integrals

$$x_{mn} = \int_{-\infty}^{\infty} \psi_m^*(x) x \psi_n(x) \, dx \quad \text{and} \quad p_{mn} = \int_{-\infty}^{\infty} \psi_m^*(x) \frac{\hbar}{i} \frac{\partial}{\partial x} \psi_n(x) \, dx$$

are important in a variety of contexts. For a particle in an infinitely deep potential well that extends over the region $-a \leq x \leq a$,

$$\psi_n(x) = \begin{cases} \frac{1}{\sqrt{a}} \cos \frac{n\pi x}{2a} & n = 1, 3, 5, \ldots \\ \frac{1}{\sqrt{a}} \sin \frac{n\pi x}{2a} & n = 2, 4, 6, \ldots \end{cases}$$

Using symbolic integration, show that, for these wave functions, $x_{mn} = 0$ and $p_{mn} = 0$ when $m$ and $n$ are both even or both odd and that

$$x_{mn} = \frac{16a}{\pi^2} (-1)^{\frac{m+n+1}{2}} \frac{mn}{(m^2 - n^2)^2}$$

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

$$f(x) = \frac{x}{l} \quad ; \quad -l \leq x \leq l$$
The Legendre polynomials

13.13. To deduce Simpson’s rule, we start by supposing three consecutive values

(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

$$L_0(x) = 1 \quad L_5(x) = \frac{1}{8}(63x^5 - 70x^3 + 15x)$$

$$L_1(x) = x \quad L_6(x) = \frac{3}{16}(231x^6 - 315x^4 + 105x^2 - 5)$$

$$L_2(x) = \frac{1}{2}(3x^2 - 1) \quad L_7(x) = \frac{1}{16}(429x^7 - 693x^5 + 315x^3 - 35x)$$

$$L_3(x) = \frac{1}{2}(5x^3 - 3x) \quad L_8(x) = \frac{1}{128}(6435x^8 - 12012x^6 + 6930x^4 - 1260x^2 + 35)$$

$$L_4(x) = \frac{1}{8}(35x^4 - 30x^2 + 3) \quad L_9(x) = \frac{1}{128}(12155x^9 - 25740x^7 + 18018x^5 - 4620x^3 + 315x)$$

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—paralleling Eq. (13.88)—we set

\[ \int_{-1}^{1} f(x) \, dx = w_1 f(x_1) + w_2 f(x_2) + w_3 f(x_3) \]

We then choose the six quantities \( w_i \), and \( x_i \) so that the expression gives an exact result for \( f(x) = 1, x, x^2, x^3, x^4, \text{and} \, x^5 \). (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

\[ \psi_1(x) = \frac{\sqrt{2}}{\sqrt{\pi \hbar}} \frac{m \omega}{\sqrt{\pi \hbar}} \frac{1}{\sqrt{2}} y e^{-y^2/2} \quad \psi_2(x) = \frac{1}{\sqrt{2}} \frac{m \omega}{\sqrt{\pi \hbar}} \frac{1}{2y^2 - 1} e^{-y^2/2} \]

where \( y = x/\sqrt{\hbar \omega/\bar{k}} \), 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

\[ f(v) = 4\pi \left(\frac{m}{2\pi kT}\right)^{3/2} \int_0^v e^{-mv^2/2kT} v^2 \, dv \]

for the fraction of the molecules having speed less than \( v \). Using numerical means, explore this integral as a function of \( v \). \( \text{Hint:} \) Re-express the integral using \( \sqrt{2kT/m} \) as the unit of velocity.

13.20. Planck’s black body radiation law gives the expression

\[ I(\nu_2, \nu_1) = \frac{8\pi \hbar}{c^3} \int_{\nu_1}^{\nu_2} \frac{\nu^3}{e^{\hbar\nu/kT} - 1} \, d\nu \]

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. \( \text{Hint:} \) One way to approach this exercise would be to choose a reference frequency \( \nu_0 \) arbitrarily (say \( 10^{14} \, \text{Hz} \)) and recast the integral on the dimensionless variable \( s = \nu/\nu_0 \). Examination of \( I \) in units of \( 8\pi h\nu_0^3/c^3 \) as a function of \( T \) in units of \( h\nu_0/k \) would then be indicated.

13.21. As used in statistical data analysis, the Gaussian distribution for a variable \( t \) is usually expressed in terms of the standard deviation \( \sigma \), the distribution function being

\[ \frac{1}{\sqrt{2\pi} \sigma} e^{-t^2/2\sigma^2} \]

Thus, the probability of finding a value between \( a \) and \( b \) is given by

\[ P(a, b) = \frac{1}{\sqrt{2\pi}\sigma} \int_a^b e^{-t^2/(2\sigma^2)} \, dt \]

Show analytically that \( P(-x, x) = \text{erf}(x/(\sqrt{2}\sigma)) \), 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? \( \text{Hint:} \) Since the gravitational potential is \(-GmM/x\), conservation of energy yields

\[ \frac{1}{2} \frac{dx}{dt}^2 = \frac{GmM}{x} = -\frac{GmM}{x_0} \quad \Rightarrow \quad \frac{dx}{dt} = -\sqrt{2GM} \sqrt{\frac{1}{x} - \frac{1}{x_0}} \]
CHAPTER 13. EVALUATING INTEGRALS

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 \textit{a priori} knowledge of the exact value. \textit{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

\[ P(r) = \frac{4}{a^3} \int_0^r e^{-2r'/a} r'^2 dr' = 4 \int_0^{r/a} e^{-2\rho^2} \rho^2 d\rho \]

where \(a\) is the Bohr radius and \(\rho = r'/a\). 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

\[ K(k) = \int_0^{\pi/2} \frac{d\phi}{(1 - k^2 \sin^2 \phi)^{1/2}} \quad ; \quad E(k) = \int_0^{\pi/2} (1 - k^2 \sin^2 \phi)^{1/2} d\phi \]

Explore these integrals as functions of the \textit{modulus} \(k\). As part of your exploration, obtain a graph of the period \(T\) of a simple pendulum as a function of the amplitude \(\alpha\) of that pendulum. Analytically, the period of that pendulum is given as a function of \(\alpha\) by \(T/T_0 = (2/\pi)K(\sin(\alpha/2))\), where \(T_0\) is the period of the pendulum at small amplitude.

13.26. The angular position \(\theta(t)\) of a simple pendulum swinging with amplitude \(\alpha\) is given by the integral

\[ \omega t = \int_0^\beta \frac{d\phi}{(1 - k^2 \sin^2 \phi)^{1/2}} \]

where, with \(l\) the length of the pendulum and \(g\) the acceleration of gravity, \(\omega = \sqrt{g/l}\), \(k = \sin(\alpha/2)\), and \(\beta = \sin^{-1}([\sin(\theta/2)]/k)\). Remember that, because of the choice of signs (see Section 13.1.4), this integral is valid only during the portion of the swing from \(\theta = 0\) to \(\theta = \alpha\). Obtain graphs of \(\theta\) versus \(\omega t\) over the first quarter of the pendulum’s swing for several different values of \(\alpha\).

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

$$V(r, \phi, z) = \frac{Q}{4\pi \varepsilon_0} a = \frac{1}{\pi} \int_0^{\pi} \left( 1 - \frac{2r}{a} \cos \phi' + \frac{r^2}{a^2} + \frac{z^2}{a^2} \right)^{-1/2} \, d\phi'$$

Explore this integral as a function of $r/a$ for several values of $z/a$.

13.30. A circular current loop of radius $a$ lies in the $xy$-plane with its center at the origin and carries a current $I'$ counterclockwise as viewed from a point on the positive $z$ axis. The magnetic field at a point in the $xz$ plane is given by

$$B(x, z) = \frac{\mu_0 I'/2\pi a}{a^2} \int_0^{\pi} \frac{z \cos \phi' \hat{i} + (a - x \cos \phi') \hat{k}}{\sqrt{x^2 + z^2 + a^2 - 2ax \cos \phi' \sqrt{2}}} \, d\phi'$$

Explore both components of this magnetic field numerically as functions of $x/a$ for various values of $z/a$, including $z/a = 0$ (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,
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 tgif), the graphs in Section 14.1 have all been produced with IDL. Those in each later section have been produced by whatever software is the subject of that section.
14.1.1 Classical Turning Points

Let \( V(x) \) be the potential energy under which an object of mass \( m \) is moving in one dimension. Turning points in the motion occur at values of \( x \) where the total energy \( E \) is entirely potential energy (kinetic energy is zero), i.e., when

\[
V(x) = E \quad \text{or} \quad V(x) - E = 0 \tag{14.2}
\]

Finding physical turning points thus involves finding the mathematical roots of the function \( f(x) = V(x) - E \), i.e., finding solutions to Eq. (14.2). If, for example, the potential energy of interest is given by the cubic polynomial

\[
V(x) = \frac{x^3}{10000} + \frac{x^2}{200} - \frac{x}{500} - \frac{1}{2} \tag{14.3}
\]

the turning points for the motion of a particle with total energy \( E = 0 \) moving in this potential energy would satisfy \( V(x) = 0 \). The first step in finding those turning points would therefore be to produce the graph of Fig. 14.1—a task that may require a bit of trial and error before a suitable range for the independent variable has been found. From this graph, we conclude that \( V(x) \) has three real roots, one in the vicinity of \( x = -50 \), a second in the vicinity of \( x = -10 \), and a third in the vicinity of \( x = +10 \). With more refined graphs drawn in the vicinity of each of these roots, we could conclude that the three roots are more tightly bound by the limits

\[
-50.00 < x_1 < -47.50 ; \quad -12.50 < x_2 < -10.00 ; \quad 7.50 < x_3 < 10.00 \tag{14.4}
\]

14.1.2 A Max-Min Problem: Equilibrium Points

From a different perspective, a particle moving in one dimension under the action of the force \( F = -dV/dx \) associated with the potential energy \( V(x) \) will be in equilibrium at those points \( x \) at which the force is zero, i.e., where

\[
F = 0 \quad \Rightarrow \quad \frac{dV}{dx} = 0 \tag{14.5}
\]

or where the graph of \( V(x) \) has a horizontal tangent. Further, evaluated at a point of equilibrium, \( d^2V/dx^2 > 0 \) implies that the equilibrium is stable while \( d^2V/dx^2 < 0 \) implies that the equilibrium is unstable. Finding points of physical equilibrium therefore involves finding mathematical roots of the function \( F(x) = -dV/dx \), and assessing the stability of those equilibria entails examining the sign of \( d^2V/dx^2 \).

More specifically, for the potential energy given in Eq. (14.3) and graphed in Fig. 14.1, we would find the points of equilibrium by solving the equation

\[
\frac{dV}{dx} = \frac{3x^2}{10000} + \frac{x}{100} - \frac{1}{500} = 0 \tag{14.6}
\]

By looking at local extrema in the graph of \( V(x) \), we infer that this potential energy exhibits two points of equilibrium, one located in the vicinity of \( x = -35 \) and the other in the vicinity of \( x = 0 \). A more refined graph leads to the conclusion that these two roots are bounded by

\[
-35.0 < x_4 < -30.0 \quad \text{and} \quad -2.5 < x_5 < 2.5 \tag{14.7}
\]

---

\(^2\)This function is, of course, not particularly realistic as a potential energy. We can, however, provide a physical context for at least a portion of \( V(x) \). If we confine our attention to the region around the one minimum it possesses, we can interpret the function as the potential energy of an anharmonic oscillator, for which—if that minimum occurs at \( x = 0 \)—we might write \( V(x) = ax^2/2 + b x^3 \) while imposing the constraint that \( a \), the coefficient of the cubic perturbation from the potential energy of a simple harmonic oscillator, be small. The expression in Eq. (14.2) simply places the minimum at a different value of \( x \) and adds a constant to the potential energy. In what follows, we will explore this function over a wider range of values of \( x \) than is physically meaningful. The pedagogic advantage of Eq. (14.2) is that it combines many of the important features of potential energy functions with, as we shall see, tractability by a variety of different approaches.
14.1.3 Natural Frequencies of Oscillating Systems

Suppose we seek the natural frequencies of oscillation for the system shown in Fig. 14.2, which consists of two objects, each having mass \( m \). Let these objects move in one dimension on a horizontal, frictionless surface, let them be connected to one another with a spring having constant \( k' \), and let each be connected to the nearer wall with a spring having constant \( k \). Further, let the position of each be measured from its equilibrium position. Then, Newton’s second law combined with Hooke’s law leads to the equations of motion\(^3\)

\[
\begin{align*}
md\frac{d^2x_1}{dt^2} &= -kx_1 + k'(x_2 - x_1) \\
md\frac{d^2x_2}{dt^2} &= -kx_2 - k'(x_2 - x_1)
\end{align*}
\] (14.8)

To cast these equations in dimensionless form, we choose a unit of length \( a \), set \( x_i/a = \tilde{x}_i \), introduce \( \omega_0 = \sqrt{k/m} \), set \( \kappa = k'/k \), and introduce \( \tilde{t} = \omega_0 t \) to find that

\[
\begin{align*}
d\frac{d^2\tilde{x}_1}{d\tilde{t}^2} &= -(1 + \kappa)\tilde{x}_1 + \kappa \tilde{x}_2 \\
d\frac{d^2\tilde{x}_2}{d\tilde{t}^2} &= \kappa \tilde{x}_1 - (1 + \kappa)\tilde{x}_2
\end{align*}
\] (14.9)

Next, seeking sinusoidal (or simple harmonic) solutions, we suppose that

\[
\tilde{x}_1(\tilde{t}) = \tilde{x}_{10} \cos \omega \tilde{t} \quad \text{and} \quad \tilde{x}_2(\tilde{t}) = \tilde{x}_{20} \cos \omega \tilde{t}
\] (14.10)

\[^3\]These equations were also discussed in Section 11.1.6.
where the (yet to be determined) frequency $\omega$ is measured in units of $\omega_0$. Substituting these suppositions into Eq. (14.9), we conclude that the (presently unknown) amplitudes must satisfy

$$
\begin{vmatrix} 1 + \tau - \omega^2 & -\tau \\ -\tau & 1 + \tau - \omega^2 \end{vmatrix} \begin{pmatrix} x_{10} \\ x_{20} \end{pmatrix} = 0
$$

(14.11)

The solution of this equation for the unknowns $x_{10}$ and $x_{20}$ will be trivial (i.e., both zero—a particularly uninteresting motion) unless the determinant of the matrix of coefficients happens itself to be zero, i.e., unless the characteristic equation

$$
D(\omega) = \begin{vmatrix} 1 + \tau - \omega^2 & -\tau \\ -\tau & 1 + \tau - \omega^2 \end{vmatrix} = (1 + \tau - \omega^2)^2 - \tau^2 = \omega^4 - 2\omega^2(1 + \tau) + 1 + 2\tau = 0
$$

(14.12)

is satisfied. We arrive at a fourth-order polynomial $D(\omega)$ in $\omega$, the roots of which will give the natural frequencies of oscillation for the simple system under consideration. Since only even powers of $\omega$ appear, however, we can for the sake of a simpler solution regard the polynomial as quadratic in $\omega^2$.

Suppose we seek the dependence of the roots of this polynomial on $\tau = k'/k$, 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 $\tau$ 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 $\tau$, while the upper of the two roots increases steadily as $\tau$ increases.

14.1.4 Range of Projectile in Viscous Medium

For a fourth example, suppose we seek the range of a projectile of mass $m$ fired with the initial speed $v_0$ at angle $\theta$ up from the horizontal in a viscous medium having damping constant $b$. Supposing the
projectile to move in the \( xy \) plane, with \( x \) horizontal and \( y \) vertical, we begin by invoking Newton’s second law\(^4\) to write the equations of motion

\[
\frac{d^2x}{dt^2} = -b \frac{dx}{dt} \quad \text{and} \quad \frac{d^2y}{dt^2} = -mg - b \frac{dy}{dt}
\]  

(14.13)

Here, \( g \) is the acceleration of gravity (which we take to be a positive number). These equations are to be solved subject to the initial values

\[
x(0) = 0 \quad , \quad v_x(0) = v_0 \cos \theta \quad , \quad y(0) = 0 \quad \text{and} \quad v_y(0) = v_0 \sin \theta
\]

(14.14)

The solution to this problem—see Chapter 11 on ordinary differential equations—is

\[
x(t) = \frac{mv_0 \cos \theta}{b} \left( 1 - e^{-bt/m} \right)
\]

(14.15)

\[
y(t) = -\frac{mgt}{b} + \frac{m}{b} \left( v_0 \sin \theta + \frac{mg}{b} \right) \left( 1 - e^{-bt/m} \right)
\]

(14.16)

Finally, to find the range \( R \), we seek the value of \( x \) at that non-zero value of \( t \) for which \( y(t) = 0 \). Thus, we need to find the non-zero solution, say \( t_1 \), of the equation

\[
-\frac{mgt}{b} + \frac{m}{b} \left( v_0 \sin \theta + \frac{mg}{b} \right) \left( 1 - e^{-bt/m} \right) = 0
\]

(14.17)

and then evaluate \( x(t_1) \). Equivalently, if we introduce the dimensionless time \( \tau = bt/m \) and the dimensionless parameter \( \alpha = bv_0/(mg) \), the equation whose root we seek becomes

\[
f(\tau) = \tau - \left( 1 + \alpha \sin \theta \right) \left( 1 - e^{-\tau} \right) = 0
\]

(14.18)

Once we have found the desired root, say \( \tau_1 \), we then determine the range \( R \) by substituting \( \tau_1 \) into Eq. (14.15), finding that

\[
\frac{R(\theta, \alpha)}{v_0^2/g} = \frac{\cos \theta \left( 1 - e^{-\tau_1} \right)}{\alpha} = \frac{\cos \theta}{\alpha(1 + \alpha \sin \theta)} \tau_1
\]

(14.19)

The mathematical task confronting us involves finding the non-zero root of the function \( f(\tau) \) defined in Eq. (14.18). More specifically, finding the angle \( \theta \) at which the projectile should be fired to achieve maximum range would entail

- Choosing a value of \( \alpha = bv_0/(mg) \).
- Finding the non-zero root of \( f(\tau) \) for several values of \( \theta \) ranging from 0 to \( \pi/2 \).
- Calculating and plotting values of \( R(\theta, \alpha) \).
- Finding the value of \( \theta \) corresponding to the peak in a graph of \( R(\theta, \alpha) \) versus \( \theta \).

Note that \( \alpha \) increases as \( b \) (the damping) increases and as \( v_0 \) increases but decreases as \( m \) increases.

To gain some insight into the nature of the desired roots, we begin by plotting the family of graphs of \( f(\tau) \) versus \( \tau \) for various values of \( \theta \) with fixed \( \alpha \). After some exploration using techniques introduced in earlier chapters, we arrive at the several graphs shown in Fig. 14.4. Every graph exhibits a root at \( \tau = 0 \), corresponding to the moment of launch. Though it is hard to judge, each graph for \( \theta = 0^\circ \) also exhibits a second root at \( \tau = 0 \), since a projectile launched at \( \theta = 0^\circ \) returns to its initial altitude immediately. As the angle of launch is increased in each case, however, the second root moves to larger and larger values of \( \tau \). Our task in subsequent sections will be to find numerical values for that second root for various values of \( \alpha \) and \( \theta \), determine the range for each, and find the particular angle at which the range is greatest for a given \( \alpha \).

\(^4\)Compare Eqs. (11.3) and (11.4).
Figure 14.4: Graphs of \( f(\tau) \) versus \( \tau \). The individual frames in this display correspond to different values of \( \alpha \); the graphs in each frame correspond to different values of \( \theta \), ranging in 15° increments from 0° for the highest graph to 90° for the lowest graph in each frame. The scalings have been chosen to reveal the nature of the roots most clearly and are different in each frame.

As a quick aside, we can reassure ourselves that this approach is appropriate by examining the limits of Eqs. (14.18) and (14.19) as \( b \) becomes small. In that limit, both \( \tau = bt/m \) and \( \alpha = b v_0/(mg) \) 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

\[
\frac{R(\theta, \alpha \rightarrow 0)}{v_0^2/g} = \frac{\cos \theta}{\alpha(1 + \alpha \sin \theta)} (2\alpha \sin \theta) = \frac{2 \sin \theta \cos \theta}{(1 + \alpha \sin \theta) \to 2 \sin \theta \cos \theta}
\] (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

\[
V(x) = \begin{cases} 
\infty & x < 0 \\
-V_0 & 0 \leq x \leq a \\
0 & a \leq x 
\end{cases} \tag{14.22}
\]

leads to the conclusion that the allowed energies \(E\) should satisfy the equation\(^5\)

\[
s \cot s = -\sqrt{c^2 - s^2} \quad \tag{14.23}
\]

where \(c^2 = 2ma^2V_0/\hbar^2\) and \(s^2 = c^2(1 - E/V_0)\) or \(E = -V_0(1 - s^2/c^2)\). To find the energies, we must thus solve Eq. (14.23) for acceptable values of \(s\) once the depth of the well conveyed by the (fixed) value of \(c\) has been specified.

A simpler version of this equation emerges if we square it (thereby possibly introducing spurious roots because the squared equation is also consistent with a plus sign on the right hand side) and then invoke the trigonometric identity \(1 + \cot^2 \theta = \sin^{-2} \theta\) to find that

\[
s^2 \cot^2 s = s^2(\sin^{-2} s - 1) = c^2 - s^2 \quad \Rightarrow \quad \sin s = \pm s/c \quad \tag{14.24}
\]

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

\[
\sin s = \pm 0.04s \quad \text{or} \quad \sin s \mp 0.04s = 0 \quad \tag{14.25}
\]

We suppose we seek roots of this function in the interval \(s \geq 0\). The graphs in Fig. 14.5 reveal these solutions in several ways. The lower graph in Fig. 14.5(a) shows the function \(\sin s - 0.04s\) (upper sign in Eq. (14.25)), and its roots lie where this graph crosses the \(s\) axis; there are eight roots in this group. Similarly, the upper graph in Fig. 14.5(a) shows the function \(\sin s + 0.04s\) (lower sign in Eq. (14.25)), and its roots appear where the graph crosses the \(s\) axis; there are nine roots in this group. In Fig. 14.5(b), which shows the functions \(\sin s\), \(+0.04s\), and \(-0.04s\), the roots appear at the values of \(s\) where one or the other of the sloped straight lines intersects the sine curve. From either of these graphs, we conclude that, for the upper sign in Eq. (14.25) the eight roots are bounded—crudely—by

| Root 1u | -0.5 < s < 0.5 |
| Root 2u | 2.5 < s < 3.5 |
| Root 3u | 6.0 < s < 7.0 |
| Root 4u | 8.5 < s < 9.5 |

and that, for the lower sign, the nine roots are bounded—again crudely—by

| Root 1l | -0.5 < s < 0.5 |
| Root 2l | 3.0 < s < 4.0 |
| Root 3l | 5.5 < s < 6.5 |
| Root 4l | 9.5 < s < 10.5 |
| Root 5l | 11.5 < s < 12.5 |

\(^5\)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

\[
\frac{a + x}{b + x^2} - c = 0 \implies a + x - c(b + x^2) = -cx^2 + x + a - cb = 0
\]

(14.32)

converts the problem into one involving a quadratic polynomial. Beyond these few cases, very few roots can be found symbolically.

## 14.3 Finding Roots Symbolically with MAXIMA

The MAXIMA command for solving a wide variety of algebraic and transcendental equations is `solve`. It takes two arguments, the first giving the equation to be solved (or the function whose roots are desired) and the second specifying the variable for which a solution is desired. Thus, the statement

\[
(\%i1) \text{solve}( a*x + b = 0, x );
\]

(\%o1) \[[x = -\frac{b}{a}]\]

returns the solution \( x = -b/a \) and the statement

\[
(\%i2) \text{solve}( a*x^2 + b*x + c, x );
\]

(\%o2) \[[x = -\frac{\sqrt{b^2 - 4ac} + b}{2a}, x = \frac{\sqrt{b^2 - 4ac} - b}{2a}]\]

in which we have illustrated that MAXIMA assumes the omitted ‘= 0’ at the end of the first argument, returns two solutions

\[
x = \frac{-b + \sqrt{b^2 - 4ac}}{2a} \quad ; \quad x = \frac{-b - \sqrt{b^2 - 4ac}}{2a}
\]

(14.33)

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 that reduces all of the factors to polynomials of order no higher than quartic. Note also that `solve` returns the solution(s) as a list, even if there is only one solution.

### 14.3.1 Classical Turning Points

The turning points for the potential energy in Eq. (14.3) are readily found symbolically with the statements

\[
(\%i1) \text{solve}( x^2 = 0, x );
\]

(\%o1) \[[x = 0]\]

returns the solution \( x = 0 \) and the statement

\[
(\%i2) \text{solve}( x^3 = a, x );
\]

(\%o2) \[[x = -\frac{\sqrt[3]{a}}{3}, x = \frac{\sqrt[3]{a}}{3} \cdot \cos \left( \frac{\pi}{3} \right), x = \frac{\sqrt[3]{a}}{3} \cdot \cos \left( \frac{\pi}{3} + \frac{2\pi}{3} \right)]\]

in which we have illustrated that MAXIMA assumes the omitted ‘= 0’ at the end of the first argument, returns two solutions

\[
x = -\frac{\sqrt[3]{a}}{3} \quad ; \quad x = \frac{\sqrt[3]{a}}{3} \cdot \cos \left( \frac{\pi}{3} \right)
\]

(14.34)
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, generate a more interpretable solution by extracting and simplifying the real and imaginary parts with the statements\(^6\)

\[
\text{(i3) map( rhs, realpart(float(soln)) );}
\]
\[
\text{(o3) \([-11.08043663084843, -48.26826756675163, 9.348704197600053]\)}
\]
\[
\text{(i4) map( rhs, imagpart(float(soln)) );}
\]
\[
\text{(o4) \([7.1054 \times 10^{-15}, -3.5527 \times 10^{-15}, -1.7763 \times 10^{-15}\)}
\]

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 the precision of the command \texttt{float} is on the order of \(10^{-15}\), so the imaginary parts differ from zero almost certainly because of internal computer roundoff.

**14.3.2 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

\[
\text{(i5) dVdx : diff( V, x );}
\]
\[
\text{(i6) solve( dVdx = 0, x );}
\]
\[
\text{(o6) \[-\frac{16\sqrt{10} + 50}{3}, \frac{16\sqrt{10} - 50}{3}\}]
\]
\[
\text{(i7) float(%);}
\]
\[
\text{(o7) \([-33.53214752089802, x = 0.198814187564908\)}
\]

finding first \(dV/dx\) and then finding the values of \(x\) at which \(dV/dx = 0\). The values 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

\[
\text{(i8) d2Vdx2 : diff( dVdx, x );}
\]
\[
\text{(i9) d2Vdx2, part( %o8, 1 );}
\]
\[
\text{(o9) \(-0.01011928851253881\)}
\]
\[
\text{(i10) d2Vdx2, part( %o8, 2 );}
\]
\[
\text{(o10) \(0.01011928851253881\)}
\]

concluding that the first root \(x = -33.53214\ldots\) locates an unstable equilibrium \((d^2V/dx^2 < 0);\) local maximum in the potential energy) while the second root \(x = 0.198814\ldots\) locates a stable equilibrium \((d^2V/dx^2 > 0);\) local minimum in the potential energy).

---

\(6\)Different versions of MAXIMA run on different platforms may produce slightly different results in the last digit or two.
14.3. 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 \( \omega^2 \), which we call \( \omega_2 \), and then take the square root of that solution with the statements

\[
\begin{align*}
\%i1 & : \omega_2^2 - 2\omega_2(1+kappa) + 1 + 2kappa \\
\%i2 & : \text{solve} \left( \omega_2 = 2kappa + 1, \omega_2 = 1 \right) \\
\%i3 & : \text{map} \left( \text{rhs}, \%o2 \right) \\
\%i4 & : \sqrt{2kappa + 1}, 1
\end{align*}
\]

Note—as must on physical grounds be the case—that the solutions obtained for \( \omega^2 \) at \( \%o2 \) 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 increases steadily as \( \kappa \)—the strength of the coupling—increases, but that the second of the two frequencies is the same for all \( \kappa \).

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 statements

\[
\begin{align*}
\%i5 & : \text{plot2d} \left( \omega, \left[ \kappa, 0, 4 \right], \left[ \text{xlabel}, “kappa” \right], \left[ \text{ylabel}, “\omega” \right], \\
& \quad \left[ y, 0, 3 \right], \left[ \text{legend}, \text{false} \right], \left[ \text{color}, \text{black} \right], \left[ \text{style}, \left[ \text{lines}, 4 \right] \right] \right); 
\end{align*}
\]

The resulting graph is shown in Fig. 14.6.

14.3.4 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 MAXIMA. We begin by defining the trajectory as given by Eqs. (14.15) and (14.16) with the statements

\[
\begin{align*}
\%i11 & : \left( m v_0 \cos(\theta)/b \right) \left( 1 - \exp(-b t/m) \right) \\
\%i12 & : \left( -m g t/b + (m/b) v_0 \sin(\theta) + m g/b \right) \left( 1 - \exp(-b t/m) \right)
\end{align*}
\]

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 MAXIMA statements

---

7 At this point, for simplicity, we drop the overbar on \( \pi \).

8 To facilitate visualization, some of the output in the following dialog with MAXIMA has been recast from the actual form presented by MAXIMA.
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 $\omega_0 = \sqrt{k/m}$.

\[
\begin{align*}
\text{(i3)} & \quad \text{xbl} : \text{factor(taylor(x, b, 0, 1))}; \\
\text{(o3)} & \quad \frac{-t(bt - 2m) \cos \theta v_0}{2m} \\
\text{(i4)} & \quad \text{yl} : \text{expand(taylor(y, b, 0, 1))}; \\
\text{(o4)} & \quad -\frac{bt^2 \sin \theta v_0}{2m} + t \sin \theta v_0 + \frac{bg t^3}{6m} - \frac{gt^2}{2}
\end{align*}
\]

If we notice some of the features of these expansions, we can clean them up a bit as follows\(^9\)

\[
\begin{align*}
\text{(i5)} & \quad \text{f1} : v_0 t \cos \theta \\
\text{(i6)} & \quad \text{f2} : \text{expand(xbl/f1)} \\
\text{(i7)} & \quad \text{xbl} : \text{f1*f2}; \\
\text{(o7)} & \quad v_0 t \cos \theta \left(1 - \frac{bt}{2m}\right)
\end{align*}
\]

\[
\begin{align*}
\text{(i8)} & \quad \text{f1} : \text{part(ybl,2)} + \text{part(ybl,4)} \\
\text{(i9)} & \quad \text{f2} : \text{factor(ybl-f1)} \\
\text{(i10)} & \quad \text{ybl} : \text{f1+f2}; \\
\text{(o10)} & \quad v_0 t \sin \theta - \frac{1}{2} gt^2 - \frac{bt^2}{6m} \left(3v_0 \sin \theta - gt\right)
\end{align*}
\]

We notice first—reassuringly—that the terms that are zeroth-order in $b$, i.e., $x = v_0 t \cos \theta$ and $y = v_0 t \sin \theta - \frac{1}{2} gt^2$, 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 removing the root at $t = 0$ and solving for the roots of the Taylor expansion of $y$ with the statements

\(^9\)Note that, in the version of MAXIMA you are using, the order of the terms in %o4 may be different from that shown, in which case the extraction of the proper parts—those independent of $b$—in the statement in %i8 may require different numbers.
14.3. FINDING ROOTS SYMBOLICALLY WITH MAXIMA

(%i11) expand( yb/t );
(%o11) \( v_0 \sin \theta - \frac{1}{2}gt - \frac{bt}{6m} (3v_0 \sin \theta - gt) \)

(%i12) solve( % = 0, t );
(%o12) \[ t = \text{mess}, t = \text{mess} \]

To sort out which of these two roots is the physically significant one, we evaluate the limit of this expression as \( b \to 0 \) with the statements

(%i13) limit( %, b, 0 );
Is \( g \) positive or negative? p;
Is \( m \) positive or negative? p;
(%o13) \[ t = \frac{2v_0 \sin \theta}{g}, t = \text{infinity} \]

Clearly, only the first root is of interest, and we extract it from the more general expression %o12 with the statement

(%i14) part( %o12, 1, 2 );
(%o14) \( \frac{3m}{2b} + \frac{3v_0}{2g} \sin \theta - \sqrt{\frac{3}{2}} \sqrt{3v_0^2 b^2 \sin^2 \theta - 2v_0bgm \sin \theta + 3g^2 m^2} \)

Finally, for consistency with approximations already made, we find this root to first order in \( b \) with the statement

(%i15) time : expand( taylor( %, b, 0, 1 ) );
(%o15) \( \frac{2v_0 \sin \theta}{g} - \frac{2v_0^2 \sin^2 \theta}{3g^2 m} b \)

and find the range by substituting this result into the first order expression for the horizontal coordinate with the statements

(%i16) range : xb, t=time;
(%o16) \text{mess}

In examining the mess, however, we note that some terms in \( b^2 \) have crept in. The statement

(%i17) expand( taylor( range, b, 0, 1 ) );
(%o17) \( \frac{2v_0^2 \cos \theta \sin \theta}{g} - \frac{8bv_0^2 \cos \theta \sin^2 \theta}{3g^2 m} \)

removes those terms and simplifies the mess somewhat. Now, however, we note the common factor \( 2v_0^2 \sin \theta \cos \theta / g \) in both terms, so the statements

(%i18) f1 : 2*v0^2*sin(theta)*cos(theta)/g$% 
(%i19) f2 : expand(%o17/f1)$% 
(%i20) range : f1*f2;
(%o10) \( \frac{v_0^2}{g} (2 \sin \theta \cos \theta) \left( 1 - \frac{4v_0b}{3mg} \sin \theta \right) + \ldots \)

yields a simple final result. 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
\[
x_{\text{mid}} = \frac{1}{2}(x_{\text{min}} + x_{\text{max}}) \quad \text{and} \quad f_{\text{mid}} = f(x_{\text{mid}})
\]

2. calculating \( f_{\text{min}} f_{\text{mid}} \), which will be positive if \( f_{\text{min}} \) and \( f_{\text{mid}} \) have the same sign (and the root then lies in the upper half of the original interval) and negative if \( f_{\text{min}} \) and \( f_{\text{mid}} \) have opposite signs (and the root then lies in the lower half of the original interval). Note that we have here assumed the root to be a single root (or at least a root of odd multiplicity), since the criterion invoked depends on the function having opposite signs on opposite sides of the root. (For a root of even order, the function has the same sign on opposite sides of the root, and the criterion here described will not identify the half of the interval in which the root lies. Without embellishment, bisection will fail in this case.)

3. refocusing our attention on the interval \( x_{\text{min}} < x < x_{\text{mid}} \) (i.e., replacing \( x_{\text{max}} \) with \( x_{\text{mid}} \)) or on the interval \( x_{\text{mid}} < x < x_{\text{max}} \) (i.e., replacing \( x_{\text{min}} \) with \( x_{\text{mid}} \)), depending on the outcome of the test at step (2).

4. returning to step (1).

The process is illustrated in Fig. 14.7. In the two cases illustrated, the root lies in the upper half of the original interval, so the second iteration will apply the same procedure to the interval from \( x_{\text{mid}} \) to \( x_{\text{max}} \). With each successive iteration, the interval within which we know the root to lie is shrunk to half of its size at the start of that iteration. This method of bisection is guaranteed to converge provided only that there is a root in the interval, though it will not work if the root is a root of even order and it may be confused if the original interval happens to contain several roots. The iteration is continued until the interval has been reduced to a size that we are willing to accept as a tolerance or, though it happens rarely, if \( f_{\text{mid}} \) actually is zero at some point in the process.
14.6.1 Algorithms for Finding Roots Numerically

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., \( \frac{df(x)}{dx}|_{x_n} = f'_n \).

2. extrapolating the tangent line to the graph of the function—a line whose slope is \( f'_n \)—to the point at which it intersects the horizontal axis, i.e., finding \( x_{n+1} \) by requiring it to satisfy

\[
\frac{\Delta f}{\Delta x} = \frac{0 - f_n}{x_{n+1} - x_n} = f'_n \quad \Rightarrow \quad x_{n+1} = x_n - \frac{f_n}{f'_n}
\]  

(14.35)

The relevant geometry is illustrated in Fig. 14.8. Given an initial “guess”, Newton’s method will converge more rapidly than the method of bisection, though it will have difficulties if the initial guess—or, for that matter, the root itself—is too close to a point at which the derivative of the function is zero. With a poor initial guess, the method may diverge altogether or may converge on a root remote from the one sought.

14.6.3 Other Methods

Numerous other methods for finding roots, some of them restricted to polynomials, have been developed. Because available routines in some programs sometimes use methods other than the two described above in detail, we include here a brief outline of the main idea in each of several other methods:

- **The secant method** starts from two approximations to the root, say \( x_1 \) and \( x_2 \), calculates \( f_1 = f(x_1) \) and \( f_2 = f(x_2) \), fits a straight line through the points \((x_1, f_1)\) and \((x_2, f_2)\), finds the intersection point \( x_3 \) of that line with the \( x \) axis, and then repeats the process with the two approximations \( x_2 \) and \( x_3 \). The method is similar to Newton’s method, except that it uses the values of the function at two points to estimate the slope of the tangent line to the function \( f(x) \). It does not require explicit knowledge of the derivative of \( f(x) \).

- **Müller’s method** is mildly more sophisticated than the secant method but involves a similar idea. It starts with three estimates of the root, say \( x_1 \), \( x_2 \), and \( x_3 \), calculates \( f_1 = f(x_1) \), \( f_2 = f(x_2) \), and \( f_3 = f(x_3) \), fits a parabola to the points \((x_1, f_1)\), \((x_2, f_2)\), and \((x_3, f_3)\), finds the roots of that parabola using the form of the quadratic formula given second in Eq. (14.29),
and finally takes as $x_4$—which replaces $x_1$ for the next iteration—the one root that emerges when the ambiguous sign in Eq. (14.29) is chosen so that the denominator has the larger of the two possible absolute values.

- Laguerre’s method, which is limited to polynomials, (1) recognizes that, in terms of its roots $x_i$, a polynomial scaled so that the coefficient of its highest order term is 1 (which has no effect on its roots) can be expressed in the form

$$f(x) = \prod_{i} (x - x_i) \quad (14.36)$$

(2) finds a couple of relationships satisfied by polynomials in this form, (3) assumes with only weak justification a priori that the root sought is isolated from all the others (which are clustered together), and (4) deduces that the next iterate $x_{n+1}$ for the root sought should be determined from the current guess $x_n$ by

$$x_{n+1} = x_n - \frac{N}{G \pm \sqrt{(N-1)(NH-G^2)}} \quad (14.37)$$

where the sign in the denominator is chosen so that the denominator has the larger of the two possible absolute values, $N$ is the order of the polynomial (equal to the number of roots), and $G$ and $H$ are defined by $G = f'/f$ and $H = (f'/f)^2 - f''/f$, each evaluated at $x = x_n$.

### 14.6.4 Assessing Accuracy

As with all numerical operations, assessing the accuracy of the roots found by an algorithm is essential before we can have confidence in the roots. We can, of course, always substitute the root we have found into the function whose root we seek and simply notice how close to zero the value of the function at the proposed root actually is. In the end, that comparison may provide the most important test of accuracy, though whether the value $f(x_{\text{root}}) = 0.0001$, say, is “close enough” to zero requires a judgment that would take into account the magnitude of $f(x)$ over the important domain of $x$ and, even more, the magnitude of the derivative of $f(x)$ evaluated at the root.

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

---

10The 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.
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

  ```python
  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

  ```python
  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

\[
q = \text{so.newton}(f, x0, fprime)
\]

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

\[
A = [\frac{1.0}{10000.0}, \frac{1.0}{200.0}, -\frac{1.0}{500.0}, -0.5]
\]

\[
q = \text{np.roots}(A)
\]

\[
\text{print}(q)
\]

\[
[-48.26826757 -11.08043663 9.3487042]
\]

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

```
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
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

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

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 ±1 × 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 roots. We seek the roots as functions of \( \kappa \), which we shall represent by the variable 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 \( \kappa \) over the interval 0 ≤ \( \kappa \) ≤ 4.0, and we elect to divide that interval into 40 steps of length 0.1 each. The coefficients of the polynomial are, of course, different for each value of \( \kappa \). Let us therefore begin by creating two lists, the first of which contains the 41 values of \( \kappa \) and the second of which—an array with 3 columns and 41 rows—contains in each row the coefficients of the polynomial (seen as a quadratic polynomial in \( \omega^2 \)) for the corresponding value of \( \kappa \). These ends are achieved with the statements
kvals = np.linspace(0.0,4.0,41)
coeff1 = -2.0*(1.0+kvals)
coeff2 = 1.0+2.0*kvals

which set the values of \( \kappa \) 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

\[
s = []
\]

\[
for \ i \ in \ np.arange(41):
    s = np.append(s, [1,coeff1[i], coeff2[i] ] )
\]

\[
s = s.reshape(41,3)
\]

\[
s1 = []
\]

\[
for \ i \ in \ np.arange(41):
    s1 = np.append( s1,np.roots( s[i] ) )
\]

The result is a 2 column \( \times \) 41 row array of real values, the first column containing the first root for each value of \( \kappa \) 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\(^\text{12}\)

\[
\texttt{rts} = \texttt{np.sqrt( np.real(s1).reshape(41,2) )}
\]

\[
\texttt{plt.plot( kvals, rts, linewidth=3, color='black' )}
\]

\[
\texttt{plt.xlabel( '$\kappa$', fontsize=14 )}
\]

\[
\texttt{plt.ylabel( '$\omega$', fontsize=14 )}
\]

\[
\texttt{plt.grid()}
\]

\[
\texttt{plt.ylim( [0.0,3.0] )}
\]

\[
\texttt{plt.tick_params(labelsize=12)}
\]

\[
\texttt{plt.show()}
\]

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 \( \alpha = \frac{b v_0}{m g} \) returns to its initial altitude. In other words, we must find the one non-zero root of the function \( f(\tau) \) 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 \texttt{proj.py}. For example, the upper left frame in Fig. 14.4 could have been created by setting several preliminary variables with the statements

\[
\texttt{thetavals = np.pi*np.linspace(0.0,6.0,7)/12.0} \quad \text{Set } \theta = 0^\circ, 15^\circ, \ldots, 90^\circ \text{ in radians.}
\]

\[
\texttt{tau= np.linspace( 0.0, 2.0, 201 )} \quad \text{Set values of } \tau.
\]

and then executing the statements

\(^\text{12}\)Because the roots returned in this case by \texttt{np.roots} are complex with negligible imaginary part, we extract the real part in the first statement in this set.
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.

![Graph showing natural frequencies as a function of coupling strength.]

Table 14.1: The PYTHON function proj.py.

```python
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
```

execfile('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()

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\(^\text{13}\)

```
alpha = 0.4; theta = np.pi*30.0/180.0  # Set $\alpha$ and $\theta$.
tau0 = 0.2  # Set initial guess for root.
q = so.newton(proj, tau0, args=(alpha,theta) )  # Invoke newton
print(q)  # Display result.
```

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  # Evaluate range for each root.
print( range )  # Display result.
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 consecutive values of $\theta$. *Continuing in the same session with PYTHON*, we prepare for a calculational loop with the statements\(^\text{14}\)

```
thetadeg = np.linspace(0.0, 90.0, 91 )  # Set values of $\theta$ in degrees.
thetarad = np.pi*thetadeg/180.0  # Convert $\theta$ 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

```
soln = []  # Prepare list for values. Set $\alpha$.
alpha=0.4; tau0=0.4  # Set initial guess for root.
for i in np.arange(0,91):
    theta = thetarad[i]  # Find root for each $\theta$.
    rt = so.newton( proj, tau0, args=(alpha, thetarad[i]) )
    soln = np.append(soln, rt )
range4 = np.cos(thetarad[i])*(1-np.exp(-soln))/alpha  # 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 `range1`, `range2`, and `range8`, respectively, we obtain the data to plot Fig. 14.10 with the statements

```
plt.plot( thetadeg, range1, color='black', linewidth=3.0 )
plt.plot( thetadeg, range2, color='black', linewidth=3.0 )
plt.plot( thetadeg, range4, color='black', linewidth=3.0 )
plt.plot( thetadeg, range8, color='black', linewidth=3.0 )
plt.grid()
plt.xlabel('$\theta$ (deg)', fontsize=16)
```

\(^{13}\)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.

\(^{14}\)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).

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 $\text{range}$ to find the maximum range and the approximate firing angle to achieve it in each case. For example, the statements

```
plt.ylabel('\$R(\theta)$', fontsize=16)
plt.xlim([0.0,90.0]); plt.ylim([0.0,1.0])
plt.tick_params(labelsize=12)
plt.show()
```

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

```
tmp = [ np.argmax( range1), np.argmax( range2), np.argmax( range4), np.argmax( range8) ]
print( tmp )
[44, 43, 40, 37]
```

then displays the maximum value and the values on either side of that maximum in $\text{range1}$. 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

15 See an exercise towards the end of the chapter(s) on specific symbol manipulating programs.
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 \texttt{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 \texttt{qmlower} containing the lines

```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 \texttt{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[2] = so.bisect( qmupper, 5.5, 6.5 )
upper[3] = so.bisect( qmupper, 8.5, 9.5 )
upper[4] = so.bisect( qmupper, 11.5, 12.5 )
upper[5] = so.bisect( qmupper, 14.5, 15.5 )
upper[6] = so.bisect( qmupper, 17.5, 18.5 )
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[2] = so.bisect( qmlower, 6.0, 7.0 )
lower[3] = so.bisect( qmlower, 9.0, 10.0 )
lower[4] = so.bisect( qmlower, 12.0, 13.0 )
lower[5] = so.bisect( qmlower, 15.0, 16.0 )
lower[6] = so.bisect( qmlower, 18.0, 19.0 )
lower[7] = so.bisect( qmlower, 21.0, 22.0 )
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 12.11879409 15.06138916
  19.7610887  20.99428643]
print( lower )
[ 0.  3.27288492  6.03920377  9.8288367  12.06284802 16.42478429
  18.04325707  23.17777839  23.86449494]
```

Remember, however, that \textit{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
14.11 Finding Roots Numerically with MAXIMA

MAXIMA makes available several functions for finding roots numerically. For polynomials, the available functions include

\begin{verbatim}
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

\begin{verbatim}
s = lower[ 1:9 ]
for i in [1,3,5,7]: s[i-1]=upper[i]
print( s )
  20.99428643 23.86449494]

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

\begin{verbatim}
energy = -( 1 - s**2./25.0**2 )
print( energy )
[-0.98540274 -0.94164483 -0.86883475 -0.76718032 -0.63704729 -0.4791054
  -0.2947839 -0.08877741]
\end{verbatim}

The statements

\begin{verbatim}
for i in np.arange(8):
    plt.plot( [0.2, 1.0], [energy[i],energy[i]], color='black',
        linewidth=2.0 )
plt.box(on=None)
plt.plot( [0.0,1.2,1.2,1.5], [-1.0,-1.0,0.0,0.0], color='black' )
plt.plot( [0.0,0.0], [-1.5,0.5], color='black' )
plt.ylabel('$E/V_0$', fontsize=16 )
plt.tick_params(axis='x', which='both', bottom=False, top=False, 
    labelbottom=False )
plt.tick_params(labelsize=14)
\end{verbatim}

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.

14.11 Finding Roots Numerically with MAXIMA
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.

- `nroots( f, a, b )`, which returns the number of real roots of the polynomial $f$ lying in the range $a < x \leq b$.
- `realroots( f, eps )`, which returns values for all of the real roots possessed by the polynomial $f$, calculating each to an absolute precision specified by `eps` (default in `rootsepsilon`, which will be $10^{-7}$ unless explicitly changed), and
- `allroots( f )`, which returns single-precision values for all of the roots (real and complex) possessed by the polynomial $f$.
- `bfallroots( f )`, which returns bigfloat values for all of the roots (real and complex) possessed by the polynomial $f$.

Note that `nroots` and `realroots` can deal properly with multiple roots and that `realroots` sets the system variable `multiplicities` to a list of the multiplicities of each root found, but that `allroots` may have difficulties dealing with multiple roots. Further details can be found in the MAXIMA manuals.

MAXIMA also contains functions for finding the roots of more general univariate functions, which—of course—includes polynomials as a subclass. We mention in particular the three functions

- `find_root( expr, x, a, b, [abserr, relerr] )`, which, with `expr` an expression depending on $x$, returns a root of `expr = 0` in the closed interval $[a, b]$ with absolute error `abserr` (default in `find_root_abs`, which will be zero unless explicitly changed) and relative error `relerr` (default in `find_root_rel`, which will be zero unless explicitly changed). Here, `abserr` and `relerr` can be presented in either order and are entered in the form `abserr = value` and `relerr = value`. This function uses the binary search method but will switch to linear interpolation if it determines that the function is smooth enough. The square brackets simply indicate that these quantities are optional, not that they are values in a list.
• \texttt{bf\_find\_root}(expr, x, a, b, \{\text{abserr}, \text{relerr}\}), which is the same as \texttt{find\_root} but returns a bigfloat value.

• \texttt{newton}(expr, x, x_0, eps), which, with \texttt{expr} an expression depending on \texttt{x}, returns a root of \texttt{expr} = 0 by Newton’s method, iterating from the starting point provided by \texttt{x_0} and continuing until |\texttt{expr}| < \texttt{eps}. There is no default value for \texttt{eps}. The package \texttt{newton1} must be loaded to make this command available.

Simply for peace of mind, let us test the first set of functions with a polynomial that has a multiple root. We execute the statements

\begin{verbatim}
(%i11) pl : expand( (x-1)^2*(x+3)*(x-4) )
(%i12) nroots( pl, -10, 10);
(%o12) 4
(%i13) realroots( pl, 0.001);
(%o13) [x = 1, x = -3, x = 4]
(%i14) multiplicities;
(%o14) [ 2, 1, 1 ]
\end{verbatim}

Create polynomial with multiple roots. Test \texttt{nroots}. Find roots with \texttt{realroots}. Display multiplicities.

Clearly, \texttt{nroots} counted all four roots of the given polynomial even though one of the three distinct roots is a double root, while \texttt{realroots} reports only the distinct roots. We must examine the system variable \texttt{multiplicities} to find that one of them is a double root. As illustrated, however, in the statements

\begin{verbatim}
(%i15) allroots(pl);
(%o15) [x = 1.0, x = 1.0, x = -3.0, x = 4.0]
(%i16) bfallroots(pl);
(%o16) [x = -3.060, x = 1.060, x = 1.060, x = 4.060]
\end{verbatim}

both \texttt{allroots} and \texttt{bfallroots}, on the other hand, report the double root twice. Evidently, this specific (double-rooted) polynomial is not one with which \texttt{allroots} or \texttt{bfallroots} has trouble.

Finally, introducing a non-polynomial function $xe^{-x^2} - x^2 + 10$, for which a graph reveals a root in the interval $2 < x < 4$, we test the second set of functions with the statements

\begin{verbatim}
(%i17) expr : x*exp(-x^2) - x^2 + 10$
(%i18) find_root( expr, x, 2.0, 4.0 );
(%o18) 3.1623003569056242
(%i19) expr, x = %;
(%o19) 1.7604... \times 10^{-16}
(%i20) bf\_find\_root( expr, x, 2.0, 4.0 );
(%o20) 3.1623003569056242
(%i21) expr, x = %;
(%o21) 1.7597... b - 16
(%i22) load(newton1)$
(%i23) newton( expr, x, 2.0, 1.0e-7 );
(%o23) 3.162300357067024
(%i24) expr, x = %;
(%o24) 1.7597... \times 10^{-10}
\end{verbatim}

Define expression. Invoke \texttt{find\_root}. Verify root. Invoke \texttt{bf\_find\_root}. Verify root. Load \texttt{newton1} package. Invoke \texttt{newton}. Verify root.

As implied by the verifications on %o9 and %o11, the roots on %o8 and %o10 satisfy $xe^{-x^2} - x^2 + 10 = 0$ to within a few parts in $10^{16}$, consistent with the default absolute and relative errors of 0.0. The root given by \texttt{newton}, however, satisfies the equation to only about one part in $10^9$, consistent with the specified error in the call to \texttt{newton}. 

14.11. FINDING ROOTS NUMERICALLY WITH MAXIMA

363
### 14.11.1 Classical Turning Points

To find the turning points for the potential energy in Eq. (14.3), we begin by defining the polynomial and asking about the number of roots with the statements

```plaintext
(%i1) V : x^3/10000 + x^2/200 - x/500 - 1/2
(%i2) nroots( V, -60, 20 );
(%o2) 3
```

Define polynomial of Eq. (14.3).

Determine number of roots in \(-60 < x < 20\).

Then, we find the roots with the statements

```plaintext
(%i3) float( realroots( V, 0.001 ) );
(%o3) [ x = -48.268, x = -11.081, x = 9.349 ]
(%i4) multiplicities;
(%o4) [ 1, 1, 1 ]
(%i5) float( realroots( V ) );
(%o5) [ x = -48.2682675, x = -11.0804366, x = 9.3487042 ]
```

Invoke `realroots` with `eps = 0.001`. Output is rounded to three digits after the decimal point.

Display multiplicity of each root.

Invoke `realroots` with default `eps` of \(10^{-7}\). Output is rounded to seven digits after the decimal point.

The three different commands yield consistent though differently formatted roots.

Alternatively, since a polynomial is simply a subclass of a general function, we might seek the turning points by remembering the bounds determined in Eq. (14.4) and invoke the statements

```plaintext
(%i6) find_root( V, x, -50.00, -47.50 );
(%o6) -48.2682675675163
(%i7) find_root( V, x, -12.50, -10.00 );
(%o7) -11.08043663084842
(%i8) find_root( V, x, 7.50, 10.0 );
(%o8) 9.348704197600052
(%i9) V, x = [ %o6, %o7, %o8 ];
(%o9) [ 0.0, 0.0, -1.11022... × 10^{-16} ]
```

Find lowest root.

Find middle root.

Find highest root.

Verify roots.

In these statements, we accept the default tolerances (both zero), so the iteration stopped when either the root or the function stayed the same (to the precision of double precision calculations) for two successive iterations. The output at `%o9` is consistent with an error no larger than the 16-digit precision of double precision calculations.

Newton’s method gives the same results for the roots, though we only need one starting value.

The statements

```plaintext
(%i10) load( newton1 )$
(%i11) newton( V, x, -50.0, 1.0e-7 );
(%o11) x = -48.2682678
(%i12) newton( V, x, -10.0 );
(%o12) x = -11.0804366
(%i13) newton( V, x, 10.0 );
(%o13) x = 9.3487042
```

find the three roots.

### 14.11.2 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
14.11. FINDING ROOTS NUMERICALLY WITH MAXIMA

words, we must find the one non-zero root of the function $f(\tau)$ defined in Eq. (14.18). Finding that root and then the range for a particular choice of $\alpha$ and $\theta$ is quickly accomplished in MAXIMA with the statements

\[
\begin{align*}
%i1 & : \text{range : } \cos(\theta) \cdot (1 - \exp(-\tau))/\alpha; \quad \text{Define range [Eq. (14.19)].} \\
%i2 & : \text{fct : } \tau - (1 + \alpha \sin(\theta)) \cdot (1 - \exp(-\tau)); \quad \text{Define function [Eq. (14.18)].} \\
%i3 & : \text{[ alpha : 0.4, theta : float(\pi*30.0/180.0) ]; } \text{Choose } \alpha, \theta. \\
%i4 & : \text{load( newton1 );} \\
%i5 & : \text{newton( ev(fct), tau, 0.5, 1.0e-7 );} \quad \text{Find root.} \\
%i6 & : \text{ev( range, tau = % );} \quad \text{Evaluate range.}
\end{align*}
\]

This result is expressed in units of $v_0^2/g$.

To find the angle of fire to attain maximum range, we need to repeat the sample calculation of the previous paragraph for values of $\theta$ ranging from $0^\circ$ to $90^\circ$, choosing, say, increments of $1^\circ$ between consecutive values of $\theta$. We can actually accomplish all of these operations in a single loop with the statements\footnote{MAXIMA's construction \texttt{block(...) allows us to group several statements together so they will be seen as a single statement. In that grouping, individual statements are separated by commas.}}

\[
\begin{align*}
%i7 & : \text{for } i: 0 \text{ thru } 90 \text{ do block} \\
 & \quad ( \text{thetadeg}[i] : i, \theta : \text{float(\pi*i/180.0 )}, \\
 & \quad \text{tmp : newton( ev(fct), tau, 0.8, 1.0e-7 ),} \\
 & \quad \text{rng}[i] : \text{ev( range, tau = tmp) } )$
\end{align*}
\]

(\text{Remember that } \alpha \text{ was set to 0.4 at } %i3.) Then, we produce the graph in Fig. 14.12 with the statements

\[
\begin{align*}
%i8 & : \text{ang : makelist( thetadeg[i], i, 0, 90 );} \\
%i9 & : \text{rg : makelist( rng[i], i, 0, 90 );} \\
%i10 & : \text{plot2d( [discrete, ang, rg], [xlabel, "theta (deg)"],} \\
 & \quad [ylabel, "R(theta)"], [color, black], [style, [lines, 4] ] )$
\end{align*}
\]

Further, rounding to seven digits after the decimal point, by noting the three values,

\[
\begin{align*}
%i11 & : \text{[ rng[39], rng[40], rng[41] ];} \\
%o11 & : [0.7251743, 0.7259486, 0.7258890]
\end{align*}
\]

we conclude that the maximum range for $\alpha = 0.4$ occurs near $\theta = 40^\circ$ (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).

\text{Had we started with some other value of } \alpha, \text{ of course, statements identical to those in %i3 and %i7-%i10 would produce a graph corresponding to that other value of the viscous damping.}
14.11.3 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). We elect the method of bisection, first creating lists of the lower and upper bounds given in Eqs. (14.26) and (14.27). To that end we execute the statements\(^{17}\)

\[
\begin{align*}
(\%i1) \quad \text{ulb} & : [-0.5, 2.5, 6.0, 8.5, 12.5, 14.5, 19.0, 20.5] \\
(\%i2) \quad \text{uub} & : [0.5, 3.5, 7.0, 9.5, 13.5, 15.5, 20.0, 21.5] \\
(\%i3) \quad \text{llb} & : [-0.5, 3.0, 5.5, 9.5, 11.5, 16.0, 17.5, 22.5, 23.5] \\
(\%i4) \quad \text{lub} & : [0.5, 4.0, 6.5, 10.5, 12.5, 17.0, 18.5, 23.5, 24.5]
\end{align*}
\]

Then, we find the roots in each category with the simple loops

\[
\begin{align*}
(\%i5) \quad \text{for } i : 1 \text{ thru } 8 \text{ do} \\
& \quad \text{rootu}[i] : \text{find_root}( \sin(s)-0.04*s, s, \text{ulb}[i], \text{uub}[i] ); \\
(\%i6) \quad \text{for } i : 1 \text{ thru } 9 \text{ do} \\
& \quad \text{rootl}[i] : \text{find_root}( \sin(s)+0.04*s, s, \text{llb}[i], \text{lub}[i] );
\end{align*}
\]

Then we concatenate the elements into lists and display them with the statements

---

\(^{17}\)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, \text{ulb} will contain the lower bounds for the roots associated with the upper sign.
14.11. FINDING ROOTS NUMERICALLY WITH MAXIMA

(%i7) rtu : makelist( rootu[i], i, 1, 8 );
(%o7) [0.0, 3.02047..., 6.54820..., 9.05418..., 13.11879..., 15.06138..., 19.76108..., 20.99428...]
(%i8) rtl : makelist( rootl[i], i, 1, 9 );
(%o8) [0.0, 3.27288..., 6.03920..., 9.82883..., 12.06284..., 16.42478..., 18.04325..., 23.17777..., 23.86449...]

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. Because \( \cot(0.0) \), which will appear when Eq. (14.23) is evaluated for the first root in \( rtu \) and \( rtl \), is undefined, however, we must begin by removing that first root in each list. The appropriate statements to MAXIMA are

(%i9) rtu : rest(rtu)$
(%i10) rtl : rest(rtl)$

(%i11) rtutest : rtu*cot(rtu)+sqrt(25.0^2-rtu^2);
(%o11) [−1.42108... \times 10^{−14}, 48.25436... − 3.55271... \times 10^{−15},
42.56276..., 3.552713... \times 10^{−15}, 30.62674..., −4.79616... \times 10^{−14} ]

(%i12) rtltest : rtl*cot(rtl)+sqrt(25.0^2-rtl^2);
(%o12) [ 49.56967... \times 10^{−15}, 3.55271... \times 10^{−14}, 45.97364..., −2.84217... \times 10^{−14},
37.69490..., 4.97379... \times 10^{−14}, 18.73932, 1.95399... \times 10^{−14} ]

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 \( rtu[2], rtu[4], rtu[6], rtl[1], rtl[3], rtl[5], \) and \( rtl[7] \). Thus, we assemble a single list containing the physically meaningful roots with the MAXIMA statements

(%i13) s : rest( rtl, 1 )$
(%i14) for i : 1 thru 7 step 2 do s[i] : rtu[i]$
(%i15) s;
(%o15) [3.0204776..., 6.0392037..., 9.0541858..., 12.0628480..., 15.0613891..., 18.0432535..., 23.1777700...]

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 statements

(%i16) energy : -( 1 - s^2/25.0^2 );
(%o16) [−0.98540, −0.94164, −0.86883, −0.76718, −0.63705, −0.47911, −0.29478, −0.08878]

where we have rounded the output to five digits after the decimal point. The statements

(%i17) for i : 1 thru 8 do en[i] : [ energy[i], energy[i] ]$
(%i18) rng : [0.2, 0.8]$

and then plot the energy-level diagram of Fig. 14.13 with the statement
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 $V_0$. Other lines in the diagram show the contour of the well, with energy $E/V_0 = -1$ at the bottom and energy $E/V_0 = 0$ at the top.

where the last list invoking the plot option `discrete` draws the boundaries of the potential well.

### 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. 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.

---

18 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. SOLVING SIMULTANEOUS EQUATIONS

14.16.1 Systems of Linear Equations

By far the simplest situation occurs when the equations in the system are all linear, i.e., when the unknowns in the system occur only to the first power and never in product with one another. As illustrated in the exercises at the end of this chapter, Kirchoff’s laws in DC circuit theory, least squares fitting of polynomials to experimental data, and some boundary value problems are among the contexts in which linear systems—sometimes quite large linear systems—of equations arise.

Whatever their physical origin, systems of linear equations can most conveniently be presented in matrix form, e.g.,

\[
\begin{pmatrix}
  a_{11} & a_{12} & a_{13} \\
  a_{21} & a_{22} & a_{23} \\
  a_{31} & a_{32} & a_{33}
\end{pmatrix}
\begin{pmatrix}
  x_1 \\
  x_2 \\
  x_3
\end{pmatrix}
= 
\begin{pmatrix}
  b_1 \\
  b_2 \\
  b_3
\end{pmatrix}
\]

(14.38)

where, with \( i \) and \( j \) independently assuming the values 1, 2, and 3, the quantities \( x_i \) are the unknowns, the quantities \( b_i \) are the inhomogenieties, and the quantities \( a_{ij} \) are the coefficients defining the equations. Here, we have three equations and three unknowns. More generally, for a system of \( n \) equations and \( n \) unknowns, we would have an \( n \times n \) matrix \( A \) of coefficients, an \( n \) element vector \( x \) of unknowns and an \( n \) element vector \( b \) of inhomogenieties, we would write the equations compactly in the form

\[ Ax = b \]

(14.39)

and we would write their solution formally as

\[ x = A^{-1}b \]

(14.40)

Less compactly but more usefully (at least occasionally), we might remember Cramer’s rule and write the solution in terms of determinants in the form

\[
x_1 = \frac{b_1 a_{12} a_{13} - b_2 a_{11} a_{13} + b_3 a_{11} a_{12}}{a_{11} a_{22} a_{33} - a_{12} a_{23} a_{31} + a_{13} a_{21} a_{32}}, \quad x_2 = \frac{a_{11} b_1 a_{13} - a_{12} b_1 a_{13} + a_{13} b_1 a_{12}}{a_{11} a_{22} a_{33} - a_{12} a_{23} a_{31} + a_{13} a_{21} a_{32}}; \quad x_3 = \frac{a_{11} a_{12} b_1 - a_{11} a_{12} b_2 + a_{11} a_{12} b_3}{a_{11} a_{22} a_{33} - a_{12} a_{23} a_{31} + a_{13} a_{21} a_{32}}
\]

(14.41)

In this rule, which can be readily extended to \( n \) equations, the denominator in the expression for the \( i \)-th unknown is the determinant of the coefficient matrix and the numerator is the determinant of the matrix created from the coefficient matrix by replacing its \( i \)-th column with the column of inhomogenieties. Cramer’s rule provides a direct, symbolic, and exact solution to the system of logarithmic 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; \( b \neq 0 \)) or an infinite number of solutions (equations homogeneous; \( b = 0 \)).

While compact, Cramer’s rule is not particularly useful for numerical solution of even modest sized systems, since the most direct approach to evaluating determinants is vulnerable to roundoff error.\(^{20}\) 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. (14.38), but the schema is readily extended to \( n \) equations. The process of Gaussian elimination entails

\(^{20}\)The most direct approach involves sums and differences of products of \( n \) elements taken so that each column and each row is represented once and only once in each product. The result is a combination of terms that are individually large, some of which are positive and some negative. We end up trying to evaluate a difference between two large numbers, an operation that invites roundoff error.
• dividing each equation by the coefficient of \( x_1 \), obtaining\(^{21} (14.42) \)

\[
\begin{pmatrix}
1 & a_{12} & a_{13} \\
1 & a_{22} & a_{23} \\
1 & a_{32} & a_{33}
\end{pmatrix}
\begin{pmatrix}
x_1 \\
x_2 \\
x_3
\end{pmatrix}
= 
\begin{pmatrix}
b'_1 \\
b'_2 \\
b'_3
\end{pmatrix}
\]

• keeping the first equation and replacing the second and third with the result of subtracting the first from each in turn, obtaining

\[
\begin{pmatrix}
1 & a'_{12} & a'_{13} \\
0 & a''_{22} & a''_{23} \\
0 & a'''_{32} & a'''_{33}
\end{pmatrix}
\begin{pmatrix}
x_1 \\
x_2 \\
x_3
\end{pmatrix}
= 
\begin{pmatrix}
b'_1 \\
b'_2 \\
b'_3
\end{pmatrix}
\]

(14.43)

• dividing the second and third equations by the coefficient of \( x_2 \), obtaining

\[
\begin{pmatrix}
1 & a'_{12} & a'_{13} \\
0 & 1 & a''_{23} \\
0 & 1 & a'''_{33}
\end{pmatrix}
\begin{pmatrix}
x_1 \\
x_2 \\
x_3
\end{pmatrix}
= 
\begin{pmatrix}
b'_1 \\
b''_2 \\
b'''_3
\end{pmatrix}
\]

(14.44)

• keeping the first and second equations but replacing the third with the result of subtracting the second from the third, obtaining

\[
\begin{pmatrix}
1 & a'_{12} & a'_{13} \\
0 & 1 & a''_{23} \\
0 & 0 & a'''_{33}
\end{pmatrix}
\begin{pmatrix}
x_1 \\
x_2 \\
x_3
\end{pmatrix}
= 
\begin{pmatrix}
b'_1 \\
b''_2 \\
b'''_3
\end{pmatrix}
\]

(14.45)

In essence, Gaussian elimination converts the original system of equations with a general coefficient matrix into an equivalent system whose coefficient matrix is 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\(^{22} \). That is, we can write

\[
A = LU \quad \text{where} \quad L = \begin{pmatrix} 1 & 0 & 0 \\ l_{12} & 1 & 0 \\ l_{13} & l_{23} & 1 \end{pmatrix} \quad \text{and} \quad U = \begin{pmatrix} u_{11} & u_{12} & u_{13} \\ 0 & u_{22} & u_{23} \\ 0 & 0 & u_{33} \end{pmatrix}
\]

(14.46)

\(^{21}\)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

\[
Ax = b \iff LUx = b \iff Ly = b \quad \text{where} \quad y = Ux
\]  

(14.47)

Since the first equation in the (matrix) equation \(Ly = b\) tells us \(y_1\) directly, the second tells us \(y_2\) directly once \(y_1\) is known, and the third tells us \(y_3\) directly once \(y_1\) and \(y_2\) are known, the equation \(Ly = b\) is readily solved for \(y\). Then, however, once \(y\) is known, a similar process that starts with \(x_3\), then moves to \(x_2\) and finally to \(x_1\) directly solves the equation \(Ux = y\) for the original unknowns in \(x\).

The methods outlined in the previous paragraphs will all work for (almost) any system of linear equations. Sometimes, however, the coefficient matrix may have special properties that can be exploited to simplify the algorithm or—important for large systems—reduce the requirements on memory for storage of matrices and intermediate results. The coefficient matrix may be symmetric \((a_{ij} = a_{ji})\) or it may be sparse (only a small fraction of its elements differing from zero). In the latter category, the matrix may be tridiagonal (non-zero elements only on the main diagonal, on the diagonal immediately above the main diagonal, and on the diagonal immediately below the main diagonal). Special savings in storage can be achieved if the matrix happens to be both symmetric and tridiagonal, since in that case the only elements that need be stored are those on the main diagonal and those on the diagonal immediately above the main diagonal, \(2n - 1\) elements rather than \(n^2\) elements.

The methods outlined in the previous paragraphs are also all direct methods, i.e., each leads directly to the desired solution in a finite number of steps. Except for roundoff error, each would yield an exact solution to the equations at hand. When the coefficient matrix is sparse, an iterative approach may be computationally more efficient. Such approaches entail finding a means by which an initial guess can be repeatedly refined until some criterion of convergence has been met. Among the most common such procedures applies to Laplace’s equation. As judiciously as possible, we “guess” a solution at a regular grid of points laid over the domain of the problem. Then, to carry out the first iteration, we step systematically through that grid, replacing the value at each grid point with the average of the values at its nearest neighbors. Then, we repeat the process with the results of the first iteration as input, generating the second iteration, and continuing until—say—no value changes by more than some specified tolerance. The only drawback to an iterative method is that we have added the uncertainties associated with convergence to those potentially generated by computer roundoff. In terms of computational labor, however, the sacrifice is often worth the gains.

Finally, we merely mention the more sophisticated approach—singular value decomposition—we must adopt if the system of equations confronting us happens to be nearly singular, in which case the presence of computer roundoff introduces instabilities in the simpler methods.\(^\text{23}\)

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.

**MAXIMA:** Within MAXIMA, solving systems of linear equations is accomplished with the commands `linsolve` and `solve` (which actually calls `linsolve` if the equations are linear). These commands will produce symbolic solutions but will also produce numerical results to the extent permitted by any numbers explicitly present in the equations. The commands 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.

CHAPTER 14. FINDING ROOTS

Figure 14.14: Zero contours for the two functions in Eq. (14.49). 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.

14.16.2 Systems of Nonlinear Equations

When one or more of the equations in a system to be solved is nonlinear, the task is much more difficult. Occasionally, systematic elimination of one variable at a time, followed by back-substitution, will yield a solution. More often, the system is not amenable to such a simple approach. We must resort to more involved approaches. Fitting experimental data by the method of least squares to functions in which the parameters do not appear linearly and finding the points of equilibrium in a system with more than one independent variable are common sources of such problems.

Since numerical methods for finding the root (or roots) of a nonlinear system of equations are all iterative, possession of a good starting guess is imperative. In two dimensions, where the equations to be solved are

\[
\begin{align*}
    f_1(x_1, x_2) &= 0 \\
    f_2(x_1, x_2) &= 0
\end{align*}
\]  

(14.48)

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

\[
\begin{align*}
    f_1(x_1, x_2) &= \sin \left( \frac{x_1^2}{20} \right) - \cos \left( \frac{x_2}{5} \right) \\
    f_2(x_1, x_2) &= x_1 \tanh(x_2) - 5.0
\end{align*}
\]  

(14.49)

The actual intersections of the dashed and solid curves near the labels A reveal two roots. In addition, the dashed and solid curves pass close to one another—but do not actually intersect—in the vicinity of the labels B. The roots near A can probably be found relatively easily by an iterative search procedure. That there are "almost" roots near B may confuse some algorithms and, if those points are close enough to the real roots near A, they might even cause difficulties in finding the real roots.

As the number of independent variables increases, the search described in the previous paragraph would move from intersection points of curves in a plane to intersection points of three surfaces in three-space to intersection points of four hypersurfaces in four-space to .... Sometimes it may be
possible (and wise) to solve for some of the variables in terms of the others and temporarily eliminate
some variables (i.e., reduce the dimensionality of the search). The task is complicated and, beyond
the simple suggestion of striving to reduce the dimensionality, no general guidelines can be given.
Any means, however devious, that can be exploited to give clues as to the existence of roots and,
even better, to their whereabouts should be exploited as fully as possible before actually embarking
on a numerical search.

Once a root has been located approximately, we might adopt a brute force technique, writing a
program that

1. Accepts a guessed solution, one value for each unknown,
2. Evaluates the functions and displays the result, and
3. Returns to step 1 for a new guess.

On first pass, we enter the initial guess. Then, after seeing how well that guess works, we make a
second (informed or, maybe, random) guess, repeating the process until the values of all functions
have been reduced to acceptably small levels. Depending on the dimensionality of the search, we
will usually develop a feel for the effect of changes in each member of the guessed solution. Fairly
quickly, we may develop a skill at narrowing in on an acceptable solution.

More systematic searches in multi-dimensional parameter spaces are harder to design. One
route in particular expands Newton’s method. Suppose, to be specific, we seek solutions to the
three nonlinear equations

\[ f_1(x_1, x_2, x_3) = 0 \quad ; \quad f_2(x_1, x_2, x_3) = 0 \quad ; \quad f_3(x_1, x_2, x_3) = 0 \]  \hspace{1cm} (14.50)

Suppose, further, that we have examined the equations and determined that there does indeed exist
a root in the immediate vicinity of the point \((x_1^{(0)}, x_2^{(0)}, x_3^{(0)})\). We might then suppose that the actual
root differs from this guess by a small amount, say,

\[ x_1 = x_1^{(0)} + \delta x_1 \quad ; \quad x_2 = x_2^{(0)} + \delta x_2 \quad ; \quad x_3 = x_3^{(0)} + \delta x_3 \]  \hspace{1cm} (14.51)

and demand that the “corrections” satisfy the equations

\[ 0 = f_1\left(x_1^{(0)} + \delta x_1, x_2^{(0)} + \delta x_2, x_3^{(0)} + \delta x_3\right) \]
\[ 0 = f_2\left(x_1^{(0)} + \delta x_1, x_2^{(0)} + \delta x_2, x_3^{(0)} + \delta x_3\right) \]  \hspace{1cm} (14.52)
\[ 0 = f_3\left(x_1^{(0)} + \delta x_1, x_2^{(0)} + \delta x_2, x_3^{(0)} + \delta x_3\right) \]

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 approx-
imate 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} (14.53)
\[ 0 = f_3 + \frac{\partial f_3}{\partial x_1} \delta x_1 + \frac{\partial f_3}{\partial x_2} \delta x_2 + \frac{\partial f_3}{\partial x_3} \delta x_3 \]

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.

**MAXIMA:** For symbolic solution of systems of nonlinear—but polynomial—equations, MAXIMA provides the commands `algsys` and `solve` (which actually calls `algsys` if the equations are nonlinear). For numerical solutions, MAXIMA provides the command `newton`, which can be supplied with a list of equations as input and is able to address systems as well as individual equations.

**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.\(^{24}\)

## 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

\[ f(\tau) = \tau - (1 + \alpha \sin \theta)(1 - e^{-\tau}) = 0 \]

\(^{24}\)See particularly the URL [docs.scipy.org/doc/scipy/reference/optimize.nonlin.html](https://docs.scipy.org/doc/scipy/reference/optimize.nonlin.html).
where $\alpha = b v_0 / (m g)$, and then (2) evaluating the range from the expression

$$\frac{R(\theta, \alpha)}{v_0^2 / g} = \frac{\cos \theta (1 - e^{-\tau})}{\alpha} = \frac{\tau \cos \theta}{\alpha (1 + \alpha \sin \theta)}$$

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

$$m \frac{d^2 x_1}{dt^2} = -k x_1 + k(x_2 - x_1)$$

$$m' \frac{d^2 x_2}{dt^2} = -k(x_2 - x_1) + k(x_3 - x_2)$$

$$m \frac{d^2 x_3}{dt^2} = -k(x_3 - x_2) - k x_3$$

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
14.1.7.2 . . . using Numerical Methods

14.11. Use numerical methods to solve Eq. (14.6) for the equilibrium points in the potential energy in Eq. (14.3).

14.12. One way to find the square root of a (positive) number \( a \) is to find the root of the function \( f(x) = x^2 - a \). (a) Apply Newton’s method symbolically to show that \( x_{n+1} = (x_n + a/x_n)/2 \).
(b) Using a pocket calculator and starting with the guess \( x_0 = 2 \), work out the first few iterates by hand and note how quickly this algorithm converges to \( \sqrt{2} = 1.41421 \). (This algorithm is the algorithm that most pocket calculators invoke when the square root key is pressed.) (c) Using whatever computational tool appeals to you, write a program that asks for the value of \( a \), an initial guess for \( \sqrt{a} \), and a tolerance and then implements Newton’s method to find \( \sqrt{a} \), printing out each iterate along the way and stopping automatically when successive iterates differ by less than the specified tolerance.

14.13. For the first six Legendre polynomials \( L_n(x) \), find all roots lying in the interval \(-1 \leq x \leq 1\). Those polynomials are

\[
\begin{align*}
L_0(x) &= 1 \\
L_1(x) &= x \\
L_2(x) &= \frac{1}{2}(3x^2 - 1) \\
L_3(x) &= \frac{1}{2}(5x^3 - 3x) \\
L_4(x) &= \frac{1}{8}(25x^4 - 15x^2 + 3) \\
L_5(x) &= \frac{1}{8}(35x^5 - 21x^3 + 5x)
\end{align*}
\]

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? \textit{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\textsuperscript{25} are 2.404825577, 5.5200781103, 8.6537279129, 
11.7915344391, 14.9309177086, 18.0710639679. \textit{Hint:} Most computational tools have built-in ca-

pabilities for evaluating the Bessel functions. Consult the appropriate vendor manuals.

14.20. Suppose a particle moves in one dimension under the influence of the potential energy 
\[
V(x) = \frac{-V_0a^2(a^2 + x^2)}{8a^4 + x^4} \quad \Rightarrow \quad \frac{V(x)}{V_0} = \frac{1 + \frac{x^2}{a^2}}{8 + \frac{x^4}{a^4}}
\]
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 \(E = -0.1V_0\). \textit{Optional:} Obtain graphs of the position of each 
turning point as a function of particle energy over the allowed range of energies for bound states.

14.21. Generate a graph showing the turning points of the potential energy given by Eq. (14.3) as a 
function of the energy of the particle.

14.22. Suppose a straight railroad track of length 1 mile (5280 ft) is 
held absolutely immovable at its two ends. On a hot summer 
day, the track expands in length by 1 ft. If the track bows 
upward from the earth in a circular arc, how high above the 
earth will the track rise at its midpoint? \textit{Hint:} The geometry of 
this exercise is shown in the accompanying figure, where \(l\) 
is the original length of the track, \(d\) is the rise at the center, 
and \(a\) is the length, \(R\) the radius, and \(\theta\) half the 
subtended angle of the circular arc. Thus, \(R\theta = a/2\), \(\sin \theta = l/(2R)\), and 
\(R - d = R \cos \theta\). The task is to find \(\theta\) and \(R\) from the first two 
of these equations and then use the third to find \(d\). The only 
quantities known \textit{a priori} are \(l\) and \(a\).
14.23. A particle of mass \( m \) moves in a potential energy given as a function of position by \( V(x) = V_0 \cosh(x/a) \). Because this potential energy is an even function of \( x \), the upper and lower turning points have the same absolute value but opposite sign. Find the upper turning point as a function of energy and generate a graph showing that turning point as a function of energy. \( \text{Hint: Measure position in units of } a \) and energy in units of \( V_0 \).

14.24. A particular problem—see Problem 3-19 in the fourth edition of Fluid Flow by Rolf H. Sabersky, Allan J. Acosta, Edward G. Hauptmann, and E. M. Gates (Prentice-Hall, Upper Saddle River, NJ, 1999)—in fluid flow leads to the need to find the roots of the fourth-order polynomial \( 12x^4 - 12x^3 + 4x - 1 \). Use graphical methods to find bounds on the roots and at least three different computational approaches to find all real roots of this polynomial.

14.27. The image of a distant object produced on a viewing screen by a small aperture is actually a diffraction pattern. When the aperture is a circle of diameter \( d \) and light from the object strikes the screen containing the aperture at normal incidence, the intensity in the diffraction pattern at angle \( \theta \) from the normal is given by

\[
I(\theta) = \frac{I_0}{\pi d^2} \left( \frac{2 J_1(\pi d \sin \theta / \lambda)}{\pi d \sin \theta / \lambda} \right)^2
\]

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. \( \text{Hint: The angle will be small, so you can safely use the approximation } \sin \theta \approx \theta. \)

14.28. The Lennard-Jones potential energy \( V_{LJ} \), which is given in terms of the coordinate \( r \) by the expression

\[
V_{LJ} = 4\epsilon \left( \frac{\sigma^{12}}{r^{12}} - \frac{\sigma^{6}}{r^{6}} \right)
\]

or

\[
V_{LJ} = \frac{4}{\epsilon} \left[ \frac{1}{r^{12}} - \frac{1}{r^{6}} \right]
\]

where \( \epsilon \) and \( \sigma \) are constants and \( r = r/\sigma \) 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.

14.17.4 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. \( \text{Optional: Extend your entire consideration into three dimensions, which will require knowledge of distance from each of (at least) three known locations. } \text{Hint: Quite a bit of information is available on the website } \text{www.trimble.com/gps. There is also an article in the January, 1994, issue of Physics Today ("Where I Stand" by Daniel Kleppner, page 9).} \)
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

\[
R(a,b,c) = \sum_{i=1}^{n} \left( x_i - (at_i^2 + bt_i + c) \right)^2
\]

i.e., as the particular values satisfying the three equations

\[
\frac{\partial R}{\partial a} = 0 ; \quad \frac{\partial R}{\partial b} = 0 ; \quad \frac{\partial R}{\partial c} = 0
\]

These equations will turn out to be linear in \( a, b, \) and \( c, \) with coefficients and inhomogenieties determined by sums of various products of the measured independent and dependent variables. Derive the three equations symbolically but then find the numerical values of the coefficients and inhomogenities for the data in the file \$HEAD/data/freefall.dat \and use at least two different numerical approaches to find the solution for \( a, b, \) and \( c \). Finally, generate a graph in which each measured point is represented by a simple symbol and the least squares parabola is shown by a solid line so you can judge the adequacy of your fit.

14.38. A particle of mass \( m \) moves along the \( x \) axis under the action of a time-dependent force \( F(t) \). We observe that \( x(0) = 0 \) and \( x(t_f) = a \). The detailed motion therefore is described by the solution to the boundary value problem

\[
m \frac{d^2x}{dt^2} = F(t) , \quad x(0) = 0 , \quad x(t_f) = a
\]

To predict the detailed motion numerically, we might divide the interval \( 0 \leq t \leq t_f \) into \( n \) segments of size \( \Delta t = t_f/n, \) let \( x_i = i \Delta t \) with \( i = 0, 1, 2, \ldots, n, \) evaluate the differential equation at \( t = t_i, \) and introduce a finite difference approximation for the derivative to conclude that

\[
x_{i-1} - 2x_i + x_{i+1} = \frac{F(t_i)}{m} \Delta t^2
\]

which is valid for \( i = 1, 2, \ldots, n - 1. \) For the two end points, we remember the boundary values and require that

\[
x_0 = 0 , \quad x_n = a
\]

In total, we have \( n + 1 \) equations determining the \( n + 1 \) unknowns \( x_0, x_1, x_2, \ldots, x_n. \) Cast these equations in the matrix form

\[
\begin{bmatrix}
? & ? & ? & \ldots & ? \\
? & ? & ? & \ldots & ? \\
? & ? & ? & \ldots & ? \\
\vdots & \vdots & \vdots & \ddots & \vdots \\
? & ? & ? & \ldots & ? \\
\end{bmatrix}
\begin{bmatrix}
x_0 \\
x_1 \\
x_2 \\
\vdots \\
x_n \\
\end{bmatrix}
= \begin{bmatrix}
? \\
? \\
? \\
\vdots \\
? \\
\end{bmatrix}
\]

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.
14.39. 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.)

14.40. 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 $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
\[ R = \sum_{i=1}^{N} \left( x_i - Ae^{-bt_i} \cos(\omega t_i + \phi) + Ct_i + D \right)^2 \]

\textit{Warning:} This exercise is not for the faint-hearted. Some useful background will be found in
Chapter 8 of \textit{Data Reduction and Error Analysis for the Physical Sciences} (Second Edition) by
Chapter 15

Solving Partial Differential Equations

The properties—temperature, electrostatic potential, magnetic field, membrane displacements, quantum wave functions, fluid pressure, ...—of physical systems are usually encapsulated mathematically in functions of space and time. As the properties are constrained physically by the systems themselves, the functions representing the properties are constrained mathematically by various physical laws, which typically specify one or more relationships among the partial derivatives of the dependent variable(s) with respect to two or more independent variables and hence will take the form of one or more partial differential equations (PDEs). The important PDEs of mathematical physics are second-order equations and are frequently—though certainly not always—linear. In this chapter, we limit ourselves to linear equations.

To define a problem completely (i.e., so that it has a unique solution), the associated PDEs must be supplemented with appropriate boundary conditions, which will usually fall into one of three categories, specifically,

- *Dirichlet* boundary conditions or boundary conditions of the *first* kind, which specify values that the *solution* must assume on the boundary of the region in which a solution is sought,

- *Neumann* boundary conditions or boundary conditions of the *second* kind, which specify values that a *derivative* of the solution must assume on the boundary of that region, and

- *mixed* boundary conditions or boundary conditions of the *third* kind, which specify values that some linear combination of the solution and its derivatives must assume on the boundary of that region.

When time is among the independent variables, complete specification of a problem will also entail stipulation of appropriate initial conditions, which specify values that the function and—for equations that are second order in time—its time derivative must assume throughout the spatial domain at a specific time (usually taken to be time 0). Depending on which conditions are necessary, we face a boundary-value problem (BVP) or an initial-value problem (IVP) or, sometimes, a problem involving both boundary and initial conditions.

Although closed form, symbolic solutions exist for some BVPs and IVPs, most of the time such problems can only be solved approximately by numerical methods. Most commonly, approaches to the numerical solution of PDEs involve (1) selecting a (usually large) set of points or *nodes* (or, in some contexts, *vertices*) that cover the ranges of all (or all save one) of the independent variables, (2) approximating the PDE(s) in such a way as to convert it (them) into a (usually large) set of algebraic equations (AEs) or *ordinary* differential equations (ODEs), and (3) solving the resulting set
of AEs or ODEs for the (approximate) solution to the original problem at each node. For example, rather than seeking the solution \( u(x,y,t) \) to the diffusion equation in two dimensions, we might introduce a grid or mesh, i.e., a set of discrete points \((x_i,y_j,t_k)\) distributed somehow over the ranges of the variables, and then convert the PDE into a set of algebraic equations for the several values \( u_{i,j,k} = u(x_i,y_j,t_k) \). Alternatively, we could discretize only the spatial variables by introducing a grid defined by the points \((x_i,y_j)\) and convert the PDE into a set of ODEs for the several functions \( u_{i,j}(t) = u(x_i,y_j,t) \) of the continuous variable \( t \). In either case, the boundary and initial conditions constrain the values and/or derivatives of the solution at nodes on the boundary. Thus, the number of independent AEs or ODEs will turn out to be exactly equal to the number of unknown values or functions in the problem.

At least two distinctly different approaches to the conversion of a PDE into a set of AEs or ODEs are in common use. Finite difference methods (FDMs) are quick, (comparatively) easy to motivate, and fast to code in computer languages, but they are borderline impossible to apply unless (1) the nodes are uniformly spaced and (2) the boundaries of the region in which a solution is sought coincide with the coordinate lines in one of the standard coordinate systems (Cartesian, polar, cylindrical, spherical, ...). An algebraic expression, which references two or more adjacent nodes, is used to approximate each derivative in the differential equation at each of these nodes. Ultimately, the partial differential equation(s) is (are) replaced by a system of AEs, (or, in some cases, by a system of ODEs), which is then solved for the dependent variable at each node.

In finite element methods (FEMs), however, the nodes forming the grid may be non-uniformly spaced. Collections of these nodes form geometric shapes—lines in one dimension; triangles or quadrilaterals or ... in two dimensions; tetrahedrons or hexahedrons (sometimes called bricks) or ... in three dimensions—that divide the region of interest into subregions, i.e., into the (finite) elements that give the method its name. Interpolation or shape functions are defined to facilitate approximating the dependent variable at points within each element from presumed known values of the dependent variable at the nodes or vertices associated with that element. The differential equation is then replaced with an equivalent integral statement, which is in turn converted into a system of algebraic equations by substituting the shape functions into this integral form, integrating, assembling the results from all elements, and imposing the constraints dictated by the boundary and initial conditions. Thus, as with the finite difference approach, the original partial differential equation(s) is (are) replaced by a system of algebraic equations, which is then solved for the dependent variable at each node.

Finite element methods have significant advantages over finite difference methods. For example, since finite element methods allow a nonuniform mesh, portions of the region in which a solution is sought can be treated with a fine mesh while, at the same time, other portions of that region can be represented with a coarse mesh, so the computational effort can be focused where it is most needed. Furthermore, especially if the boundaries are irregular, the boundary conditions are more easily incorporated with finite element methods than with finite difference methods. While we gain considerably in generality by invoking a method of this second type, we pay a heavy price: finite element methods are difficult to describe and even more difficult to code.

In Section 15.1, we deduce and provide physical contexts for a number of important PDEs. Then, in subsequent sections, we introduce both finite difference and finite element methods, illustrating each with explicit solutions to representative problems. For the sake of the quickest exposition of the essential ideas, we start by addressing problems with one independent variable before moving on to problems having two or more independent variables and requiring coordinate systems other than Cartesian. Because the methods are elaborate and their coding in computer languages is involved and lengthy, this chapter will undoubtedly seem more mathematical and abstract than physical. However obscured it may appear to be by the detailed discussion of method and implementation, the desire to address physical situations does assuredly lie underneath the entire exposition.
15.1 Sample Problems

We begin by laying out several of the most common PDEs of mathematical physics, placing each in at least some of the physical contexts in which it appears. As we shall discover in Section 15.1.5, the most common equations fall into one of three categories, represented respectively by the (classical) wave equation, the diffusion equation, and the Laplace equation.

15.1.1 Motion of a String: The Wave Equation

Consider a string that extends along the (horizontal) $x$ axis when it is in its equilibrium position, and let a point of the string when the string is in that position be located at $x$. Suppose that the string is moving in a plane (not the most complicated possible motion) such that, at a general time $t$, the point nominally located at $x$ is displaced transversely (i.e., perpendicular to the equilibrium orientation of the string)\(^1\) by an amount $u(x,t)$ and longitudinally (i.e., parallel to that equilibrium orientation) by an amount $v(x,t)$. The geometry is shown in Fig. 15.1. The string is, of course, under tension $\tau(x,t)$, which in general will vary from point to point and from time to time, but, for a perfectly flexible string (which we explicitly assume), will always be directed tangent to the string. A force diagram for an isolated element—the element located between $x$ and $x+\Delta x$ when the string is in equilibrium—of the string might look like Fig. 15.2. For the transverse motion of the string, Newton’s second law $F = ma$ then requires that

$$
\rho(x)\Delta x \frac{\partial^2 u}{\partial t^2} = \tau(x + \Delta x, t) \sin \theta(x + \Delta x, t) - \tau(x, t) \sin \theta(x, t) - \rho(x) \Delta x g
$$

(15.1)

or, if we divide by $\Delta x$ and let $\Delta x$ approach zero, that

$$
\rho(x) \frac{\partial^2 u}{\partial t^2} = \frac{\partial}{\partial x} \left( \tau(x, t) \sin \theta(x, t) \right) - \rho(x) g
$$

(15.2)

\(^1\)We take this transverse direction to be vertical, and include gravity among the forces acting on the string.
Here, $\rho(x) \Delta x$ is the mass of the element, and $\rho(x) g \Delta x$—the gravitational force on the element when the string is near the surface of the earth—illustrates a possible external force on the string. Similarly, for the longitudinal motion of the string, Newton’s second law requires that

$$
\rho(x) \Delta x \frac{\partial^2 v}{\partial t^2} = \tau(x + \Delta x, t) \cos \theta(x + \Delta x, t) - \tau(x, t) \cos \theta(x, t)
$$

(15.3)

or that

$$
\rho(x) \frac{\partial^2 v}{\partial t^2} = \frac{\partial}{\partial x} \left( \tau(x, t) \cos \theta(x, t) \right)
$$

(15.4)

for the horizontal motion.

This set of equations still contains too many unknowns. We must eliminate $\theta(x, t)$, because we really want to find only $u(x, t)$ and $v(x, t)$. Figure 15.3 supports the conclusion that

$$
\sin \theta = \frac{\Delta u}{\Delta s} = \frac{\Delta u}{\sqrt{\Delta u^2 + (\Delta x + \Delta v)^2}} = \frac{\partial u}{\partial x} \left( \frac{\partial u}{\partial x}^2 + \left( 1 + \frac{\partial v}{\partial x} \right)^2 \right)^{1/2}
$$

(15.5)

and that

$$
\cos \theta = \frac{\Delta x + \Delta v}{\Delta s} = \frac{1 + \frac{\partial v}{\partial x}}{\left( \frac{\partial u}{\partial x}^2 + \left( 1 + \frac{\partial v}{\partial x} \right)^2 \right)^{1/2}}
$$

(15.6)

Thus, the equations describing the motion of the string are

$$
\rho(x) \frac{\partial^2 u}{\partial t^2} = \frac{\partial}{\partial x} \left[ \tau(x, t) \frac{\partial u}{\partial x} \left( \frac{\partial u}{\partial x}^2 + \left( 1 + \frac{\partial v}{\partial x} \right)^2 \right)^{1/2} \right] - \rho g
$$

(15.7)

$$
\rho(x) \frac{\partial^2 v}{\partial t^2} = \frac{\partial}{\partial x} \left[ \tau(x, t) \frac{1 + \frac{\partial v}{\partial x}}{\left( \frac{\partial u}{\partial x}^2 + \left( 1 + \frac{\partial v}{\partial x} \right)^2 \right)^{1/2}} \right]
$$

(15.8)
The system is, of course, not complete as yet. We would also need to know not only appropriate initial conditions, i.e., initial values for the four functions
\[ u(x, 0), \ v(x, 0), \ \frac{\partial u}{\partial x}(x, 0), \ \frac{\partial v}{\partial x}(x, 0) \] (15.9)
but also appropriate boundary conditions on the string at its two ends and a connection between \( \tau \) on the one hand and \( u \) and \( v \) on the other. This system is decidedly non-linear and very difficult to solve.

Approximations are almost always necessary to turn the problem we would really like to solve into one that we can solve. If, for example, (1) the amplitude of the motion is small so that
\[ \Delta u \ll \Delta x \implies \frac{\partial u}{\partial x} \ll 1 \] (15.10)
(2) \( \tau \) is sufficiently large that, in small amplitude motion, \( \tau \) remains essentially constant, and (3) the motion is transverse so that \( v = 0 \) everywhere and always, then the equation for \( v \) is automatically satisfied and the equation for \( u \) decouples from that for \( v \) and becomes the (inhomogeneous) wave equation,
\[ \rho \frac{\partial^2 u}{\partial t^2} = \tau \frac{\partial^2 u}{\partial x^2} - \rho g \] (15.11)
which is correct under the given circumstances even if \( \rho \) varies with \( x \).

We have derived this result for motion of a string in one dimension. Two special cases are worth noting:

- If the situation is static so there is no time dependence (\( \partial u/\partial t = 0 \)), then the shape of a string hanging under its own weight is given by the solution to the equation
  \[ \tau \frac{d^2 u}{dx^2} = \rho g \] (15.12)
  which reduces in this case to an ODE.

- If there is no outside force and \( \rho \) is constant, then Eq. (15.11) becomes
  \[ \rho \frac{\partial^2 u}{\partial t^2} = \tau \frac{\partial^2 u}{\partial x^2} \implies \frac{\partial^2 u}{\partial t^2} = \frac{\tau}{\rho} \frac{\partial^2 u}{\partial x^2} \implies \frac{1}{c^2} \frac{\partial^2 u}{\partial t^2} = \frac{\partial^2 u}{\partial x^2} \] (15.13)
  Note that the constant \( c = \sqrt{\tau/\rho} \) has the units of velocity.\(^2\) It turns out—though we don’t know this as yet—that \( c \) is the speed of propagation of the wave conveyed by the solution to this equation.

In both cases, of course, appropriate boundary and initial conditions must be specified before the equation has a unique solution.

While its deduction is more complicated, Eq. (15.13) has a natural extension to two and three dimensions, in which case we would consider, for example, the displacement of a two-dimensional membrane or the pressure in a gas in a three-dimensional enclosure. The function we seek would then be \( u = u(x, y, z, t) = u(r, t) \) and the second derivative with respect to one spatial dimension becomes the Laplacian. The equation in that case is
\[ \frac{1}{c^2} \frac{\partial^2 u}{\partial t^2} = \nabla^2 u \text{ or } \left( \nabla^2 - \frac{1}{c^2} \frac{\partial^2}{\partial t^2} \right) u = 0 \] (15.14)
\(^2\)Since \( [\tau] = \text{N} = \text{kg.m/s}^2 \) and \( [\rho] = \text{kg/m} \), \( [\tau/\rho] = (\text{kg.m/s}^2)/(\text{kg/m}) = \text{m}^2/\text{s}^2 \). (Here, the symbol \([\ldots]\) stands for the units of \ldots.)
Here

\[ \nabla^2 = \frac{\partial^2}{\partial x^2} + \frac{\partial^2}{\partial y^2} + \frac{\partial^2}{\partial z^2} \]  
(15.15)

\[ = \frac{1}{r} \frac{\partial}{\partial r} \left( r \frac{\partial}{\partial r} \right) + \frac{1}{r^2} \frac{\partial^2}{\partial \phi^2} + \frac{\partial^2}{\partial z^2} \]  
(15.16)

\[ = \frac{1}{r^2} \frac{\partial}{\partial r} \left( r^2 \frac{\partial}{\partial r} \right) + \frac{1}{r^2 \sin^2 \theta} \frac{\partial}{\partial \theta} \left( \sin \theta \frac{\partial}{\partial \theta} \right) + \frac{1}{r^2 \sin^2 \theta} \frac{\partial^2}{\partial \phi^2} \]  
(15.17)

where \((x, y, z), (r, \phi, z)\), and \((r, \theta, \phi)\) are the physicists’ standard Cartesian, cylindrical, and spherical coordinates. We thus arrive at the (classical) wave equation, one of the three prototype equations in mathematical physics.

### 15.1.2 Heat Flow: The Diffusion Equation

To motivate a second of the important equations, we seek the temperature \(u(x, t)\) in the one-dimensional rod shown in Fig. 15.4. Let the rod be insulated along its sides and characterized by a thermal conductivity \(K\), which gives the rate of heat flow per unit area in the direction of increasing \(x\) by the expression

\[
\text{rate of heat flow in rod towards positive } x \text{ through cross section at right angles to rod} = -KA \frac{\partial u}{\partial x} \]

at \(x\) by the expression

where \(A\) is the area of a cross section of the rod. [In effect, Eq. (15.18) defines \(K\).] Here, the derivative \(\partial u/\partial x\) is the gradient of the temperature, and the minus sign appears because heat flows in the positive direction when the gradient is negative and in the negative direction when the gradient is positive, i.e., heat flows from regions of higher temperature to regions of lower temperature; the explicit minus sign then assures that \(K\) will be a positive quantity. With this definition of a material property, which may depend on position in the rod, we conclude that, as heat flows in the rod,

\[
\text{heat flow in time } \Delta t \text{ into shaded element across surface at } x = -KA \left. \frac{\partial u}{\partial x} \right|_x \Delta t \]

(15.19)

and

\[
\text{heat flow in time } \Delta t \text{ into shaded element across surface at } x + \Delta x = +KA \left. \frac{\partial u}{\partial x} \right|_{x+\Delta x} \Delta t \]

(15.20)

so that, in the end,

\[
\text{net heat conducted into shaded element in time } \Delta t = \left[ KA \left. \frac{\partial u}{\partial x} \right|_{x+\Delta x} - KA \left. \frac{\partial u}{\partial x} \right|_x \right] \Delta t \]

(15.21)
15.1. Sample Problems

This energy, of course, will affect the temperature of the element. Indeed, if we introduce two other properties of the material, its heat capacity per unit mass \(c\), and its density \(\rho\), we then conclude that

\[
\text{heat necessary to increase temperature of shaded element by amount } \Delta u = (A \Delta x \rho c) \Delta u
\]

(15.22)

where \(A \Delta x\) is the volume of the element, and (hence) \(\rho A \Delta x\) is its mass. Since any heat transported into the element must affect its temperature as per this equation, these two evaluations of the heat flux must be equal, and we conclude that

\[
\rho c \frac{\partial u}{\partial t} = \frac{\partial}{\partial x} \left( K \frac{\partial u}{\partial x} \right)
\]

(15.23)

or, if \(K\) happens to be constant, that

\[
\frac{\rho c}{K} \frac{\partial u}{\partial t} = \frac{\partial^2 u}{\partial x^2} \Rightarrow \frac{\partial^2 u}{\partial x^2} = \frac{1}{\alpha^2} \frac{\partial u}{\partial t}
\]

(15.24)

where \(\alpha^2 = K/\rho c\). The statement of the problem is, of course, not complete until appropriate initial and boundary conditions have been specified.

One special case is worth noting. If the situation is static, i.e., if thermal equilibrium has been reached, then \(\partial u/\partial t = 0\) and the temperature distribution in the rod satisfies

\[
\frac{\partial}{\partial x} \left( K \frac{\partial u}{\partial x} \right) = 0 \quad \text{or} \quad (K \text{ constant}) \quad \frac{\partial^2 u}{\partial x^2} = 0
\]

(15.25)

with, of course, appropriate boundary values at the ends of the rod.

While its deduction is more complicated, Eq. (15.25) has a natural extension to two and three dimensions, in which case we would consider, for example, the evolution of the temperature distribution in a plate or in a three-dimensional object when the initial temperature throughout the object is given and either Dirichlet or Neumann or mixed boundary conditions are specified at all points on its boundary. We would then seek the function \(u(x,y,z,t) = u(r,t)\) giving the temperature at all points in space and time, and the fundamental equation becomes

\[
\nabla \cdot \left( K \nabla u \right) = \rho c \frac{\partial u}{\partial t} \quad \text{or} \quad (K \text{ constant}) \quad \nabla^2 u = \frac{1}{\alpha^2} \frac{\partial u}{\partial t}
\]

(15.26)

We thus arrive at the (classical) diffusion equation, the second of the three prototype equations in mathematical physics. The diffusion equation is similar to the wave equation, but differs from it in that the time derivative is only first order.

15.1.3 Steady State Heat Flow: The Laplace Equation

The third of the three important equations in mathematical physics is quickly deduced from Eq. (15.26). In two or three dimensions when a steady-state temperature distribution has been reached, \(u\) satisfies the Laplace equation

\[
\nabla^2 u = 0
\]

(15.27)

which, as always, must be supplemented with appropriate boundary conditions before its solution is unique. (This time there will be no initial conditions, since there is no time variable in the picture.)

15.1.4 Other Situations

Variants on the three equations we have just deduced appear in many places. For example, Maxwell’s equations for the electrostatic field \(\mathbf{E}\) support the argument

\[
\nabla \times \mathbf{E} = 0 \quad \Rightarrow \quad \mathbf{E} = -\nabla V
\]

\[
\nabla \cdot \mathbf{D} = \nabla \cdot \epsilon \mathbf{E} = \rho \quad \Rightarrow \quad \nabla \cdot (\epsilon \nabla V) = -\rho
\]

(15.28)
leading to an equation that determines the electrostatic potential $V$ from the charge density $\rho$ and the dielectric permittivity $\epsilon$. If $\epsilon$ is constant, the equation becomes the Poisson equation,

$$\nabla^2 V = -\frac{\rho}{\epsilon}$$

(15.29)

If, in addition, $\rho = 0$, the equation becomes the Laplace equation

$$\nabla^2 V = 0$$

(15.30)

for the electrostatic potential.

For another example, suppose we seek a sinusoidal solution to the wave equation, i.e., suppose we seek a solution to Eq. (15.14) of the form

$$u(x,y,z,t) = \psi(x,y,z) \cos \omega t$$

Then, if $u$ satisfies the wave equation and we set $k^2 = \omega^2/c^2$, $\psi$ will satisfy the equation

$$\nabla^2 \psi + k^2 \psi = 0$$

an equation called the Helmholtz equation. In one dimension, the Helmholtz equation becomes

$$\frac{d^2 \psi}{dx^2} + k^2 \psi = 0$$

(15.31)

which describes standing waves in a string and many other physical situations.

We need not limit our enumeration to equations of importance in classical physics. For example, the time-dependent Schrödinger equation,

$$-\frac{\hbar^2}{2m} \nabla^2 \psi + V \psi = i\hbar \frac{\partial \psi}{\partial t}$$

(15.32)

for the quantum wave function $\psi$ of a particle of mass $m$ in a potential energy $V$ is in some sense a diffusion equation (second order in the space derivatives, first order in the time derivative), though the term $V \psi$ and the imaginary unit $i = \sqrt{-1}$ create significant differences between the two equations.

Beyond Maxwell’s equations and the Schrödinger equation, second-order PDEs can be found in the Dirac equation (relativistic quantum mechanics), the equations of fluid dynamics (the Navier-Stokes’ equations; see Section 15.1.6), the equations of magnetohydrodynamics (MHD, which combine Maxwell’s equations and the Navier-Stokes’ equations), and in many other contexts in classical and contemporary physics.

### 15.1.5 Classification of Second-Order PDEs

In general terms, we have arrived in all cases at second-order, linear, often homogeneous PDEs, equations that we might collectively denote by the symbolism

$$\mathcal{L}\{u\} = 0$$

(15.33)

where the operator $\mathcal{L}$ symbolizes the (linear) differential operator that defines the equation. The most general example in two variables would have the form

$$A(x,y) \frac{\partial^2 u}{\partial x^2} + 2B(x,y) \frac{\partial^2 u}{\partial x \partial y} + C(x,y) \frac{\partial^2 u}{\partial y^2} + D(x,y) \frac{\partial u}{\partial x} + E(x,y) \frac{\partial u}{\partial y} + F(x,y) u = 0$$

(15.34)

where $A$, $B$, $C$, $D$, $E$, and $F$ may depend on $x$ and $y$ but do not depend on $u$ or its derivatives. Many, many important equations (though, significantly, not all) fall into this category. The solutions
of those equations that are in this category exhibit the very important property of superposition, namely

\[ \mathcal{L}\{u_1\} = 0, \mathcal{L}\{u_2\} = 0 \implies \mathcal{L}\{au_1 + bu_2\} = a\mathcal{L}\{u_1\} + b\mathcal{L}\{u_2\} = 0 \quad (15.35) \]
i.e., any linear combination of two solutions is itself a solution of the differential equation.

As it turns out, second-order, linear PDEs in two variables fall into three distinct categories, depending on the algebraic sign of the quantity \( AC - B^2 \). If, for example, \( AC - B^2 > 0 \), the equation is said to be elliptic; if \( AC - B^2 = 0 \), the equation is parabolic; and if \( AC - B^2 < 0 \), the equation is hyperbolic. When the coefficients actually depend on \( x \) and/or \( y \), an equation may fall into one category for some portions of the region of interest and another category for other portions of that region. Each of the prototype equations we have identified above, however, falls cleanly into one of the three categories. For the wave equation, we note that

\[ \frac{\partial^2 u}{\partial x^2} - \frac{1}{c^2} \frac{\partial^2 u}{\partial t^2} = 0 \implies A = 1, B = 0, C = -\frac{1}{c^2} \implies AC - B^2 = -\frac{1}{c^2} < 0 \quad (15.36) \]
and conclude that the wave equation is hyperbolic. For the diffusion equation,

\[ \frac{\partial^2 u}{\partial x^2} - \frac{1}{\alpha^2} \frac{\partial u}{\partial t} = 0 \implies A = 1, B = C = 0 \implies AC - B^2 = 0 \quad (15.37) \]
and the diffusion equation is parabolic. Finally, for the Laplace equation,

\[ \frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} = 0 \implies A = C = 1, B = 0 \implies AC - B^2 = 1 \quad (15.38) \]
and the Laplace equation is elliptic. In focusing on these three equations, we will therefore be exhibiting techniques that would be applicable to all possible second-order, linear, homogeneous PDEs in two variables.

Note, incidentally, that the variable transformation \( t \to iy \) will convert the wave equation (hyperbolic) into the Laplace equation (elliptic) and remember that this transformation also converts trigonometric functions into hyperbolic functions. Within the framework of complex variable theory, we might therefore expect that solutions to the wave equation and solutions to the Laplace equation could be related to one another (though we shall not pursue that connection in this book).

### 15.1.6 Equations of Fluid Dynamics: Navier-Stokes’ Equations

Let us next work out some of the fundamental equations of fluid dynamics. The state of a fluid at time \( t \) is described by a number of functions, the most important of which are the (vector) velocity field \( \mathbf{v}(\mathbf{r}, t) \), the (scalar) pressure \( \rho(\mathbf{r}, t) \), and the (scalar) density \( \rho(\mathbf{r}, t) \). In general, all of these quantities are functions of position \( \mathbf{r} \) within the fluid and of time \( t \). Further, \( p \) and \( \rho \) are related by the equation of state of the fluid involved.

Consider a volume—see Fig. 15.5—in the shape of a rectangular parallelepiped with its faces parallel to the coordinate planes. Let the lower left back corner be at \((x, y, z)\) and the upper right front corner be at \((x + \Delta x, y + \Delta y, z + \Delta z)\). We regard the volume as fixed in space, and the fluid as flowing through the volume in a way described by the velocity field already introduced. Further, the fluid in this volume experiences (internal) forces from its contact with the rest of the fluid at the surface of the volume and may also experience (external) forces from things like a nearby earth.\(^3\) The basic equations for fluid motion reflect the conservation of mass (the continuity equation), Newton’s second law, and conservation of linear and angular momentum. We deduce here only the first two of these fundamental relationships.

\(^3\)We assume the fluid is not electrically charged, and we ignore internal forces arising from the gravitational attraction of one portion of the fluid for another. Thus, the only internal forces on one element of the fluid will arise from its direct contact with contiguous elements.
15.1.6.1 Conservation of Mass: The Equation of Continuity

In the time interval from $t$ to $t + \Delta t$, the change in the mass of the fluid in the volume can be calculated in two different ways. First, we focus on the density, concluding that

$$\text{mass added during interval } t \text{ to } t + \Delta t = \left[ \rho(t + \Delta t) - \rho(t) \right] \Delta x \Delta y \Delta z \approx \frac{\partial \rho}{\partial t} \Delta t \Delta x \Delta y \Delta z \quad (15.39)$$

Alternatively, we can calculate this increment in mass from the velocity field. Focus for example on the two sides parallel to the $yz$ plane. On the side at coordinate $x$, all of the fluid in a volume of height $(v_x(x) \Delta t) \Delta y \Delta z$ passes through the surface into the volume in the time interval $\Delta t$. Similarly, on the side at coordinate $x + \Delta x$, all of the fluid in a volume $(v_x(x + \Delta x) \Delta t) \Delta y \Delta z$ passes out of the volume in that same time interval. On the first side, the density of the fluid is $\rho(x)$ while on the second side the density is $\rho(x + \Delta x)$. Thus, the net mass transported into the volume in time $\Delta t$ over these two surfaces by the fluid flow is given by

$$\text{net mass transported into volume through sides parallel to } yz \text{ plane in time } t \text{ to } t + \Delta t = \rho(x) v_x(x) \Delta t \Delta y \Delta z - \rho(x + \Delta x) v_x(x + \Delta x) \Delta t \Delta y \Delta z$$

$$= - \left( \frac{\partial (\rho v_x)}{\partial x} \right) \Delta t \Delta x \Delta y \Delta z \quad (15.40)$$

All quantities in this last expression are evaluated at $x$, $y$, $z$, and $t$. We justify ignoring differences in those variables from one point to another by noting that the expression is already first order in $\Delta t$, $\Delta x$, $\Delta y$, and $\Delta z$. Therefore, any variation in $\rho$, $v_x$, $\partial (\rho v_x)/\partial x$, or $\partial \rho/\partial t$ in the (small) interval from $x$ to $x + \Delta x$ would contribute at the strongest to second order in these small quantities—and we will consistently ignore contributions at that level.

Similar expressions apply for the other two pairs of faces (the pair parallel to the $xy$ plane and the pair parallel to the $xz$ plane). In total, the flow of the fluid through the volume transports a net mass given by

$$\text{net mass transported into volume through all sides in time } t \text{ to } t + \Delta t = - \left( \frac{\partial (\rho v_x)}{\partial x} + \frac{\partial (\rho v_y)}{\partial y} + \frac{\partial (\rho v_z)}{\partial z} \right) \Delta t \Delta x \Delta y \Delta z$$

$$= - \nabla \cdot (\rho v) \Delta t \Delta x \Delta y \Delta z \quad (15.41)$$
Figure 15.6: Stresses on the front (dotted) surface of an element in a fluid. (Be sure you visualize the intersection of the three vectors to lie on that front surface.) The signs are defined to give the stresses exerted on an element by the next element in the direction in which the coordinate increases.

...into the volume.

The values obtained in Eqs. (15.39) and (15.41) for the mass added in the chosen volume in the time $\Delta t$ must, of course, be the same, and we arrive at the equation of continuity,

$$\nabla \cdot (\rho \mathbf{v}) + \frac{\partial \rho}{\partial t} = 0$$

which expresses the conservation of mass, i.e., the conviction that—however the fluid flows—any change in the mass in a fixed volume can come only by the flow of matter through the surface bounding that volume, i.e., that mass can be neither created nor destroyed in the volume.\(^4\) If the fluid is incompressible ($\rho$ constant), then the equation of continuity reduces to $\nabla \cdot \mathbf{v} = 0$.

### 15.1.6.2 Newton’s Second Law

The volume on which we are focusing our attention experiences forces from the surrounding fluid at its surface. Both normal forces (perpendicular to the surface and related to pressure) and shear forces (tangent to the surface and related to viscosity) may be experienced. All are expressed in terms of stress, normal stress for the first and shear stress for the second, where a stress in general is defined as a force per unit area. For stress, we use the notation $T_{rs}(x, y, z, t)$, where the first subscript conveys the direction of the normal to the surface involved and the second subscript conveys the direction of the stress, i.e.,

$$T_{rs} = \begin{pmatrix}
  \text{the component in the } s \text{ direction of} \\
  \text{the force on a surface of unit area} \\
  \text{oriented perpendicular to the } r \text{ direction}
\end{pmatrix}$$

Even more specifically, $T_{xx}$ stands for the force in the (positive) $x$ direction (second subscript) on a surface of unit area oriented perpendicular to the $x$ axis (first subscript) and would be a normal stress. Similarly, the symbol $T_{xy}$ stands for the force in the (positive) $y$ direction (second subscript)

\(^4\)The equation, of course, is quite similar to the parallel equation in electrodynamics expressing the conservation of charge.
on a surface of unit area oriented perpendicular to the $x$ axis (first subscript) and would be a shear stress. The three stresses $T_{xx}$, $T_{xy}$, and $T_{xz}$ are shown in Fig. 15.6. As a vector, the total stress on a surface of unit area oriented perpendicular to the $x$ axis would be given by

$$ T_x = T_{xx} \mathbf{i} + T_{xy} \mathbf{j} + T_{xz} \mathbf{k} \quad (15.44) $$

Note particularly that the stresses shown are the stresses on the indicated side of the illustrated element arising from its contact with the adjacent portion of the fluid in the direction in which the $x$ coordinate increases. The stresses that the illustrated element exerts on its neighbor in the positive $x$ direction will, via Newton’s third law, be equal in magnitude but opposite in direction to those shown. In particular, then, the net force arising from internal stresses on the front and back surfaces of the illustrated element would be given by

$$ F_x = \left[ T_{xx}(x + \Delta x, y, z) - T_{xx}(x, y, z) \right] \mathbf{i} + \left[ T_{xy}(x + \Delta x, y, z) - T_{xy}(x, y, z) \right] \mathbf{j} + \left[ T_{xz}(x + \Delta x, y, z) - T_{xz}(x, y, z) \right] \mathbf{k} \Delta y \Delta z \quad (15.45) $$

Here, we can evaluate the tensions at $y, z$ because variation of $y$ and $z$ over the front and back surfaces will contribute only to second order in $\Delta y$ and $\Delta z$—and we are ignoring terms beyond first order. Further, the derivatives can all be evaluated at argument $(x, y, z)$, since the variation between the front and back surfaces will contribute only to second order in $\Delta x$. For clarity in the final result, those arguments have been omitted.\(^5\)

Only the subscripts and the variable with respect to which stresses are differentiated must be adjusted to yield expressions for the forces arising from internal stresses on the other two pairs of surfaces on the element in question. We find that

$$ F_y = \left( \frac{\partial T_{yx}}{\partial y} \mathbf{i} + \frac{\partial T_{yy}}{\partial y} \mathbf{j} + \frac{\partial T_{yz}}{\partial y} \mathbf{k} \right) \Delta x \Delta y \Delta z \quad (15.46) $$

and

$$ F_z = \left( \frac{\partial T_{zx}}{\partial z} \mathbf{i} + \frac{\partial T_{zy}}{\partial z} \mathbf{j} + \frac{\partial T_{zz}}{\partial z} \mathbf{k} \right) \Delta x \Delta y \Delta z \quad (15.47) $$

Finally, bringing together all of the $x$ components of the forces arising from internal stress on all surfaces of the element on which our attention is focused and adding a possible external force\(^6\) $\rho f_x \Delta x \Delta y \Delta z$, we conclude that the $x$ component of the net force on the element in question would be given by

$$ F_x = \left( \rho f_x + \frac{\partial T_{xx}}{\partial x} + \frac{\partial T_{yx}}{\partial y} + \frac{\partial T_{zx}}{\partial z} \right) \Delta x \Delta y \Delta z \quad (15.48) $$

According to Newton’s second law, this force must, of course, also be given as the product of the mass of the element times the $x$ component of its acceleration, i.e., by $(\rho \Delta x \Delta y \Delta z) a_x$. Thus, we have for the $x$ component of the equation of motion the result

$$ \rho a_x = \rho f_x + \frac{\partial T_{xx}}{\partial x} + \frac{\partial T_{yx}}{\partial y} + \frac{\partial T_{zx}}{\partial z} \quad (15.49) $$

Similar equations could be deduced for the $y$ and $z$ components of the equation of motion for the fluid.

\(^5\)It is extremely easy to become very confused about these signs. Think about them several times.

\(^6\)For convenience, we define $f_x$ to be a force per unit mass.
Evaluating the acceleration, however, introduces a subtle complication, since \( a \neq \partial v_x / \partial t \)!! This derivative is the rate of change of the velocity field at a fixed position in space. Unfortunately, at the end of the time interval, the element of the fluid on which the force acts is at a new position in the fluid. The quantity that figures in the definition of the acceleration we want is the change in the velocity of a particular element of the fluid that moves with the fluid; that quantity is given by

\[
\Delta v = v(x + v_x \Delta t, y + v_y \Delta t, z + v_z \Delta t, t + \Delta t) - v(x, y, z, t)
\]

(15.50)

which implies that the acceleration we really want is given by

\[
a = \frac{\Delta v}{\Delta t} = v_x \frac{\partial v}{\partial x} + v_y \frac{\partial v}{\partial y} + v_z \frac{\partial v}{\partial z} + \frac{\partial v}{\partial t} = \frac{\partial v}{\partial t} + (v \cdot \nabla)v
\]

(15.51)

a derivative that is sometimes said to “follow the fluid” and is sometimes called the substantial derivative. Thus, the translation of the \( x \) component of Newton’s second law into the vocabulary used to describe the state of a fluid is

\[
\rho \left( \frac{\partial}{\partial t} + v \cdot \nabla \right) v_x = \rho f_x + \frac{\partial T_{xx}}{\partial x} + \frac{\partial T_{yx}}{\partial y} + \frac{\partial T_{zx}}{\partial z}
\]

(15.52)

The \( y \) and \( z \) components have similar statements.\(^7\)

Equation (15.52) and its \( y \) and \( z \) counterparts constitute a set of complicated, non-linear, coupled PDEs. Further, they are not by themselves complete, since they involve the stresses (which are not among the official quantities used to describe the state of the fluid). The equations (including the continuity equation) have too many unknowns (\( \rho, v, \) and the stresses). We need some further relationships, particularly relationships linking stresses to fluid velocities (and involving new fluid parameters like viscosity). For example, in some simple cases, we can sometimes suppose that the shear stress in one direction (say \( x \)) on a surface perpendicular to another direction (say \( y \)) is related to the rate at which the component of the velocity in the first direction changes with displacement in the second direction, i.e., \( T_{yx} = \mu \partial v_x / \partial y \), where \( \mu \) is the viscosity of the fluid (which in effect is defined by this relationship).\(^8\)

### 15.1.6.3 A Special Case: Non-viscous Flow

If the fluid of interest has zero viscosity, then there can be no shear forces and all the off-diagonal elements of the stress tensor \( T \) will be zero. Further, in most instances, the on-diagonal elements will all be equal to the negative of the pressure. That is, \( T_{xx} = T_{yy} = T_{zz} = -p \) and \( T_{xz} = T_{xy} = \ldots = 0 \). In that case, Eq. (15.52) reduces to

\[
\rho \left( \frac{\partial}{\partial t} + v \cdot \nabla \right) v_x = \rho f_x - \frac{\partial p}{\partial x}
\]

(15.53)

Combined in vector notation with the other two components, the basic equation for non-viscous flow then is that

\[
\rho \left( \frac{\partial}{\partial t} + v \cdot \nabla \right) v = \rho f - \nabla p
\]

(15.54)

Once the external force and the equation of state relating \( \rho \) and \( p \) have been determined, this equation provides the starting point for solving many problems in non-viscous fluid flow.

---

\(^7\)To write a single equation combining the \( x, y, \) and \( z \) components, we would have to introduce the notation of tensors. Given our limited use of these relationships, we have little motivation to take that step.

\(^8\)A fluid described by this relationship is said to be a *Newtonian* fluid. The relationship is an approximation, but fortunately—there are several fluids to which it—and its generalizations to 2D and 3D flow—seem to be accurately applicable.
15.1.6.4 A Second Special Case: Sound Waves

Suppose the flow of interest is 1D, so \(v_y\) and \(v_z\) are both zero and \(v_x\) depends only on \(x\) and \(t\). Further, suppose that external forces are absent and that the flow is inviscid, i.e., that the viscosity is zero and hence that shear stresses are zero \((T_{yx} = 0 \text{ and } T_{zx} = 0)\). Then Eq. (15.42) expressing conservation of mass and Eq. (15.52) expressing Newton’s second law become

\[
\frac{\partial}{\partial x} (\rho v_x) = -\frac{\partial \rho}{\partial x}; \quad \rho \left( \frac{\partial v_x}{\partial t} + v_x \frac{\partial v_x}{\partial x} \right) = -\frac{\partial p}{\partial x} \quad (15.55)
\]

(We have recognized that \(T_{xx} = -p\).) Further, suppose that the velocity of the fluid is small, so we can ignore the non-linear term and reduce these equations to

\[
\frac{\partial}{\partial x} (\rho v_x) = -\frac{\partial \rho}{\partial x}; \quad \rho \frac{\partial v_x}{\partial t} = -\frac{\partial p}{\partial x} \quad (15.56)
\]

Next, let us suppose that \(p = p(\rho)\), i.e., that the pressure in the fluid and its density are functionally related. Then,

\[
\frac{\partial p}{\partial x} = \frac{dp}{d\rho} \frac{\partial \rho}{\partial x} = K \frac{\partial \rho}{\partial x} \quad (15.57)
\]

where \(K = dp/d\rho\), which will in general depend on \(\rho\), is a property of the fluid characterizing the extent to which small changes in applied pressure are determined from small changes in density. The equations now become

\[
\frac{\partial}{\partial x} (\rho v_x) = -\frac{\partial \rho}{\partial t}; \quad \rho \frac{\partial v_x}{\partial t} = -K \frac{\partial \rho}{\partial x} \quad (15.58)
\]

Finally, we suppose that the pressure departs only slightly from its nominal equilibrium value \(\rho_0\) and that the velocity in equilibrium is zero (the fluid is quiescent), i.e., we set

\[
\rho = \rho_0 + \delta \rho(x,t) \quad ; \quad v_x = \delta v_x(x,t) \quad (15.59)
\]

Then, ignoring all but the linear terms in small quantities, we find that the equations now are

\[
\rho_0 \frac{\partial \delta v_x}{\partial x} = -\frac{\partial \delta \rho}{\partial t}; \quad \rho_0 \frac{\partial \delta v_x}{\partial t} = -K_0 \frac{\partial \delta \rho}{\partial x} \quad (15.60)
\]

where \(K_0 = K(\rho_0)\) is the value of \(K\) associated with the equilibrium density \(\rho_0\)—and is constant. Differentiating the first of these with respect to \(t\) and the second with respect to \(x\), we find that

\[
\rho_0 \frac{\partial^2 \delta v_x}{\partial t \partial x} = -\frac{\partial^2 \delta \rho}{\partial t^2}; \quad \rho_0 \frac{\partial^2 \delta v_x}{\partial x \partial t} = -K_0 \frac{\partial^2 \delta \rho}{\partial x^2} \quad (15.61)
\]

Since the mixed second-partial derivatives are equal, we then conclude that

\[
\frac{\partial^2 \delta \rho}{\partial t^2} = K_0 \frac{\partial^2 \delta \rho}{\partial x^2} \quad (15.62)
\]

That is, the density fluctuations \(\delta \rho\) in this system satisfy the wave equation. In effect, we have discovered that, among the motions possible in this medium, there is a longitudinal wave that propagates with speed \(v = \sqrt{K_0}\). [See Eq. (15.13) and footnote 2.]

As an aside, note that, for an ideal gas, \(pV = nRT\) or \(p = nRT/V = MRT/(Vn_m)\), where \(M\) is the mass of the sample of gas and \(m_n\) is the mass of the gas per mole. Further, when the gas undergoes an adiabatic process, as it does when it supports a sound wave, \(pV^\gamma\), where \(\gamma\) is the ratio of the specific heat at constant pressure to that at constant volume, is constant. Thus, since \(pV^\gamma = p_0 V_0^\gamma\) and \(V = M/\rho\), we find that

\[
p \left( \frac{M}{\rho} \right)^\gamma = p_0 V_0^\gamma \quad \Rightarrow \quad p = \frac{p_0 V_0^\gamma}{M^\gamma \rho^\gamma} \quad \Rightarrow \quad K = \frac{dp}{d\rho} = \gamma \frac{p_0 V_0^\gamma}{M^\gamma \rho^{\gamma-1}} \quad (15.63)
\]
or, evaluating at $\rho = \rho_0$ and noting the relationship $V_0 = M/\rho_0$, that

$$K_0 = \frac{dp}{d\rho_0} = \frac{\gamma}{M} \frac{p_0 V_0^\gamma}{\rho_0^{\gamma-1}} = \frac{\gamma}{M} \frac{p_0 M^{\gamma-1}}{\rho_0^{\gamma-1}} = \frac{p_0}{\rho_0} = \frac{\gamma RT_0}{m_n}$$

(15.64)

and we have found $K_0$ for this simple system. Note that $K_0$ is temperature dependent. Further, with this value for $K_0$, we can predict that the speed of sound in an ideal gas at temperature $T_0$ is given by $v = \sqrt{K_0} = \sqrt{\gamma RT_0/m_n}$. In particular, we conclude that the speed of sound in a gas increases as the square root of the (absolute) temperature. With $R = 8.31$ N m/(mol K) and—for air—$m_n = 0.0288$ kg and $\gamma = 7/5 = 1.4$, we find at room temperature ($70$ °F = $21$ °C = $294$ K) that

$$v = \sqrt{\frac{\gamma RT_0}{m_n}} = \sqrt{1.4 \frac{(8.31)(294)}{0.0288}} \text{ m/s} = 344 \text{ m/s}$$

(15.65)

A further interesting connection appears if we remember from the equipartition theorem of kinetic theory that the rms speed $v_{rms}$ of molecules of mass $m$ in a gas at absolute temperature $T_0$ satisfies $\frac{1}{2} mv_{rms}^2 = \frac{3}{2} kT_0$ or, on multiplying by Avogadro’s number, $\frac{1}{2} m_n v_{rms}^2 = \frac{3}{2} RT_0$. We conclude that

$$v_{rms} = \sqrt{\frac{3RT_0}{m_n}} \implies v = \sqrt{\frac{\gamma RT_0}{m_n}} = \sqrt{\frac{3RT_0}{3m_n}} = \sqrt{\frac{\gamma}{3}} v_{rms}$$

(15.66)

We thus find that the speed of sound in a gas is a simple multiply—0.683 in the case of air—of the root mean square speed of the molecules of the gas and that both increase as the square root of the temperature!

### 15.1.7 A General 1D Equation

While a differential equation in one dimension is, of course, an ODE, not a PDE, description and illustration of the essential approaches underlying both finite difference methods and finite element methods are simplest in one dimension. We elect to begin there, seeking as an example to solve the general linear, second-order, self-adjoint,\(^9\) inhomogeneous equation

$$-\frac{d}{dx} \left( \alpha(x) \frac{d\varphi(x)}{dx} \right) + \beta(x) \varphi(x) = f(x) \quad \text{or} \quad -\alpha(x) \frac{d^2\varphi(x)}{dx^2} - \alpha'(x) \frac{d\varphi(x)}{dx} + \beta(x) \varphi(x) = f(x)$$

(15.67)

in the interval $0 \leq x \leq L$. Here, $\varphi(x)$ is an unknown function, $\alpha(x)$ and $\beta(x)$ are parameters related to the physical properties of the problem (and may be functions of $x$), $\alpha'(x) = \alpha(x)/dx$, and $f(x)$ is a forcing or source function (inhomogeneity). With appropriate choices of $\alpha$, $\beta$, and $f$, this equation can be reduced to the one-dimensional versions of several of the equations we have developed in the previous subsections.

To be complete, we shall suppose we are dealing with a boundary value problem, taking the desired boundary conditions to be given by

$$\varphi(0) = p$$

(15.68)

and

$$\left[\alpha(x) \frac{d\varphi(x)}{dx} + \gamma \varphi(x)\right]_{x=L} = q$$

(15.69)

where $p$, $\gamma$, and $q$ are known constants. Together, these conditions will allow us to demonstrate how to treat each of the three kinds of boundary conditions. The condition at $x = 0$ is a Dirichlet boundary condition or boundary condition of the first kind; the condition at $x = L$ is a mixed boundary condition or boundary condition of the third kind. When $\gamma = 0$ in this second condition, the condition reduces to a Neumann condition or boundary condition of the second kind.

\(^9\)The word self-adjoint characterizes an equation in which—see the second form in Eq. (15.67)—the coefficient of the first derivative term is the derivative of the coefficient of the second derivative term. While this requirement appears to be restrictive, it turns out—see Exercise 15.3—that any linear, second-order equation can with an appropriately chosen integrating factor be cast in self-adjoint form, so the restriction is only apparent, not real.
15.1.8 A General 2D Equation

As a simple—and general—two-dimensional example, we will illustrate the application of finite difference and finite element analysis to solve an equation of the general form

\[-\frac{\partial}{\partial x} \left( \alpha_x(x, y) \frac{\partial \varphi}{\partial x} \right) - \frac{\partial}{\partial y} \left( \alpha_y(x, y) \frac{\partial \varphi}{\partial y} \right) + \beta(x, y) \varphi = f(x, y)\]  

(15.70)

where \(\alpha_x, \alpha_y\) and \(\beta\) are known quantities—possibly constants; possibly functions of \(x\) and \(y\)—and \(f\)—also possibly constant; possibly a function of \(x\) and \(y\)—is a driving term (inhomogeneity). While this equation is presented in its most general form so that the results of our discussion can be applied to a variety of different physical problems, appropriate restrictions of \(\alpha_x, \alpha_y, \beta,\) and \(f\) will reduce it to the Laplace, Poisson, or Helmholtz equation. To complete the statement of the problem, we suppose that a solution is to be found subject to the Dirichlet conditions

\[\varphi = p \quad \text{(on } \Gamma_1)\]  

(15.71)

on the portion \(\Gamma_1\) of the boundary and the Neumann conditions

\[\left( \alpha_x \frac{\partial \varphi}{\partial x} \hat{i} + \alpha_y \frac{\partial \varphi}{\partial y} \hat{j} \right) \cdot \hat{n} = q \quad \text{(on } \Gamma_2)\]  

(15.72)

on the remainder \(\Gamma_2\) of the boundary. Here, \(p\) and \(q\) are quantities defined on the boundary, and \(\hat{n}\) is a unit vector perpendicular to the boundary and directed outward from the perspective of a viewer in the region in which a solution is sought.

15.2 Finite Difference Methods (FDMs) in One Dimension

In the finite difference approach to the one-dimensional problem laid out in Section 15.1.7, we begin by dividing the interval \(0 \leq x \leq L\) into \(N\) segments, each of length \(\Delta x = L/N,\) so that the \(i\)-th node \((i = 0, 1, 2, \ldots, N)\) has \(x\) coordinate \(x_i = i \Delta x.\) In particular \(x_0 = 0\) and \(x_N = L.\) Then, we evaluate Eq. (15.67) at the point \(x_i\) to find that

\[-\alpha_i \frac{d^2 \varphi(x)}{dx^2} \bigg|_{x_i} - \alpha_i' \frac{d \varphi(x)}{dx} \bigg|_{x_i} + \beta_i \varphi_i = f_i\]  

(15.73)

where \(\alpha_i = \alpha(x_i), \alpha_i' = \alpha'(x_i), \beta_i = \beta(x_i),\) and \(f_i = f(x_i)\) are all known and \(\varphi_i = \varphi(x_i)\) is to be found. Next, we approximate the derivatives by invoking finite differences. We illustrate only the most common of several possible ways to achieve that objective. In terms of the quantities \(\varphi_i,\) we might, for example, write the first derivative at \(x_i\) in any of the ways

\[\frac{d \varphi(x)}{dx} \bigg|_{x_i} \approx \frac{\varphi_{i+1} - \varphi_i}{\Delta x} \quad \text{or} \quad \frac{d \varphi(x)}{dx} \bigg|_{x_i} \approx \frac{\varphi_i - \varphi_{i-1}}{\Delta x} \quad \text{or} \quad \frac{d \varphi(x)}{dx} \bigg|_{x_i} \approx \frac{\varphi_{i+1} - 2 \varphi_i + \varphi_{i-1}}{2 \Delta x}\]  

(15.74)

or probably in other ways as well (see Exercise 15.25). The first of these approximations involves a forward difference, the second involves a backward difference, and the third involves a central difference.\(^{10}\) Each is correct, though they are not all equally convenient or useful—nor is any single one always the most appropriate or convenient. For the present example, we choose the central difference to approximate the first derivative in Eq. (15.73). To find an approximation for the second derivative, however, we use both the forward and the backward approximations for the first derivative, writing that

\[\frac{d^2 \varphi(x)}{dx^2} \bigg|_{x_i} \approx \frac{d \varphi(x)}{dx} \bigg|_{x_i + \Delta x/2} - \frac{d \varphi(x)}{dx} \bigg|_{x_i - \Delta x/2} \approx \frac{\varphi_{i+1} - \varphi_i}{\Delta x} - \frac{\varphi_i - \varphi_{i-1}}{\Delta x} = \frac{\varphi_{i+1} - 2 \varphi_i + \varphi_{i-1}}{\Delta x^2}\]  

(15.75)

\(^{10}\)Note that the central difference formula is the average of the forward and backward formulae.
Here, we have recognized that the forward (backward) difference approximation to the derivative at \( x_i \) is a central difference approximation to the derivative at \( x_i + \frac{1}{2} \Delta x (x_i - \frac{1}{2} \Delta x) \), and we have taken the second derivative at \( x_i \) to be approximated by the difference of these two approximations to the first derivative divided by the separation of the points at which the two first derivatives are evaluated.\(^\text{13}\) Finally, we substitute the approximation of Eq. (15.75) and the central difference approximation of Eq. (15.74) into Eq. (15.73) to find that

\[
-\alpha_i \left( \frac{\varphi_{i+1} - 2\varphi_i + \varphi_{i-1}}{\Delta x^2} \right) \alpha_i' \left( \frac{\varphi_{i+1} - \varphi_{i-1}}{2\Delta x} \right) + \beta_i \varphi_i = f_i
\]

(15.76)

or, on multiplying by \( \Delta x^2 \) and collecting terms with the same index on \( \varphi \), that

\[
\left( -\alpha_i + \frac{\alpha_i' \Delta x}{2} \right) \varphi_{i-1} + \left( 2\alpha_i + \beta_i \Delta x^2 \right) \varphi_i + \left( -\alpha_i - \frac{\alpha_i' \Delta x}{2} \right) \varphi_{i+1} = f_i \Delta x^2
\]

(15.77)

Since \( i \) can assume any value in the interval \( 0 \leq i \leq N \), we thus have \( N + 1 \) equations, just the right number to determine the \( N + 1 \) unknowns \( \varphi_i \).

The conclusion of the last sentence of the previous paragraph, however, is premature. Unfortunately, when evaluated at \( i = 0 \) or \( i = N \), Eq. (15.77) makes reference to \( \varphi_{-1} \) or \( \varphi_{N+1} \), each of which is outside the domain of the problem! Thus, we really have \( N + 3 \) unknowns. The additional information we need lies in the boundary conditions, though the precise way in which these conditions resolve this problem depends on the type of boundary condition. For the Dirichlet condition of Eq. (15.68), we abandon the first equation \( (i = 0) \) altogether, replacing it with the equation prescribed by the boundary conditions, namely

\[
\varphi_0 = p
\]

(15.78)

For the mixed boundary condition of Eq. (15.69), we use a central difference approximation to the derivative in the boundary condition, finding that

\[
\alpha_N \left( \frac{\varphi_{N+1} - \varphi_{N-1}}{2\Delta x} \right) + \gamma \varphi_N = q \implies \varphi_{N+1} = \varphi_{N-1} + \frac{2\Delta x}{\alpha_N} (q - \gamma \varphi_N)
\]

(15.79)

where \( \alpha_N = \alpha(x_N) = \alpha(L) \). Then, we write out the last equation \( (i = N) \) and substitute from Eq. (15.79) to eliminate \( \varphi_{N+1} \), finding ultimately that

\[
-2\alpha_N \varphi_{N-1} + \left[ 2\left( \alpha_N + \gamma \Delta x \right) + \left( \beta_N + \frac{\gamma \alpha_N'}{\alpha_N} \right) \Delta x^2 \right] \varphi_N = \left( f_N + \frac{\alpha_N' q}{\alpha_N} \right) \Delta x^2 + 2q \Delta x
\]

(15.80)

With these resolutions of the first and last equations, we arrive at the now fully defined and complete set

\[
\begin{align*}
\varphi_0 &= p \\
\left( -\alpha_i + \frac{\alpha_i' \Delta x}{2} \right) \varphi_{i-1} + \left( 2\alpha_i + \beta_i \Delta x^2 \right) \varphi_i + \left( -\alpha_i - \frac{\alpha_i' \Delta x}{2} \right) \varphi_{i+1} &= f_i \Delta x^2, \quad 1 \leq i \leq N - 1 \\
-2\alpha_N \varphi_{N-1} + \left[ 2\left( \alpha_N + \gamma \Delta x \right) + \left( \beta_N + \frac{\gamma \alpha_N'}{\alpha_N} \right) \Delta x^2 \right] \varphi_N &= \left( f_N + \frac{\alpha_N' q}{\alpha_N} \right) \Delta x^2 + 2q \Delta x
\end{align*}
\]

(15.81)

of \( N + 1 \) equations for the \( N + 1 \) unknowns \( \varphi_0, \varphi_1, \ldots, \varphi_N \). Remember that, for definiteness, we have chosen to use Dirichlet boundary conditions at \( x = 0 \) and mixed boundary conditions at \( x = L \). In other situations, the conditions at the two ends might both be of the same type, or they could each

\(^{13}\)You may have to read this sentence several times. The appearance of the binomial coefficients \((1, 2, 1)\), as in \( (a + b)^2 = a^2 + 2ab + b^2 \), is worth noting.
be of a type different from those here illustrated. In any case, we have illustrated how to address all three possible types in the specific choices made.

In effect, we have deduced a set of linear algebraic equations to be solved for the unknowns \( \varphi_0, \varphi_1, \ldots, \varphi_N \). For visualization, we note that the set can be seen in matrix form. Further, since each equation involves no more than three consecutive indices, the matrix of the coefficients is tridiagonal though not necessarily symmetric. For example, if \( N = 10 \), the matrix version of these equations would have the general form

\[
\begin{pmatrix}
1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
? & ? & ? & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
\end{pmatrix}
\begin{pmatrix}
\varphi_0 \\
\varphi_1 \\
\varphi_2 \\
\varphi_3 \\
\varphi_4 \\
\varphi_5 \\
\varphi_6 \\
\varphi_7 \\
\varphi_8 \\
\varphi_9 \\
\varphi_{10}
\end{pmatrix}
= \begin{pmatrix}
p \\
? \\
? \\
? \\
? \\
? \\
? \\
? \\
? \\
? \\
? \\
\end{pmatrix}
(15.82)
\]

In this display, only those elements marked with ‘?’ can differ from zero (though some of those may be zero as well), and every non-zero element except those in the column of \( \varphi_i \)'s is known at the outset.

Two cases are important. In the first case, the equations are inhomogeneous and the determinant of the coefficient matrix is not zero; and the equations have a unique solution that will approximate the solution to the original boundary value problem. In the second case, the equations are homogeneous, and we hope that the the coefficient matrix contains a parameter that can be adjusted to make the determinant of that matrix zero; and our problem involves seeking the eigenvalues and eigenvectors of the coefficient matrix; we expect to find several solutions, one corresponding to each eigenvalue of the coefficient matrix.\(^\text{12}\)

Finite difference approaches to PDEs are, of course, subject to error. In general, these errors fall into three distinct and independent categories. In the first instance, we have replaced a continuous variable by a discrete set of values of that variable, i.e., we have discretized the independent variable. In so doing, we have rendered the solution vulnerable to discretization error, an error that will be reduced as the separation of the discrete values is made smaller. From that perspective alone, we would want to make that separation as small as possible, the smaller the better. Unfortunately, as we make the separation smaller, we also increase the computational labor of generating the solution and hence the time required to complete the solution. What’s worse, increasing the computational labor also increases the likelihood that errors will be generated by computer roundoff, which arises because computers retain floating point numbers only to finite precision. We thus must seek a compromise: We want to use a grid with small intervals so as to reduce discretization error but we can’t make the grid too small because roundoff errors may then become significant. Fortunately, in most cases, we will be able to find a grid that is simultaneously fine enough to render discretization error of little consequence and large enough to keep roundoff errors at bay. The standard assessment of these issues involves solving the problem twice, once with a coarser grid and then again with a finer grid. Until roundoff becomes a problem, we can (almost) always safely assume that the solution on the finer grid is more accurate than that on the coarser grid, and a comparison of the two solutions will provide some indication of the accuracy of the solution.

Sometimes, we will use a direct algorithm to solve the discretized equations that convey the solution, and the third type of error will not be present. When an iterative algorithm is used,

\(^{12}\)Two other cases exist but are of no significance. The equations might be inhomogeneous and the determinant of the coefficient matrix zero, in which case no solution exists, or the equations might be homogeneous and the determinant of the coefficient matrix non-zero, in which case only the trivial solution (all \( \varphi \)'s zero) exists.
however, we need be concerned not only about discretization and roundoff errors; we must also assess convergence error. In those cases, we are using a method that may or may not converge to the solution of the approximate equations. We must therefore worry not only about the extent to which the solution to the equations we are solving actually represents the solution to the original problem (discretization and roundoff errors) but also about the extent to which the solution we ultimately deduce is actually a solution to the approximating equations (convergence error). Again, we assess the accuracy of the solution to the approximating equations by comparing successive iterates that we hope are converging on that solution. That hope will be more or less justified depending on the rapidity of the apparent convergence.

We can carry our analysis no further without selecting a particular programming language in which to implement explicit coding. That task is undertaken in the next section(s).

15.6 Using PYTHON to Solve 1D PDEs via an FDM

15.6.1 A General Coding

The final step is to solve Eq. (15.82) for the unknowns \( \varphi_0, \varphi_1, \ldots, \varphi_N \). The numerical operations associated with constructing and then solving Eq. (15.82) are clearly a job for a computer, especially when the problem to be solved involves more than a very few simultaneous equations. A program to construct Eq. (15.82) and then solve the resulting system for \( \varphi \) at each node would begin by asking the user to input the values of the various parameters involved in the problem. In all cases, the parameters \( p, \gamma, \) and \( q \), which relate to the boundary conditions, are constants. In general, the quantities \( \alpha, \alpha', \) and \( \beta \), which relate to the physical system involved, and \( f \), which represents a source or excitation function, will depend on \( x \). For simplicity in illustration, however, we will suppose these four quantities to be constants as well.\(^{13}\) Indeed, if \( \alpha \) is constant, then \( \alpha' \) is zero, and we will simply leave out terms multiplied by \( \alpha' \) when we construct the equations.

The first segment of the program we wish to write will, then, request input of all the constants needed in the remainder of the program. Appropriate PYTHON statements are\(^{14}\)

```
N = input('Enter number of segments (N): '); N=int(N)
alpha = input('Enter alpha: ' ); alpha=float(alpha)
betalpha = input('Enter beta: ' ); beta=float(betalpha)
f = input('Enter f: ' ); f=float(f)
L = input('Enter L: ' ); L=float(L)
p = input('Enter p: ' ); p=float(p)
gamma = input('Enter gamma: ' ); gamma=float(gamma)
q = input('Enter q: ' ); q = float(q)
```

Next, with the statements\(^{15}\)

```
dx = L/N; dx2 = dx**2
x = np.linspace(0.0, L, num=N+1)
```

we calculate the size of each segment \( dx \), the square of that size \( dx^2 \), and the values of \( x \) at which (equally spaced) nodes will be placed in the interval \( 0 \leq x \leq L \). Then, to prepare for using PYTHON's function `np.linalg.solve` to solve the equations, we must create \( cf \), an \((N+1) \times (N+1)\) matrix containing the coefficients of the system to be solved, and \( inhomo \), a vector containing the inhomogeneities. We invoke the statements

---

\(^{13}\)If \( \alpha, \alpha', \beta, \) and \( f \) are not constant, each of these quantities would have to be represented by an \( N+1 \) element vector and values would have to be given for all of those elements.

\(^{14}\)In PYTHON 3, the `input` command returns a string, so explicit conversion to an appropriate data type is necessary. That conversion is not necessary in PYTHON 2, but it does no harm.

\(^{15}\)The module `numpy` will be imported and named `np` before the command file under construction is executed.
inhomo = np.zeros(N+1) + 1.0
inhomo = f*dx2*inhomo
inhomo[0] = p
inhomo[N] = inhomo[N] + 2.0*q*dx

cf = np.zeros([N+1,N+1])
for i in range(N+1):
    cf[i,i] = 2.0*alpha + beta*dx2
cf[0,0] = 1.0
cf[N,N] = cf[N,N] + 2.0*gamma*dx
for i in range(N):
    cf[i,i+1] = -alpha
cf[0,1] = 0.0
for i in range(N):
    cf[i+1,i] = -alpha
cf[N,N-1] = -2.0*alpha

to create the necessary matrix and vector, which reflect Eqs. (15.81) and (15.82). Finally, we exploit
the statement

phi = np.linalg.solve(cf, inhomo)

to solve the system of equations defined by the the coefficient matrix cf and the vector of inhomoge-
nieties inhomo..

A more fully commented command file containing these statements is named fdm1d.py, is listed
in Appendix 15.A.4, and can be copied from the directory $HEAD/python.\textsuperscript{16}

\subsection*{15.6.2 An Example: Simple Harmonic Motion}

We now illustrate the application of fdm1d.py to a simplification of Eq. (15.67), specifically the equation

\[
\frac{d^2\varphi}{dt^2} + k \varphi = 0 \quad ; \quad t \in (0,T)
\]

for the simple harmonic motion of a mass \( m \) attached to a spring of stiffness \( k \). To match this
situation to that discussed in Section 15.1.7, we must interpret \( \alpha \) as a constant equal to \(-m\), \( \beta \) as
a constant equal to \( k \), \( L \) as the time \( T \) at the end of the interval of interest, and the independent
variable \( x \) as the independent variable \( t \). Further, we must set \( f = 0 \). The dependent variable \( \varphi \)
gives the displacement of the oscillator from its equilibrium position. Basically, we are interested in
the position over a range of times \( 0 \leq t \leq T \). In a consistent set of units, we will take

\[
m = -\alpha = 4.0 \text{ kg} \quad ; \quad k = \beta = 3.0 \text{ N/m}
\]

To complete the definition of the problem, we need to specify the boundary conditions.\textsuperscript{17} Suppose
we seek a solution for which \( \varphi(0) = 0 \text{ m} \) and \( d\varphi(t)/dt|_{t=T} = 1.0 \text{ m/s} \), i.e., we specify the position at
\( t = 0 \) and the velocity at \( t = T \). To effect these conditions, we need to assign the values

\[
p = 0.0 \text{ m} \quad ; \quad \gamma = 0.0 \text{ kg/s} \quad ; \quad q = -4.0 \text{ kg m/s}
\]

\textsuperscript{16}See the Local Guide for the meaning of $HEAD at your site.

\textsuperscript{17}Note that we are using a method that requires one item of information at each end of the region of interest—a boundary value problem. More often in problems in motion, one has an initial value problem in which one specifies two items at one end of the region of interest, say an initial position and an initial velocity.
the first of which reduces Eq. (15.68) to $\phi(0) = 0$ m and the rest of which (with $\alpha = -4.0$ kg) reduce Eq. (15.69) to $d\phi(t)/dt|_{t=T} = 1.0$ m/s. In this example, we will seek a solution over the interval $0 \leq t \leq 10$ s, so $L \rightarrow T = 10.0$ s. In physical terms, we start the oscillator at its equilibrium position with an unspecified initial velocity such that, at $t = 10.0$ s, its velocity will be 1.0 m/s.

With these choices, we are now ready to execute the command file and solve the problem. The statement

```python
import numpy as np
execfile('fdm1d.py')
```

will execute the statements in the file, one at a time. The first several statements request input, to which we respond with the values

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>N</td>
<td>20</td>
</tr>
<tr>
<td>$\alpha$</td>
<td>-4.0</td>
</tr>
<tr>
<td>$\beta$</td>
<td>3.0</td>
</tr>
<tr>
<td>$f$</td>
<td>0.0</td>
</tr>
<tr>
<td>$L$</td>
<td>10.0</td>
</tr>
<tr>
<td>$p$</td>
<td>0.0</td>
</tr>
<tr>
<td>$\gamma$</td>
<td>0.0</td>
</tr>
<tr>
<td>$q$</td>
<td>-4.0</td>
</tr>
</tbody>
</table>

Once we have entered these parameters, the remaining statements will construct the coefficient matrix, generate the solution, and store it in an array named $\phi$. We will have no further interaction with the program. Presently, the PYTHON prompt will return. At that point, all variables to which values have been assigned in the execution of the command file will be accessible at PYTHON's main prompt. In particular, we could plot the solution by invoking the statements

```python
import matplotlib.pyplot as plt
plt.plot( x, phi, color='black', linewidth=3 )
plt.title('N=20', fontsize=20 )
plt.grid( color='black' )
```

The resulting graph is shown in the upper left panel of Fig. ??.

To assess the accuracy of this solution, we repeat the process with an increasing number of divisions of the interval in which the solution is sought. To save the solution just generated and generate others for $N = 50$ and $N = 100$, we execute the statements

```python
x020=x; phi020=phi
execfile( 'fdm1d.py' )

Enter number of segments (N): 50
```

```python
x050=x; phi050=phi
execfile( 'fdm1d.py' )

Enter number of segments (N): 100
```

Then, to plot all solutions in a single display, we execute the statements

```python
plt.subplots_adjust(hspace=0.4)
plt.subplot(2,2,1)
```
Figure 15.7: Harmonic motion via finite difference analysis. This graph was produced with PYTHON.

All of these graphs are shown in Fig. 15.7. Further, as revealed most directly in the lower right panel, even the solution with only 20 segments lies quite close to the solutions for 50 and 100 segments. We conclude that the solution we have obtained is accurate, at least to the resolution of the graph paper.

As a further test on the accuracy of the solution, which we expect to improve as the number of segments increases, we compare the solution for $N = 50$ with that for $N = 100$ with the statements

```python
dif = np.zeros(51)
for i in range(50):
```
which suggest that the solution for \( N = 50 \) is accurate to an absolute error of about ±0.012 and that the solution falls in the range \(-1.6 \leq \varphi \leq 1.6\). If we generate a solution for \( N = 200 \), its difference from the solution for \( N = 100 \) ranges from −0.0036 to 0.0029. so the solution for \( N = 100 \) evidently has an absolute error of about ±0.004.

The above tests of accuracy have, of course, assessed only discretization error. To test for roundoff error, we would have to increase the number of nodes even further, looking for the point at which the solution begins to depart from that to which it appears at the moment to be converging. Since we have in this example used a direct method for finding the solution to the system of linear equations, we need not be concerned about convergence error.

Finally, using the solution for \( N = 100 \), we might determine the initial velocity and check the final velocity with the statements

\[
\text{InitialVelocity} = \frac{(\text{phi100}[1] - \text{phi100}[0])}{(\text{x100}[1] - \text{x100}[0])}
\]

\[
\text{print(InitialVelocity)}
\]

\[-1.3820091459050068\]

\[
\text{FinalVelocity} = \frac{(\text{phi100}[100] - \text{phi100}[99])}{(\text{x100}[100] - \text{x100}[99])}
\]

\[
\text{print(FinalVelocity)}
\]

\[0.958655935015019\]

(The results are, of course, in m/s.) Despite the graphical agreement of the solutions for \( N = 50 \) and \( N = 100 \), the final velocity for \( N = 100 \) doesn’t quite match the prescribed boundary value of 1.0 m/s. For comparison, had we used \( N = 200 \), we would have found the initial and final velocities, rounded to four digits after the decimal, to be −1.3847 m/s and 0.9793 m/s, respectively; had we used \( N = 500 \), we would have found those velocities to be −1.3855 and 0.9917 m/s. Apparently, with increasing \( N \), the calculated final velocity converges on the prescribed boundary value.

### 15.9 Finite Element Methods (FEMs) in One Dimension

Finite element methods provide an alternative to finite difference methods for approximating the solution of a boundary value problem—ordinary or partial differential equation plus boundary conditions—in one or more dimensions. The total process involves

1. **Discretizing the domain (preprocessing).** The region of interest is subdivided into a number of small elements by appropriately chosen nodes, at each of which an approximation to the dependent variable \( \varphi \) will be sought. The discretization of the domain through the identification of suitable nodes and elements is known as preprocessing.

2. **Selecting the interpolation or shape functions.** The interpolation or shape functions for approximating the dependent variable within an element are selected. Because of its simplicity, linear interpolation is commonly used. Higher-order polynomials are more accurate but they also result in a more complicated formulation.

3. **Formulating the equations for a single element.** Equations for each element—the elemental equations—are formulated using the Ritz variational method, the Galerkin method, or some alternative and less common method.
4. **Assembling the system of equations.** The full system of equations is then obtained by (1) assembling the elemental equations into a set applying to the entire region and (2) imposing continuity conditions at the interfaces between elements.

5. **Incorporating the boundary conditions.** Constraints imposed at the boundaries of the region of interest are brought to bear to resolve unknowns left undetermined by the previous steps.

6. **Solving the system of equations.** The (probably large) system of simultaneous, linear, algebraic equations deduced at the previous step is solved for the dependent variable at each node. Such numerical methods as Gauss-Jordan elimination, Gaussian elimination with back substitution, and LU (lower-upper) decomposition can all be invoked.

7. **Displaying the solution (postprocessing).** The resulting solution is displayed in various ways. Separate from the finite element method, this last step is known as postprocessing. Postprocessing can be a totally separate process from the other steps, and may even be carried out with different software packages than were used to find the solution itself.

In this and the next section(s), we illustrate the method of finite element analysis by applying these steps to the one-dimensional boundary value problem defined in Section 15.1.7.

### 15.9.1 Discretizing the Domain: Preprocessing

The first step in the finite element method is to divide the region of interest, \(0 \leq x \leq L\), into a selected number of elements. In the one-dimensional case, these elements will be line segments whose endpoints are called nodes. Let \(M\) be the number of elements, \(N = M + 1\) be the number of nodes, and \(l^e\) \((e = 1, 2, 3, \ldots, M)\) be the length of the \(e\)-th element. Furthermore, let \(x_i\) \((i = 1, 2, 3, \ldots, N)\) be the coordinate of the \(i\)-th node. In particular, \(x_1 = 0\) and \(x_N = L\).\(^{18}\) The indices \(i\) are known as the global node numbers. In addition, we introduce a local numbering system, in which the nodes of the \(e\)-th element are denoted by \(x_1^e\) and \(x_2^e\), with \(x_1^e < x_2^e\). In the present one-dimensional context, the locally and globally numbered coordinates of the nodes are related by

\[
x_1^e = x_e, \quad x_2^e = x_{e+1} \quad ; \quad e = 1, 2, 3, \ldots, M
\]

(15.84)

Here, on the left-hand sides, the superscript \(e\) refers to the element and the subscript (1 or 2) refers to the node’s local number. The quantities on the right-hand sides are the globally labeled nodes, and the subscript is a global node number. For example, the two nodes of element 2 are identified locally as \(x_1^2\) and \(x_2^2\) and globally as \(x_2\) and \(x_3\). The numbering both of elements and of nodes is illustrated in Fig. 15.8, and the relationship between global and local node numbers is conveyed explicitly in the connectivity matrix shown in Table 15.1. Further, with this notation, \(l^e = x_{e+1} - x_e = x_2^e - x_1^e\) and, in general, will vary from element to element.

### 15.9.2 Selecting Interpolation or Shape Functions

The next step is to select interpolation functions or shape functions that can be used to approximate \(\varphi(x)\) within an element. For simplicity, we employ linear functions.\(^{19}\) Thus, in the \(e\)-th element, we approximate the unknown function by

\[
\hat{\varphi}^e = a^e + b^e x
\]

(15.85)

where \(a^e\) and \(b^e\) are constants to be determined. For subsequent convenience, however, it is preferable to express this linear relationship in terms of the values \(\hat{\varphi}_1^e\) and \(\hat{\varphi}_2^e\) at the end points of

---

\(^{18}\) For variety (and to develop the ability to think in both numbering schemes), we here elect to start the numbering of nodes at \(i = 1\) rather than at \(i = 0\).

\(^{19}\) As noted earlier, higher order polynomials may be used as well, though more nodes would then be necessary. See Exercises 15.10, 15.11, and 15.12.
the element rather than in terms of the slope $b^{(e)}$ and intercept $a^{(e)}$. Since these end points occur locally at $x_1^{(e)}$ and $x_2^{(e)}$, we have that
\[ \tilde{\varphi}_1^{(e)} = a^{(e)} + b^{(e)} x_1^{(e)} \]
\[ \tilde{\varphi}_2^{(e)} = a^{(e)} + b^{(e)} x_2^{(e)} \] (15.86)

If we solve Eq. (15.86) for $a^{(e)}$ and $b^{(e)}$, substitute the results into Eq. (15.85), and group the terms appropriately, we discover that the approximation in Eq. (15.85) can be written—see Exercise 15.26—in the form
\[ \tilde{\varphi}^{(e)}(x) = \sum_{j=1}^{2} N_j^{(e)}(x) \tilde{\varphi}_j^{(e)} \] (15.87)

where
\[ N_1^{(e)}(x) = \frac{x-x_1^{(e)}}{l^{(e)}} \] (15.88)
\[ N_2^{(e)}(x) = \frac{x-x_2^{(e)}}{l^{(e)}} \] (15.89)

and $l^{(e)} = x_2^{(e)} - x_1^{(e)}$ is the length of the element. Equations (15.88) and (15.89) define the shape functions for the $e$-th element. Graphs of these shape functions are shown in Fig. 15.9. Note that
\[ N_1^{(e)}(x_1^{(e)}) = 1 \quad N_1^{(e)}(x_2^{(e)}) = 0 \]
\[ N_2^{(e)}(x_1^{(e)}) = 0 \quad N_2^{(e)}(x_2^{(e)}) = 1 \] (15.90)
i.e., that one of the shape functions has the value one at the lower end of the element and diminishes linearly to zero at the upper end while the other has the value zero at the lower end and rises linearly to one at the upper end. These properties assure that the sum in Eq. (15.87) has the proper value at each end of the element to which it applies.

15.9.3 Formulating the Equations for a Single Element

The next step in the finite element method is to formulate the equations constraining the solution at the nodes defining a single element. We first define the residual \( r(x) \) of Eq. (15.67) as the difference between the right-hand side and the left-hand side when the approximate solution \( \tilde{\varphi} \) is substituted into the equation, i.e., by

\[
     r(x) = -\frac{d}{dx} \left( \alpha \frac{d \tilde{\varphi}}{dx} \right) + \beta \tilde{\varphi} - f \tag{15.91}
\]

Were \( \tilde{\varphi}(x) \) an exact solution, the residual \( r(x) \) would be identically zero. Since \( \tilde{\varphi} \) is only an approximation to \( \varphi \), however, \( r(x) \) will be nonzero. We define the approximate solution by requiring that \( r(x) \) be as small as possible in some sense. We might, for example, choose to make \( r(x) = 0 \) at a discrete set of points. Still better, we can choose to make an appropriate number of weighted “averages” of \( r(x) \) equal to zero, i.e., we can choose \( \varphi_{1}^{(e)} \) and \( \varphi_{2}^{(e)} \) so that

\[
     R_{i}^{(e)} = \int_{x_{i1}^{(e)}}^{x_{i2}^{(e)}} W_{i}^{(e)}(x) r(x) \, dx = 0 \quad ; \quad i = 1, 2 \tag{15.92}
\]

where the \( R_{i}^{(e)} \) are the weighted residual integrals and the \( W_{i}^{(e)}(x) \) are as yet unspecified weighting functions for the \( e \)-th element. Note that we need as many weighting functions as we have nodes, since there are that many values of \( \tilde{\varphi}_{i}^{(e)} \) to determine. Different choices for these weighting functions yield different—though in the end essentially equivalent—methods of solution. In Galerkin’s method, the weighting functions are chosen as the shape functions \( N_{i}^{(e)}(x) \) used for the expansion of \( \tilde{\varphi} \) in Eq. (15.87). With that choice, the weighted residual integral for the \( e \)-th element is given by

\[
     R_{i}^{(e)} = \int_{x_{i1}^{(e)}}^{x_{i2}^{(e)}} N_{i}^{(e)} \left[ -\frac{d}{dx} \left( \alpha(x) \frac{d \tilde{\varphi}_{i}^{(e)}}{dx} \right) + \beta(x) \tilde{\varphi}_{i}^{(e)} \right] dx - \int_{x_{i1}^{(e)}}^{x_{i2}^{(e)}} N_{i}^{(e)} f(x) \, dx \quad i = 1, 2 \tag{15.93}
\]

20Alternatively, we could use the Ritz method to formulate the system of equations. The Ritz method is a variational method in which the boundary value problem is formulated in terms of a variational expression or functional. The minimum of this functional corresponds to the governing differential equation under the given boundary conditions. To obtain the approximate solution, the functional is minimized with respect to its variables. In this approach, the choice of the weighting functions, which seemed quite arbitrary in the Galerkin approach, is embedded naturally in the development. In all cases where a variational formulation exists, the Galerkin and Ritz methods are equivalent. The Galerkin method is, however, applicable even in cases for which a variational formulation cannot be found.
where Eq. (15.91) was substituted into Eq. (15.92) for \( r(x) \). If the first term is integrated by parts, we find that

\[
R_i^{(e)} = \int_{x_1^{(e)}}^{x_2^{(e)}} \left( \alpha(x) \frac{dN_i^{(e)}}{dx} \frac{d\bar{\phi}_i^{(e)}}{dx} + \beta(x) N_i^{(e)} \bar{\varphi}_i^{(e)} \right) \, dx
\]

\[
- \int_{x_1^{(e)}}^{x_2^{(e)}} N_i^{(e)} f(x) \, dx - \left( \alpha N_i^{(e)} \frac{d\bar{\phi}_i^{(e)}}{dx} \right) \bigg|_{x_1^{(e)}}^{x_2^{(e)}} ; \quad i = 1, 2. \tag{15.94}
\]

Now if Eq. (15.87) is substituted into Eq. (15.94), we find that

\[
R_i^{(e)} = \sum_{j=1}^{2} \tilde{q}_j^{(e)} \int_{x_1^{(e)}}^{x_2^{(e)}} \left( \alpha(x) \frac{dN_i^{(e)}}{dx} \frac{dN_j^{(e)}}{dx} + \beta(x) N_i^{(e)} N_j^{(e)} \right) \, dx
\]

\[
- \int_{x_1^{(e)}}^{x_2^{(e)}} N_i^{(e)} f(x) \, dx - \left( \alpha N_i^{(e)} \frac{d\bar{\phi}_i^{(e)}}{dx} \right) \bigg|_{x_1^{(e)}}^{x_2^{(e)}} ; \quad i = 1, 2 \tag{15.95}
\]

These equations for the weighted residual integrals can also be expressed in matrix form as

\[
\begin{bmatrix}
R_1^{(e)} \\
R_2^{(e)}
\end{bmatrix} =
\begin{bmatrix}
K_{11}^{(e)} & K_{12}^{(e)} \\
K_{21}^{(e)} & K_{22}^{(e)}
\end{bmatrix}
\begin{bmatrix}
\bar{\phi}_1^{(e)} \\
\bar{\phi}_2^{(e)}
\end{bmatrix} - \begin{bmatrix}
b_1^{(e)} \\
b_2^{(e)}
\end{bmatrix} - \begin{bmatrix}
g_1^{(e)} \\
g_2^{(e)}
\end{bmatrix} \tag{15.96}
\]

or more compactly as

\[
\{R^{(e)}\} = \{K^{(e)}\} \{\bar{\phi}^{(e)}\} - \{b^{(e)}\} - \{g^{(e)}\} \tag{15.97}
\]

where

\[
K_{ij}^{(e)} = \int_{x_1^{(e)}}^{x_2^{(e)}} \left( \alpha(x) \frac{dN_i^{(e)}}{dx} \frac{dN_j^{(e)}}{dx} + \beta(x) N_i^{(e)} N_j^{(e)} \right) \, dx \tag{15.98}
\]

\[
b_i^{(e)} = \int_{x_1^{(e)}}^{x_2^{(e)}} N_i^{(e)} f(x) \, dx \tag{15.99}
\]

and

\[
g_i^{(e)} = \left( \alpha N_i^{(e)} \frac{d\bar{\phi}_i^{(e)}}{dx} \right) \bigg|_{x_1^{(e)}}^{x_2^{(e)}} = \left( \alpha N_i^{(e)} \frac{d\bar{\phi}_i^{(e)}}{dx} \right) \bigg|_{x=x_2^{(e)}} - \left( \alpha N_i^{(e)} \frac{d\bar{\phi}_i^{(e)}}{dx} \right) \bigg|_{x=x_1^{(e)}} \tag{15.100}
\]

Note that the matrix \( K \) is necessarily symmetric in \( i \) and \( j \).

In general, the elements will be short, and the functions \( \alpha(x) \) and \( \beta(x) \) will be slowly varying functions of \( x \). Over the range of integration appearing in Eq. (15.98), these functions can often be treated as constants, though with different values for each element. Under those circumstances, Eq. (15.98) can be evaluated analytically by taking the derivatives of the shape functions, substituting those derivatives and the shape functions themselves into Eq. (15.98), and integrating over the element. The result is a \( 2 \times 2 \) matrix whose elements are\(^{22}\)

\[
K_{11}^{(e)} = K_{22}^{(e)} = \frac{\alpha^{(e)} l^{(e)}}{6} + \frac{\beta^{(e)} l^{(e)}}{3} \tag{15.101}
\]

\[
K_{12}^{(e)} = K_{21}^{(e)} = -\frac{\alpha^{(e)} l^{(e)}}{6} + \frac{\beta^{(e)} l^{(e)}}{3} \tag{15.102}
\]

where, in these equations, \( \alpha^{(e)} \) and \( \beta^{(e)} \) stand for approximate (constant) values of \( \alpha(x) \) and \( \beta(x) \) appropriate to the \( e \)-th element, and \( l^{(e)} = x_2^{(e)} - x_1^{(e)} \) is the length of the \( e \)-th element. Similarly, if

\(^{22}\)Recall that \( \int u \, dv = uv - \int v \, du \) where for this case \( u = N_i^{(e)} \) and \( dv = (d/dx) (\alpha^{(e)} \frac{d\bar{\phi}_i^{(e)}}{dx}) \, dx \).

\(^{22}\)The integrals that appear here can, of course, be evaluated manually. An alternative evaluation using a symbol manipulating program is laid out in Appendix 15.B.
$f^{(c)}$ is the (approximate) constant value of $f$ within the $e$-th element, Eq. (15.99) can be evaluated to give

$$b_1^{(e)} = b_2^{(e)} = f^{(e)} \frac{l^{(e)}}{2} \quad (15.103)$$

If, finally, we replace $N_1^{(e)}$ and $N_2^{(e)}$ in Eq. (15.100) with the shape functions in Eqs. (15.88) and (15.89), we have that

$$g_1^{(e)} = -\alpha(x_1^{(e)}) \frac{d\varphi^{(e)}}{dx} \bigg|_{x_1^{(e)}}$$
$$g_2^{(e)} = \alpha(x_2^{(e)}) \frac{d\varphi^{(e)}}{dx} \bigg|_{x_2^{(e)}} \quad (15.104)$$

With $[K]$, $\{b\}$, and $\{g\}$ given by Eqs. (15.101)–(15.104), the _elemental equation_ obtained by requiring $R_i^{(e)}$ as given by Eq. (15.96) to be zero for all $i$ is

$$\begin{bmatrix} K_{11}^{(e)} & K_{12}^{(e)} \\ K_{21}^{(e)} & K_{22}^{(e)} \end{bmatrix} \begin{bmatrix} \varphi_1^{(e)} \\ \varphi_2^{(e)} \end{bmatrix} = \begin{bmatrix} b_1^{(e)} \\ b_2^{(e)} \end{bmatrix} + \begin{bmatrix} g_1^{(e)} \\ g_2^{(e)} \end{bmatrix} \quad (15.105)$$

### 15.9.4 Assembling the System of Equations

The next step is to assemble the elemental equations for _each_ element into a single (large) set describing _all_ elements. The process of assembly is best illustrated with an example. Suppose we divide the interval $0 \leq x \leq L$ into $M = 3$ elements with $N = M + 1 = 4$ nodes. The elemental equation for the first element is given by Eq. (15.105) with the superscript ($e$) set to one, i.e., by

$$\begin{bmatrix} K_{11}^{(1)} & K_{12}^{(1)} \\ K_{21}^{(1)} & K_{22}^{(1)} \end{bmatrix} \begin{bmatrix} \varphi_1^{(1)} \\ \varphi_2^{(1)} \end{bmatrix} = \begin{bmatrix} b_1^{(1)} \\ b_2^{(1)} \end{bmatrix} + \begin{bmatrix} g_1^{(1)} \\ g_2^{(1)} \end{bmatrix} \quad (15.106)$$

which is equivalent to the two equations

$$K_{11}^{(1)} \varphi_1^{(1)} + K_{12}^{(1)} \varphi_2^{(1)} = b_1^{(1)} + g_1^{(1)}$$
$$K_{21}^{(1)} \varphi_1^{(1)} + K_{22}^{(1)} \varphi_2^{(1)} = b_2^{(1)} + g_2^{(1)} \quad (15.107)$$

Note that in Eqs (15.106) and (15.107) we have employed the _local_ numbering system. For the other two elements—also with local node numbers—we have similarly that

$$K_{11}^{(2)} \varphi_1^{(2)} + K_{12}^{(2)} \varphi_2^{(2)} = b_1^{(2)} + g_1^{(2)}$$
$$K_{21}^{(2)} \varphi_1^{(2)} + K_{22}^{(2)} \varphi_2^{(2)} = b_2^{(2)} + g_2^{(2)} \quad (15.108)$$

and

$$K_{11}^{(3)} \varphi_1^{(3)} + K_{12}^{(3)} \varphi_2^{(3)} = b_1^{(3)} + g_1^{(3)}$$
$$K_{21}^{(3)} \varphi_1^{(3)} + K_{22}^{(3)} \varphi_2^{(3)} = b_2^{(3)} + g_2^{(3)} \quad (15.109)$$

From the relationship between the local and global numbering systems as defined in Eq. (15.84), however, it is clear that $x_2^{(1)} = x_1^{(2)} = x_2$ and $x_2^{(2)} = x_1^{(3)} = x_3$. Imposing the condition of continuity at nodes 2 and 3, we conclude then that $\varphi_1^{(1)} = \varphi_2^{(2)}$ and $\varphi_2^{(2)} = \varphi_1^{(3)}$. Now if we once again invoke the relationship in Eq. (15.84) and recognize the correspondences

$$\varphi_1^{(1)} \rightarrow \varphi_1, \quad \varphi_2^{(1)} \rightarrow \varphi_2, \quad \varphi_2^{(2)} \rightarrow \varphi_3, \quad \text{and} \quad \varphi_2^{(3)} \rightarrow \varphi_4$$
linking the (approximate) values at locally numbered nodes to the (approximate) values \( \tilde{\varphi}_1, \tilde{\varphi}_2, \tilde{\varphi}_3, \) and \( \tilde{\varphi}_4 \) at the globally numbered nodes, Eqs. (15.107)–(15.109) become

\[
\begin{align*}
K_{11}^{(1)} \tilde{\varphi}_1 & + \ K_{12}^{(1)} \tilde{\varphi}_2 & = b_1^{(1)} + g_1^{(1)} \\
K_{21}^{(1)} \tilde{\varphi}_1 & + \ K_{22}^{(1)} \tilde{\varphi}_2 & = b_2^{(1)} + g_2^{(1)} \\
\quad & + \ K_{12}^{(2)} \tilde{\varphi}_3 & = b_1^{(2)} + g_1^{(2)} \\
K_{21}^{(2)} \tilde{\varphi}_2 & + \ K_{22}^{(2)} \tilde{\varphi}_3 & = b_2^{(2)} + g_2^{(2)} \\
\quad & + \ K_{12}^{(3)} \tilde{\varphi}_4 & = b_1^{(3)} + g_1^{(3)} \\
K_{21}^{(3)} \tilde{\varphi}_3 & + \ K_{22}^{(3)} \tilde{\varphi}_4 & = b_2^{(3)} + g_2^{(3)}.
\end{align*}
\]

(15.110)

Thus, we have six equations but only four unknowns. Some of these equations must be redundant. If we choose carefully, we might ignore two of them. Alternatively, we can reduce the number of equations to four by replacing the second and third equations with their sum and, similarly, replacing the fourth and fifth equations with their sum to find that Eq. (15.110) becomes

\[
\begin{bmatrix}
K_{11}^{(1)} & K_{12}^{(1)} & 0 & 0 \\
K_{21}^{(1)} & K_{22}^{(1)} + K_{11}^{(1)} & K_{12}^{(2)} & 0 \\
0 & K_{21}^{(2)} & K_{22}^{(2)} + K_{11}^{(3)} & K_{12}^{(3)} \\
0 & 0 & K_{21}^{(3)} & K_{22}^{(3)}
\end{bmatrix}
\begin{bmatrix}
\tilde{\varphi}_1 \\
\tilde{\varphi}_2 \\
\tilde{\varphi}_3 \\
\tilde{\varphi}_4
\end{bmatrix}
= \begin{bmatrix}
b_1^{(1)} \\
b_2^{(1)} + b_1^{(2)} \\
b_2^{(2)} + b_1^{(3)} \\
b_2^{(3)}
\end{bmatrix} + \begin{bmatrix}
g_1^{(1)} \\
g_2^{(1)} + g_1^{(2)} \\
g_2^{(2)} + g_1^{(3)} \\
g_2^{(3)}
\end{bmatrix}.
\]

(15.111)

or more compactly

\[
[K][\tilde{\varphi}] = \{b\} + \{g\}.
\]

(15.112)

In Eq. (15.112), \([K]\) is the assembled stiffness matrix for the three-element problem. The extension to more than three elements is now evident. Note that, no matter how many elements we have in a one-dimensional problem divided into elements with two nodes, the matrix \([K]\) will always be tridiagonal.

We complete this step in the process by working out the elements in the assembled equation. From Eqs. (15.101), (15.102), and (15.111), we conclude that the nonzero elements of \([K]\) can be written as

\[
K_{11} = K_{11}^{(1)} = \frac{\alpha^{(1)} l^{(1)}}{l^{(1)}} + \frac{\beta^{(1)} l^{(1)}}{3}
\]

\[
K_{ii} = K_{ii}^{(i-1)} + K_{11}^{(i)}
= \frac{\alpha^{(i-1)} l^{(i-1)}}{l^{(i-1)}} + \frac{\beta^{(i-1)} l^{(i-1)}}{3} + \frac{\alpha^{(i)} l^{(i)}}{l^{(i)}} + \frac{\beta^{(i)} l^{(i)}}{3} ; \quad i = 2, 3, 4, \ldots, N - 1
\]

\[
K_{NN} = K_{22}^{(M)} = \frac{\alpha^{(M)} l^{(M)}}{l^{(M)}} + \frac{\beta^{(M)} l^{(M)}}{3}
\]

\[
K_{i+1,i} = K_{i,i+1} = K_{12}^{(i)} = -\frac{\alpha^{(i)} l^{(i)}}{l^{(i)}} + \frac{\beta^{(i)} l^{(i)}}{6} ; \quad i = 1, 2, 3, \ldots, N - 1
\]

(15.113)

where (by way of reminder) \(N = M + 1\) is the number of nodes and \(M\) is the number of elements. Similarly, using Eqs. (15.103) and (15.111), we can write the elements of \(\{b\}\) in the form

\[
b_1 = b_1^{(1)} = \frac{f^{(1)} l^{(1)}}{2}
\]
\[ b_i = b_2^{(i-1)} + b_1^{(i)} = f^{(i-1)} \frac{l^{(i-1)}}{2} + f^{(i)} \frac{l^{(i)}}{2} \quad i = 2, 3, 4, \ldots, N - 1 \]

\[ b_N = b_1^{(M)} = f^{(M)} \frac{l^{(M)}}{2} \quad (15.114) \]

Finally, we need to evaluate the elements of \( \{ g \} \). Note that all but the first and last entries in \( \{ g \} \) can be written as

\[ g_i = g_2^{(i-1)} + g_1^{(i)} \quad (15.115) \]

If Eq. (15.104) is substituted into Eq. (15.115), we have

\[ g_i = \alpha \frac{d\tilde{\varphi}}{dx} \bigg|_{x = x_2^{(i-1)}} - \alpha \frac{d\tilde{\varphi}}{dx} \bigg|_{x = x_1^{(i)}} \quad (15.116) \]

However, \( x_2^{(i-1)} = x_1^{(i)} = x_i \) [see Eq. (15.84)] and \( \alpha (d\tilde{\varphi}/dx) \) must be continuous at \( x_i \). As a result, the right-hand side of Eq. (15.116) is zero for \( i = 2, 3 \) and we are left with the vector

\[ \{ g \} = \begin{cases} -\alpha(x_1) \frac{d\tilde{\varphi}}{dx} \bigg|_{x = x_1} \\ 0 \\ 0 \\ \alpha(x_4) \frac{d\tilde{\varphi}}{dx} \bigg|_{x = x_4} \end{cases} \quad (15.117) \]

In general for a problem with \( M \) elements (\( N \) nodes), we would find that

\[ g_i = -\alpha(x_1) \frac{d\tilde{\varphi}}{dx} \bigg|_{x = x_1} \\
\quad i = 2, 3, 4, \ldots, N - 1 \\
\]

\[ g_N = \alpha(x_N) \frac{d\tilde{\varphi}}{dx} \bigg|_{x = x_N} = 0 \quad (15.118) \]

In consequence of the considerations to this point, our system of equations for \( \{ \tilde{\varphi} \} \)—the (approximate) values at the nodes—now assumes the form\(^23\)

\[
\begin{bmatrix}
K_{11} & K_{12} & K_{13} & K_{14} \\
K_{21} & K_{22} & K_{23} & K_{24} \\
K_{31} & K_{32} & K_{33} & K_{34} \\
K_{41} & K_{42} & K_{43} & K_{44}
\end{bmatrix}
\begin{bmatrix}
\tilde{\varphi}_1 \\
\tilde{\varphi}_2 \\
\tilde{\varphi}_3 \\
\tilde{\varphi}_4
\end{bmatrix}
= 
\begin{bmatrix}
b_1 \\
b_2 \\
b_3 \\
b_4
\end{bmatrix}
+ 
\begin{bmatrix}
g_1 \\
0 \\
0 \\
g_4
\end{bmatrix}
\quad (15.119)
\]

Similar expressions can be readily written for cases in which there are more nodes than four.

### 15.9.5 Incorporating the Boundary Conditions

Before we can solve Eq. (15.119) for \( \{ \tilde{\varphi} \} \), we need to incorporate the boundary conditions. First consider the boundary condition of the third kind given in Eq. (15.69). If we replace \( \varphi \) by its approximation \( \tilde{\varphi} \), solve for \( \alpha (d\tilde{\varphi}/dx) \) at \( x = L \), and substitute this into Eq. (15.118), we have that

\[ g_N = \alpha \frac{d\tilde{\varphi}}{dx} \bigg|_{x = L} = q - \gamma \tilde{\varphi}_N \quad (15.120) \]

\(^{23}\)We elect to introduce a symbol for every element in \( K \) even though many of those elements are known in advance to be zero.
or specifically for our case that
\[ g_4 = \alpha \frac{d\tilde{\varphi}}{dx} \bigg|_{x=L} = q - \gamma \tilde{\varphi}_4 \] (15.121)

Thus, the last equation in our system becomes
\[ K_{41} \tilde{\varphi}_1 + K_{42} \tilde{\varphi}_2 + K_{43} \tilde{\varphi}_3 + K_{44} \tilde{\varphi}_4 = b_4 + q - \gamma \tilde{\varphi}_4 \] (15.122)

which we can recast as
\[ K_{41} \tilde{\varphi}_1 + K_{42} \tilde{\varphi}_2 + K_{43} \tilde{\varphi}_3 + (K_{44} + \gamma) \tilde{\varphi}_4 = b_4 + q \] (15.123)

With this rewriting of the fourth equation, our system of equations becomes
\[
\begin{bmatrix}
K_{11} & K_{12} & K_{13} & K_{14} \\
K_{21} & K_{22} & K_{23} & K_{24} \\
K_{31} & K_{32} & K_{33} & K_{34} \\
K_{41} & K_{42} & K_{43} & K_{44} + \gamma
\end{bmatrix} \begin{bmatrix}
\tilde{\varphi}_1 \\
\tilde{\varphi}_2 \\
\tilde{\varphi}_3 \\
\tilde{\varphi}_4
\end{bmatrix} = \begin{bmatrix}
b_1 \\
b_2 \\
b_3 \\
b_4 + q
\end{bmatrix} + \begin{bmatrix}
g_1 \\
0 \\
0 \\
0
\end{bmatrix}
\] (15.124)

In general for a domain with \( N \) nodes, we have
\[ K_{NN} \rightarrow K_{NN} + \gamma = \frac{\alpha^{(M)}}{l^{(M)}} + \frac{\beta^{(M)}}{3} + \gamma \] (15.125)

\[ b_N \rightarrow b_N + q = \frac{f^{(M)}}{2} + q \] (15.126)

and hence the \( N \)th element of \( \{b\} \) is absorbed in \( [K] \) and \( \{b\} \). This operation can always be performed for mixed (and Neumann) boundary conditions.

Imposing the Dirichlet boundary condition of Eq. (15.68) is simpler. Recall that for \( M \) elements, we have \( M+1 \) unknowns \( \tilde{\varphi}_i \) (\( i = 1, 2, 3, \ldots, N \)) and \( M+1 \) equations. However, the Dirichlet boundary condition given in Eq. (15.68) specifies the value of one of these unknowns, specifically \( \tilde{\varphi}_1 \). Thus we actually have \( M + 2 \) equations that need to be simultaneously satisfied. But since we only need as many equations as unknowns, we can replace the first equation of Eq. (15.124) with Eq. (15.68). As a result, \( g_1 \) no longer plays a role in the system of equations, and the entire \( g \) vector has been absorbed.\(^{24}\) The new system of equations for our three-element example is therefore
\[
\begin{bmatrix}
1 & 0 & 0 & 0 \\
K_{21} & K_{22} & K_{23} & K_{24} \\
K_{31} & K_{32} & K_{33} & K_{34} \\
K_{41} & K_{42} & K_{43} & K_{44}
\end{bmatrix} \begin{bmatrix}
\tilde{\varphi}_1 \\
\tilde{\varphi}_2 \\
\tilde{\varphi}_3 \\
\tilde{\varphi}_4
\end{bmatrix} = \begin{bmatrix}
p \\
b_2 \\
b_3 \\
b_4
\end{bmatrix}
\] (15.127)

where \( K_{44} \) and \( b_4 \) have been modified according to Eqs. (15.125) and (15.126).

The symmetry of the coefficient matrix is easily restored. Consider, for example, the second equation in Eq. (15.127), namely
\[ K_{21} \tilde{\varphi}_1 + K_{22} \tilde{\varphi}_2 + K_{23} \tilde{\varphi}_3 + K_{24} \tilde{\varphi}_4 = b_2 \] (15.128)

Since \( \tilde{\varphi}_1 = p \), however, this equation can be recast as
\[ K_{22} \tilde{\varphi}_2 + K_{23} \tilde{\varphi}_3 + K_{24} \tilde{\varphi}_4 = b_2 - K_{21} p \] (15.129)

\(^{24}\)Actually the original first equation \( K_{11} \tilde{\varphi}_1 + K_{12} \tilde{\varphi}_2 + K_{13} \tilde{\varphi}_3 + K_{14} \tilde{\varphi}_4 = b_1 + g_1 \) is merely reinterpreted as an equation determining \( g_1 \) after the solution \( \{\tilde{\varphi}\} \) has been found.
A similar recasting of the other equations leads finally to

\[
\begin{bmatrix}
1 & 0 & 0 & 0 \\
0 & K_{22} & K_{23} & K_{24} \\
0 & K_{32} & K_{33} & K_{34} \\
0 & K_{42} & K_{43} & K_{44}
\end{bmatrix}
\begin{bmatrix}
\tilde{\phi}_1 \\
\tilde{\phi}_2 \\
\tilde{\phi}_3 \\
\tilde{\phi}_4
\end{bmatrix}
= 
\begin{bmatrix}
p \\
b_2 - K_{21}p \\
b_3 - K_{31}p \\
b_4 - K_{41}p
\end{bmatrix}
\]

(15.130)

and once again we have a symmetric system. Further, since all elements in the coefficient matrix and all elements in the vector of inhomogeneities are now known, these four equations uniquely determine the vector \( \{ \tilde{\phi} \} \) of unknowns.\(^{25}\)

We can carry our analysis no further without selecting a particular programming language in which to implement explicit coding. That task is undertaken in the next several sections.

### 15.13 Using PYTHON to Solve 1D PDEs via an FEM

#### 15.13.1 A General Coding

The final step is to solve Eq. (15.130) for the unknowns \( \tilde{\phi}_0, \tilde{\phi}_1, \ldots, \tilde{\phi}_M \). A program to construct Eq. (15.130) and then solve the resulting system for \( \tilde{\phi} \) at each node would begin by asking the user to input the values of the various parameters involved in the problem. In all cases, the parameters \( p, \gamma, \) and \( q \), which relate to the boundary conditions, are constants. In general, the quantities \( \alpha, \alpha', \) and \( \beta \), which relate to the physical system involved, and \( f \), which represents a source or excitation function will depend on \( x \). Further, \( l(e) \), which reflects the particular discretization adopted, will in general depend on the element. For simplicity in illustration, however, we will suppose these five quantities to be constants as well.\(^{26}\) Indeed, if \( \alpha \) is constant, then \( \alpha' \) is zero, and we will simply leave out terms multiplied by \( \alpha' \) when we construct the equations.

The first segment of the program we wish to write will, then, request input of all the constants needed in the remainder of the program. Appropriate PYTHON statements\(^{27}\) (BLOCK 1), including an evaluation of the length of the rod and the coordinates of the nodes, are

```python
M = input( 'Enter number of segments (M): ' ); M = int(M)
alpha = input( 'Enter alpha: ' ); alpha = float(alpha)
beta = input( 'Enter beta: ' ); beta = float(beta)
f = input( 'Enter f: ' ); f = float(f)
l = input( 'Enter l: ' ); l = float(l)
p = input( 'Enter p: ' ); p = float(p)
gamma = input( 'Enter gamma: ' ); gamma = float(gamma)
q = input( 'Enter q: ' ); q = float(q)
length = l*M
x = np.linspace(0.0, l*M, M+1)
```

The next step is to assemble \( [K] \) as given in Eqs. (15.111) and (15.113). To this end, we define \( K \) as an \((M+1) \times (M+1)\) floating array with all elements set to zero. Then, we assign values to the diagonal

---

\(^{25}\) We assume, of course, that the coefficient matrix is not singular.

\(^{26}\) If \( \alpha, \alpha', \beta, f, \) and \( l(e) \) are not constant, each would have to be represented by an appropriately sized vector, and values would have to be given for all elements in these vectors.

\(^{27}\) The \texttt{int} and \texttt{float} functions guarantee that entered quantities will be stored in the proper format (integer or real) for the subsequent calculations, regardless of how they are entered.
and non-zero off-diagonal elements in accordance with Eq. (15.113) with the statements \(^{28,29,30}\) (BLOCK 2)

\[
K = \text{np.zeros}([M+1,M+1]) \quad \# \text{Create } (M+1)\times(M+1) \text{ array of zeros}
\]

\[
S = \alpha/l + \beta*l/3.0 \quad \# \text{Evaluate common quantities}
\]

\[
S2 = 2.0*S
\]

\[
T = -\alpha/l + \beta*l/6.0
\]

\[
K[0,0] = S \quad \# \text{Set diagonal elements of } K
\]

for \(i\) in range(1,\(M\)):

\[
K[i,i] = S2
\]

\[
K[M,M] = S
\]

for \(i\) in range(\(M\)):

\[
K[i+1,i] = T \quad \# \text{main diagonal of } K
\]

\[
K[i,i+1] = T
\]

To complete the assembly, we create the vector \(\{b\}\) as given by Eq. (15.114) in a similar fashion, invoking the statements (BLOCK 3)

\[
b = \text{np.zeros}(M+1) \quad \# \text{Create } M+1 \text{ element vector of zeros}
\]

\[
U = f*l/2.0 \quad \# \text{Evaluate common quantities}
\]

\[
U2 = 2.0*U
\]

\[
b[0] = U \quad \# \text{Set elements of } b
\]

for \(i\) in range(1,\(M\)):

\[
b[i] = U2
\]

\[
b[M] = U
\]

Finally, to produce the system of equations given in Eq. (15.130), we must incorporate the boundary conditions and thus modify \(K\) and \(b\). First, we impose the boundary condition of the third kind through Eqs. (15.125) and (15.126) with the statements (BLOCK 4)

\[
K[M,M] = K[M,M] + \gamma
\]

\[
b[M] = b[M] + q
\]

Then we impose the Dirichlet condition as described in Section 15.9.5 with the statements (BLOCK 5)

\[
K[0,0] = 1.0
\]

\[
b[0] = p
\]

for \(j\) in range(1,\(M\)):

\[
K[j,0] = 0.0
\]

for \(i\) in range(1,\(M\)):

\[
b[i] = b[i] - K[0,i]*p
\]

\[
K[0,i] = 0.0
\]

\(^{28}\) Even though the coefficient matrix in this case is tridiagonal, we elect here not to use PYTHON’s routine trisol but instead to illustrate an approach that will work even if the coefficient matrix is not tridiagonal.

\(^{29}\) Remember that indices in PYTHON start at zero rather than one, so all PYTHON indices are one less than the corresponding subscript on \(K\). More subtly, remember that \(K_{ij}\) is the element in the \(i\)th row and the \(j\)th column, which is denoted \(K[i-1,i-1]\) in PYTHON; i.e., the row and column indices are reversed in PYTHON’s notation. We must here pay attention to that issue.

\(^{30}\) The module numpy will be imported and named np before the command file under construction is executed.
As in Eq. (15.130), $K_{11}$, or $K[0,0]$ in PYTHON terminology, is set equal to one and $b_1$, or $b[0]$, is set equal to $p$. Then, we assign the value zero to $K_{1j}$ for $j = 2, 3, 4, \ldots, M$. Finally, the last two lines multiply all but the first entry in the first column of $K$ by $p$, subtract each of those values from the corresponding element of $b$ and then—note the order of operations—set all but the first element in the first column of $K$ to zero. As a result, we now have in $K$ and $b$ the coefficient matrix and vector of inhomogeneities as in Eq. (15.130), a symmetric system of equations ready to be solved.

To solve this system of equations, which is $[K][\ddot{\phi}] = \{b\}$, we invoke the function `np.linalg.solve` in PYTHON’s `numpy` module with the statement (BLOCK 6)

```python
phi = np.linalg.solve(K,b)
```

Upon return to the PYTHON command prompt, `phi` contains the (approximate) solution to the boundary value problem defined in Section 15.1.7 and $x$ contains the coordinates of the nodes along the rod.

A more fully commented command file containing these statements is named `fem1d.py`, is listed in Appendix 15.C.4, and can be copied from the directory `$HEAD/python`.

### 15.13.2 An Example: Simple Harmonic Motion

We now illustrate the application of `fem1d` to a simplification of Eq. (15.67), specifically the equation

$$m \frac{d^2 \varphi}{dt^2} + k \varphi = 0 \quad ; \quad 0 \leq t \leq T$$  \hspace{1cm} (15.131)

for the simple harmonic motion of a mass $m$ attached to a spring of stiffness $k$. To match this situation to that discussed in Section 15.1.7, we must interpret $\alpha$ as a constant equal to $-m$, $\beta$ as a constant equal to $k$, $L$ as the time $T$ at the end of the interval of interest, and the independent variable $x$ as the independent variable $t$. Further, we must set $f = 0$. The dependent variable $\varphi$ gives the displacement of the oscillator from its equilibrium position. Basically, we are interested in the position over a range of times $0 \leq t \leq T$. In a consistent set of units, we will take $m = -\alpha = 4.0 \text{ kg}$ and $k = \beta = 3.0 \text{ N/m}$.

To complete the definition of the problem, we need to specify the boundary conditions.\(^{32}\) Suppose we seek a solution for which $\varphi(0) = 0 \text{ m}$ and $d\varphi(t)/dt|_{t=T} = 1.0 \text{ m/s}$, i.e., we specify the position at $t = 0$ s and the velocity at $t = T$. To effect these conditions, we need to assign the values

$$p = 0.0 \text{ m} \quad ; \quad \gamma = 0.0 \text{ kg/s} \quad ; \quad q = -4.0 \text{ kg\cdot m/s}$$

the first of which reduces Eq. (15.68) to $\varphi(0) = 0 \text{ m}$ and the rest of which (with $\alpha = -4.0 \text{ kg}$) reduce Eq. (15.69) to $d\varphi(t)/dt|_{t=T} = 1.0 \text{ m/s}$. In this example, we will seek a solution over the interval $0 \leq t \leq 10$ s, so $L \rightarrow T = 10.0$ s. In physical terms, we start the oscillator at its equilibrium position with an unspecified initial velocity such that, at $t = 10.0$ s, its velocity will be $1.0 \text{ m/s}$.

With these choices, we are now ready to execute the command file and solve the problem. The statement

```
execfile('fem1d.py') or exec( open('fem1d.py').read() )
```

\(^{31}\)At some sites, the file may also be located in the PYTHON directory structure such that PYTHON can find it when it is identified only by its name without a prepended path.

\(^{32}\)Note that we are using a method that requires one item of information at each end of the region of interest—a boundary value problem. More often in problems in motion, one has an initial value problem in which one specifies two items at one end of the region of interest, say an initial position and an initial velocity.
will execute the statements in the file, one at a time. The first several statements will request input, to which we will respond with the values

\begin{verbatim}
Enter number of segments (M): 20
Enter alpha: -4.0
Enter beta: 3.0
Enter f: 0.0
Enter l: 0.5
Enter p: 0.0
Enter gamma: 0.0
Enter q: -4.0
\end{verbatim}

Once we have entered these parameters, the remaining statements will construct all necessary matrices and vectors and then generate the solution, storing it in an array named \textit{phi}. We will have no further interaction with the program and, presently, the PYTHON prompt will return. At that point, all variables to which values have been assigned in the execution of the command file will be accessible at PYTHON’s main prompt. In particular, we could plot the solution by invoking the statements

\begin{verbatim}
import matplotlib.pyplot as plt
plt.plot( x, phi, color='black', linewidth=3 )
plt.grid(color='black')
\end{verbatim}

The resulting graph is shown in the upper left panel of Fig. 15.10.

To assess the accuracy of this solution, we repeat the process with an increasing number of divisions of the interval in which the solution is sought. To save the solution just generated and generate others for $M = 50$ and $M = 100$, we execute the statements

\begin{verbatim}
x020 = x ; phi020 = phi
execfile('fem1d.py') or exec( open('fem1d.py').read() )
Enter number of segments (M): 50
. (rest same as for M=20, except l = 10/50 = 0.2)
x050 = x ; phi050 = phi
execfile('fem1d.py') or exec( open('fem1d.py').read() )
Enter number of segments (M): 100
. (rest same as for M=20, except l = 10/100 = 0.1)
x100 = x ; phi100 = phi
\end{verbatim}

Then, to plot all solutions in a single display, we execute the statements

\begin{verbatim}
plt.subplots_adjust(hspace=0.4)
plt.subplot(2,2,1)
plt.plot( x020, phi020, color='black', linewidth=2 )
plt.title( 'M=20', fontsize=12 )
plt.grid( color='black' )
plt.subplot(2,2,2)
plt.plot( x050, phi050, color='black', linewidth=2 )
plt.title( 'M=50', fontsize=12 )
plt.grid( color='black' )
\end{verbatim}

\footnote{Remember that we have written the command file \textit{fem1d.py} to expect $l = L/M$.}
Figure 15.10: Harmonic motion via finite element analysis. This graph was produced with PYTHON.

```python
plt.subplot(2,2,3)
plt.plot( x100, phi100, color='black', linewidth=2 )
plt.title( 'M=100', fontsize=12 )
plt.grid( color='black' )
plt.subplot(2,2,4)
plt.plot( x020, phi020, color='black', linewidth=1 )
plt.plot( x050, phi050, color='black', linewidth=1 )
plt.plot( x100, phi100, color='black', linewidth=1 )
plt.title( 'M=20,50,100', fontsize=12 )
plt.grid( color='black' )
plt.show()

All of these graphs are shown in Fig. 15.10. Further, as revealed most directly in the lower right panel, even the solution with only 20 segments lies quite close to the solutions for 50 and 100 segments. We conclude that the solution we have obtained is accurate, at least to the resolution of the graph paper.

As a further test on the accuracy of the solution, which we expect to improve as the number of segments increases, we compare the solution for \( M = 50 \) with that for \( M = 100 \) with the statements

```python
diff = np.zeros(50)
for i in range(50):
    diff[i] = phi100[2*i] - phi050[i]
[np.max(diff), np.min(diff)]
[0.018974468637044106, -0.016188364494459817]
[np.max(phi100), np.min(phi100)]
[1.6034972190016608, -1.6044899436721143]
```

which suggests that the solution for \( M = 50 \) is accurate to an absolute error of about ±0.02 and that
the solution falls in the range \(-1.6 \leq \varphi \leq 1.6\). If we generate a solution for \(M = 200\), its difference from the solution for \(M = 100\), rounded to five digits after the decimal point, ranges from \(-0.00402\) to \(0.00476\), so the solution for \(N = 100\) evidently has an absolute error of about \(\pm 0.005\).

The above tests of accuracy have, of course, assessed only discretization error. To test for roundoff error, we would have to increase the number of nodes even further, looking for the point at which the solution begins to depart from that to which it appears at the moment to be converging. Since we have in this example used a direct method for finding the solution to the system of linear equations, we need not be concerned about convergence error.

Finally, using the solution for \(M = 100\), we might determine the initial velocity and check the final velocity with the statements

\[
\frac{\text{phi100}[1] - \text{phi100}[0]}{\text{x100}[1] - \text{x100}[0]} \quad \# \text{Display initial velocity} \\
-1.38747
\]

\[
\frac{\text{phi100}[100] - \text{phi100}[99]}{\text{x100}[100] - \text{x100}[99]} \quad \# \text{Display final velocity} \\
0.95704
\]

(The results are, of course, in m/s.) Despite the graphical agreement of the solutions for \(M = 50\) and \(M = 100\), the final velocity for \(M = 100\) doesn’t quite match the prescribed boundary value of 1.0 m/s. For comparison, had we used \(M = 200\), we would have found the initial and final velocities to be \(-1.38606\) m/s and \(0.97890\) m/s, respectively; had we used \(N = 500\), we would have found those velocities to be \(-1.38567\) and \(0.99164\) m/s. Apparently, with increasing \(M\), the calculated final velocity converges on the prescribed boundary value.

### 15.16 Finite Difference Methods (FDMs) in Two Dimensions

We can easily extend the one-dimensional FDM already described to apply to problems involving two independent variables. We must, however, treat initial value problems, such as those involving the wave and diffusion equations, differently from boundary value problems, such as those involving the Laplace equation.

#### 15.16.1 The Wave Equation

As laid out in Section 15.1.1, the wave equation for waves in one dimension involves two independent variables, a spatial coordinate \(x\) locating a point in the one-dimensional medium and a time coordinate \(t\). The displacement \(u(x, t)\) of the medium satisfies Eq. (15.13),

\[
\frac{1}{c^2} \frac{\partial^2 u}{\partial t^2} = \frac{\partial^2 u}{\partial x^2} \quad (15.132)
\]

which is to be solved subject both to the initial conditions

\[
u(x, 0) = f(x) \quad ; \quad \frac{\partial u}{\partial t}(x, 0) = g(x) \quad (15.133)
\]

that specify how the motion of the string is initiated and to appropriate boundary conditions that specify how the medium is “fastened” at its two ends, say at \(x = 0\) and \(x = L\). To illustrate two distinct types of boundary conditions, we suppose here that the displacement of the medium must be zero for all time at \(x = 0\) and the derivative of that displacement must be zero for all time at \(x = L\), i.e., that

\[
u(0, t) = 0 \quad ; \quad \frac{\partial u}{\partial x}(L, t) = 0 \quad (15.134)
\]
For a string, for which \( u(x,t) \) represents a displacement, these conditions stipulate physically that the string is tied down at \( x = 0 \) and free at \( x = L \); for an air column, for which \( u(x,t) \) also represents a displacement, these conditions stipulate that the pipe containing the column is closed at \( x = 0 \) (where there will therefore be a displacement node and a pressure antinode) and open at \( x = L \) (where there will be a displacement antinode and a pressure node).

In one numerical approach to this problem via an FDM, we introduce a set of equally spaced nodes in the spatial dimension but leave the temporal dimension continuous. Suppose we divide the interval \( 0 \leq x \leq L \) into \( N \) elements by \( N+1 \) nodes, each separated from its nearest neighbor(s) by the distance \( \Delta x = L/N \). We denote the coordinates of those nodes by \( x_i = i \Delta x \) with \( i = 0, 1, 2, \ldots, N \). In particular, \( x_0 = 0 \) and \( x_N = L \). Then, we introduce the set of functions \( u_i(t) = u(x_i,t) \), evaluate the PDE of Eq. (15.132) at \( x_i \), and express the spatial derivative at \( x_i \) in terms of finite differences as in Eq. (15.75) to find that

\[
\frac{d^2 u_i}{dt^2} = \frac{c^2}{\Delta x^2} \left( u_{i+1} - 2u_i + u_{i-1} \right) \tag{15.135}
\]

The boundary conditions can now be invoked to resolve problems with this equation when \( i = 0 \) and \( i = N \), at which points the equation refers to \( u \) at points outside the domain of the problem, i.e., to \( u_{-1} \) or \( u_{N+1} \), respectively. At \( x = 0 \), the boundary condition implies that \( u_0 = 0 \), so we can replace Eq. (15.135) at \( i = 0 \) with the equation \( u_0 = 0 \), though—for the sake of a similar treatment of all nodes—it is more appropriate to regard \( u_0 = 0 \) as the solution to the initial value problem

\[
\frac{d^2 u_0}{dt^2} = 0 ; \quad u_0(0) = 0 , \quad \frac{du_0}{dt}(0) = 0 \tag{15.136}
\]

To address Eq. (15.135) at \( i = N \), we look to a central difference approximation of the spatial derivative, finding that

\[
\frac{\partial u}{\partial x}(L,t) \approx \frac{u_{N+1}(t) - u_{N-1}(t)}{2\Delta x} = 0 \implies u_{N+1} = u_{N-1} \tag{15.137}
\]

Then, we write Eq. (15.135) for \( i = N \), substitute \( u_{N-1} \) for \( u_{N+1} \), and solve the result for \( \frac{d^2 u_N}{dt^2} \). In this approach, we have discretized only the spatial variable, and the task of solving a PDE in two variables becomes one of solving the coupled set

\[
\frac{d^2 u_0}{dt^2} = 0 \tag{15.138}
\]

\[
\frac{d^2 u_i}{dt^2} = \frac{c^2}{\Delta x^2} \left( u_{i+1} - 2u_i + u_{i-1} \right) , \quad i = 1, 2, 3, \ldots, N-1 \tag{15.139}
\]

\[
\frac{d^2 u_N}{dt^2} = \frac{c^2}{\Delta x^2} \left( -2u_N + 2u_{N-1} \right) \tag{15.140}
\]

of ODEs subject to the initial conditions

\[
u_i(0) = f(x_i) ; \quad \frac{du_i}{dt}(0) = g(x_i) \tag{15.141}\]

where, to reflect properly the conditions imposed on \( u_0 \), we suppose that \( f(x_0) = f(0) = 0 \) and \( g(x_0) = g(0) = 0 \). By discretizing the spatial variable, we have reduced our problem to a problem of the sort addressed in Chapter 11, and we therefore say no more about it here.

We could go one step further, discretizing also the time variable by introducing a time step \( \Delta t \) and the discrete time instants \( t_j = j \Delta t, \ j = 0, 1, 2, \ldots \). Then, approximating both the spatial and the temporal derivatives by finite differences and introducing the quantities \( u_{i,j} = u(x_i, t_j) \), we
Figure 15.11: Geometry for full discretization of the wave equation. Equation 15.143 involves the points marked with solid figures and has been solved to express the solution at the (one) point marked with a solid square in terms of the solutions at the (four) points marked with solid circles.

might evaluate the original PDE of Eq. (15.132) at \((x,t) = (x_i,t_j)\) and find the fully discretized form

\[
\frac{u_{i,j+1} - 2u_{i,j} + u_{i,j-1}}{\Delta t^2} = \frac{c^2}{\Delta x^2} \left( u_{i+1,j} - 2u_{i,j} + u_{i-1,j} \right)
\]  

(15.142)

Solving for \(u_{i,j+1}\) and introducing \(\alpha = \frac{c^2 \Delta t^2}{\Delta x^2}\), we find the simpler form

\[
u_{i,j+1} = \alpha u_{i+1,j} + 2(1 - \alpha)u_{i,j} + \alpha u_{i-1,j} - u_{i,j-1}\]  

(15.143)

Evidently, if we know values of \(u_{i,j}\) for \(j = 0\) and \(j = 1\), we can determine values for \(j = 2\), then for \(j = 3\), … . That is, if we know \(u_{i,j}\) along two consecutive horizontal lines in Fig. 15.11, we can step the solution forward in time from line to line by increments of \(\Delta t\), going as far as our interest dictates and our patience endures.

As in the partial discretization of the previous paragraph, the boundary conditions help us deal with unknown values that appear when Eq. (15.143) is evaluated at \(i = 0\) or \(i = N\) by stipulating that

\[
u_{0,j} = 0 \quad \text{and} \quad \frac{\partial u}{\partial x}(L,t_j) \approx \frac{u_{N+1,j} - u_{N-1,j}}{2\Delta x} = 0 \quad \Rightarrow \quad u_{N+1,j} = u_{N-1,j}
\]

(15.144)

Free of values outside the interval \(0 \leq x \leq L\), the equations we use to step the solution to \(t_{j+1}\) from knowledge of the solution at previous times then are

\[
u_{0,j+1} = 0
\]

(15.145)

\[
u_{i,j+1} = \alpha u_{i+1,j} + 2(1 - \alpha)u_{i,j} + \alpha u_{i-1,j} - u_{i,j-1}, \quad i = 1, 2, \ldots, N - 1
\]

(15.146)

\[
u_{N,j+1} = 2\alpha u_{N-1,j} + 2(1 - \alpha)u_{N,j} - u_{N,j-1}
\]

(15.147)

In their turn, the initial conditions start the process by stipulating that

\[
u_{i,0} = f(x_i) \quad \text{and} \quad u_{i,1} = u_{i,0} + g(x_i)\Delta t
\]

(15.148)
For consistency with the imposed boundary conditions, we must, of course, require that $f(0) = 0$ and $dg(x)/dx|_{x=L} = 0$. We leave it to Exercise 15.18 to demonstrate that the method described in this paragraph will be unstable unless $\alpha \leq 1$.

The discussion at the end of Section 15.2 on discretization and roundoff errors in finite difference methods applies as much in two dimensions as in one. Since solution of the wave equation involves a direct method, however, convergence error as discussed in that earlier section is not here an issue.

We can proceed no further in this example without resorting to a specific coding language, so we postpone further discussion to later sections.

15.16.2 The Diffusion Equation

As laid out in Section 15.1.2, the diffusion equation for heat flow in one dimension involves two independent variables, a spatial coordinate $x$ locating a point in the one-dimensional medium and a time coordinate $t$. The temperature $u(x, t)$ at position $x$ and time $t$ in the medium satisfies Eq. (15.26),

$$\frac{\partial u}{\partial t} = \alpha^2 \frac{\partial^2 u}{\partial x^2}$$

which is to be solved subject to the initial condition

$$u(x, 0) = f(x)$$

that specifies the initial temperature distribution in the medium and to appropriate boundary conditions that specify the way the temperature is controlled at the two ends, say at $x = 0$ and $x = L$. To illustrate both possibilities in a single example, we suppose that, at $x = 0$, the temperature is maintained at the value $T_0$ and that, at $x = L$, the medium is insulated so that no heat flow takes place either into or out of the rod at that end, i.e., we suppose that

$$u(0, t) = T_0 \quad \text{and} \quad \frac{\partial u}{\partial x}(L, t) = 0$$

In one numerical approach to this problem via an FDM, we introduce a set of equally spaced nodes in the spatial dimension but leave the temporal dimension continuous. Suppose we divide the interval $0 \leq x \leq L$ into $N$ elements by $N+1$ nodes, each separated from its nearest neighbor(s) by the distance $\Delta x = L/N$. We denote the coordinates of those nodes by $x_i = i \Delta x$ with $i = 0, 1, 2, \ldots, N$. In particular, $x_0 = 0$ and $x_N = L$. Then, we introduce the set of functions $u_i(t) = u(x_i, t)$, evaluate the PDE of Eq. (15.149) at $x_i$, and express the spatial derivative at $x_i$ in terms of finite differences as in Eq. (15.75) to find that

$$\frac{du_i}{dt} = \frac{\alpha^2 c}{\Delta x^2} \left( u_{i+1} - 2u_i + u_{i-1} \right)$$

The boundary conditions can now be invoked to resolve problems with this equation when $i = 0$ and $i = N$, at which points the equation refers to $u$ at points outside the domain of the problem, i.e., to $u_{-1}$ or $u_{N+1}$, respectively. At $x = 0$, the boundary condition implies that $u_0 = T_0$, so we can replace Eq. (15.152) at $i = 0$ with the equation $u_0 = T_0$, though—for the sake of a similar treatment of all nodes—it is more appropriate to regard $u_0 = T_0$ as the solution to the initial value problem

$$\frac{du_0}{dt} = 0 \quad ; \quad u_0(0) = T_0$$

To address Eq. (15.152) at $i = N$, we look to a central difference approximation of the spatial derivative, finding that

$$\frac{\partial u}{\partial x}(L, t) \approx \frac{u_{N+1}(t) - u_{N-1}(t)}{2\Delta x} = 0 \implies u_{N+1} = u_{N-1}$$
Then, we write Eq. (15.152) for $i = N$, substitute $u_{N-1}$ for $u_{N+1}$, and solve the result for $du_N/dt$. In this approach, in which we have discretized only the spatial variable, the problem of solving a PDE in two variables becomes one of solving the coupled set

\[
\frac{du_0}{dt} = 0 \quad (15.155)
\]

\[
\frac{du_i}{dt} = \frac{\alpha^2}{\Delta x^2} \left( u_{i+1} - 2u_i + u_{i-1} \right) , \quad i = 1, 2, 3, \ldots, N - 1 \quad (15.156)
\]

\[
\frac{du_N}{dt} = 2\frac{\alpha^2}{\Delta x^2} \left( u_{N-1} - u_N \right) \quad (15.157)
\]

of ODEs subject to the initial condition

\[
u_i(0) = f(x_i) \quad (15.158)
\]

where, to reflect properly the conditions imposed on $u_0$, we suppose that $f(x_0) = f(0) = T_0$ and $df(x)/dx|_{x=L} = 0$. By discretizing the spatial variable, we have reduced our problem to a problem of the sort addressed in Chapter 11, and we therefore say no more about it here.

We could go one step further, discretizing also the time variable by introducing a time step $\Delta t$ and the discrete time instants $t_j = j \Delta t, \ j = 0, 1, 2, \ldots$. Then, approximating both the spatial and the temporal derivatives by finite differences and introducing the quantities $u_{i,j} = u(x_i, t_j)$, we might evaluate the original PDE of Eq. (15.149) at $(x, t) = (x_i, t_j)$ and find the fully discretized form

\[
\frac{u_{i,j+1} - u_{i,j}}{\Delta t} = \frac{\alpha^2}{\Delta x^2} \left( u_{i+1,j} - 2u_{i,j} + u_{i-1,j} \right) \quad (15.159)
\]

Solving for $u_{i,j+1}$ and introducing $\gamma = \alpha^2 \Delta t/\Delta x^2$, we find the simpler form

\[
u_{i,j+1} = \gamma u_{i-1,j} + (1 - 2\gamma)u_{i,j} + \gamma u_{i+1,j} \quad (15.160)
\]

Evidently, if we know values of $u_{i,j}$ for $j = 0$, we can determine values for $j = 1$, then for $j = 2$, \ldots. That is, if we know $u_{i,j}$ along one horizontal line in Fig. 15.12, we can step the solution forward in time from line to line by increments of $\Delta t$, going as far as our interest dictates and our patience endures.

As in the partial discretization of the previous paragraph, the boundary conditions help us deal with unknown values that appear when Eq. (15.160) is evaluated at $i = 0$ or $i = N$ by stipulating that

\[
u_{0,j} = T_0 \quad \text{and} \quad \frac{\partial u}{\partial x}(L, t_j) \approx \frac{u_{N+1,j} - u_{N-1,j}}{2 \Delta x} = 0 \implies u_{N+1,j} = u_{N-1,j} \quad (15.161)
\]

Free of values outside the interval $0 \leq x \leq L$, the equations we use to step the solution to $t_{j+1}$ from knowledge of the solution at previous times then are

\[
u_{0,j+1} = T_0 \quad (15.162)
\]

\[
u_{i,j+1} = \gamma u_{i-1,j} + (1 - 2\gamma)u_{i,j} + \gamma u_{i+1,j} \quad , \quad i = 1, 2, \ldots, N - 1 \quad (15.163)
\]

\[
u_{N,j+1} = 2\gamma u_{N-1,j} + (1 - 2\gamma)u_{N,j} \quad (15.164)
\]

In their turn, the initial conditions start the process by stipulating that

\[
u_{i,0} = f(x_i) \quad (15.165)
\]
Figure 15.12: Geometry for full discretization of the diffusion equation. Equation 15.160 involves the points marked with solid figures and has been solved to express the solution at the (one) point marked with a solid square in terms of the solutions at the (three) points marked with solid circles.

For consistency with the imposed boundary conditions, we must, of course, require that $f(0) = T_0$. We leave it to Exercise 15.19 to demonstrate that the method described in this paragraph will be unstable unless $\gamma \leq 1/2$.

The discussion at the end of Section 15.2 on discretization error and roundoff errors in finite difference methods applies as much in two dimensions as in one. Since solution of the diffusion equation involves a direct method, however, convergence error as discussed in that earlier section is not here an issue.

We can proceed no further in this example without resorting to a specific coding language, so we postpone further discussion to later sections.

15.16.3 The Laplace Equation

As laid out in Section 15.1.3, the Laplace equation for the steady state temperature distribution in a two-dimensional plate involves two independent spatial variables $x$ and $y$ which, together, locate a point in the two-dimensional medium. The temperature $u(x,y)$ at the point $(x,y)$ satisfies Eq. (15.27),

$$\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} = 0 \quad (15.166)$$

which is to be solved subject to boundary conditions that specify the way the temperature is controlled at the boundaries of the region.\(^{34,35}\)

Stipulation of boundary conditions is more complicated for a problem involving two spatial dimensions than for the problems in Sections 15.16.1 and 15.16.2, each of which involved only one spatial dimension. In one spatial dimension, the boundary consists of two points, one at each end of the region. In two spatial dimensions, the boundary is defined by a closed curve—whatever is

\(^{34}\)In this case, there are no initial conditions, since we seek only the final steady state temperature, which is determined solely by the (time-independent) boundary conditions. The initial temperature distribution is entirely irrelevant.

\(^{35}\)Note that this approach to the Laplace equation was also discussed in Section 9.3.1, though the example discussed there involved only Dirichlet boundary conditions.
necessary to bound the region of interest—in the \( xy \) plane. Easy application of FDMs is limited to fairly simple geometries in which the boundaries lie along coordinate lines in one or another of the standard coordinate systems. In the present example, we will suppose the region of interest to be a square, each of whose edges has length \( L \), and we will suppose that its four corners lie at \((x, y) = (0, 0), (L, 0), (L, L), \) and \((0, L)\) in the \( xy \) plane. Then, so as to illustrate both possible types of boundary condition, we will suppose that

\[
\begin{align*}
  u(x, 0) &= 0 ; \quad u(x, L) = 100 ; \quad u(0, y) = 100 \frac{y}{L} ; \quad \frac{\partial u}{\partial x}(L, y) = 0
\end{align*}
\]

i.e., that the temperature along the line \( y = 0 \) is maintained at the value 0, that the temperature along the line \( y = L \) is maintained at the value 100, that the temperature along the line \( x = 0 \) rises linearly from 0 at \( y = 0 \) to 100 at \( y = L \), and that the edge along the line \( x = L \) is insulated.

The partial discretization of a PDE as illustrated in the previous two sections is applicable only to initial value problems and does not provide a suitable approach to the present boundary value problem. We can approach the present problem successfully only through full discretization of the PDE. Thus, we divide each edge into \( N \) segments each of length \( \Delta x = L/N \), placing the \((N + 1)^2\) nodes at the points \((x_i, y_j)\), where \( x_i = i \Delta x \; (i = 0, 1, 2, \ldots, N) \) and \( y_j = j \Delta x \; (j = 0, 1, 2, \ldots, N) \). Then, we introduce the \((N + 1)^2\) values \( u_{i,j} = u(x_i, y_j) \), evaluate Eq. (15.166) at the point \((x_i, y_j)\), and express each second partial derivative in terms of finite differences to find that

\[
\frac{u_{i+1,j} - 2u_{i,j} + u_{i-1,j}}{\Delta x^2} + \frac{u_{i,j+1} - 2u_{i,j} + u_{i,j-1}}{\Delta x^2} = 0
\]

or, solving for \( u_{i,j} \), that

\[
u_{i,j} = \frac{1}{4} \left( u_{i+1,j} + u_{i-1,j} + u_{i,j+1} + u_{i,j-1} \right)
\]

Interestingly, in solutions to Laplace’s equation via an FDM, the value at a particular node is equal to the average of the values at the four nearest neighbors of that node. This geometry is shown in Fig. 15.13.

The boundary conditions can now be invoked to resolve problems with Eq. (15.169) when \( i = 0 \) or \( i = N \) and/or \( j = 0 \) or \( j = N \), at which points the equation refers to one or more nodes lying outside the boundaries of the region in which a solution is sought. We must deal with eight different categories of such nodes:

- nodes on the bottom edge but not at a corner, for which \( j = 0 \) and \( i = 1, 2, \ldots, N - 1 \). Along this edge, the boundary condition stipulates that \( u_{i,0} = 0 \), and we replace Eq. (15.169) with this alternative.
- nodes on the top edge but not at a corner, for which \( j = N \) and \( i = 1, 2, \ldots, N - 1 \). Along this edge, the boundary condition stipulates that \( u_{i,0} = 100 \), and we replace Eq. (15.169) with this alternative.
- nodes on the left edge but not at a corner, for which \( i = 0 \) and \( j = 1, 2, \ldots, N - 1 \). Along this edge, the boundary condition stipulates that \( u(0, y) = 100 \frac{y}{L} \), or \( u_{0,j} = 100 \frac{y_j}{L} \), and we replace Eq. (15.169) with this alternative.
- nodes on the right edge but not at a corner, for which \( i = N \) and \( j = 1, 2, \ldots, N - 1 \). This edge is to be insulated, i.e., \( \frac{\partial u}{\partial x} = 0 \) on that edge. We use a central difference approximation to the derivative to find that

\[
\frac{\partial u}{\partial x}(x, y) \bigg|_{x=L} = 0 \quad \Rightarrow \quad \frac{u_{N+1,j} - u_{N-1,j}}{2 \Delta x} = 0 \quad \Rightarrow \quad u_{N+1,j} = u_{N-1,j}
\]
Figure 15.13: Geometry for full discretization of the Laplace equation. Equation 15.169 involves the points marked with solid figures and has been solved to express the solution at the (one) point marked with a solid square in terms of the solutions at the (four) points marked with solid circles.

For these nodes, Eq. (15.169) becomes

\[
\begin{align*}
    u_{N,j} &= \frac{1}{4} \left( u_{N+1,j} + u_{N-1,j} + u_{N,j+1} + u_{N,j-1} \right) \\
    &= \frac{1}{4} \left( 2u_{N-1,j} + u_{N,j+1} + u_{N,j-1} \right) \\
    &= \frac{1}{4} \left( 2u_{N-1,j} + u_{N,j+1} + u_{N,j-1} \right)
\end{align*}
\]  

(15.171)

and, as long as \( j \) stays in the specified range, this equation no longer involves points outside the region defined by the boundaries.

- the node at the lower left corner for which \( i = 0 \) and \( j = 0 \). This point lies on two of the bounding edges. Fortunately, in the present case, the boundary conditions on those two edges are consistent, and we simply replace Eq. (15.169) with \( u_{0,0} = 0 \). In fact, however, this point will never enter into any equation we will need to consider, so the value we assign at this one point is of no consequence. Indeed, in some problems, there may be a discontinuity in the temperature at an isolated point such as this one. Conveniently, the method we have adopted is not upset by such a discontinuity.

- the node at the upper left corner, for which \( i = 0 \) and \( j = N \). This point is similar to the point \((i,j) = (0,0)\), even to the consistency of the values on the two edges to which it belongs. We set \( u_{0,N} = 100 \).

- the node at the lower right corner, for which \( i = N \) and \( j = 0 \). Here, the boundary condition on the lower edge dictates that we should set \( u_{N,0} = 0 \). Applied to this node, however, Eq. (15.171) with \( j = 0 \) suggests that we should require that

\[
    u_{N,0} = \frac{1}{4} \left( 2u_{N-1,0} + u_{N,1} + u_{N,-1} \right)
\]  

(15.172)

Consistency with the presumed value \( u_{N,0} = 0 \) and the known value \( u_{N-1,0} = 0 \) then implies that we should expect that

\[
    u_{N,-1} = -u_{N,1}
\]  

(15.173)

a result that we might also infer from symmetries if we saw the problem of interest as the upper half of a larger problem obtained by reflecting the problem we are addressing into the
region \(-L < y < 0\), making the temperature on the bottom edge of that larger region \(-100\). We conclude that taking \(u_{N,0} = 0\) is appropriate and justified.

- the node at the upper right corner, for which \(i = N\) and \(j = N\). This point is similar to the point \((i, j) = (N, 0)\) and, without further argument, we accept the replacement \(u_{N,N} = 100\).

In short, the equations we seek to solve, now involving no quantities at points outside the boundaries of the problem, are

\[
\begin{align*}
 u_{i,j} &= \frac{1}{4} \left( u_{i+1,j} + u_{i-1,j} + u_{i,j+1} + u_{i,j-1} \right) ; \quad 1 \leq i,j \leq N - 1 \\
 u_{i,0} &= 0 ; \quad 1 \leq i \leq N - 1 \\
 u_{i,N} &= 100 ; \quad 1 \leq i \leq N - 1 \\
 u_{0,j} &= 100 \frac{y_j}{L} ; \quad 1 \leq j \leq N - 1 \\
 u_{N,j} &= \frac{1}{4} \left( 2u_{N-1,j} + u_{N,j+1} + u_{N,j-1} \right) ; \quad 1 \leq j \leq N - 1 \\
 u_{0,0} &= 0 \\
 u_{0,N} &= 100 \\
 u_{N,0} &= 0 \\
 u_{N,N} &= 100
\end{align*}
\]

We can proceed no further in this example without resorting to a specific coding language, so we postpone further discussion to later sections. We do note here, however, that iterative methods will almost always be used to solve this set of equations. Consequently (and in contrast to the situation with the wave and diffusion equations), we will have to pay attention not only to discretization and roundoff errors but also to convergence error.

### 15.20 Using PYTHON to Solve 2D PDEs via an FDM

To complete the solution of the problems laid out in Section 15.16, we must choose a specific language for the development of explicit coding to implement the algorithms described. In this section, we illustrate that process in PYTHON.

#### 15.20.1 The Wave Equation

The final step in addressing the example laid out in Section 15.16.1 is to set up and solve Eqs. (15.145)–(15.147). Using a command file in PYTHON, we would begin by requesting input of all necessary parameters and assuring that each is stored with the proper data type. The statements

\[
\begin{align*}
 N &= \text{input(‘Enter number of segments (N):’)} ; \quad N = \text{int}(N) \\
 dt &= \text{input(‘Enter time step (dt):’) ; \quad dt = \text{float}(dt)} \\
 T &= \text{input(‘Enter number of time steps (T):’) ; \quad T = \text{int}(T)} \\
 c &= \text{input(‘Enter speed of propagation (c):’) ; \quad c = \text{float}(c)} \\
 L &= \text{input(‘Enter length of string (L):’) ; \quad L = \text{float}(L)}
\end{align*}
\]

accomplish those objectives. Then, we would determine the length of each segment, establish values for the coordinates at which solutions will be generated, and evaluate and display the one parameter \(\alpha\) with the statements\(^{36}\)

\(^{36}\)The modules \texttt{numpy} and \texttt{matplotlib.pyplot} will be imported and named \texttt{np} before the command file under construction is executed.
```
dx = L/N; x = np.linspace(0.0,L,num=N+1)
alpha = c**2*dt**2/dx**2

Prudence dictates the wisdom of adding the statement

    print( alpha )
    if alpha > 1.0:
        print( 'Error: alpha > 1; execution halted' )
        exit()
```
to display the value of alpha and halt execution if the parameter α has a value that guarantees an unstable—and hence inaccurate—solution.

As we saw in the general discussion in Section 15.16.1, generation of the solution at the next time instant requires knowledge of the solution at the current time instant and of the solution at the immediate past time instant. At each step of the way, we need preserve only the current and immediate past solutions, but we must keep those solutions until the solution at the next time instant has been generated. In essence, we need three vectors of dimension N + 1 for storing solutions. We declare, therefore, that at any step in the process, the vector u1 will store the past solution, the vector u2 will store the current solution, and the vector u3 will receive the solution at the next time instant as it is generated. Then, once the new solution has been generated and displayed (graphed or written to a file), we no longer need the values in u1, so we will move those in u2 to u1 and those in u3 to u2, thus preparing for the next pass through a loop that advances the solution from time instant to time instant. These three variables are prepared with the statements

```
u1 = np.zeros( N+1 ) # For past solution
u2 = np.zeros( N+1 ) # For current solution
u3 = np.zeros( N+1 ) # For next solution
```

Next, before coding the loop that will generate the solution, step by step, we must initialize the values in u1 to reflect the initial displacement of the string, initialize the values in u2 to reflect the impact of the initial velocity on the motion during the first time step, and display u1 and u2. To be specific, let us suppose that, initially, the string is displaced in the shape of a single hump of a sine wave, but only in the middle quarter of its length, and is not displaced over the first three-eighths and the last three eighths of its length. Thus, we suppose that

\[
  f(x) = \begin{cases} 
    0 & 0 \leq x \leq \frac{3}{8}L \\
    1 + \cos \frac{8\pi}{L} \left( x - \frac{L}{2} \right) & \frac{3}{8}L \leq x \leq \frac{5}{8}L \\
    0 & \frac{5}{8}L \leq x \leq L 
  \end{cases}
\]  

which results in a smooth transition from zero displacement outside the center one quarter of the string and the sinusoidal displacement in that interval. Further, we suppose that the string is released from rest so that

\[
  g(x) = 0
\]

or, equivalently—see Eq. (15.148)—\( u_{i,0} = u_{i,1} \) or \( u2 = u1 \). The coding that will impose these initial conditions, display the initial solution, and then display the solution after the first time step then is\(^{37}\)

---

\(^{37}\)The calculation of the range of i to be used is complicated. The PYTHON function int will truncate the quantities 3.0*N/8.0 and 5.0*N/8.0. We really want the lower limit to be raised rather than truncated. The upper limit can be truncated. Thus, we add 1 to the lower limit.
\[ b = 8.0 \cdot \pi / L \]

for \( i \) in range( int(3.0 \cdot N/8.0), int(5.0 \cdot N/8.0) ):
   \[ u_1[i] = 1.0 + \cos(b \cdot (x[i]-L/2.0)) \]
plt.plot( x, u1, color='black', linewidth=3 )
plt.ylim( [-2.0,2.0] )
plt.show()

Here, we have recognized that the displacement over time will range from \(-2.0\) to \(+2.0\), and we have set the vertical scale to display that range. We will need to include that specification for each subsequent invocation of the command `plot`. Next, because zero initial velocity means that the solution at the second time is the same as that at the first time, we execute the statements\(^{38}\)

\[ u_2 = u_1.copy() \]
plt.plot( x, u2, color='black', linewidth=3 )
plt.ylim( [-2.0,2.0] )
plt.show()

to take the first step and plot the solution at that step.

Now, we are ready to code the algorithm that uses Eqs. (15.145)–(15.147) to advance the solution, step by step. Appropriate coding, or at least a first pass at such coding, might be

for \( j \) in range(2,T+1):
   \[ u_3[0] = 0 \]
   for \( i \) in range(1,N):
      \[ u_3[i] = \alpha \cdot u_2[i+1] + 2.0 \cdot (1.0 - \alpha) \cdot u_2[i] + \alpha \cdot u_2[i-1] - u_1[i] \]
      \[ u_3[N] = 2 \cdot \alpha \cdot u_2[N-2] + 2 \cdot (1.0 - \alpha) \cdot u_2[N-1] - u_1[N] \]
   u1 = u2.copy(); u2 = u3.copy()
plt.plot( x, u2, color='black', linewidth=3 )
plt.ylim( [-2.0,2.0] )
plt.show()

Here, each pass through the outermost loop advances the solution by one time step. Within that loop, we (1) construct the solution at the next time instant by exploiting Eqs. (15.145)–(15.147), (2) shift the values to prepare for the next pass through the loop, and (3) plot the solution. A more fully commented command file containing these statements—and a few others to be discussed in a moment—is named `fdmwave1d.py`, is listed in Appendix 15.D.1, and can be copied from the directory `$HEAD/python`\(^{39}\).

As a preliminary trial, we import the needed modules `numpy` and `matplotlib.pyplot` and run the coding developed to this point with a trial set of values, say \( N = 100, \ dt = 0.1, \ T = 20, \ c = 0.5, \) and \( L = 10 \) (for which \( \alpha = 0.25 \)). First, as expected, the execution halts at each `plt.show()` statement and will not proceed to the next step until we have manually closed the graphic window. Second, even with 20 time steps, the solution is not advanced very far.

We postpone addressing the first problem. We can, of course, fix the second problem simply by requesting a larger number of time steps or enlarging the time step (or both)—though we must be careful not to increase \( \alpha \) beyond 1.0). As the number of time steps requested increases—and, perhaps, the time interval between steps decreases—the program as we now have it, which displays the solution at every step of the way, will produce increasing volumes of output. With the more

\(^{38}\)Remember that the simple statement, for example, `u1=u2` will not create an independent copy of the first array in the second.

\(^{39}\)At some sites, the file may also be located in the PYTHON directory structure such that PYTHON can find it when it is identified only by its name without a prepended path.
accurate solutions (numerous segments, short time interval between generated solutions, many time steps to be computed), we would be wise to introduce a mechanism for suppressing the display of many of the generated solutions. To do so, we add at the beginning the statement

\[ f = \text{input}('\text{Plot frequency (f):}') \]; \ f = \text{int}(f) \]

to request the number of solutions whose display should be suppressed after a particular solution has been displayed and, in addition, we modify the `plt.plot` statement in the final loops to make the display conditional upon \( j \)—the loop index—being a multiple of \( f \). In effect, we simply condition all `plt.plot` statements except the first (which displays the initial condition) in a way that allows each step to be calculated but only every \( f \)-th step to be displayed. The second `plt.plot` statement will be executed only if \( f = 1 \), i.e., all steps are to be displayed. All of the remaining `plt.plot` statements will be recast in the form\(^{40}\)

\[
\text{if } f*\text{int}(j/f) == j: \\
\quad \text{plt.plot( } x, u2, \text{ color='black', linewidth=3, )} \\
\quad \text{plt.ylim( [-2.0,2.0] )} \\
\quad \text{plt.show()} 
\]

Finally, we add statements like

\[
\text{plt.title( 't = 0.0 s' ) } \quad \# \text{ Label first plot} \\
\text{plt.title( 't = '+str(dt)+' s' ) } \quad \# \text{ Label second plot} \\
\text{plt.title( 't = '+str(j*dt)+' s' ) } \quad \# \text{ Label remaining plots} 
\]

so that each plot produced is titled by the corresponding value of the time. (See the full listing in Appendix 15.D.1 for the details.)

With these embellishments, we might run the program and—with a bit of trial and error—submit the controlling values

\[
\text{execfile('fdmwave1d.py') or exec( open('fdmwave1d.py').read() )} \\
\text{Enter number of segments (N): 100} \\
\text{Enter time step (dt): 0.1} \\
\text{Enter number of time steps (T): 800} \\
\text{Enter speed of propagation (c): 0.5} \\
\text{Enter length of string (L): 10} \\
\text{Plot frequency (f): 50} \\
\text{0.25} \quad \# \text{ Reported value of alpha} 
\]

By closing each plot window after it appears and examining the output, we conclude that these parameters advance the solution through what we might anticipate to be one full period of the motion. The process evidently then repeats, since this first cycle has returned the string (essentially—recognize we are generating an approximate solution) to its initial configuration.

Several representative configurations of the string are shown in Fig. 15.14. The pulse divides into two equal halves, one propagating to the left and the other to the right. Each reflects at the end, though with a phase change at the fixed (left) end and without a phase change at the free (right) end. The pulses pass through one another, create at \( t = 20.0 \text{ s} \) an instant when—except for discretization error and/or computer roundoff error—the string is straight but its center portion has non-zero velocity, reflect from the ends again, and return to the middle, this time in phase. At \( t = 40.0 \text{ s} \), half the cycle has been completed and the original shape has been inverted. In another 40.0 s (i.e. at \( t = 80.0 \text{ s} \)), the string will return to its initial configuration. Note in particular the

\(^{40}\text{We exploit here the fact that the quantity } f*\text{int}(j/f) \text{ will be equal to } j \text{ only when } j \text{ is a multiple of } f.\)
amplitude of the motion of the free end of the string when the pulse reflects there. Because the reflected pulse is in phase with the incident pulse, the two add constructively and, when half the pulse has reflected, the displacement at the end is twice that of the incident pulse. At the fixed end, the reflected pulse is out of phase with the incident pulse, so their interference at all instants during the reflection is completely destructive at \( x = 0 \), and the end remains fixed.

A few small edits to the program `fdmwave1d.py` will create a version that will display an animation of the motion of this string. Following the pattern described in Section 5.14.5, we

- Replace each statement `plt.show()` with `plt.pause(delay)` to pause execution by a selected amount after each plot is produced.
- Add the statement

  ```python
delay = input( 'Delay between plots: ' ); delay = float(delay)
  ```

  at the end of the input block, thereby giving the user control over the duration of the delay to be inserted after each plot is displayed so the speed of the animation can be controlled.
- Add the statement `fig, ax = plt.subplots()` at the beginning of the section titled “Set and display initial displacement”.
- Replace each `plt.plot(...)` statement with the statement `ax.plot(...)`.  
- Add the statement `ax.clear()` before each invocation of `ax.plot`. 
- To assure that all titles will involve only one digit after the decimal point, add the statement `time=round(j*dt,1)` before the `plt.title` statement in the last loop and change the argument of the `str` function in that loop to `time`. 

• Add the statement `plt.show()` at the end of the entire program and at the top level of execution (i.e., not indented at all), executing that statement only if PYTHON 2 is in use.

• Add the statement `import sys` before executing the program to provide the module `sys.version`.

The full program is listed in Appendix 15.D.2.

Execution of this program, which we have named `fdmwave1d_anim.py` and stored in the default directory, with the statements:

```python
import numpy as np
import matplotlib.pyplot as plt
import sys
execfile('fdmwave1d_anim.py') or exec( open('fdmwave1d_anim.py').read() )
Enter number of segments (N): 100
Enter time step (dt): 0.1
Enter number of time steps (T): 800
Enter speed of propagation (c): 0.5
Enter length of string (L): 10
Plot frequency (f): 4
Delay between plots: 0.1
```

yields an on-screen display of the successive frames in the motion for one complete cycle of that motion. When the final plot window is closed, the command window returns to command level.

Accuracy is difficult to assess in this situation. Given the speed of propagation to be 0.5 m/s and the length of the string to be 10 m, we know that each (half) pulse should take 10 s (5 m/0.5 m/s) to travel from its initial position to the end of the string, 20 s to return to the middle of the string, etc. We also know that the two pulses, now of opposite polarity, should superpose at \( t = 20 \) s and exactly cancel each other, should superpose at \( t = 40 \) s to exactly invert the initial pulse, ... The solutions shown in Fig. 15.14 at these times reflect these expectations reasonably well. To check one of these expectations out more fully, let us run `fdmwave1d_anim` to obtain values for the solution at \( t = 20 \) s for three different accuracies. To recreate the frame at \( t = 20 \) s in Fig. 15.14, we would execute the statements and provide the input:

```python
execfile('fdmwave1d_anim.py') or exec( open('fdmwave1d_anim.py').read() )
Enter number of segments (N): 100
Enter time step (dt): 0.1
Enter number of time steps (T): 200
Enter speed of propagation (c): 0.5
Enter length of string (L): 10
Plot frequency (f): 4
Delay between plots: 0.1
```

Here, after the simulation has run to 20.0 s, we save the final values of \( u_2 \) and \( x \) for later plotting and display the range of values assumed by the solution at \( t = 20 \) s. Then, with the input,
15.20. USING PYTHON TO SOLVE 2D PDES VIA AN FDM

execfile('fdmwave1d_anim.py') or exec(open('fdmwave1d_anim.py').read() )

Enter number of segments (N): 200
Enter time step (dt): 0.05
Enter number of time steps (T): 400
Enter speed of propagation (c): 0.5
Enter length of string (L): 10
Plot frequency (f): 8
Delay between plots: 0.1

u2_200 = u2.copy(); x_200 = x.copy()
[ np.max(u2_200), np.min(u2_200) ]
[0.04371336097361708, -0.04369697574769488]

and again with the input

execfile('fdmwave1d_anim.py') or exec(open('fdmwave1d_anim.py').read() )

Enter number of segments (N): 400
Enter time step (dt): 0.025
Enter number of time steps (T): 800
Enter speed of propagation (c): 0.5
Enter length of string (L): 10
Plot frequency (f): 16
Delay between plots: 0.1

u2_400 = u2.copy(); x_400 = x.copy()
[ np.max(u2_400), np.min(u2_400) ]
[0.019038120058669727, -0.01903614784040629]

we generate and save the solution at \( t = 20 \) s, using progressively smaller divisions of the string (larger \( N \)), adjusting the other parameters to track the solution to \( t = 20 \) s, omitting numerous intermediate graphs, preserving the parameter \( \alpha \) at the value 0.25, and displaying the maximum and minimum values attained by the approximate solution (which we expect to be zero everywhere). Significantly, the departure from zero changes from \( \pm 0.12 \) to \( \pm 0.04 \) to \( \pm 0.02 \) as the number of segments—and presumably the accuracy of the solution—increases. For graphical comparison (and choosing a vertical scale to reveal the differences most clearly), we generate Fig. 15.15 with the statements

```python
plt.subplots_adjust(hspace=0.4)
plt.subplot(3,1,1)
plt.plot( x_100, u2_100, linewidth= 3, color='black' )
plt.title( 'N = 100' )
plt.ylim( (-0.2,0.2) ); plt.grid( color='black' )

plt.subplot(3,1,2)
plt.plot( x_200, u2_200, linewidth= 3, color='black' )
plt.title( 'N = 200' )
plt.ylim( (-0.2,0.2) ); plt.grid( color='black' )

plt.subplot(3,1,3)
plt.plot( x_400, u2_400, linewidth= 3, color='black' )
plt.title( 'N = 400' )
plt.ylim( (-0.2,0.2) ); plt.grid( color='black' )
```

Increasing \( N \) clearly improves the solution, i.e., makes it more nearly zero everywhere. Given these results, we might find it wise to go back and regenerate Fig. 15.14 with \( N = 400 \).
Figure 15.15: Shape of the string at $t = 20$ s for different segment sizes. This graph was produced with PYTHON.

15.20.2 The Diffusion Equation

The final step in addressing the example laid out in Section 15.16.2 is to set up and solve Eqs. (15.162)–(15.164). Using a command file in PYTHON, we would begin by requesting input of all necessary parameters and assuring that each is stored with the proper data type. The statements

\begin{verbatim}
N = input( 'Enter number of segments (N): ' ); N = int(N)
dt = input( 'Enter time step (dt): ' ); dt= float(dt)
T = input( 'Enter number of time steps (T): ' ); T = int(T);
alpha = input( 'Enter value of alpha (alpha): ' ); alpha = float(alpha)
L = input( 'Enter length of rod (L): ' ); L = float(L)
\end{verbatim}

accomplish those objectives. Then, we would determine the length of each segment, establish values for the coordinates at which solutions will be generated, and evaluate and display the one parameter $\gamma$ with the statements\footnote{The modules \textit{numpy} and \textit{matplotlib.pyplot} will be imported and named \texttt{np} and \texttt{plt} before the command file under construction is executed.}

\begin{verbatim}
dx = L/N; x = np.linspace(0.0, L, num=N+1)
gamma = alpha**2*dt/dx**2
print( ' gamma = '+str(gamma) )
\end{verbatim}

Prudence dictates the wisdom of adding the statement
if gamma > 0.5:
    print( 'Error: gamma > 0.5; execution halted' )
    exit()

to display the value of \( \gamma \) and halt execution if the parameter \( \gamma \) has a value that guarantees an unstable—and hence inaccurate—solution.

As we saw in the general discussion in Section 15.16.2, generation of the solution at the next time instant requires knowledge only of the solution at the current time instant. At each step of the way, we need preserve only the current solution, but we must keep that solution until the solution at the next time instant has been generated. In essence, we need two vectors of dimension \( N + 1 \) for storing solutions. We declare, therefore, that at any step in the process, the vector \( u_1 \) will store the current solution, and the vector \( u_2 \) will receive the solution at the next time instant as it is generated. Then, once the new solution has been generated and displayed (graphed or written to a file), we no longer need the values in \( u_1 \), so we will move the solution in \( u_2 \) to \( u_1 \), thus setting the stage for the next pass through a loop that advances the solution from time instant to time instant. These two variables are prepared with the statements

\[
\begin{align*}
    u_1 &= \text{np.zeros}( N+1 ) \quad \# \text{For current solution} \\
    u_2 &= \text{np.zeros}( N+1 ) \quad \# \text{For next solution}
\end{align*}
\]

Next, before coding the loop that will generate the solution, step by step, we must initialize the values in \( u_1 \) to reflect the initial temperature distribution in the rod and then display that distribution. To be specific, let us suppose that, initially, the temperature varies like one hump of a sine wave, but only in the middle quarter of its length, and is zero over the first three-eighths and the last three eighths of its length. Thus, we suppose that

\[
f(x) = \begin{cases} 
0 & 0 \leq x \leq \frac{3}{8}L \\
1 + \cos \frac{8\pi}{L} \left( x - \frac{L}{2} \right) & \frac{3}{8}L \leq x \leq \frac{5}{8}L \\
0 & \frac{5}{8}L \leq x \leq L 
\end{cases}
\]

which results in a smooth transition from zero temperature outside the center one quarter of the string to the sinusoidal displacement in that interval. The coding that will impose these initial conditions and then display the initial solution is

\[
b = 8.0*\text{np.pi}/L \\
\text{for i in range( int(3.0*N/8.0), int(5.0*N/8.0) ) :} \\
    u1[i] = 1.0 + \text{np.cos( b*(x[i]-L/2.0) )} \\
\text{plt.plot( x, u1, linewidth=3, color='black' )} \\
\text{plt.ylim( [0.0,2.0] )} \\
\text{plt.show()} 
\]

Here, we have recognized that the temperature will always remain between 0 and 2, and we have set the vertical scale to display that range for each anticipated graph.

Now, taking the temperature at \( x = 0 \) to be zero \( (T_0 = 0) \), we are ready to code the algorithm that uses Eqs. (15.162)–(15.164) to advance the solution, step by step. Appropriate coding, or at least a first pass at such coding, might be

\[\text{The calculation of the range of } i \text{ to be used is complicated. Because of integer arithmetic, the quantities } 3\times N/8 \text{ and } 5\times N/8 \text{ will be truncated, but we really want the lower limit to be raised rather than truncated. The upper limit can be truncated. Thus, we add 1 to the lower limit.}\]
for j in range(1,T+1):
    u2[1] = 0.0
    for i in range(1,N):
        u2[i] = gamma*u1[i-1] + (1.0-2.0*gamma)*u1[i] + gamma*u1[i+1]
    u2[N] = 2*gamma*u1[N-1] + (1.0-2.0*gamma)*u1[N]
    u1 = u2.copy()
    plt.plot( x, u1, linewidth=3, color='black' )
    plt.ylim( [0.0,2.0] )
    plt.show()

Here, each pass through the outermost loop advances the solution by one time step. Within that loop, we (1) construct the solution at the next time instant by exploiting Eqs. (15.162)–(15.164), (2) shift the values to prepare for the next pass through the loop, and (3) plot the solution, and (4) display it. A more fully commented command file containing these statements—and a few others to be discussed in a moment—is named fdmdiffus1d.py, is listed in Appendix 15.D.3, and can be copied from the directory \$HEAD/python.

As a preliminary trial, we import the needed modules numpy and matplotlib.pyplot and run the coding developed to this point with a trial set of values, say $N = 100$, $dt = 0.1$, $T = 25$, $\alpha = 0.1$, and $L = 10$ (for which $\gamma = 0.1$). First, as expected, the execution halts at each plt.show() statement and will not proceed to the next step until we have manually closed the graphic window. Second, even with 25 time steps, the solution is not advanced very far.

We postpone addressing the first problem. We can, of course, fix the second problem simply by requesting a larger number of time steps or by enlarging the time step (or both—though we must be careful not to increase $\gamma$ beyond 0.5). As the number of time steps requested increases, however, the program as we now have it will display the solution at every step of the way. With the more accurate solutions (numerous segments, short time interval between generated solutions, many time steps to be computed), we would be wise to introduce a mechanism for suppressing the display of many of the generated solutions. To do so, we add at the beginning the statement

$$f = \text{input( 'Plot frequency (f): ' )}; f = \text{int}(f)$$

to request the number of solutions whose display should be suppressed after a particular plot has been produced and, in addition, we modify the plt statements in the final loop to make the display conditional upon $j$—the loop index—being a multiple of $f$. In effect, we simply condition the plt.* statements in the final loop in a way that each step is calculated but only every $f$-th step is displayed. That group of plt.* statements is therefore recast in the form

$$\text{if } f*\text{int}(j/f) == j:$$

$$\text{plt.plot( x, u1, linewidth=3, color='black' )}$$

$$\text{plt.ylim( [0.0,2.0] )}$$

$$\text{plt.show()}$$

Finally, we add the statements

$$\text{plt.title( 't = 0.0 s' )}$$ # Label first plot
$$\text{plt.title( 't = '+'str(j*dt)+' s' )}$$ # Label remaining plots

so that each plot produced is titled by the corresponding value of the time. (See the full listing in Appendix 15.D.3 for the details.)

With these embellishments (and after importation of needed modules) we might run the program and—with a bit of trial and error—submit the starting values

44 At some sites, the file may also be located in the PYTHON directory structure such that PYTHON can find it when it is identified only by its name without a prepended path.
45 We exploit here the fact that the quantity $f*\text{int}(f/j)$ will be equal to $j$ only when $j$ is a multiple of $f$. 
Figure 15.16: Temperatures in a rod at various times as it approaches equilibrium. Note that the vertical scale after the graph for $t = 0.0$ s has been expanded the better to show the gradual evolution of the temperature distribution. This graph was produced with PYTHON.

execfile('fdmdiffus1d.py') or exec(open('fdmdiffus1d.py').read())

Enter number of segments (N): 100
Enter time step (dt): 0.4
Enter number of time steps (T): 4400
Enter value of alpha (alpha): 0.1
Enter length of rod (L): 10
Plot frequency (f): 200

gamma = 0.4

to trace the evolution of the initial temperature distribution for a longer time.

Several representative graphs of the temperature distribution versus position are shown in Fig. 15.16. Initially, the thermal energy in the central peak diffuses to either end of the rod. Energy diffuses out of the rod altogether at $x = 0$, because the “refrigerator” that maintains that end at $u = 0$ °C functions as a sink for thermal energy. Any thermal energy that reaches the insulated end ($x = 10$), however, does not escape the rod. In time, the insulated end becomes the hottest part of the rod. Then, thermal energy diffuses from the insulated end back towards the refrigerated end. Ultimately, all of the initial thermal energy in the rod has “leaked” into the refrigerator and the rod has attained equilibrium. At that point, the entire rod has the temperature of the refrigerator. Evidently, the insulated end is hottest at about $t = 1200$ s. Thereafter, the temperature at all points in the rod slowly falls until all points reach the temperature of the refrigerator. We might well have been able to predict this end result, even without solving the problem explicitly.
A few small edits to the program `fdmdiffus1d.py` will create a version that will display an animation of the motion of this string. Following the pattern described in Section 5.14.5, we

- Replace each statement `plt.show()` with `plt.pause(delay)` to pause execution by a selected amount after each plot is produced.
- Add the statement
  
  ```python
delay = input('Delay between plots: '); delay = float(delay)
  ```
  
at the end of the input block, thereby giving the user control over the duration of the delay to be inserted after each plot is displayed so the speed of the animation can be controlled.
- Add the statement `fig, ax = plt.subplots()` at the beginning of the section titled “Display initial temperature distribution”.
- Replace each `plt.plot(...)` statement with the statement `ax.plot(...)`.
- Add the statement `ax.clear()` before each invocation of `ax.plot`.
- To assure that all titles will involve only one digit after the decimal point, add the statement `time=round(j*dt,1)` before the `plt.title` statement in the last loop and change the argument of the `str` function in that loop to `time`.
- Add the statement `plt.show()` at the end of the entire program and at the top level of execution (i.e., not indented at all), executing that statement only if PYTHON 2 is in use.
- Add the statement `plt.grid(color='black')` before each `plt.title(...)` statement.
- Add the statement `import sys` before executing the program to provide the module `sys.version`.

The full program is listed in Appendix 15.D.4.

Execution of this program, which we have named `fdmdiffus1d_anim.py` and stored in the default directory, with the statements

```python
import numpy as np
import matplotlib.pyplot as plt
import sys
execfile('fdmdiffus1d_anim.py')
or exec( open('fdmdiffus1d_anim.py').read() )
```

yields an on-screen display of the successive frames in the motion until the temperature starts falling throughout the rod. When the final plot window is closed, the command window returns to command level.

To assess the accuracy of this solution, let us generate and compare solutions at \( t = 160 \) s for \( N = 100, 200, \) and \( 400 \). To recreate the frame at \( t = 160 \) s in Fig. ??, we would execute the statements and provide the input

```python
execfile('fdmdiffus1d_anim.py')
or exec( open('fdmdiffus1d_anim.py').read() )
```

```
Enter number of segments (N): 100
Enter time step (dt): 0.4
Enter number of time steps (T): 4400
Enter value of alpha (alpha): 0.1
Enter length of rod (L): 10
Plot frequency (f): 20
Delay between plots: 0.1
```

\[ \gamma = 0.4 \]

\[ ^{46}\text{It may be necessary to restart PYTHON and import required modules before executing this program for the first time. Closing the figure window after each execution is also important.} \]
Enter number of time steps (T): 400
Enter value of alpha (alpha): 0.1
Enter length of rod (L): 10
Plot frequency (f): 25
Delay between plots: 0.1

gamma = 0.4
u1_100 = u1.copy(); x_100 = x.copy()

Here, at the last PYTHON prompt, we save the final values for later plotting. Then, with the input,

execfile('fdmdiffus1d_anim.py') or exec(open('fdmdiffus1d_anim.py').read())

Enter number of segments (N): 200
Enter time step (dt): 0.1
Enter number of time steps (T): 1600
Enter value of alpha (alpha): 0.1
Enter length of rod (L): 10
Plot frequency (f): 100
Delay between plots: 0.1

gamma = 0.4
u1_200 = u1.copy(); x_200 = x.copy()

and again with the input

execfile('fdmdiffus1d_anim.py') or exec(open('fdmdiffus1d_anim.py').read())

Enter number of segments (N): 400
Enter time step (dt): 0.025
Enter number of time steps (T): 6400
Enter value of alpha (alpha): 0.1
Enter length of rod (L): 10
Plot frequency (f): 400
Delay between plots: 0.1

gamma = 0.4
u1_400 = u1.copy(); x_400 = x.copy()

we generate and save the solution at $t = 160$ s, using progressively smaller divisions of the rod (larger $N$), adjusting the other parameters to track the solution precisely to $t = 160$ s, preserving the parameter $\gamma$ at the value 0.4.\textsuperscript{47} For graphical comparison (and choosing a vertical scale to reveal the differences most clearly), we generate Fig. ?? with the statements

plt.subplots_adjust(hspace=0.5)
plt.subplot(3,1,1)
plt.plot( x_100, u1_100, linewidth= 3, color='black' )
plt.title( 'N = 100' )
plt.ylim( (0.0,0.6) ); plt.grid( color='black' )

plt.subplot(3,1,2)
plt.plot( x_200, u1_200, linewidth= 3, color='black' )
plt.title( 'N = 200' )
plt.ylim( (0.0,0.6) ); plt.grid( color='black' )

plt.subplot(3,1,3)

\textsuperscript{47}Note that, since $\gamma = \alpha^2 dt/dx^2$, doubling $N$ which halves $dx$ and reduces $dx^2$ to one quarter of its original value requires that $dt$ be reduced by a factor of four as well if $\gamma$ is to retain its original value.
Figure 15.17: Temperature distribution at $t = 160$ s for different segment sizes. This graph was produced with PYTHON.

The three frames in Fig. 15.17 appear to be identical. Indeed, were the solutions plotted on the same axes, no difference—to the resolution of the graph—would be noticed anywhere along the rod. Evidently, at $t = 160$ s at least, the original solution with $N = 100$ is accurate to the resolution of the graph.

To assess the differences at $t = 160$ s more carefully, we might invoke PYTHON to calculate the difference, say, between the solution at that time for $N = 100$ and the solution for $N = 200$. To do so, we would invoke the statements

```python
import numpy as np

N = 100
x = np.linspace(0, 10, N+1)
y = np.sin(x)**2

plt.plot( x, y, linewidth=3, color='black' )
plt.title( 'N = 100' )
plt.ylim( (0.0,0.6) ); plt.grid( color='black' )
```

The two solutions are extremely close throughout the rod. We conclude that the solution obtained for $N = 100$ is probably accurate to an absolute error of about $\pm 2 \times 10^{-4}$.

---

48 Remember that the solution for $N = 200$ has twice as many values at the solution for $N = 100$. 
15.20.3 The Laplace Equation

The final step in addressing the example laid out in Section 15.16.3 is to set up and solve Eq. (15.174). Using a command file in PYTHON, we would begin by requesting input of the two necessary parameters and assuring that each is stored with the proper data type. The statements

\[
\begin{align*}
N &= \text{input( 'Enter number of segments (N): ' )}; \; N = \text{int}(N) \\
L &= \text{input( 'Enter length of side(L): ' )}; \; L = \text{float}(L) \\
\text{maxits} &= \text{input( 'Maximum number of iterations (maxits): ' )} \\
\text{maxits} &= \text{int}(\text{maxits})
\end{align*}
\]

accomplish those objectives. Then, we would determine the length of each segment and establish values for the coordinates at which a solution will be generated with the statements\(^{49}\)

\[
dx = L/N; \; x = \text{np.linspace}(0.0, L, N+1); \; y = x.\text{copy()}
\]

(Remember that we are solving the Laplace equation in a square and we have elected to use the same segment size for both coordinates.)

Next, we establish a suitably dimensioned array \(u\) to contain the solution as it is generated, set the Dirichlet boundary values on the left, bottom, and top edges, and display the initial values in the array \(u\) with the statements

\[
u = \text{np.zeros( (N+1, N+1) )} \\
u[0, :] = 100.0 \\
u[:, 0] = 100.0 - 100.0*x/L \\
\text{print}(\ u \\
\]

In constructing these statements, we have arranged the values in the array \(u\) so that, in the output generated by the statement \text{print}(u), the left, right, top, and bottom edges correspond to the left, right, top, and bottom edges in Fig. 15.13. Thus, for example, execution of the statements introduced to this point with \(N\) set to 5 and \(L\) set to 10 yields the output

\[
\begin{bmatrix}
100. & 100. & 100. & 100. & 100. \\
80. & 0. & 0. & 0. & 0. \\
60. & 0. & 0. & 0. & 0. \\
40. & 0. & 0. & 0. & 0. \\
20. & 0. & 0. & 0. & 0. \\
0. & 0. & 0. & 0. & 0. \\
\end{bmatrix}
\]

The values in the top and bottom rows and in the left column of this array correctly reflect the Dirichlet conditions to be imposed on the solution. The values in the right column are unknown initially, since we have imposed Neumann (derivative) conditions along that edge. Those values must be determined as the solution unfolds.

Our task from this point is to determine values to assign to the entries not on the top, bottom, or left edges in such a way that Eq. (15.174) is satisfied. This task differs substantially from the task we confronted in solving the wave and diffusion equations in Sections 15.20.1 and 15.20.2. Those previous equations involved a mixture of boundary and initial conditions and, once the process was started, we could step forward in time as far as we pleased. With the Laplace equation, we are dealing with a boundary value problem alone, and conditions are imposed around the entire perimeter of the region of interest. Rather than having boundary conditions on the left and right

\(^{49}\)The module \texttt{numpy} will be imported and named \texttt{np} before the command file under construction is executed.
of the region of interest, initial conditions on the bottom, and no definite boundary on the top, we have instead boundary conditions on all four sides of the region of interest. One strategy for the present problem would be to guess the derivative $\partial u/\partial x$ at the left edge, treat the problem as an initial value problem in $x$ by stepping systematically across the above array to the right edge, and hope that we arrived at that edge with values satisfying the boundary condition at that edge. If we missed, we would make another guess for the starting derivative and try again. We conclude that, in contrast to the problems treated in Sections 15.20.1 and 15.20.2, solution of this problem will require an iterative approach.

We have already written Eq. (15.174) to support an iterative approach more suitable than that suggested in the previous paragraph.\footnote{This iterative approach has already been described—though for slightly different boundary conditions—in Section 9.3.1 and implemented in PYTHON in Section 9.11.} This approach entails (1) guessing a solution at each node not constrained by a Dirichlet condition, (2) stepping systematically through those nodes while replacing the value at each node with the value determined by Eq. (15.174), and (3) repeating step (2) until some criterion of convergence has been met.\footnote{We elect to work in place in the array that contains our initial guess, so each newly generated value is used in subsequent calculations as soon as it becomes available. Alternatively, we could store the newly generated values in a second array, completing one pass through the array before using any of the newly generated values. As it turns out, the former approach converges rather more rapidly than the latter approach.} For the moment, let us simply carry out a user-specified number of iterations. Thus, we would (1) add to the input statements the coding

```python
maxits = input( 'Maximum number of iterations (maxits): ' )
maxits = int( maxits )
```

to obtain the desired number of iterations, (2) change the `print` statement above to read

```python
print( '\nIteration # 0' );
```

so as to label the output, space it away from what precedes, and round the output for a more legible display, and (3) add the multiple loop

```python
for itcnt in range(1, maxits+1):
    for i in range(1,N):
        for j in range(1,N):
            u[i,j] = 0.25*(u[i+1,j] + u[i-1,j] + u[i,j+1] + u[i,j-1])
    for i in range(1,N):
        u[i,N] = 0.25*(2.0*u[i,N-1] + u[i+1,N] + u[i-1,N])
    print( '\nIteration # '+str(itcnt) );
```

to effect the solution and display it. Here, each pass through the outermost loop will effect one iteration in the algorithm described above. A total of `maxits` iterations will be executed before the loop terminates. Within that outermost loop, the double loop on `i` and `j` steps through the “interior” points in the grid, replacing the value at each with the average of the values at its four nearest neighbors. Then, the single loop on `i` steps through the nodes along the right edge (except for the two corner nodes), replacing the value at each with the value dictated by the appropriate member of Eq. (15.174). Finally, the `print` statement displays the solution after the current iteration is completed. A more fully commented command file containing these statements—and a few others to be discussed in a moment—is named `fdmlap2d.py`, is listed in Appendix 15.D.5, and can be copied from the directory `$HEAD/python`.

As a quick test of our as yet incomplete command file and as a way to illustrate the manner of approach to a final solution, let us execute the program as it stands with the input

```python
50
```

execfile('fdmlap2d.py') or exec(open('fdmlap2d.py').read())
Enter number of segments (N): 5
Enter length of side(L): 10
Maximum number of iterations: 5

obtaining the output

Iteration # 0
[[100. 100. 100. 100. 100. 100.]
 [ 80.  0.  0.  0.  0.  0.]
 [ 60.  0.  0.  0.  0.  0.]
 [ 40.  0.  0.  0.  0.  0.]
 [ 20.  0.  0.  0.  0.  0.]
 [  0.  0.  0.  0.  0.  0.]]

Iteration # 1
[[100. 100. 100. 100. 100. 100.]
 [ 80.  45.  36.25  34.062  33.516  41.758]
 [ 60.  26.25  15.625  12.422  11.484  16.182]
 [ 40.  16.562  8.047  5.117  4.15   6.121]
 [ 20.  9.141  4.297  2.354  1.626  2.343]
 [  0.  0.  0.  0.  0.  0.  ]]

Iteration # 2
[[100. 100. 100. 100. 100. 100.]
 [ 80.  60.625  52.578  49.629  50.718  54.404]
 [ 60.  38.203  27.812  23.511  23.64   26.951]
 [ 40.  23.848  15.269  12.422  11.484  12.662]
 [ 20.  12.036  7.415  5.117  4.15   5.429]
 [  0.  0.  0.  0.  0.  0.  ]]

Iteration # 3
[[100. 100. 100. 100. 100. 100.]
 [ 80.  67.695  61.284  58.878  59.231  61.353]
 [ 60.  44.839  36.226  32.516  32.344  34.676]
 [ 40.  28.036  20.749  17.258  16.698  18.375]
 [  0.  0.  0.  0.  0.  0.  ]]}

Iteration # 4
[[100. 100. 100. 100. 100. 100.]
 [ 80.  71.531  66.659  64.601  64.575  65.956]
 [ 60.  48.948  42.218  39.105  38.763  40.465]
 [ 40.  30.89  25.073  22.201  21.713  23.06  ]
 [ 20.  15.204  12.051  10.441  10.126  10.828]
 [  0.  0.  0.  0.  0.  0.  ]]

Iteration # 5
[[100. 100. 100. 100. 100. 100.]
 [ 80.  73.902  70.18   68.465  68.296  69.264]
 [ 60.  51.752  46.528  43.989  43.616  44.889]
 [ 20.  16.265  13.788  12.516  12.27   12.835]
 [  0.  0.  0.  0.  0.  0.  ]]

15.20. USING PYTHON TO SOLVE 2D PDES VIA AN FDM
The progression in the solution from iterate to iterate is quite clear, though it is also clear from
the changes taking place from one iterate to the next that we haven’t gone far enough to achieve
convergence.

While the coding developed to this point certainly displays the essence of the algorithm for
solving the Laplace equation, the above output also makes clear that we must carry the solution
through a larger number of iterates to achieve convergence. That change, however, will result in
much more output unless we suppress all but every f-th iteration (1) by adding the coding

\[
f = \text{input( 'Display frequency (f):' ); } \ f = \text{int}(f)
\]

to request the number of iterates whose display is to be suppressed and (2) by modifying the print
statement in the final loops to make the display conditional upon itcnt being a multiple of f. In
effect, we simply replace all but the first print statement everywhere it appears with the conditional
statement

\[
\text{if } f \times \text{int(itcnt/f)} == \text{itcnt}:
    \text{print( 'Iteration # '+str(itcnt) ); print(np.around(u,3))}
\]

(See the full listing of this program, which we name fdmlap2d.py, in Appendix 15.D.5 for the
details.)

With this embellishment, we might test our program by submitting the starting values

execfile('fdmlap2d.py') or exec( open('fdmlap2d.py').read() )
Number of segments: 5
Length of side: 10
Maximum number of iterations: 80
Display frequency: 20

obtaining the output

Iteration # 0
[[100. 100. 100. 100. 100. 100.]
 [ 80.  0.  0.  0.  0.  0. ]
 [ 60.  0.  0.  0.  0.  0. ]
 [ 40.  0.  0.  0.  0.  0. ]
 [ 20.  0.  0.  0.  0.  0. ]
 [ 0.  0.  0.  0.  0.  0.]]

Iteration # 20
[[100. 100. 100. 100. 100. 100. ]
 [ 80. 79.882 79.802 79.76 79.752 79.77 ]
 [ 60. 59.831 59.718 59.658 59.646 59.673]
 [ 0.  0.  0.  0.  0.  0. ]]

Iteration # 40
[[100. 100. 100. 100. 100. 100. ]
 [ 80. 79.999 79.999 79.999 79.999 79.999]
 [ 60. 59.999 59.998 59.998 59.998 59.998]
 [ 40. 39.999 39.999 39.999 39.999 39.999]
In this output, convergence to a steady solution is more evident. Indeed, the solution at iteration 60 conforms to exactly what we might expect. That that result is reproduced in subsequent iterations is also reassuring.

While the coarse grid used for the above solution renders its display easy on the screen or on the page of a book and, in particular, allows us to reveal the nature of the convergence to the final steady solution, that coarse grid also leaves open the possibility that we have found an accurate solution to the discretized equations that is not, however, a particularly good approximation to the continuous solution to the original problem. We would prefer to use a more refined grid and to use graphical rather than tabular display of the solution. To achieve those objectives, we need simply specify a larger number of segments, a larger number of iterations, and a larger gap between displayed iterates. Anticipating making a mesh plot of $u$ over the $xy$-plane, we generate the arrays containing the $x$ and $y$ coordinates in the proper grid with the statement

\[
xx, yy = \text{np.meshgrid}(x, y)
\]

Then to generate the mesh plot, we replace the `print` statements with

\[
u1 = \text{np.flip}(u, \text{axis}=0)
ax1 = \text{plt.axes(projection='3d')}
ax1.plot_surface(xx, yy, u1, color='white', shade=False, edgecolor='black')
ax1.set_xlabel('x'); ax1.set_ylabel('y'); ax1.set_zlabel('u')
ax1.set_title( 'Iteration 0' ) or ax1.set_title( 'Iteration ' + str(itcnt) )
ax1.view_init(azim=-45, elev=25)
plt.show()
\]

A more fully commented command file containing these modifications is named `fdmlap2d_plot.py`, is listed in Appendix 15.D.6, and can be copied from the directory `$HEAD/python`.

As a preliminary trial, we import the needed modules `numpy`, `matplotlib.pyplot`, and `mpl_toolkits.mplot3d.axes3d` and run the coding developed to this point with the statements

---

52 In fact, the solution we have obtained is exactly correct, but the problem we have solved is also especially simple. Actually obtaining the exact solution with such a coarse grid is much more an accident of the particular problem than a behavior we can expect.

53 The invocation of `np.flip` with `axis=0` to invert the row index in $u$ turned out to be necessary to assure that low values of the row index correspond to low values of $y$. The invocation of `ax1.view_init` reorients the produced diagram to show the shape of the surface most clearly. The need for neither of these embellishments was recognized until figures were produced without them.
import numpy as np
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import axes3d
execfile( 'fdmlap2d_plot.py' ) or exec( open('fdmlap2d_plot.py').read() )

with a trial set of values, say

Enter number of segments (N): 20
Enter length of side(L): 10
Maximum number of iterations (maxits): 150
Display frequency (f): 50

As expected, the execution halts at each plt.show() statement and will not proceed to the next step until we have manually closed the graphic window.

The output of this program for several steps along the way from the initial guess to the final equilibrium is shown in Fig. 15.18. The upper left panel in this figure shows the initial guess, to the resolution specified by division of each edge into 20 segments. Subsequent panels are separated by 50 iterations. The gradual penetration of thermal energy from the heated edges maintained at fixed temperatures into the full space is clear as the iterations proceed. While 150 iterations are clearly not enough for the solution to relax completely, we nonetheless anticipate—correctly—that the ultimate equilibrium solution will appear in this display as a plane whose front edge is at 0 °C and whose back edge is at 100 °C.

We can create an animated display showing the gradual convergence from the initial conditions and initial guessed solution to the final solution by making only a few changes in the program fdmlap2d_plot.py. Following the pattern described in Section 5.14.5, we

- Replace each statement plt.show() with plt.pause(delay) to pause execution by a selected amount after each plot is produced.
- Add the statement

```
delay = input( 'Delay between plots: ' ); delay = float(delay)
```

at the end of the input block, thereby giving the user control over the duration of the delay to be inserted after each plot is displayed so the speed of the animation can be controlled.
- Add the statement fig, ax1 = plt.subplots() at the beginning of the section titled “Display solution on the screen”.
- Replace each plt.plot(...) statement with the statement ax1.plot(...).
- Add the statement plt.clf() before each invocation of ax1.plot.\(^{54}\)
- Add the statement plt.show() at the end of the entire program and at the top level of execution (i.e., not indented at all), executing that statement only if PYTHON 2 is in use.
- Add the statement import sys before executing the program to provide the module sys.version.

The full program is listed in Appendix 15.D.7.

Execution of this program, which we have named fdmlap2d_anim.py and stored in the default directory, with the statements\(^{55}\)

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

\(^{54}\)In contrast to the description in Section 5.14.5, plt.clf() seems to work better here than ax.clear(), especially in PYTHON 3.9.

\(^{55}\)It may be necessary to restart PYTHON and import required modules before executing this program for the first time. Closing the figure window after each Execution is also important.
Figure 15.18: Approach of temperature distribution to equilibrium. The panels show the initial distribution (upper left), the distribution at iterate 50 (upper right), the distribution at iterate 100 (lower left), and the distribution at iterate 150 (lower right). The relationship of these figures to the previously displayed arrays is easiest if you reflect each array in a vertical line between the third and fourth columns of the array. With that rearrangement, the array can then simply be laid down on the $xy$ plane in the figures. This graph was produced with PYTHON.

from mpl_toolkits.mplot3d import axes3d
import sys
execfile( 'fdmlap2d_anim.py' ) or exec(open('fdmlap2d_anim.py').read() )
Enter number of segments (N): 20
Enter length of rod (L): 10
Maximum number of iterations: 200
Plot frequency (f): 2
Delay between plots: 0.1

yields an on-screen display of the successive frames in the progress of the iteration from the initial condition to a final converged solution. When the final plot window is closed, the command window returns to command level.

In the above discussion, we have simply allowed some specified number of iterations to be carried out and applied qualitative reality checks to the resulting solutions. A common means to assess convergence of the iterative solution of the discretized equations involves comparing the values generated in each iterate with the ones from the previous iterate and accepting the most recent iterate as final once it differs from its predecessor by no more than a specified tolerance. One then supposes
that the iterative algorithm has converged to an accuracy—fractional or absolute depending on the precise coding—given by the tolerance. Modification of \texttt{fdm2dlap\_plot.m} to incorporate this embellishment is the subject of one of the end-of-chapter exercises.

Convergence of the iterative process to a particular tolerance, however, only provides evidence that the \textit{discretized} equations have been solved to a particular accuracy. That characteristic assures us only that \textit{convergence} error is adequately small. (Probably \textit{roundoff} error is also under control at that point.) By itself, \textit{convergence} of the iterative process offers only marginal assurance that \textit{discretization} error is adequately small. To assess discretization error, we would need to compare not successive iterates on a given grid but converged iterates on at least two grids, one of which is finer than the other. \textit{Multigrid algorithms} to achieve that objective are beyond the scope of this chapter.

### 15.23 Finite Element Methods (FEMs) in Two Dimensions

While their implementation is more involved in two dimensions than in one dimension, finite element approaches to problems in two dimensions involve the same several steps as those identified at the beginning of Section 15.9. In this section, we illustrate the method of finite element analysis by applying those steps to the two-dimensional boundary value problem defined in Section 15.1.8.

#### 15.23.1 Discretizing the Domain: Pre-processing

The first step is to divide the domain into elements or, in the jargon, to \textit{create a mesh} over the domain or to \textit{mesh} the domain. To facilitate the creation of that mesh (and the discussion of the method), we elect to use the simplest possible two-dimensional element, meshing the region of interest—denote it by \( \Omega \)—with triangular elements as illustrated, for example, in the left panel of Fig. 15.19. Each element is defined by three nodes (vertices), which also serve as nodes on adjacent elements, and each element has three \textit{edges}—the lines along which pairs of elements meet. Together, the edges define the perimeter or boundary of the element. For the meshing to be legal, no node in one element can be located along the edge of another element, as, for example, at the point circled in the right panel of Fig. 15.19. Further, since the error in the finite element solution varies inversely as the sine of the smallest angle in each element, all elements should be at least approximately equilateral. While increasing the number of elements in the region improves the accuracy of the solution, it also increases both the computation time and the memory needed to obtain the solution. Thus, the number of elements should be large (and their size small) only in regions in which the solution is expected to vary rapidly. A coarser mesh can be accepted in regions in which the solution varies slowly. Choosing an appropriate mesh of elements of suitable sizes is difficult and may require an iterative approach, especially if little is known \textit{a priori} about the solution.

As a modest example, suppose the region of interest is a rectangle. For simplicity, divide that region into eight triangular elements by one vertical line, one horizontal line, and three diagonal lines, as shown in Fig. 15.20. Then, we number the elements in an order that seems convenient. In addition, we number the nodes both globally (i.e., with respect to the entire domain) and locally (i.e., with respect to a particular element). The \textit{global} labeling is done by assigning a specific and unique integer to each node, as in the left panel of Fig. 15.21. \textit{Locally}, the nodes—remember that each element has three nodes—are numbered with the integers 0, 1, and 2, as in the right panel of Fig. 15.21.\footnote{In this example, we elect to begin all numbering of elements and nodes—both global and local—with zero, fully aware that some programming languages (IDL, PYTHON, C) follow that pattern but that others (MATLAB, OCTAVE, FORTRAN) start numbering with 1. In the latter case, each index in the program will be incremented by 1 relative to those in this theoretical discussion of the general strategies.} Each element will have its own zeroth, first, and second nodes, each of which also has a global label that identifies it uniquely. A connectivity matrix such as the one shown in Table 15.2 is created to express the relation between global and local node numbers. In the present context,
Figure 15.19: Discretization of a general region with triangular elements.

Figure 15.20: Subdivision of a rectangular domain into eight triangular elements.

Figure 15.21: Rectangular domain of Fig. 15.20 with global (left) and local (right) node numbers.
the matrix consists of \( M \) rows, one for each element, with each row containing 3—one for each node of the element—entries, i.e. the matrix has \( M \) rows and three columns. Every entry in the matrix is a global node number corresponding to the zeroth, first, or second node of each element. In this example, the (global) numbering both of the elements and of the nodes is done in the same way, from top to bottom and from left to right. The local node numbers are assigned counterclockwise starting at the lower left node of each element. For element 0, for example, the global number of local node 0 is 1, the global number of local node 1 is 3, and the global number of local node 2 is 0. Similarly, for element 1, we have global node numbers 1, 4, and 3. The rest of the entries are generated in the same way and are recorded in Table 15.2.

15.23.2 Selecting the Interpolation or Shape Functions

The simplest interpolation or shape functions are linear in both \( x \) and \( y \). For element \( e \), we write the approximating equation in the form

\[
\hat{\varphi}^{(e)}(x, y) = a^{(e)} + b^{(e)}x + c^{(e)}y
\]  

(15.178)

The constants are determined so that this equation gives the correct values at all three nodes of any element, i.e., so that

\[
\hat{\varphi}_0^{(e)} = a^{(e)} + b^{(e)}x_0^{(e)} + c^{(e)}y_0^{(e)}
\]

\[
\hat{\varphi}_1^{(e)} = a^{(e)} + b^{(e)}x_1^{(e)} + c^{(e)}y_1^{(e)}
\]

\[
\hat{\varphi}_2^{(e)} = a^{(e)} + b^{(e)}x_2^{(e)} + c^{(e)}y_2^{(e)}
\]

(15.179)

where subscripts refer to local node numbers. Solution of this system of equations for \( a^{(e)} \), \( b^{(e)} \), and \( c^{(e)} \) and substitution of the solution into Eq. (15.178) gives the approximating function\(^{57}\)

\[
\hat{\varphi}^{(e)}(x, y) = \sum_{j=0}^{2} N_j^{(e)}(x, y) \hat{\varphi}_j^{(e)}
\]

(15.180)

where, with \( \Delta^{(e)} \)—the determinant of the matrix of coefficients in Eq. (15.179)—defined by

\[
\Delta^{(e)} = \begin{vmatrix}
1 & x_0^{(e)} & y_0^{(e)} \\
1 & x_1^{(e)} & y_1^{(e)} \\
1 & x_2^{(e)} & y_2^{(e)}
\end{vmatrix}
\]

(15.181)

\(^{57}\)See Exercise 15.15.
the interpolation or shape functions $N_j^{(e)}(x, y)$ are given by

\[
N_0^{(e)}(x, y) = \frac{1}{\Delta^{(e)}} \begin{vmatrix} 1 & x & y \\ 1 & x_1^{(e)} & y_1^{(e)} \\ 1 & x_2^{(e)} & y_2^{(e)} \end{vmatrix}
\]

(15.182)

\[
N_1^{(e)}(x, y) = \frac{1}{\Delta^{(e)}} \begin{vmatrix} 1 & x^{(e)} & y^{(e)} \\ 1 & x_0^{(e)} & y_0^{(e)} \\ 1 & x_2^{(e)} & y_2^{(e)} \end{vmatrix}
\]

(15.183)

\[
N_2^{(e)}(x, y) = \frac{1}{\Delta^{(e)}} \begin{vmatrix} 1 & x^{(e)} & y^{(e)} \\ 1 & x_0^{(e)} & y_0^{(e)} \\ 1 & x_1^{(e)} & y_1^{(e)} \end{vmatrix}
\]

(15.184)

From these expressions and the properties of determinants, we can quickly recognize that each interpolation function has the value one at the node corresponding to its index and the value zero at the other two nodes. Thus, the sum on the right side of Eq. (15.180) reduces at each node to the value given on the left side. Equally true but less obvious, each interpolation function is in fact zero along the entire line joining the two nodes at which it has the value zero. This property will turn out to be useful when the boundary conditions are taken into account.

### 15.23.3 Formulating the Equations for a Single Element

From the general form of Eq. (15.70), the expression for the residual $r$ can be written easily as

\[
r = -\frac{\partial}{\partial x}\left(\alpha_x \frac{\partial \tilde{\psi}}{\partial x}\right) - \frac{\partial}{\partial y}\left(\alpha_y \frac{\partial \tilde{\psi}}{\partial y}\right) + \beta \tilde{\psi} - f
\]

(15.185)

Evaluated for element $e$, the weighted residuals then are given by the integral

\[
R_i^{(e)} = \int \int_{\Omega^{(e)}} N_i^{(e)} r \, dx \, dy \quad ; \quad i = 0, 1, 2
\]

(15.186)

over the surface $\Omega^{(e)}$ of element $e$. Here, following the Galerkin formulation, we take the weighting functions to be the interpolation functions themselves. Substitution of Eq. (15.185) into Eq. (15.186) leads to

\[
R_i^{(e)} = \int \int_{\Omega^{(e)}} N_i^{(e)} \left[-\frac{\partial}{\partial x}\left(\alpha_x \frac{\partial \tilde{\psi}}{\partial x}\right) - \frac{\partial}{\partial y}\left(\alpha_y \frac{\partial \tilde{\psi}}{\partial y}\right) + \beta \tilde{\psi} - f\right] \, dx \, dy
\]

(15.187)

\[
= -\int \int_{\Omega^{(e)}} N_i^{(e)} \left[\frac{\partial}{\partial x}\left(\alpha_x \frac{\partial \tilde{\psi}}{\partial x}\right) + \frac{\partial}{\partial y}\left(\alpha_y \frac{\partial \tilde{\psi}}{\partial y}\right)\right] \, dx \, dy
\]

\[
+ \int \int_{\Omega^{(e)}} N_i^{(e)} \beta \tilde{\psi} \, dx \, dy - \int \int_{\Omega^{(e)}} N_i^{(e)} f \, dx \, dy
\]

(15.188)

Then, interpreting $\psi$, $V_x$, and $V_y$ in Eq. (15.232), the identity developed in Appendix 15.E, as $N_i^{(e)}$, $\alpha_x \partial \tilde{\psi} / \partial x$, and $\alpha_y \partial \tilde{\psi} / \partial y$, respectively, we can rewrite the first integral in Eq. (15.188) so as to recast

---

58 See Exercise 15.15.
the entirety of Eq. (15.188) in the form

\[ R_i^{(e)} = \int \int_{\Omega^{(e)}} \left( \alpha_x \frac{\partial N_i^{(e)}}{\partial x} \frac{\partial \tilde{\varphi}^{(e)}}{\partial x} + \alpha_y \frac{\partial N_i^{(e)}}{\partial y} \frac{\partial \tilde{\varphi}^{(e)}}{\partial y} + \beta N_i^{(e)} \tilde{\varphi}^{(e)} \right) dx \, dy \\
- \int \int_{\Omega^{(e)}} N_i^{(e)} f \, dx \, dy - \oint_{\Gamma^{(e)}} N_i^{(e)} \left( \alpha_x \frac{\partial \tilde{\varphi}^{(e)}}{\partial x} \hat{i} + \alpha_y \frac{\partial \tilde{\varphi}^{(e)}}{\partial y} \hat{j} \right) \cdot \hat{n}^{(e)} \, dl \]  

(15.189)

Here, \( \Gamma^{(e)} \) is the path bounding the region \( \Omega^{(e)} \), and \( \hat{n}^{(e)} \) is a unit vector lying in the plane of element \( e \), perpendicular at each point to \( \Gamma^{(e)} \), and directed outward from the perspective of a viewer in \( \Omega^{(e)} \). Substitution of the approximating function of Eq. (15.180) yields for the weighted residuals the expression

\[ R_i^{(e)} = \sum_{j=0}^{2} \varphi_j^{(e)} \int \int_{\Omega^{(e)}} \left( \alpha_x \frac{\partial N_i^{(e)}}{\partial x} \frac{\partial N_j^{(e)}}{\partial x} + \alpha_y \frac{\partial N_i^{(e)}}{\partial y} \frac{\partial N_j^{(e)}}{\partial y} + \beta N_i^{(e)} N_j^{(e)} \right) dx \, dy \\
- \int \int_{\Omega^{(e)}} N_i^{(e)} f \, dx \, dy - \oint_{\Gamma^{(e)}} N_i^{(e)} \left( \alpha_x \frac{\partial \tilde{\varphi}^{(e)}}{\partial x} \hat{i} + \alpha_y \frac{\partial \tilde{\varphi}^{(e)}}{\partial y} \hat{j} \right) \cdot \hat{n}^{(e)} \, dl \]  

(15.190)

or, in the matrix form,

\[ \{R^{(e)}\} = [K^{(e)}] \{\tilde{\varphi}^{(e)}\} - \{b^{(e)}\} - \{g^{(e)}\} \]  

(15.191)

where \([\ldots]\) denotes a matrix and \{\ldots\} denotes a vector. The matrix \([K^{(e)}]\) is a 3 \times 3 matrix and \(\{R^{(e)}\}, \{\tilde{\varphi}^{(e)}\}, \{b^{(e)}\}, \) and \(\{g^{(e)}\}\) are three-element vectors. The matrix elements \(K_{ij}^{(e)}\) and the vector elements \(b_i^{(e)}\) and \(g_i^{(e)}\) can be obtained from Eq. (15.190), namely

\[ K_{ij}^{(e)} = \int \int_{\Omega^{(e)}} \left( \alpha_x \frac{\partial N_i^{(e)}}{\partial x} \frac{\partial N_j^{(e)}}{\partial x} + \alpha_y \frac{\partial N_i^{(e)}}{\partial y} \frac{\partial N_j^{(e)}}{\partial y} + \beta N_i^{(e)} N_j^{(e)} \right) dx \, dy \]  

(15.192)

\[ b_i^{(e)} = \int \int_{\Omega^{(e)}} N_i^{(e)} f \, dx \, dy \]  

(15.193)

\[ g_i^{(e)} = \oint_{\Gamma^{(e)}} N_i^{(e)} \left( \alpha_x \frac{\partial \tilde{\varphi}^{(e)}}{\partial x} \hat{i} + \alpha_y \frac{\partial \tilde{\varphi}^{(e)}}{\partial y} \hat{j} \right) \cdot \hat{n}^{(e)} \, dl \]  

(15.194)

where, in all cases, \(i\) and \(j\) assume the values 0, 1, and 2.

All elements of the matrix \([K^{(e)}]\) and of the vector \(\{b^{(e)}\}\) are known at the start of the problem. The elements of the vector \(\{g^{(e)}\}\) are ultimately determined when the boundary conditions are incorporated. Those terms will be discussed further in Section 15.23.5.

Of course, the equations determining the unknown nodal values are obtained by requiring the weighted residuals to be zero. In matrix form, we have from Eq. (15.191) for the \(e\)-th element that

\[ [K^{(e)}] \{\tilde{\varphi}^{(e)}\} = \{b^{(e)}\} + \{g^{(e)}\} \]  

(15.195)

Written out in terms of the specific (local) nodes, this equation becomes

\[
\begin{bmatrix}
K_{00}^{(e)} & K_{01}^{(e)} & K_{02}^{(e)} \\
K_{10}^{(e)} & K_{11}^{(e)} & K_{12}^{(e)} \\
K_{20}^{(e)} & K_{21}^{(e)} & K_{22}^{(e)}
\end{bmatrix}
\begin{bmatrix}
\tilde{\varphi}_0^{(e)} \\
\tilde{\varphi}_1^{(e)} \\
\tilde{\varphi}_2^{(e)}
\end{bmatrix}
=
\begin{bmatrix}
b_0^{(e)} \\
b_1^{(e)} \\
b_2^{(e)}
\end{bmatrix}
+
\begin{bmatrix}
g_0^{(e)} \\
g_1^{(e)} \\
g_2^{(e)}
\end{bmatrix}
\]  

(15.196)
15.23.4 Assembling the System of Equations

The goal of assembly is to generate a system of equations that can be solved for the vector \( \{ \tilde{\varphi} \} \), whose—here 9—components approximate the solution to the original problem at the nodes. That is, we seek a set of equations of the form

\[
[K]\{ \tilde{\varphi} \} = \{ b \} + \{ g \} \tag{15.197}
\]

where \([K]\) is a—here 9×9—matrix and \( \{ b \} \) and \( \{ g \} \) are—here 9 component—vectors. As in the one-dimensional case discussed in Section 15.9, the equations for the individual elements are assembled to take into account the requirement that the solution be continuous at the boundaries between elements. Regardless of which element one focuses on, the solution along each bounding line is a linear interpolation between the values at the two nodes defining the line. Thus, requiring that the solution be continuous at each node assures continuity along the lines joining all nodes. To illustrate the process, consider the assembly of the equations relating to elements 0 and 1. As shown in Fig. 15.21 and in the connectivity matrix, the global numbers of the three nodes of element 0 are (in order) 1, 3, and 0. Thus,

\[
\tilde{\varphi}^{(0)}_0 = \tilde{\varphi}_1 ; \quad \tilde{\varphi}^{(0)}_1 = \tilde{\varphi}_3 ; \quad \tilde{\varphi}^{(0)}_2 = \tilde{\varphi}_0
\]

and, written with global identifications on the unknown \( \tilde{\varphi} \)'s, Eq. (15.196) for element 0 becomes

\[
\begin{bmatrix}
K^{(0)}_{00} & K^{(0)}_{01} & K^{(0)}_{02} \\
K^{(0)}_{10} & K^{(0)}_{11} & K^{(0)}_{12} \\
K^{(0)}_{20} & K^{(0)}_{21} & K^{(0)}_{22}
\end{bmatrix}
\begin{bmatrix}
\tilde{\varphi}_1 \\
\tilde{\varphi}_3 \\
\tilde{\varphi}_0
\end{bmatrix}
= \begin{bmatrix}
b^{(0)}_0 \\
b^{(0)}_1 \\
b^{(0)}_2
\end{bmatrix} + \begin{bmatrix}
g^{(0)}_0 \\
g^{(0)}_1 \\
g^{(0)}_2
\end{bmatrix}
\tag{15.199}
\]

Similarly, the three nodes of element 1 are (in order) 1, 4, and 3. Thus,

\[
\tilde{\varphi}^{(1)}_0 = \tilde{\varphi}_1 ; \quad \tilde{\varphi}^{(1)}_1 = \tilde{\varphi}_4 ; \quad \tilde{\varphi}^{(1)}_2 = \tilde{\varphi}_3
\]

and Eq. (15.196) for element 1 becomes

\[
\begin{bmatrix}
K^{(1)}_{00} & K^{(1)}_{01} & K^{(1)}_{02} \\
K^{(1)}_{10} & K^{(1)}_{11} & K^{(1)}_{12} \\
K^{(1)}_{20} & K^{(1)}_{21} & K^{(1)}_{22}
\end{bmatrix}
\begin{bmatrix}
\tilde{\varphi}_1 \\
\tilde{\varphi}_3 \\
\tilde{\varphi}_0
\end{bmatrix}
= \begin{bmatrix}
b^{(1)}_0 \\
b^{(1)}_1 \\
b^{(1)}_2
\end{bmatrix} + \begin{bmatrix}
g^{(1)}_0 \\
g^{(1)}_1 \\
g^{(1)}_2
\end{bmatrix}
\tag{15.201}
\]

We have, of course, found six equations constraining only four unknowns, namely \( \tilde{\varphi}_0, \tilde{\varphi}_1, \tilde{\varphi}_3, \) and \( \tilde{\varphi}_4 \). Two of the six are redundant.\(^{59}\) Rather than discard two of them, however, we elect to reduce the number to four by adding selected pairs of these equations. To see the proper combinations, let us recast each of Eqs. (15.199) and (15.201) as equations for the vector containing all four of the involved nodal values. That is, let us augment each equation to make it more obvious that it is not the only equation with which we must deal. In the process, we add, for example, to Eq. (15.199) a row and a column corresponding to global node 4 even though that node does not enter into the equations for element 0. Further, we rearrange the order of the columns in the matrix of Eq. (15.199) so the vector of unknowns can be written with the entries in the order of the global node numbers and we rearrange the order of the rows so that the entries in \( \{ b \} \) and \( \{ g \} \) are also in the order of the global node numbers (and, incidentally and automatically, so that the symmetry of the augmented matrix is preserved). The result is

\[
\begin{bmatrix}
K^{(0)}_{22} & K^{(0)}_{20} & K^{(0)}_{21} & 0 \\
K^{(0)}_{02} & K^{(0)}_{00} & K^{(0)}_{01} & 0 \\
K^{(0)}_{12} & K^{(0)}_{10} & K^{(0)}_{11} & 0 \\
0 & 0 & 0 & 0
\end{bmatrix}
\begin{bmatrix}
\tilde{\varphi}_0 \\
\tilde{\varphi}_1 \\
\tilde{\varphi}_3 \\
\tilde{\varphi}_4
\end{bmatrix}
= \begin{bmatrix}
b^{(0)}_2 \\
b^{(0)}_0 \\
b^{(0)}_1 \\
0
\end{bmatrix} + \begin{bmatrix}
g^{(0)}_2 \\
g^{(0)}_0 \\
g^{(0)}_1 \\
0
\end{bmatrix}
\tag{15.202}
\]

\(^{59}\)We assume that the equations are not contradictory.
Similarly, we augment Eq. (15.201) to obtain the result
\[
\begin{pmatrix}
0 & 0 & 0 & 0 \\
0 & K_{00}^{(1)} & K_{02}^{(1)} & K_{01}^{(1)} \\
0 & K_{02}^{(1)} & K_{22}^{(1)} & K_{21}^{(1)} \\
0 & K_{10}^{(1)} & K_{12}^{(1)} & K_{11}^{(1)}
\end{pmatrix}
\begin{pmatrix}
\hat{\varphi}_0 \\
\hat{\varphi}_1 \\
\hat{\varphi}_3 \\
\hat{\varphi}_4
\end{pmatrix}
=
\begin{pmatrix}
0 \\
0 \\
0 \\
0
\end{pmatrix}
+ 
\begin{pmatrix}
0 \\
b_0^{(1)} \\
b_2^{(1)} \\
b_1^{(1)}
\end{pmatrix}
(15.203)
\]

Basically, we reduce the original set of six equations to the required four by adding these equations, finding as the result of assembling the equations for elements 0 and 1 that
\[
\begin{pmatrix}
K_{22}^{(0)} & K_{20}^{(0)} & K_{21}^{(0)} & 0 \\
K_{02}^{(0)} & K_{00}^{(0)} + K_{00}^{(1)} & K_{01}^{(0)} + K_{02}^{(1)} & K_{01}^{(1)} \\
K_{12}^{(0)} & K_{10}^{(0)} + K_{10}^{(1)} & K_{11}^{(0)} + K_{22}^{(1)} & K_{21}^{(1)} \\
0 & K_{10}^{(1)} & K_{12}^{(1)} & K_{11}^{(1)}
\end{pmatrix}
\begin{pmatrix}
\hat{\varphi}_0 \\
\hat{\varphi}_1 \\
\hat{\varphi}_3 \\
\hat{\varphi}_4
\end{pmatrix}
=
\begin{pmatrix}
b_2^{(0)} \\
b_0^{(0)} + b_0^{(1)} \\
b_1^{(0)} + b_2^{(1)} \\
b_1^{(1)}
\end{pmatrix}
+ 
\begin{pmatrix}
g_2^{(0)} \\
g_0^{(0)} + g_0^{(1)} \\
g_1^{(0)} + g_2^{(1)} \\
g_1^{(1)}
\end{pmatrix}
(15.204)
\]

Furthermore, we can expand these equations to the full \(9 \times 9\) set for all of the nodes, finding that, at this stage in the full assembly,
\[
\begin{pmatrix}
K_{22}^{(0)} & K_{20}^{(0)} & 0 & 0 & K_{21}^{(0)} & 0 & 0 & 0 & 0 \\
K_{02}^{(0)} & K_{00}^{(0)} + K_{00}^{(1)} & 0 & K_{01}^{(0)} + K_{02}^{(1)} & K_{01}^{(1)} & 0 & 0 & 0 & 0 \\
K_{12}^{(0)} & K_{10}^{(0)} + K_{10}^{(1)} & 0 & K_{11}^{(0)} + K_{22}^{(1)} & K_{21}^{(1)} & 0 & 0 & 0 & 0 \\
0 & K_{10}^{(1)} & 0 & K_{12}^{(1)} & K_{11}^{(1)} & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0
\end{pmatrix}
\begin{pmatrix}
\hat{\varphi}_0 \\
\hat{\varphi}_1 \\
\hat{\varphi}_3 \\
\hat{\varphi}_4 \\
\hat{\varphi}_5 \\
\hat{\varphi}_6 \\
\hat{\varphi}_7 \\
\hat{\varphi}_8
\end{pmatrix}
=
\begin{pmatrix}
b_2^{(0)} \\
b_0^{(0)} + b_0^{(1)} \\
b_1^{(0)} + b_2^{(1)} \\
b_1^{(1)}
\end{pmatrix}
+ 
\begin{pmatrix}
g_2^{(0)} \\
g_0^{(0)} + g_0^{(1)} \\
g_1^{(0)} + g_2^{(1)} \\
g_1^{(1)}
\end{pmatrix}
(15.205)
\]

The pattern is now clear:

- The element \(K_{ij}^{(e)}\) of the \(3 \times 3\) matrix \([K^{(e)}]\) contributes additively to the element \(K_{lm}\) of the \(9 \times 9\) assembled matrix \([K]\), where \(l\) is the global number of the node with local number \(i\)
in element \(e\) and \(m\) is the global number of the node with local number \(j\) in element \(e\). For example, \(K(0)_0\) contributes to \(K_{11}\) because local node 0 in element 0 coincides with global node 1; \(K(1)_0\) contributes to \(K_{41}\) because local node 1 of element 1 coincides with global node 4 and local node 0 of element 1 coincides with global node 1.

- The element \(b(e)_{(0)}\) contributes additively to the element \(b_l\), where \(l\) is the global number of the node with local number \(i\) in element \(e\). For example, \(b(0)_{1}\) contributes to \(b_{3}\) because local node 1 in element 0 coincides with global node 3.

- The element \(g(e)_{(0)}\) contributes additively to the element \(g_l\), where \(l\) is the global number of the node with local number \(i\) in element \(e\). For example, \(g(1)_{2}\) contributes to \(g_{3}\) because local node 2 of element 1 coincides with global node 3.

From this point, continuing the assembly to include the contributions from the remaining elements is straightforward but tedious.

### 15.23.5 Incorporating the Boundary Conditions

Apart from the nodal values of \(\tilde{\varphi}\), only the \(g\)'s in Eq. (15.205)—or better in what Eq. (15.205) becomes when all elements have been incorporated—remain unknown. To demonstrate how the boundary conditions either render a priori knowledge of these quantities unnecessary or specify these quantities explicitly, we begin by noting that the fully assembled vector \(\{g\}\) is

\[
\{g\} = \begin{pmatrix}
    g(0)_{2} \\
    g(0)_{1} + g(1)_{2} + g(2) + g(2)_{1} + g(2)_{2} + g(3) \\
    g(2)_{1} + g(3) \\
    g(3)_{1} + g(4)_{2} + g(4)_{3} + g(4)_{4} + g(5)_{5} + g(6)_{6} + g(7)_{7} \\
    g(4)_{4} + g(5)_{5} + g(6)_{6} + g(7)_{7} \\
    g(5)_{5} + g(6)_{6} + g(7)_{7} \\
    g(7)_{7}
\end{pmatrix}
\]  \hspace{1cm} (15.206)

Remember, now, that \(g_{(e)}\) is given by the line integral

\[
g_{(e)} = \oint_{\Gamma_{(e)}} N_{i}^{(e)} \left( \frac{\partial \tilde{\varphi}^{(e)}}{\partial x} \mathbf{i} + \frac{\partial \tilde{\varphi}^{(e)}}{\partial y} \mathbf{j} \right) \cdot \mathbf{n}^{(e)} dl \]  \hspace{1cm} (15.207)

[see Eq. (15.194)] evaluated counterclockwise around the entirety of element \(e\). Furthermore, notice that the various \(g\)’s that are added together in each row of Eq. (15.206) are the line integrals around each element that has a vertex at the global node with which the row in \(\{g\}\) is associated.

Let us look at a few representative elements of \(\{g\}\). Suppose we represent the line integrals by a notation like \((i \rightarrow j + j \rightarrow k + k \rightarrow i)\) for the integral that runs from (global) node \(i\) to (global) node \(j\) to (global) node \(k\) and back to (global) node \(i\) around a particular element. Because the integrand must be continuous along and across the lines separating elements, we need not actually keep track of which element we are traversing on any of the indicated segments. In this notation,
for example, \( g_4 \), which is the sum of the line integrals around all the elements having a vertex at global node number 4, becomes

\[
(4 \rightarrow 3 + 3 \rightarrow 1 + 1 \rightarrow 4) + \\
(4 \rightarrow 1 + 1 \rightarrow 2 + 2 \rightarrow 4) + \\
(4 \rightarrow 2 + 2 \rightarrow 5 + 5 \rightarrow 4) + \\
(4 \rightarrow 5 + 5 \rightarrow 7 + 7 \rightarrow 4) + \\
(4 \rightarrow 7 + 7 \rightarrow 6 + 6 \rightarrow 4) + \\
(4 \rightarrow 6 + 6 \rightarrow 3 + 3 \rightarrow 4)
\]

Next, note that the middle integral in each line is zero, because the interpolation function along that line is identically zero at all points. Then, note that each of the other integrals appears twice, traversed once in each direction, and the two together cancel. This particular sum boils down to zero! The same behavior characterizes all nodes not on the boundary of \( \Omega \). (There is only one such node—global node 4—in the present example.)

We can examine other elements in the vector \( \{g\} \) using the same shorthand notation. For example, the element \( g_2 \) would be represented by

\[
(2 \rightarrow 4 + 4 \rightarrow 1 + 1 \rightarrow 2) + \\
(2 \rightarrow 5 + 5 \rightarrow 4 + 4 \rightarrow 2)
\]

As in the previous paragraph, the middle integral in each item is zero and the first integral in the first item cancels the third integral in the second item. Since this node is on the boundary, however, complete cancellation does not happen; we are left with \( 1 \rightarrow 2 + 2 \rightarrow 5 \). Similar considerations reduce the vector \( \{g\} \) to the much simpler vector

\[
\{g\} = \begin{cases}
3 \rightarrow 0 + 0 \rightarrow 1 \\
0 \rightarrow 1 + 1 \rightarrow 2 \\
1 \rightarrow 2 + 2 \rightarrow 5 \\
6 \rightarrow 3 + 3 \rightarrow 0 \\
0 \\
2 \rightarrow 5 + 5 \rightarrow 8 \\
7 \rightarrow 6 + 6 \rightarrow 3 \\
8 \rightarrow 7 + 7 \rightarrow 6 \\
5 \rightarrow 8 + 8 \rightarrow 7
\end{cases}
\]

(15.208)

Only integrals along edges that are on the boundary of \( \Omega \) remain. Resolution of all remaining unknowns therefore lies in the boundary conditions!

The explicit treatment of these terms from this point depends on what type of boundary conditions are specified. With Dirichlet conditions, the solution is specified on the boundary and the value at each node so constrained is therefore fixed by the boundary conditions. To illustrate the incorporation of this boundary condition, consider a Dirichlet condition at a particular node \( m \). We first replace the row corresponding to global node \( m \) in the equation \( [K]\{\tilde{\varphi}\} = \{b + g\} \) with the equation \( \tilde{\varphi}_m = p_m \), where \( p_m \) is the value specified for node \( m \) by the boundary condition. In short,

- The element in the \( m \)-th column of the \( m \)-th row of \([K]\) is replaced by 1,
- All other elements in that row are replaced by 0, and
- The \( m \)-th element in the vector \( \{b + g\} \) is replaced by \( p_m \).

Then, to complete the incorporation of this boundary condition, we must reflect the influence of this known value \( \tilde{\varphi}_m \) on the remaining equations in order to restore the symmetry of the stiffness matrix. As an example, consider the \( n \)-th equation \((n \neq m)\) in the system:

\[
K_{n0} \tilde{\varphi}_0 + K_{n1} \tilde{\varphi}_1 + \cdots + K_{nm} \tilde{\varphi}_m + \cdots + K_{n7} \tilde{\varphi}_7 + K_{n8} \tilde{\varphi}_8 = b_n + g_n
\]

(15.209)

\[\text{See Exercise 15.15.}\]

\[\text{The original } m \text{-th equation is then interpreted as an equation giving } g_m \text{ after the solution to the altered set of equations has been found.}\]
Since $\tilde{\varphi}_m = p_m$, this expression can be recast as
\[
K_{n0} \tilde{\varphi}_0 + K_{n1} \tilde{\varphi}_1 + \cdots + 0 \tilde{\varphi}_m + \cdots + K_{n7} \tilde{\varphi}_7 + K_{n8} \tilde{\varphi}_8 = b_n + g_n - K_{nm} p_m \tag{15.210}
\]
Such recasting of all equations for $n \neq m$ is accomplished by

- Multiplying by $p_m$ all elements except the element associated with node $m$ in the column associated with node $m$ of the original matrix $[K]$,
- Subtracting each resulting product from the corresponding element in the original vector $\{b + g\}$, and then
- Substituting zero for all elements in the $m$-th column of $[K]$ except the element in the $m$-th row (which had already been set to 1).

So, for each node on which a Dirichlet condition is declared,

- The associated element in the vector $\{b + g\}$ is replaced by the known value at the node corresponding to that element,
- The other elements in the vector $\{b + g\}$ are adjusted as per Eq. (15.210),
- All elements in the associated row and the associated column of $[K]$ except the element in the intersection of that row and column are set to 0, and
- The element at that intersection is set to 1.

If we have Dirichlet conditions at all bounding nodes, incorporation of these conditions results in replacing the original vector $\{b + g\}$ with a new vector containing no unknown quantities. The problem is now reduced to solving a fully determined (probably large) set of simultaneous linear equations for the vector $\{\tilde{\varphi}\}$.

When Neumann conditions are specified, we are given the quantity defined in Eq. (15.72). This gives us enough information to evaluate the integral in Eq. (15.207), not around an entire element but along a portion of the boundary of the region $\Omega$. We are therefore in a position explicitly to evaluate the integrals symbolized above by $3 \to 0$, $0 \to 1$, etc. Thus, the $g$’s associated with portions of the boundary along which Neumann conditions are specified have values directly determinable from those boundary conditions. If Neumann conditions are specified on all boundaries, all parameters in the equation $[K]\{\tilde{\varphi}\} = \{b + g\}$ except the dependent variable $\tilde{\varphi}$ are known from the beginning, so again the problem is reduced to solving a fully determined set of simultaneous linear equations.

We can proceed no further in this example without resorting to a specific coding language, so we postpone further discussion to later sections.

### 15.27 Using PYTHON to Solve 2D PDEs via an FEM

#### 15.27.1 A General Coding

The process of assembling the system of equations is tedious, if not difficult. The task of solving the resulting large set of simultaneous algebraic equations is to be sure, straightforward, but it can be enormously time-consuming, especially when the region of interest is more complicated than the simple eight-element geometry that we have so far discussed. The task is clearly one for a computer, and it will almost certainly be carried out numerically, not symbolically. Ideally, we would like to input an equation of the form of Eq. (15.70), a suitable definition of the region of interest, and all applicable boundary conditions, assigning to the computer the tasks of (1) constructing the complete equation begun in Eq. (15.205), (2) applying the boundary conditions, and (3) solving the system of equations.

In order to simplify the computer coding for this example and provide a clearer picture of finite element programming without becoming overwhelmed by geometric details, we will impose
several restrictions on the problems that our program can solve. First, we will limit ourselves to a square domain located in the first quadrant with one corner at the origin, such that \( 0 \leq x, y \leq L \). Further, to ensure that all elements will be isosceles right triangles (which simplifies computation considerably), we will divide each edge of this square into the same number \( d \) of segments of equal length \( L/d \). In addition, we will restrict \( \alpha_x, \alpha_y, \beta, \) and \( f \) to be constants and, instead of allowing for general boundary conditions, we will assume the constant Dirichlet conditions
\[
\varphi(x, 0) = p_2 \quad ; \quad \varphi(x, L) = p_1
\]
(15.211)
on the bottom \((y = 0)\) and top \((y = L)\) boundaries, the linear variation
\[
\varphi(0, y) = \frac{p_1 - p_2}{L} y + p_2
\]
(15.212)
between the bottom and top values along the left edge \((x = 0)\), and the Neumann boundary condition
\[
\frac{\partial \varphi}{\partial x}(L, y) = q
\]
(15.213)
with \( q \) constant along the right edge \((x = L)\). Collectively, these assumptions illustrate several different types of boundary condition.

In the first segment of the coding, we will request input of the various parameters describing the problem. Appropriate PYTHON statements are
\[
L = \text{input}('\text{Enter length of side (L): } '); L = \text{float}(L)
d = \text{input}('\text{Enter number of segments (d): } '); d = \text{int}(d)
\]
\[
\text{alpha}_x = \text{input}('\text{Enter alpha}_x: '); \text{alpha}_x=\text{float(alpha}_x)\]
\[
\text{alpha}_y = \text{input}('\text{Enter alpha}_y: '); \text{alpha}_y=\text{float(alpha}_y)\]
\[
\text{beta} = \text{input}('\text{Enter beta: } '); \text{beta}=\text{float(beta)}\]
\[
f = \text{input}('\text{Enter f: } '); f=\text{float(f)}\]
\[
p1 = \text{input}('\text{Enter value for top edge (p1): } '); p1=\text{float(p1)}\]
\[
p2 = \text{input}('\text{Enter value for bottom edge (p2): } '); p2=\text{float(p2)}\]
\[
q = \text{input}('\text{Enter q: } '); q=\text{float(q)}\]
The number of elements, \( M \), is twice the square of the number of segments on each edge, the number of nodes, \( N \), is equal to the square of one plus the number of segments on each edge, i.e., to \((M+1)^2\), and the length of each segment of each edge is the length of the edge divided by the number of segments into which the edge is divided, all as reflected in the additional coding
\[
M = 2*d**2 \quad \# \text{Calculate number of elements}
\]
\[
N = (d+1)**2 \quad \# \text{Calculate number of nodes}
\]
\[
dx = L/d \quad \# \text{Calculate segment size}
\]
As illustrated in Fig. 15.22, the number of segments \( d \) also provides a general way to refer to element numbers and global node numbers for varying numbers of elements.

After calculating the number of elements and nodes involved, we find the coordinates of all of the nodes. We elect to store these coordinates in two vectors, \( x \) and \( y \), where each entry is the \( x \) or \( y \) coordinate of the node whose global node number corresponds to the index of the entry in the vector. These assignments can be made with two nested for loops, the inner loop incrementing row numbers down a column of nodes and the outer loop incrementing the column number. In this way the nodes are traversed in the sequence in which we numbered them. Further, we can keep track of the correct global node number with a simple counter. To accomplish the assignment of values to \( x \) and \( y \), we would use the coding
\[\text{62}\]

\[\text{int} \quad \text{and float functions guarantees that parameters will have the proper data type, regardless of how they happen to be entered.}\]
Figure 15.22: General element and global node numbering based on the number of segments \( d \) into which each edge is divided. Consistent with PYTHON conventions, we have here numbered elements and nodes starting at 0.

\[
\begin{array}{c|c|c|c|c|c|c}
\hline
& 0 & d + 1 & 2(d + 1) & d^2 - 1 & d(d + 1) \\
\hline
0 & 0 & 2d & 2d + 1 & 2d(d - 1) & d(d + 1) \\
1 & 1 & 2d + 2 & 2d + 3 & 2d(d - 1) & 2d(d + 1) + 1 \\
2 & 2 & 2d + 3 & 2d + 4 & 2d(d - 1) & 2d(d + 1) + 2 \\
\vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\
\hline
d & 2d - 2 & 4d - 2 & 2d^2 - 2 & (d + 1)^2 - 2 \\
\hline
d + 1 & 2d + 1 & 3d + 2 & 2d^2 - 1 & (d + 1)^2 - 1 \\
\hline
\end{array}
\]

We assign global and local node numbers following the pattern in Fig. 15.23. The next step is to create and store the connectivity matrix, so we can easily convert between local and global node numbers. As illustrated in Table 15.2, this matrix is a two-dimensional array in which the index of each row is an element number, and the entries in each row are the global node numbers of the zeroth, first, and second local nodes of that element. Then, in developing the coding to create this matrix, we recognize that

- **Integer** division of an element number by \( 2 \cdot d \) yields the index of the vertical line in Fig. 15.22 on which the zeroth node of the element lies. Here, the index of the vertical line is simply a count of the number of vertical lines to its left, so the first vertical line is at index 0, the second at index 1, and the last at index \( d \).

```
import numpy as np
x = np.zeros(N)  # Create array to store x values
y = np.zeros(N)  # Create array to store y values
ct = -1  # Initialize a counter variable
for i in range(d+1):  # Start row number loop
    for j in range(d+1):  # Start column number loop
        ct = ct + 1  # Increment counter
        x[ct] = i*dx  # Find x coordinate
        y[ct] = L - j*dx  # Find y coordinate
```

Figure 15.23: Simple square domain based on Fig. 15.20 with element numbers (left and right),
global node numbers (left), and local node numbers (right), all starting at 0 as appropriate to the
way arrays are indexed in PYTHON.

- On a given vertical line, the global node number of the zeroth local node increases by *one*
every time the element number increases by *two*. Therefore, we can in general find the *global*
ode number of an element’s zeroth local node by
  - performing *integer* division of the element number by two,
  - adding one (since the sequence of zeroth local nodes begins with global node 1), and
  - adding the index of the vertical line on which the node lies.

- Once the global node number of the zeroth local node of a particular element has been found,
  the remaining two global node numbers for that element follow easily. The way in which they
  follow, however, depends on whether the element is odd or even.63

Thus, the connectivity matrix \( cm \) is created by the statements

```python
cm = np.zeros([3,M], dtype=int) # Create 3 by M null array
for e in range(M): # Loop through all elements
    vl = int(e/(2*d)) # Find index of vertical line
    cm[0,e] = int(e/2) + 1 + vl # Find global number of node 0
    if 2*int(e/2) == e: # If e is even
        cm[1,e] = cm[0,e] + d # find global number of node 1
        cm[2,e] = cm[0,e] - 1 # find global number of node 2
    else: # If e is odd
        cm[1,e] = cm[0,e] + d + 1 # find global number of node 1
        cm[2,e] = cm[0,e] + d # find global number of node 2
```

The next step is to construct the stiffness matrix \( [K^{(e)}] \) for a single element. As given in
Eq. (15.192), the elements of \([K^{(e)}] \) depend on both the interpolation functions and their derivatives.
Since our elements are all isosceles right triangles, the coordinates of all local nodes are related
through the leg length, \( \Delta x \). Therefore, we can substitute for \( x_1^{(e)} \) and \( x_2^{(e)} \) in terms of \( x_0^{(e)} \) and \( \Delta x \),
and for \( y_1^{(e)} \) and \( y_2^{(e)} \) in terms of \( y_0^{(e)} \) and \( \Delta x \) in the expression for \( K_{ij}^{(e)} \). However, because of the

---

63 Several invocations of the `int` command are necessary to make sure certain calculated values are integers.
different orientation of odd and even numbered elements, these substitutions are not identical for the two cases. For odd numbered elements, we set

\[ x_1^{(e)} = x_2^{(e)} = x_0^{(e)} + \Delta x \quad , \quad y_1^{(e)} = y_0^{(e)} \quad \text{and} \quad y_2^{(e)} = y_0^{(e)} + \Delta x \quad (e \text{ odd}) \]  

(15.214)

In contrast, for even numbered elements, we set

\[ x_1^{(e)} = x_0^{(e)} + \Delta x \quad , \quad x_2^{(e)} = x_0^{(e)} \quad \text{and} \quad y_1^{(e)} = y_2^{(e)} = y_0^{(e)} + \Delta x \quad (e \text{ even}) \]  

(15.215)

With these simplifications, the elements defining the entries of \([K^{(e)}]\) are much easier to evaluate. Whether the element number \(e\) is even or odd, we elect to do the \(y\) integral first and the \(x\) integral second. In all cases, \(x\) will then run from \(x_0^{(e)}\) to \(x_0^{(e)} + \Delta x\). The limits on \(y\), however, will depend on whether \(e\) is odd or even. In the present mesh, the triangular elements are all right, isosceles triangles and the diagonal lines defining the hypotenuses of these triangles lie at a 45° angle to the horizontal. Thus, when \(e\) is odd (see Fig. 15.22), the limits on \(y\) will run from \(y_0^{(e)}\) to \(y_0^{(e)} + x - x_0^{(e)}\), and we conclude that

\[
\int \int_{\Omega^{(e)}} \ldots \, dx \, dy = \int_{x_0^{(e)}}^{x_0^{(e)} + \Delta x} \left[ \int_{y_0^{(e)}}^{y_0^{(e)} + x - x_0^{(e)}} \ldots \, dy \right] \, dx \quad (e \text{ odd})
\]  

(15.216)

When \(e\) is even, on the other hand, the limits on \(y\) will instead run from \(y = y_0^{(e)} + x - x_0^{(e)}\) to \(y_0^{(e)} + \Delta x\), and we conclude that

\[
\int \int_{\Omega^{(e)}} \ldots \, dx \, dy = \int_{x_0^{(e)}}^{x_0^{(e)} + \Delta x} \left[ \int_{y_0^{(e)} + x - x_0^{(e)}}^{y_0^{(e)} + \Delta x} \ldots \, dy \right] \, dx \quad (e \text{ even})
\]  

(15.217)

Because the integrands are no worse than quadratic in \(x\) and \(y\), their evaluation is straightforward, but there are many of them and we elect to invoke a symbol manipulating program.\(^{64}\) It turns out that the entries in \([K^{(e)}]\) depend only on the parameters \(\Delta x\), \(\alpha_x\), and \(\alpha_y\), and not upon \(e\), the specific element. In other words, there exist only two distinct elemental stiffness matrices, one for odd elements and one for even elements, namely

\[
K_{\text{odd}}^{(e)} = \begin{bmatrix}
\frac{\beta \Delta x^2 + 6\alpha_x}{12} & \frac{\beta \Delta x^2 - 12\alpha_x}{24} & \frac{\beta \Delta x^2}{24} \\
\frac{\beta \Delta x^2 - 12\alpha_x}{24} & \frac{\beta \Delta x^2 + 6\alpha_x + 6\alpha_y}{12} & \frac{\beta \Delta x^2 - 12\alpha_y}{24} \\
\frac{\beta \Delta x^2}{24} & \frac{\beta \Delta x^2 - 12\alpha_y}{24} & \frac{\beta \Delta x^2 + 6\alpha_y}{12}
\end{bmatrix}
\]  

(15.218)

and

\[
K_{\text{even}}^{(e)} = \begin{bmatrix}
\frac{\beta \Delta x^2 + 6\alpha_y}{12} & \frac{\beta \Delta x^2}{24} & \frac{\beta \Delta x^2 - 12\alpha_y}{24} \\
\frac{\beta \Delta x^2}{24} & \frac{\beta \Delta x^2 + 6\alpha_x}{12} & \frac{\beta \Delta x^2 - 12\alpha_x}{24} \\
\frac{\beta \Delta x^2 - 12\alpha_y}{24} & \frac{\beta \Delta x^2 - 12\alpha_x}{24} & \frac{\beta \Delta x^2 + 6\alpha_x + 6\alpha_y}{12}
\end{bmatrix}
\]  

(15.219)

The coding that constructs these two matrices in our program is

```python
Kodd = np.zeros([3,3])  # Create two 3 by 3 arrays
```

\(^{64}\)Details of this evaluation are described more fully in Appendix 15.F.
CHAPTER 15. SOLVING PARTIAL DIFFERENTIAL EQUATIONS

Keven = np.zeros([3,3])

bx = beta*dx**2  # Evaluate a common quantity

Kodd[0,0] = (bx + 6*alpha_x)/12  # Assign the appropriate value to
Kodd[1,1] = (bx + 6*alpha_x + 6*alpha_y)/12  # each K(i,j). Note that the array
Kodd[2,2] = (bx + 6*alpha_y)/12  # are symmetric, and that Keven
Kodd[0,1] = Kodd[1,0]  # includes all of the same values as
Kodd[1,2] = Kodd[2,1]  # Kodd, but in different locations.
Kodd[0,2] = bx/24
Kodd[2,0] = bx/24

Keven[0,0] = Kodd[2,2]
Keven[1,1] = Kodd[0,0]
Keven[2,2] = Kodd[1,1]
Keven[0,1] = Kodd[0,2]
Keven[1,2] = Kodd[1,2]
Keven[0,2] = Kodd[1,2]

We are now ready to assemble the complete matrix $[K]$, some of whose elements were shown in Eq. (15.205). At the same time, we can also construct the vector $\{b\}$, whose entries $b^e(i)$ can be found by using a symbol manipulating program to evaluate the integrals in Eq. (15.193). It turns out that all three elements $b^e_0$, $b^e_1$, and $b^e_2$ have the same value $f \Delta x^2/6$, regardless of whether $e$ is odd or even. Following the algorithm described at the end of Section 15.23.4 for assembling the final stiffness matrix, we start by creating a null matrix $[K]$ and a null vector $\{b\}$ of the proper dimensions. Then, we step through the elements one at a time, at each step adding the contributions of the element to the accumulating entries at the proper positions in $[K]$ and $\{b\}$. This end is achieved with the coding

```python
K = np.zeros([N,N]);  # Create arrays to store values of
b = np.zeros([N,1]);  # K(i,j) and b(i)

for e in range(M):  # Count through element numbers
  for i in range(3):  # For each local node of an element
    for j in range(3):  # place its contributions at the
      if (2*int(e/2) != e):
        K[cm[i,e],cm[j,e]] = K[cm[i,e],cm[j,e]] + Kodd[i,j]
      else:
        K[cm[i,e],cm[j,e]] = K[cm[i,e],cm[j,e]] + Keven[i,j]
```

Application of the boundary conditions will complete the setup of the problem. To apply the Dirichlet conditions to the top, bottom, and left edges we must first find an algorithm to identify which nodes are on these edges for various values of $d$. By inspection, we find that, by letting $i$ run from zero to $d$, global node numbers on the left boundary are given by $i$, on the top by $i(d+1)$, and on the bottom by $(i+1)(d+1) - 1$. Once we know these numbers, we can replace the appropriate

---

65See again Appendix 15.F.
15.27. USING PYTHON TO SOLVE 2D PDES VIA AN FEM

equations by the given values of $\bar{\phi}$, and then reflect the influence of this relationship in the other equations as described in Section 15.23.5. The requisite coding is

```python
for i in range(d+1):
    u = i  # Nodes on the left boundary
    s = i*(d+1)  # Nodes on the top boundary
    t = (i+1)*(d+1) - 1  # Nodes on the bottom boundary
    p3 = (p1-p2)/L * y[u] + p2  # Find values of phi on left boundary

for j in range(N):
    K[j,s] = 0  # Set rows in K to zero where value
    K[j,t] = 0  # of phi is known
    K[j,u] = 0
    b[s] = p1  # Set values of b
    b[t] = p2
    b[u] = p3
    if (j != s):
        b[j] = b[j] - K[s,j]*p1  # Reflect influence
    if (j != t):
        b[j] = b[j] - K[t,j]*p2
    if (j != u):
        b[j] = b[j] - K[u,j]*p3
    K[s,s] = 1  # Set the appropriate entry to 1 in
    K[t,t] = 1  # the rows where phi is known
    K[u,u] = 1
```

Since the Neumann condition is specified only on the right edge, the unit normal vector $\hat{n}$ in Eqs. (15.72) and (15.194) becomes simply $\hat{i}$. Also, as shown in Section 15.23.5, we need worry only about segments on the boundary of $\Omega$ when integrating to find $g_i^{(e)}$. These two items and the fact that all elements with boundaries on the right edge have odd numbers reduce Eq. (15.194) to

$$g_i^{(e)} = \int_{y_0^{(e)}}^{y_1^{(e)}+\Delta x} N_i^{(e)} \alpha_x \frac{\partial \bar{\phi}}{\partial x} dy$$

(15.220)

Since we know that $\alpha_x (\partial \bar{\phi} / \partial x) = q$ and that $x = x_0^{(e)} + \Delta x$, we can easily evaluate this expression, provided we assume $\alpha_x$ can be treated as a constant over the area of each element. Again invoking a symbol manipulating program, we find that

$$g_0^{(e)} = 0 \quad ; \quad g_1^{(e)} = \frac{q \Delta x}{2} \quad ; \quad g_2^{(e)} = \frac{q \Delta x}{2}$$

(15.221)

The contribution of each node to the complete $\{g\}$ vector will be twice this value, since it consists of two parts, one from each right boundary element that contains the node. We don’t need $g_0^{(e)}$ since no node on the right edge is a first local node. The necessary additional coding, including the consolidation of the inhomogeneities into a single vector, is

```python
g = np.zeros([N,1])  # Create vector to store g values
for i in range(d):
    nd = d*(d+1)+i  # Nodes on right boundary
    g[nd] = q * dx
b = b+g  # Store the vector {b+g} in {b}
```

---

66 Remember that indices in PYTHON start at 1. We implement that wrinkle by adding 1 to $u$, $s$, and $t$ and by running the loop on $j$ from 1 to $N$ rather than 0 to $N-1$.

67 See Appendix 15.F once more.
At this point, K contains the coefficient matrix and b the inhomogeneities for the system of simultaneous linear equations we wish to solve.

We need only solve the system and then write the results into an appropriate two-dimensional array. The remaining coding thus is

\[
\text{phi} = \text{np.linalg.solve}(K, b) \quad \# \text{Solve the equation } K\text{phi} = b
\]

\[
\text{phi} = \text{np.linalg.solve}(K, b) \quad \# \text{writing the results to phi}
\]

\[
A = \text{np.zeros}(d+1,d+1) \quad \# \text{Create a } d+1 \text{ by } d+1 \text{ array}
\]

\[
cnt=0 \quad \# \text{Initialize a counter}
\]

\[
\text{for i in range}(d+1): \quad \# \text{Use nested for loops to write all}
\]

\[
\text{for j in range}(d+1): \quad \# \text{entries in } A
\]

\[
A[i,j] = \text{phi}[cnt] \quad \# \text{Increment counter}
\]

Since the nodes are, in this example, uniformly spaced over the region of the problem, we need not be concerned about knowing the nodal coordinates for purposes of plotting the solution. The coordinates of the node corresponding to each value in the two-dimensional array A are proportional to the indices of that value in the array.

All of the preceding PYTHON code has been incorporated into the procedure fem2d.py, a listing of which can be found in Appendix 15.G.4. (The file itself can be copied from the directory $HEAD/python.) When run, the PYTHON script simply asks for necessary input, calculates the solution, and returns that solution in the array A. The array A can then be examined, probably by plotting a surface, to view the solution.

### 15.27.2 An Example: Isotropic Heat Flow

We will now demonstrate fem2d by applying it to an equation in isotropic heat flow. The steady-state temperature \(u(x, y)\) in a square plate of uniform composition is a solution to the two-dimensional Laplace equation

\[
\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} = 0 \quad (15.222)
\]

to which Eq. (15.70) reduces when we replace \(\tilde{\varphi}\) with \(u\) and set

\[
\alpha_x = 1 \quad ; \quad \alpha_y = 1 \quad ; \quad \beta = 0 \quad ; \quad f = 0 \quad (15.223)
\]

We will take the plate to be ten units square \((L = 10)\), and position it in the first quadrant with a corner at the origin, and we will apply the boundary conditions

\[
u(x, 0) = 0 \quad ; \quad u(x, 10) = 100 \quad ; \quad u(0, y) = 10y \quad ; \quad \frac{\partial u}{\partial x}(10, y) = 0 \quad (15.224)
\]

i.e., \(p_2 = 0, p_1 = 100, L = 10, \) and \(q = 0\). With these choices, we run the program and solve the problem with the single statement

```
execfile("fem2d.py") or exec(open(‘fem2d.py’).read() )
```

At the several prompts generated by this statement, we enter the values

Enter length of side (L): 10
Enter number of segments (d): 8
Enter alpha_x: 1
Enter alpha_y: 1
Enter beta: 0
Enter f: 0
Enter value for top edge: 100
Enter value for bottom edge: 0
Enter q: 0

Each line appears one at a time, and waits for us to input the requested parameter. After the last parameter has been entered, the program generates a solution and stores it in the array `A`, which is then accessible to the main program. The statement

```
print( np.transpose(A) )
```

shows the solution as a matrix. Here, we must transpose the internal array so that the printed solution will be oriented in the way we have been visualizing the geometry of the problem.

A mesh plot of the solution over the $xy$ plane can now be displayed easily by invoking the statements

```
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import axes3d
xx = np.linspace(0.0,L,d+1)
x,y = np.meshgrid( xx, xx )
fig1 = plt.figure(1)
ax1 = plt.axes(projection='3d')
Aadj = np.flip(np.transpose(A))
ax1.plot_surface(x,y,Aadj, color='white', shade=False, edgecolor='black')
ax1.set_xlabel('x'); ax1.set_ylabel('y'); ax1.set_zlabel('A')
plt.show()
```

The resulting graph is shown in Fig. 15.24. This figure shows that the temperature varies linearly throughout the plate. This result is, of course, entirely consistent with our intuition: the temperature in the plate should vary linearly between the extremes on opposite edges.

When working with numerical approximation techniques such as finite element analysis, we must always be wary of the accuracy of results. When investigating a problem with a less evident solution, we would be tempted to test the results by performing additional analyses using a larger number of elements. In the present case, for example, we could generate the solution on a finer grid by specifying the values

```
execfile("fem2d.py") or exec( open('fem2d.py').read() )
```

Here, we need not only transpose but then flip the internal array so the graph associates the surface properly with the axes.
Figure 15.24: The steady-state temperature distribution of the isotropic plate. This graph was produced with PYTHON.

Enter alpha_x: 1
Enter alpha_y: 1
Enter beta: 0
Enter f: 0
Enter value for top edge: 100
Enter value for bottom edge: 0
Enter q: 0

which will generate a solution on a $17 \times 17$ grid with twice as many divisions as the first solution. Then, we could extract a $9 \times 9$ array from this more refined $17 \times 17$ solution and display that portion of the solution on the original grid with the statements

```python
B = np.zeros([9,9])
for i in range(9):
    for j in range(9):
print( np.transpose(B) )
```

We discover that, at least to the displayed precision, the two solutions are identical, and we conclude that discretization and roundoff error are under control to this level of accuracy. (Because we have
solved the algebraic equations by a direct method, we need not here be concerned about convergence error.)

15.30 Exercises

Note: In these exercises, we refer to the programs developed in the text without appending the file type, adopting this approach to leave to you the choice of which language to use in addressing the exercise.

15.1. Recast fdm1d to solve the ODE of Eq. (pde:diffeq) when \( f(x) = kx \), the boundary conditions are
\[
\frac{\partial \varphi}{\partial x}(0) = p ; \quad \varphi(L) = q
\]
and \( \alpha, \beta, k, p, q, \) and \( L \) are constants whose values are to be read in at execution time. Then, using your command file, explore the solution to the equation for various values of \( k \) when
\[
\alpha = -4.0 ; \quad \beta = 4.0 ; \quad L = 10.0 ; \quad p = q = 0
\]
Optional: (a) Find an analytic solution to the equation in this exercise and compare the exact results with the approximate results generated by your modification of \texttt{fdm1d}. (b) Find the points at which the solution has the value zero, both starting with the solution obtained in the main exercise and working from the exact solution obtained in optional part (a).

15.2. Suppose the one-dimensional string of length \( l \) discussed in Section 15.1.1 hangs vertically and is acted on by gravity. Suppose that \( u(x,t) \) and \( v(x,t) \) give the transverse (horizontal) and longitudinal (vertical) displacements of the particle of the string nominally located at \( x \), which is measured downward from the top of the string. (a) Examine the forces acting on this string, deduce the general equations for both longitudinal and transverse motion of the string, and show ultimately that, if the motion is entirely transverse and the amplitude of the motion is small, the equations reduce to
\[
\rho(x) \frac{\partial^2 u}{\partial t^2} = \frac{\partial}{\partial x} \left( \tau(x) \frac{\partial u}{\partial x} \right) ; \quad \tau(x) = -g \int_x^l \rho(x') \, dx'
\]
where \( \rho(x) \) is the mass per unit length of the string. (b) Taking \( \rho \) to be constant, show that the tension \( \tau(x) \), which is simply equal to the weight of the string below \( x \), is given by \( \tau(x) = pg(l - x) \).

With this restriction, the equation of motion then becomes
\[
\frac{\partial^2 u}{\partial t^2} = \frac{\partial}{\partial x} \left( (l - x) g \frac{\partial u}{\partial x} \right)
\]
(c) Recast this equation in dimensionless form by introducing the variables \( \pi = x/l \) and \( \ell = t \sqrt{g/l} \).
(d) Suppose you seek a sinusoidal solution for which \( u(\pi, \ell) = f(\pi) \cos \omega \ell \). Find the ODE satisfied by \( f(\pi) \) and then introduce the new independent variable \( y \) defined by \( y^2 = 1 - \pi \) to find that, expressed as a function of \( y, f \) satisfies the Bessel equation
\[
y^2 \frac{d^2 f}{dy^2} + y \frac{df}{dy} - 4\omega^2 y^2 f = 0
\]
Note: The variable transformation \( y^2 = 1 - \pi \) in effect recognizes that the vertical string is more appropriately treated by locating points on the string relative to the bottom rather than the top of the string!

15.3. Show that every second-order linear ODE can be cast in self-adjoint form, i.e., show that functions \( \alpha(x), \gamma(x), \) and \( f(x) \) can be found such that the general second-order linear ODE
\[
a(x) \frac{d^2 \varphi}{dx^2} + b(x) \frac{d\varphi}{dx} + c(x) \varphi = g(x)
\]
can be recast in the form of Eq. (15.67) in Section 15.1.7. Hint: Multiply the equation by an undetermined integrating factor \( h(x) \), find a (first-order) differential equation for \( h(x) \), and solve that equation—at least formally.
15.4. Recast \texttt{fdm1d} to solve the equation \[
\frac{d^2 u}{dx^2} + kxu = 0
\]
over the interval \(0 \leq x \leq L\) subject to the boundary conditions \(u(0) = 0.0\) and \(du(L)/dx = 1.0\), arranging your program so the value of \(k\) is entered at execution time. Then, explore the solution in some detail for various values of \(k\), making sure to assess the accuracy of your solution.

15.5. In (two-dimensional) polar coordinates \((\nabla, \phi)\), the Laplace equation is
\[
\nabla^2 u = \frac{1}{\nabla} \frac{\partial}{\partial \nabla} \left( \nabla \frac{\partial u}{\partial \nabla} \right) + \frac{1}{\phi^2} \frac{\partial^2 u}{\partial \phi^2} = 0 \quad \Rightarrow \quad \nabla^2 \frac{\partial^2 u}{\partial \phi^2} + \nabla \frac{\partial u}{\partial \nabla} + \frac{\partial^2 u}{\phi^2} = 0
\]
Suppose a solution is sought in the region \(0 < a \leq \nabla \leq b, \ 0 \leq \phi \leq 2\pi\) subject to the boundary conditions
\[
u(a, \phi) = f(\phi) ; \quad u(b, \phi) = g(\phi)
\]
and the requirement that the solution be periodic with period \(2\pi\) in \(\phi\). Let \(\Delta \nabla = (b - a)/n\) and \(\Delta \phi = 2\pi/m\) and then let \(\nabla_i = a + i \Delta \nabla, i = 0, 1, \ldots, n\) and \(\phi_j = j \Delta \phi, j = 0, 1, 2, \ldots, m\). Finally let \(u_{i,j} = u(\nabla_i, \phi_j)\). Discretize this equation. In particular use a central difference formula to approximate the derivative \(du/\partial \nabla\). You should find ultimately that
\[
u_{i+1,j} = \frac{\nabla^2 \Delta \phi^2 + \frac{1}{2} \nabla \Delta \phi^2 \Delta \nabla}{2 \nabla^2 \Delta \phi^2} + \frac{u_{i-1,j} (\nabla^2 \Delta \phi^2 - \frac{1}{2} \nabla \Delta \phi^2 \Delta \nabla) + \left(u_{i,j+1} + u_{i,j-1}\right) \Delta \nabla^2}{2 \nabla^2 \Delta \phi^2 + 2 \Delta \nabla^2}
\]
Determine how this equation must be modified when \(i = 0, i = n\) and/or \(j = 0, j = m\), and then write a program to solve this equation when
\[
f(\phi) = 0 ; \quad g(\phi) = \begin{cases} -100 & -\pi < \phi < 0 \\ 100 & 0 \leq \phi \leq \pi \end{cases}
\]
Finally, compile, link, and run your program to explore the solution in some detail.

15.6. Suppose you know three points \((x_0, f_0), (x_1 = x_0 + \Delta x, f_1)\), and \((x_2 = x_0 + 2\Delta x, f_2)\) on a curve and you wish to estimate the derivative of the corresponding function at \(x = x_0\) by a method that is more accurate than simply the forward difference formula \((f_1 - f_0)/\Delta x\). You might fit the parabola \(f(x) = Ax^2 + Bx + C\) through the three points and then approximate the derivative of the actual function as the derivative of this parabola at the point \(x = x_0\). Show that this approach yields the formula
\[
\frac{df}{dx} \bigg|_{x=x_0} = \frac{-3f_0 + 4f_1 - f_2}{2 \Delta x}
\]
which is a higher-order forward difference approximation than the one used in the text. \textit{Hint:} Set up the three equations \(f_i = Ax_i^2 + Bx_i + C\), solve those equations for \(A, B,\) and \(C\), and note that \(df/dx = 2Ax + B\). You may find a symbolic manipulating program of substantial assistance.

15.7. Consider the differential equation and boundary conditions
\[
\frac{d^2 u}{dx^2} + k^2 u = 0 \quad ; \quad u(0) = u(L) = 0
\]
(a) Find the analytic solution to this problem and identify the special values of \(k\) that permit non-trivial solutions. (b) Show that the finite difference approach to the problem leads to the matrix eigenvalue problem \(Au = -k^2 L^2 u/N^2\) where \(N\) is the number of equal-length segments into which the length \(L\) of the domain is divided and
\[
A = \begin{pmatrix}
-2 & 1 & 0 & \cdots & 0 & 0 & 0 \\
1 & -2 & 1 & \cdots & 0 & 0 & 0 \\
0 & 1 & -2 & \cdots & 0 & 0 & 0 \\
\vdots & \vdots & \vdots & \ddots & \vdots & \vdots & \vdots \\
0 & 0 & 0 & \cdots & -2 & 1 & 0 \\
0 & 0 & 0 & \cdots & 1 & -2 & 1 \\
0 & 0 & 0 & \cdots & 0 & 1 & -2 \\
\end{pmatrix} ; \quad u = \begin{pmatrix}
u_1 \\
u_2 \\
u_3 \\
\vdots \\
u_{N-3} \\
u_{N-2} \\
u_{N-1} \\
\end{pmatrix}
\]
Note that, when the domain $0 \leq x \leq L$ is divided into $N$ segments, there will be $N + 1$ nodes ranging from $x_0 = 0$ to $x_{N+1} = L$. Because, along the way to a solution, the boundary conditions result in the rows and columns associated with these two nodes being deleted, these matrices will have only $N - 1$ rows and columns. (c) Taking $N = 100$, use an available numeric processing program like IDL, MATLAB, OCTAVE, or PYTHON to find the first several eigenvalues $k_n$ and compare your results with the values found in part (a).

15.8. Consider the differential equation and boundary conditions
\[ \frac{d^2u}{dx^2} + k^2 u = 0 \quad ; \quad u(0) = u(L) = 0 \]

(a) Find the analytic solution to this problem and identify the special values of $k$ that permit non-trivial solutions. (b) Show that the finite element approach to the problem leads to the generalized eigenvalue problem $Au = -k^2 L^2 Bu/6N^2$ where $N$ is the number of equal-length elements into which the length $L$ of the domain is divided and

\[
A = \begin{pmatrix}
-2 & 1 & 0 & \cdots & 0 & 0 & 0 \\
1 & -2 & 1 & \cdots & 0 & 0 & 0 \\
0 & 1 & -2 & \cdots & 0 & 0 & 0 \\
\vdots & \vdots & \vdots & \ddots & \vdots & \vdots & \vdots \\
0 & 0 & 0 & \cdots & -2 & 1 & 0 \\
0 & 0 & 0 & \cdots & 1 & -2 & 1 \\
0 & 0 & 0 & \cdots & 0 & 1 & -2 \\
\end{pmatrix}; \quad B = \begin{pmatrix}
4 & 1 & 0 & \cdots & 0 & 0 & 0 \\
1 & 4 & 1 & \cdots & 0 & 0 & 0 \\
0 & 1 & 4 & \cdots & 0 & 0 & 0 \\
\vdots & \vdots & \vdots & \ddots & \vdots & \vdots & \vdots \\
0 & 0 & 0 & \cdots & 4 & 1 & 0 \\
0 & 0 & 0 & \cdots & 1 & 4 & 1 \\
0 & 0 & 0 & \cdots & 0 & 1 & 4 \\
\end{pmatrix}
\]

and

\[
u = \begin{pmatrix}
u_1 \\
u_2 \\
u_3 \\
\vdots \\
u_{N-3} \\
u_{N-2} \\
u_{N-1}
\end{pmatrix}
\]

Note that, when the domain $0 \leq x \leq L$ is divided into $N$ segments, there will be $N + 1$ nodes ranging from $x_0 = 0$ to $x_{N+1} = L$. Because, along the way to a solution, the boundary conditions result in the rows and columns associated with these two nodes being deleted, these matrices will have only $N - 1$ rows and columns. (c) Taking $N = 100$, use an available numeric processing program like IDL, MATLAB, OCTAVE, or PYTHON to find the first several eigenvalues $k_n$ and compare your results with the values found in part (a).

15.9. When $\alpha$, $\beta$, and $f$ are constants, Eqs. (15.67), (15.68), and (15.69) can be solved analytically. Show, for example, that the solution to this boundary value problem is given by

\[ \varphi(x) = A \sin \lambda x + \left( p - \frac{f}{\beta} \right) \cos \lambda x + \frac{f}{\beta} \]

where

\[ A = \frac{(q - \gamma f/\beta) + (p - f/\beta)(\alpha \lambda \sin \lambda L - \gamma \cos \lambda L)}{\alpha \lambda \cos \lambda L + \gamma \sin \lambda L} \]

and $\lambda = \sqrt{-\beta/\alpha}$ when $\beta/\alpha < 0$. Using graphical displays in particular, compare the analytic solution in this exercise with the solution obtained numerically by finite difference and finite element approaches. Optional: Find corresponding solutions when $\beta/\alpha = 0$ and $\beta/\alpha > 0$.

15.10. Suppose a linear element $e$ is characterized by three nodes at $x_1^{(e)}$, $x_2^{(e)}$, and $x_3^{(e)}$. Further, let the solution $\varphi^{(e)}$ on that element be approximated by the quadratic function

\[ \varphi^{(e)} = a^{(e)} + b^{(e)} x + c^{(e)} x^2 \]
(a) Find the constants \(a^{(e)}, b^{(e)},\) and \(c^{(e)}\) that will make this function match the specific values \(\tilde{\varphi}_1^{(e)}, \tilde{\varphi}_2^{(e)},\) and \(\tilde{\varphi}_3^{(e)}\) at the points \(x = x_1^{(e)}, x_2^{(e)},\) and \(x_3^{(e)}\), respectively. (b) Substituting these values into the above expression and grouping terms appropriately, cast the result in the form

\[
\tilde{\varphi}^{(e)} = \sum_{i=1}^{3} \tilde{\varphi}_i^{(e)} N_i^{(e)}(x)
\]

and show that the shape functions \(N_i^{(e)}(x)\) appropriate to this three-noded linear element are

\[
N_1^{(e)}(x) = \frac{1}{\Delta} \begin{vmatrix}
1 & x & x^2 \\
1 & x_2^{(e)} & (x_2^{(e)})^2 \\
1 & x_3^{(e)} & (x_3^{(e)})^2 \\
\end{vmatrix}
\]

\[
N_2^{(e)}(x) = \frac{1}{\Delta} \begin{vmatrix}
1 & x_1^{(e)} & (x_1^{(e)})^2 \\
1 & x & x^2 \\
1 & x_3^{(e)} & (x_3^{(e)})^2 \\
\end{vmatrix}
\]

\[
N_3^{(e)}(x) = \frac{1}{\Delta} \begin{vmatrix}
1 & x_1^{(e)} & (x_1^{(e)})^2 \\
1 & x_2^{(e)} & (x_2^{(e)})^2 \\
1 & x & x^2 \\
\end{vmatrix}
\]

where

\[
\Delta = \begin{vmatrix}
1 & x_1^{(e)} & (x_1^{(e)})^2 \\
1 & x_2^{(e)} & (x_2^{(e)})^2 \\
1 & x_3^{(e)} & (x_3^{(e)})^2 \\
\end{vmatrix}
\]

Finally, (c) show that—with \(\xi = (x - x_1^{(e)}/(x_3^{(e)} - x_1^{(e)})\)—these functions reduce to

\[
N_1^{(e)}(x) = 2(\xi - 1) \left(\xi - \frac{1}{2}\right) ; \quad N_2^{(e)}(x) = 4\xi(1 - \xi) ; \quad N_3^{(e)}(x) = 2\xi \left(\xi - \frac{1}{2}\right)
\]

when \(x_2^{(e)}\) is midway between \(x_1^{(e)}\) and \(x_3^{(e)}\), i.e., when \(x_2^{(e)} = x_1^{(e)} + \frac{1}{2}l^{(e)}\) and \(x_3^{(e)} = x_1^{(e)} + l^{(e)}\), and for \(\xi < x < 1\). \textit{Hint:} You may find a symbolic manipulating program to be useful at many points in this problem.

15.11. \textit{Accepting the shape functions given in part (c) of Exercise 15.10, (a) find the } 3 \times 3 \text{ matrices}

\[
^1K^{(e)}_{ij} = \int_{x_1^{(e)}}^{x_3^{(e)}} \frac{dN_i^{(e)}}{dx} \frac{dN_j^{(e)}}{dx} \, dx \quad \text{and} \quad ^2K^{(e)}_{ij} = \int_{x_1^{(e)}}^{x_3^{(e)}} N_i^{(e)} N_j^{(e)} \, dx
\]

and then construct the \(3 \times 3\) matrix whose elements are

\[
K_{ij}^{(e)} = a^{(e)} \, ^1K_{ij}^{(e)} + b^{(e)} \, ^2K_{ij}^{(e)}
\]

to produce the (elemental) stiffness matrix analogous to Eq. (15.98). (b) Find the three-element vector whose elements are

\[
b_i^{(e)} = f^{(e)} \int_{x_1^{(e)}}^{x_3^{(e)}} N_i^{(e)} \, dx
\]

analogous to Eq. (15.99) and the three element vector whose elements are

\[
g_i^{(e)} = a^{(e)} N_i^{(e)} \left. \frac{d^2}{dx^2} \right|_{x=x^{(e)}} - a^{(e)} N_i^{(e)} \left. \frac{d^2}{dx^2} \right|_{x=A^{(e)}},
\]

analogous to Eq. (15.100). By the time you are done, you should discover that the elemental equation applicable to this three-node element is

\[
\begin{bmatrix}
K_{11}^{(e)} & K_{12}^{(e)} & K_{13}^{(e)} \\
K_{21}^{(e)} & K_{22}^{(e)} & K_{23}^{(e)} \\
K_{31}^{(e)} & K_{32}^{(e)} & K_{33}^{(e)}
\end{bmatrix}
\begin{bmatrix}
\varphi_1^{(e)} \\
\varphi_2^{(e)} \\
\varphi_3^{(e)}
\end{bmatrix}
= \begin{bmatrix}
b_1^{(e)} \\
b_2^{(e)} \\
b_3^{(e)}
\end{bmatrix} + \begin{bmatrix}
g_1^{(e)} \\
g_2^{(e)} \\
g_3^{(e)}
\end{bmatrix}
\]
where
\[ y_1^{(e)} = -\alpha^{(e)} \frac{d\tilde{\varphi}^{(e)}}{dx} \bigg|_{x_1^{(e)}} \quad ; \quad g_1^{(e)} = \alpha^{(e)} \frac{d\tilde{\varphi}^{(e)}}{dx} \bigg|_{x_1^{(e)}}. \]

Except for factors of $\alpha^{(e)}$, $\beta^{(e)}$, $f^{(e)}$, and $g^{(e)}$, you should have explicit values for $K_{ij}^{(e)}$ and $b_{ij}^{(e)}$.

**Hints:** (1) Begin by recasting the integrals as integrals on the variable $\xi$. (2) You may find a symbolic manipulating program to be useful at many points in this problem. (3) Assume as in Exercise 15.10 that the point $x = x_2$ is midway between $x_1$ and $x_3$.

15.12. Continuing with the circumstances of the previous two problems, suppose the region of interest is divided into three three-noded elements, $e = 1, 2, 3$ with global nodes 1, 2, 3, 4, 5, 6, 7, nodes 1, 2, and 3 in element 1, nodes 3, 4, and 5 in element 2, and nodes 5, 6, and 7 in element 3. Suppose node 2 is midway between nodes 1 and 3, node 4 is midway between nodes 3 and 5, and node 6 is midway between nodes 5 and 7, but do not suppose the lengths of the three elements are the same. Following the pattern in Section 15.9.4, assemble the elemental equations for these three elements into an equation for the whole system analogous to Eq. (15.130) if the solution is required to satisfy the boundary conditions of Eqs. (15.68) and (15.69).

15.13. Recast fem1d so it will generate a solution to Eq. (15.67) when

- $\alpha$ is a (positive) constant and $\beta = 0$,
- $f(x)$ varies with position in accordance with $f(x) = Ae^{-\sigma(x-L/2)}$, with $A$ and $\sigma$ constants, and
- the solution is to have the fixed value $\varphi = 0$ at $x = 0$ and the fixed value $\varphi = 100$ at $x = L$.

Then, explore the character of the solutions for various values of $\sigma$. Use two-noded elements of equal size. Note: This solution to this problem corresponds physically to the steady state temperature in a one-dimensional rod whose ends are maintained at the fixed temperatures 0 and 100, respectively, and whose middle is heated with a source that provides a constant energy input.

15.14. Repeat the previous problem but pursue its solution this time by using elements of varying length designed to recognize that, especially if the Gaussian function is sharply peaked in the center, it might be wise to use smaller elements in that region.

15.15. (a) Solve Eq. (15.179) for the constants $a^{(e)}$, $b^{(e)}$, and $c^{(e)}$ and then verify the expressions given in Eqs. (15.182)–(15.184) for the shape functions. Remember that symbolic manipulating programs are available. (b) For each node $i$, verify that the function $N_i^{(e)}(x, y)$ as given by Eqs. (15.182) and (15.184) is zero not only at the two nodes not identified by its index but also along the entire line joining those two nodes. Hint: Set the determinant in the numerator of the expression giving $N_i^{(e)}$ to zero, thereby obtaining the equation of a line in the plane. Verify that that line is, in fact, the line joining the two described nodes. (c) Find the functions to which these functions reduce when $(x_0, y_0) = (-1, 0)$, $(x_1, y_1) = (1, 0)$, $(x_2, y_2) = (0, 1)$ and demonstrate that each is zero where it is supposed to be zero and one where it is supposed to be one.

15.16. Consider a four-node element in two dimensions. Let the nodes be at $(x_0, y_0)$, $(x_1, y_1)$, $(x_2, y_2)$, and $(x_3, y_3)$, and take the interpolating function to be $\tilde{\varphi} = a + bx + cy + dxy$. (a) Find the four shape functions in general terms, showing that those functions can be expressed in the form

\[
N_0(x, y) = \frac{1}{\Delta} \left| \begin{array}{ccc} 1 & x & y \\ x_0 & y_0 & x_0y_0 \\ x_1 & y_1 & x_1y_1 \\ x_2 & y_2 & x_2y_2 \\ x_3 & y_3 & x_3y_3 \end{array} \right| \quad \text{where} \quad \Delta = \left| \begin{array}{ccc} 1 & x_0 & y_0 \\ 1 & x_1 & y_1 \\ 1 & x_2 & y_2 \\ 1 & x_3 & y_3 \end{array} \right|
\]

and in similar forms for $N_1(x, y)$, $N_2(x, y)$, and $N_3(x, y)$. (b) Then, show that, if the nodes lie at the corners of a square of side $s$, i.e., the nodes are—in order—at $(0, 0)$, $(s, 0)$, $(s, s)$, and $(0, s)$, the
shape functions expressed in terms of the variables $\xi = x/s$ and $\eta = y/s$ are

$$
N_0(\xi, \eta) = (1 - \xi)(1 - \eta) \quad ; \quad N_1(\xi, \eta) = \xi(1 - \eta) \quad ; \quad N_2(\xi, \eta) = \xi \eta \quad ; \quad N_3(\xi, \eta) = \eta(1 - \xi)
$$

(c) Finally generate surface graphs of the four functions and verify that the shape function associated with each node is zero everywhere along the two edges that intersect at the diagonally opposite node. 

Hints: (1) Remember that symbol manipulating programs are available. (2) Some results along the way to a solution are very involved. Presentation of the intermediate results can certainly be suppressed.

15.17. Find the solution for steady state temperature when a square as in the text has its lower edge maintained at 0, the temperature on its left edge rises linearly from 0 to 50, that on its upper edge rises linearly from 50 to 100, and its right edge is insulated.

15.18. Proof that the full discretization of the wave equation as described towards the end of Section 15.16.1 leads to an unstable method unless $\alpha \leq 1$ is extremely difficult. (A proof is worked out in pages 16–29 of Finite Difference Methods for Partial Differential Equations, George Forsythe and Wolfgang Wasow (John Wiley and Sons, New York, 1960.).) This exercise asks not for that proof but only that you obtain evidence supporting the existence of that instability by recasting fdwaveid to solve the wave equation subject to the boundary conditions $u(0, t) = u(l, t) = 0$ and the initial conditions $u(x, 0) = 0$, $\partial u(x, 0)/\partial x = 0$. The solution should, of course, be zero at all subsequent times, since we have started the string in its equilibrium position with zero velocity. Now, suppose that a computer roundoff error occurs such that, instead of being zero at all nodes, $u(x, \Delta t) = 0$ everywhere except at one node near the middle of the string and, at that node $u(x, \Delta t)$ mistakenly acquires the value 1 (one). Use your program to solve this problem for choices of the parameters that make $\alpha = 0.5$ and solve it again for other choices that make $\alpha = 1.5$. Look at the solution with print or plot frequency 1 and compare the way in which that one disruption of a value at the end of the first time step propagates forward in time for the two values of $\alpha$.

15.19. Proof that the full discretization of the diffusion equation as described towards the end of Section 15.16.2 leads to an unstable method unless $\gamma \leq 1/2$ is extremely difficult. (A proof is worked out in pages 92–98 of Finite Difference Methods for Partial Differential Equations, George Forsythe and Wolfgang Wasow (John Wiley and Sons, New York, 1960.).) This exercise asks not for that proof but only that you obtain evidence supporting the existence of that instability by recasting fdmdiffus1d to solve the diffusion equation subject to the boundary conditions $u(0, t) = u(l, t) = 0$ and the initial condition $u(x, 0) = 0$, $\partial u(x, 0)/\partial x = 0$. The solution should, of course, be zero at all subsequent times, since we have started the string in its equilibrium position with zero velocity. Now, suppose that a computer roundoff error occurs such that, instead of being zero at all nodes, $u(x, \Delta t) = 0$ everywhere except at one node near the middle of the rod and, at that node $u(x, \Delta t)$ mistakenly acquires the value 1 (one). Use your program to solve this problem for choices of the parameters that make $\gamma = 0.25$ and other choices that make $\gamma = 1.0$. Look at the solution with print or plot frequency 1 and compare the way in which that one disruption of a value at the end of the first time step propagates forward in time for the two values of $\gamma$.

15.20. The inhomogeneous Helmholtz equation in two-dimensional Cartesian coordinates is

$$
\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} + k^2 u = r(x, y)
$$

where $k^2$ is a constant and $r(x, y)$ is the inhomogeneity. Apply finite difference methods to show that

$$
u_{i,j} \approx \frac{u_{i+1,j} + u_{i-1,j} + u_{i,j+1} + u_{i,j-1} - \Delta x^2 r_{i,j}}{4 - k^2 \Delta x^2}
$$

Here, $u_{i,j} = u(x_i, y_j)$, the spacing between consecutive values of $x$ is $\Delta x$, the spacing between consecutive values of $y$ is $\Delta y$ and $\Delta x = \Delta y$. This result could be used in an iterative approach to solving the inhomogeneous Helmholtz equation. Note: If $k^2 = 0$, the equation of this exercise reduces to the inhomogeneous Laplace equation, i.e., to the Poisson equation. If, on the other hand, $r(x, y) = 0$, then this equation reduces to the (homogeneous) Helmholtz equation.

15.21. Starting with $v = \sqrt{RT/m}$, set $T = T_0 + \Delta T$ and expand about $T_0$ and show that for small variations $\Delta T$ about this base value, the speed of sound varies linearly with $\Delta T$. 

15.22. In the text, we used the formula in Eq. (15.168) as the basis for an iterative algorithm for solving Laplace’s equation, i.e., for relaxing the initial guess to a final solution. This formula estimates the next iterate at a particular node as the average of that node’s four nearest neighbors. An alternative approach, known as over-relaxation, includes a contribution from the current value in the node itself, specifically

\[ u_{i,j} = (1 - \alpha) u_{i,j} + \alpha \frac{u_{i+1,j} + u_{i-1,j} + u_{i,j+1} + u_{i,j-1}}{4} \]

where \( \alpha = 1 \) reduces this equation to Eq. (15.168) and \( 1 \leq \alpha < 2 \).\(^{69}\) While the optimal choice of \( \alpha \) depends on the size of the grid used, in general using a value of \( \alpha > 1 \) will speed the convergence of the iterative process to an acceptably accurate solution. Recast \texttt{fdmlap2d} to (1) accept a value of \( \alpha \) as input, (2) implement over-relaxation, (3) monitor the absolute value of the node-by-node change between successive iterates, and (4) print out the maximum value of that change at the end of each iteration. Then, use your program to explore the impact of various values of \( \alpha \) on the rate of convergence for the specific example treated in the text.

15.23. Recast \texttt{fdmlap2d} so that iteration stops when the largest change occurring at any single node from one iterate to the next does not exceed an externally prescribed tolerance. To avoid all possibility of an infinite loop, you should halt iteration either when the tolerance has been reached (or exceeded) or when some prescribed number of iterations has taken place. Code so that your program displays the actual tolerance achieved. Further, if execution terminates because the prescribed tolerance is not achieved, your program should print a message that alerts you to the fact that the prescribed tolerance was not achieved. Test your program with the same example as was used in the text. 

Hints: Before starting an iteration, set a variable, say \( \text{maxch} \), equal to zero. As you calculate a new value for each node in the iteration, store the result in a temporary variable so you can compare that value with the old value it will replace, updating \( \text{maxch} \) to the absolute value of the difference between the new and the old values if and only if that difference exceeds the difference already stored in \( \text{maxch} \). Then, substitute the new value for the old in the array containing the evolving solution and go on to the next node. Once the iteration is completed, \( \text{maxch} \) will contain the absolute value of the largest change at any node during that single iteration. If \( \text{maxch} \) is less than the prespecified convergence criterion, stop the iteration; otherwise conduct one more iteration.

15.24. Consider the problem defined by the two-dimensional Poisson equation

\[ \frac{\partial^2 U}{\partial x^2} + \frac{\partial^2 U}{\partial y^2} = -(1 - x^2)(1 - y^2) \]

to be solved in the square region \( R \) defined by \(-1 \leq x, y \leq 1\) subject to the Dirichlet boundary conditions requiring \( U = 0 \) on the entire boundary of \( R \). Assume all variables are dimensionless. (a) Obtain a surface plot of this inhomogeneity over the \( x, y \) plane. (b) Recasting \texttt{fem2d} to incorporate the given boundary conditions and to address the non-zero inhomogeneity find, explore, and (using surface and contour plots as you deem appropriate) display the solution \( U(x, y) \) in the square region \( R \). Compare the resulting equation with Eq. (15.69) as a guide to interpreting the present problem as a modification of the problem addressed by \texttt{fem2d}. (c) Find the two-dimensional vector field \( \mathbf{V} = -\nabla U = -\nabla U \) in \( R \) and display that field graphically in whatever ways seem appropriate.

15.25. (a) Starting with the Taylor series

\[ \varphi(x + \Delta x) = \varphi(x) + \Delta x \varphi'(x) + O(\Delta x^2) \]

evaluate

\[ \varphi(x + \Delta x) - \varphi(x) \]
to show that

\[ \varphi'(x) = \frac{\varphi(x + \Delta x) - \varphi(x)}{\Delta x} + O(\Delta x) \]

That is, derive the forward difference formula in Eq. (15.74). (b) Deduce the more accurate forward difference formula

\[ \varphi'(x) = -\varphi(x + 2\Delta x) + 4\varphi(x + \Delta x) - 3\varphi(x) \frac{2\Delta x}{2\Delta x} + O(\Delta x^2) \]

\( ^{69}\alpha \geq 2 \) generates an unstable algorithm.
CHAPTER 15. SOLVING PARTIAL DIFFERENTIAL EQUATIONS

**Hint:** Start by using the Taylor expansion

\[ f(x + h) = f(x) + hf'(x) + \frac{1}{2} h^2 f''(x) + O(h^3) \]

to expand \( \varphi(x + \Delta x) \) and \( \varphi(x + 2\Delta x) \).

15.26. Starting with Eqs. (15.85) and (15.86), derive Eqs. (15.87), (15.88), and (15.89).

15.27. Following the pattern illustrated in Section 15.16.1, develop equations corresponding to Eqs. (15.145)-(15.148) to solve the wave equation

\[ \frac{1}{c^2} \frac{\partial^2 u(x, t)}{\partial t^2} = \frac{\partial^2 u(x, t)}{\partial x^2} \]

subject to the boundary and initial conditions

\[ u(0, t) = 0 \quad ; \quad u(L, t) = 0 \quad ; \quad u(x, 0) = f(x) \quad ; \quad \left. \frac{\partial u(x, t)}{\partial t} \right|_{t=0} = 0 \]

These equations describe a string of length \( L \) that is fixed at both ends and set into motion by drawing it aside to the initial shape given by \( f(x) \) and released from rest. Of course, for consistency, \( f(0) \) and \( f(L) \) must both be zero, but \( f(x) \) is otherwise unconstrained. Then, develop a program analogous to `fdmwave1d` to solve this problem, testing your program with

\[ f(x) = A \sin \frac{n \pi x}{L} \]

for \( n = 1, 2, \) and 3. You should recognize that this exercise asks you to find the lowest three normal modes in the vibrations of this string. You should, of course, expect the results of your program to reveal what you know to be the periodic motion in these three modes of oscillation of the string. Your program should also allow you to confirm that the frequencies of the three modes \( f_1, f_2, \) and \( f_3 \) are related by \( f_2 = 2f_1 \) and \( f_3 = 3f_1 \). Agreement with these expectations provides some level of confidence in the adequacy of your choice of parameters.
15.A  Program(s) for FDM Approach to 1D Problem

15.A.4  Listing of fdm1d.py (PYTHON)

# ***** Command file fdm1d.py *****

# ***** Note that, when this command file has completed executing,
# all variables to which it assigns values—-and in
# particular x and phi—-will be accessible at PYTHON’s main
# command level.

# ***** Request input of necessary parameters.

N = input( 'Enter number of segments (N): ' ); N=int(N)
alpha = input( 'Enter alpha: ' ); alpha=float(alpha)
beta = input( 'Enter beta: ' ); beta=float(beta)
f = input( 'Enter f: ' ); f=float(f)
L = input( 'Enter L: ' ); L=float(L)
p = input( 'Enter p: ' ); p=float(p)
gamma = input( 'Enter gamma: ' ); gamma=float(gamma)
q = input( 'Enter q: ' ); q = float(q)

# ***** Calculate segment size, square of segment size, and
# coordinates of nodes.

dx = L/N; dx2 = dx**2
x = np.linspace(0.0, L, num=N+1)

# ***** Determine vector of inhomogenieties and coefficient matrix.

inhomo = np.zeros( N+1 ) + 1.0
inhomo = f*dx2*inhomo
inhomo[0] = p
inhomo[N] = inhomo[N] + 2.0*q*dx
cf = np.zeros( [N+1,N+1] )
for i in range(N+1):
    cf[i,i] = 2.0*alpha + beta*dx2
cf[0,0] = 1.0
cf[N,N] = cf[N,N] + 2.0*gamma*dx
for i in range(N):
    cf[i,i+1] = -alpha
cf[0,1] = 0.0
for i in range(N):
    cf[i+1,i] = -alpha
cf[N,N-1] = -2.0*alpha

# ***** Solve system using np.linalg.solve.

phi = np.linalg.solve( cf, inhomo )
15.B Evaluating Integrals for 1D FEM Problem

15.B.1 ... using MAXIMA

An appropriate batch file to use MAXIMA to evaluate the important integrals in Section 15.1.7 is named FEM1DCalcs.mac, can be copied from the directory $HEAD/maxima, and invoked within MAXIMA with the statement batch( "FEM1DCalcs.mac" ) issued at MAXIMA’s prompt for commands. The batch file is listed (without output) in the remainder of this subsection.

```maxima
/* FEM1DCalcs.mac */
/* This batch file evaluates the several integrals that appear in */
/* setting up coding to solve the 1D FEM problem in CPSUP. */
/* Create interpolation functions and their derivatives */
n : [ (x2 - x)/l, (x - x1)/l ]$
dnx : diff( n, x )$
/* Create integrand for K, then integrate */
K1 : transpose(dnx) . dnx$
K2 : transpose(n) . n$
K : alpha*K1 + beta*K2$
K : integrate( K, x, x1, x2 )$
K : expand( subst( x1+l, x2, K) );
/* Evaluate b */
b : integrate( f*n, x, x1, x2 )$
b : expand( subst( x1+l, x2, b ) );
```

15.C Program(s) for FEM Approach to 1D Problem

15.C.4 Listing of fem1d.py (PYTHON)

```python
# ***** Command file fem1d.py *****

# ******** Note that, when this command file has completed executing,
# all variables to which it assigns values---and in
# particular x and phi---will be accessible at PYTHON’s main
# command level.

# BLOCK 1 Request input of necessary parameters and assure that each
# is stored with the proper data type. Add calculation of the
# length of the rod and values of x along the rod.

M = input( 'Enter number of segments (M): ' ); M = int(M)
alpha = input( 'Enter alpha: ' ); alpha = float(alpha)
beta = input( 'Enter beta: ' ); beta = float(beta)
f = input( 'Enter f: ' ); f = float(f)
l = input( 'Enter l: ' ); l = float(l)
```
15.C. PROGRAM(S) FOR FEM APPROACH TO 1D PROBLEM

```python
p = input('Enter p: '); p = float(p)
gamma = input('Enter gamma: '); gamma = float(gamma)
q = input('Enter q: '); q = float(q)
length = l*M
x = np.linspace(0.0, l*M, M+1)

# BLOCK 2 Determine coefficient (stiffness) matrix before incorporation of boundary conditions.
K = np.zeros([M+1,M+1])  # Create (M+1)x(M+1) array of zeros
S = alpha/l + beta*l/3.0  # Evaluate common quantities
S2 = 2.0*S
T = -alpha/l + beta*l/6.0
K[0,0] = S  # Set diagonal elements of K
for i in range(1,M):
    K[i,i] = S2
K[M,M] = S

for i in range(M):  # Set elements above and below main diagonal of K
    K[i+1,i] = T
    K[i,i+1] = T

# BLOCK 3 Create vector of inhomogeneities.
b = np.zeros(M+1)  # Create M+1 element vector of zeros
U = f*l/2.0  # Evaluate common quantities
U2 = 2.0*U
b[0] = U  # Set elements of b
for i in range(1,M):
    b[i] = U2
b[M] = U

# BLOCK 4 Incorporate mixed boundary conditions.
b[M] = b[M] + q

# BLOCK 5 Incorporate Dirichlet boundary condition
K[0,0] = 1.0
b[0] = p
for j in range(1,M):
    K[j,0] = 0.0
for i in range(1,M):
    b[i] = b[i] - K[0,i]*p
K[0,i] = 0.0

# ***** Solve system using np.linalg.solve
phi = np.linalg.solve(K,b)
```
CHAPTER 15. SOLVING PARTIAL DIFFERENTIAL EQUATIONS

15.D  PYTHON Programs for FDM Approach to 2D Problems

15.D.1  Listing of fdmwave1d.py

```python
# ***** Command file fdmwave1d.py *****
# ***** Note that, when this command file has completed executing,
# all variables to which it assigns values---and in
# particular x and u2---will be accessible at PYTHON's main
# command level.

# ***** Request input of necessary parameters.
N = input( 'Enter number of segments (N): ' ); N = int(N)
dt= input( 'Enter time step (dt): ' ); dt= float(dt)
T = input( 'Enter number of time steps (T): ' ); T = int(T)
c = input( 'Enter speed of propagation (c): ' ); c = float(c)
L = input( 'Enter length of string (L): ' ); L = float(L)
f = input( 'Plot frequency (f): ' ); f = int(f)

# ***** Calculate segment size, coordinates of nodes and
# terminate with message when alpha too large.
dx = L/N; x = np.linspace(0.0,L,num=N+1)
alpha = c**2*dt**2/dx**2
print( alpha )  # Display alpha
if alpha > 1.0:
    print( 'Error: alpha > 1; execution halted' )
    exit()

# ***** Prepare arrays for algorithm
u1 = np.zeros( N+1 )  # For past solution
u2 = np.zeros( N+1 )  # For current solution
u3 = np.zeros( N+1 )  # For next solution

# ***** Set and display initial displacement
b = 8.0*np.pi/L
for i in range( int(3.0*N/8.0), int(5.0*N/8.0) ) :
    u1[i] = 1.0 + np.cos(b*(x[i]-L/2.0))
plt.plot( x, u1, color='black', linewidth=3 )
plt.title( 't = 0.0 s' )
plt.ylim( [-2.0,2.0] )
plt.show()

# ***** Set solution at time dt with initial velocity
# equal to zero; display if requested
```
u2 = u1.copy()
if f==1:
    plt.plot( x, u2, color='black', linewidth=3 )
    plt.title( 't = '+str(dt)+' s' )
    plt.ylim( [-2.0,2.0] )
    plt.show()

# ***** Calculate solution repeatedly, displaying every
#     f-th result
for j in range(2,T+1):
    u3[0] = 0
    for i in range(1,N):
        u3[i] = alpha*u2[i+1] + \
               2.0*(1.0-alpha)*u2[i] + alpha*u2[i-1] - u1[i]
    u3[N] = 2*alpha*u2[N-1]+2*(1.0-alpha)*u2[N] - u1[N]
    u1 = u2.copy(); u2 = u3.copy()
    if f*int(j/f) == j:
        plt.plot( x, u2, color='black', linewidth=3 )
        plt.title( 't = '+str(j*dt)+' s' )
        plt.ylim( [-2.0,2.0] )
        plt.show()

15.D.2 Listing of fdmwave1d_anim.py

# ***** Command file fdmwave1d_anim.py *****

# ***** Note that, when this command file has completed executing,
#     all variables to which it assigns values---and in
#     particular x and u2---will be accessible at PYTHON's main
#     command level.

# ***** Request input of necessary parameters.
N = input( 'Enter number of segments (N):' ); N = int(N)
dt = input( 'Enter time step (dt):' ); dt= float(dt)
T = input( 'Enter number of time steps (T):' ); T = int(T)
c = input( 'Enter speed of propagation (c):' ); c = float(c)
L = input( 'Enter length of string (L):' ); L = float(L)
f = input( 'Plot frequency (f):' ); f = int(f)
delay = input( 'Delay between plots:' ); delay = float(delay)

# ***** Calculate segment size, coordinates of nodes and
#     terminate with message when alpha too large.
dx = L/N; x = np.linspace(0.0,L,num=N+1)
alpha = c**2*dt**2/dx**2
print( alpha ) # Display alpha
if alpha > 1.0:
    print( 'Error: alpha > 1; execution halted' )
    exit()
# ***** Prepare arrays for algorithm
u1 = np.zeros( N+1 )  # For past solution
u2 = np.zeros( N+1 )  # For current solution
u3 = np.zeros( N+1 )  # For next solution

# ***** Set and display initial displacement

fig, ax = plt.subplots()

b = 8.0*np.pi/L
for i in range( int(3.0*N/8.0), int(5.0*N/8.0) ) :
    u1[i] = 1.0 + np.cos(b*(x[i]-L/2.0))
ax.clear()
ax.plot( x, u1, color='black', linewidth=3 )
plt.title( 't = 0.0 s' )
plt.ylim( [-2.0,2.0] )
plt.pause(delay)

# ***** Set solution at time dt with initial velocity
# equal to zero; display if requested

u2 = u1.copy()
if f==1:
    ax.clear()
    ax.plot( x, u2, color='black', linewidth=3 )
    plt.title( 't = '+str(dt)+' s' )
    plt.ylim( [-2.0,2.0] )
    plt.pause(delay)

# ***** Calculate solution repeatedly, displaying every
# f-th result

for j in range(2,T+1):
    u3[0] = 0
    for i in range(1,N):
        u3[i] = alpha*u2[i+1] + 2.0*(1.0-alpha)*u2[i] + alpha*u2[i-1] - u1[i]
    u3[N] = 2*alpha*u2[N-1]+2*(1.0-alpha)*u2[N] - u1[N]
    u1 = u2.copy(); u2 = u3.copy()
    if f*int(j/f) == j:
        ax.clear()
        ax.plot( x, u2, color='black', linewidth=3 )
        time = round(j*dt,1)
        plt.title( 't = '+str(time)+' s' )
        plt.ylim( [-2.0,2.0] )
        plt.pause(delay)

if int(sys.version[0])==2: plt.show()
# ***** Command file fdmdiffus1d.py *****

# ***** Note that, when this command file has completed executing, # all variables to which it assigns values---and in # particular x and u2---will be accessible at PYTHON's main # command level.

# ***** Request input of necessary parameters.

N = input( 'Enter number of segments (N): ' ); N = int(N)
dt = input( 'Enter time step (dt): ' ); dt= float(dt)
T = input( 'Enter number of time steps (T): ' ); T = int(T);
alpha = input( 'Enter value of alpha (alpha): ' ); alpha = float(alpha)
L = input( 'Enter length of rod (L): ' ); L = float(L)
f = input( 'Plot frequency (f): ' ); f = int(f)

# ***** Calculate segment size, coordinates of nodes and # terminate with message when gamma too large.

dx = L/N; x = np.linspace(0.0, L, num=N+1)
gamma = alpha**2*dt/dx**2
print( ' gamma = '+str(gamma) )
if gamma > 0.5:
    print( 'Error: gamma > 0.5; execution halted' )
    exit()

# ***** Prepare arrays for algorithm; set initial conditions

u1 = np.zeros( N+1 ) # For current solution
u2 = np.zeros( N+1 ) # For next solution
b = 8.0*np.pi/L
for i in range( int(3.0*N/8.0), int(5.0*N/8.0) ):
    u1[i] = 1.0 + np.cos( b*(x[i]-L/2.0) )

# Display initial temperature distribution

plt.plot( x, u1, linewidth=3, color='black' )
plt.ylim( [0.0,2.0] )
plt.title( 't = 0.0 s' )
plt.show()

# ***** Calculate solution repeatedly, displaying every # f-th result

for j in range(1,T+1):
    u2[1] = 0.0
    for i in range(1,N):
        u2[i] = gamma*u1[i-1] + (1.0-2.0*gamma)*u1[i] + gamma*u1[i+1]
    u2[N] = 2*gamma*u1[N-1] + (1-2*gamma)*u1[N]
    u1 = u2.copy()
    if f*int(j/f) == j:
        plt.plot( x, u1, linewidth=3, color='black' )
15.D.4  Listing of fdmdiffus1d_anim.py

# ***** Command file fdmdiffus1d_anim.py *****

# ***** Note that, when this command file has completed executing,
# all variables to which it assigns values---and in
# particular x and u2---will be accessible at PYTHON’s main
# command level.

# ***** Request input of necessary parameters.

N = input( 'Enter number of segments (N): ' ); N = int(N)
dt = input( 'Enter time step (dt): ' ); dt= float(dt)
T = input( 'Enter number of time steps (T): ' ); T = int(T);
alpha = input( 'Enter value of alpha (alpha): ' ); alpha = float(alpha)
L = input( 'Enter length of rod (L): ' ); L = float(L)
f = input( 'Plot frequency (f): ' ); f = int(f)
delay = input( 'Delay between plots: ' ); delay = float(delay)

# ***** Calculate segment size, coordinates of nodes and
# terminate with message when gamma too large.

dx = L/N; x = np.linspace(0.0, L, num=N+1)
gamma = alpha**2*dt/dx**2
print( ' gamma = '+str(gamma) )
if gamma > 0.5:
    print( 'Error: gamma > 0.5; execution halted' )
    exit()

# ***** Prepare arrays for algorithm; set initial conditions

u1 = np.zeros( N+1 )  # For current solution
u2 = np.zeros( N+1 )  # For next solution
b = 8.0*np.pi/L
for i in range( int(3.0*N/8.0), int(5.0*N/8.0) ) :
    u1[i] = 1.0 + np.cos( b*(x[i]-L/2.0) )

# Display initial temperature distribution

fig, ax = plt.subplots()
ax.clear()
ax.plot( x, u1, linewidth=3, color='black' )
plt.ylim( [0.0,2.0] )
plt.grid(color='black')
plt.title( 't = 0.0 s' )
plt.pause( delay )
# ***** Calculate solution repeatedly, displaying every
# f-th result

for j in range(1,T+1):
    u2[1] = 0.0
    for i in range(1,N):
        u2[i] = gamma*u1[i-1] + (1.0-2.0*gamma)*u1[i] + gamma*u1[i+1]
    u2[N] = 2*gamma*u1[N-1] + (1-2*gamma)*u1[N]
    u1 = u2.copy()
    if f*int(j/f) == j:
        ax.clear()
        ax.plot( x, u1, linewidth=3, color='black' )
        plt.ylim( [0.0,2.0] )
        time=round(j*dt,1)
        plt.grid(color='black')
        plt.title( 't = '+str(time)+' s' )
        plt.pause( delay )

if int(sys.version[0])==2: plt.show()

15.D.5 Listing of fdmlap2d.py

# ***** Command file fdmlap2d.py *****

# ***** Acquire controlling parameters *****

N = input( 'Enter number of segments (N): ' ); N = int(N)
L = input( 'Enter length of side(L): ' ); L = float(L)
maxits = input( 'Maximum number of iterations (maxits): ' )
    maxits = int( maxits )
f = input( 'Display frequency (f): ' )
    f=int(f)

# ***** Calculate grid spacing, values of x and y at grid points *****

dx = L/N; x = np.linspace(0.0, L, N+1); y = x.copy()

# ***** Create and initialize array for solution and
# Display solution on the screen *****

u = np.zeros( (N+1, N+1) )
u[0,:] = 100.0
u[:,0] = 100.0 - 100.0*x/L
print( '\nIteration # 0' ); print( np.around(u,3) )

# ***** Iterate the specified number of times, displaying
# every f-th iterate on the screen *****

for itcnt in range(1, maxits+1):
    for i in range(1,N):
        for j in range(1,N):
            u[i,j] = 0.25*(u[i+1,j] + u[i-1,j] + u[i,j+1] + u[i,j-1])
    for i in range(1,N):
15.D.6 Listing of fdmlap2d_plot.py

# ***** Command file fdmlap2d_plot.py *****

# ***** Execute
#   import numpy as np
#   import matplotlib.pyplot as plt
#   from mpl_toolkits.mplot3d import Axes3D
# ***** before running this file *****

# ***** Acquire controlling parameters *****
N = input( 'Enter number of segments (N): ' ); N = int(N)
L = input( 'Enter length of side(L): ' ); L = float(L)
maxits = input( 'Maximum number of iterations (maxits): ' )
maxits = int( maxits )
f = input( 'Display frequency (f): ' ); f=int(f)

# ***** Calculate grid spacing, values of x and y at grid points
# and create 2D grids for plotting *****
dx = L/N; x = np.linspace(0.0, L, N+1); y = x.copy()
xx, yy = np.meshgrid( x, y )

# ***** Create and initialize array for solution and
# Display solution on the screen *****
u = np.zeros( (N+1, N+1) )
u[0,:] = 100.0
u[:,0] = 100.0 - 100.0*x/L
u1 = np.flip(u, axis=0)
ax1 = plt.axes(projection='3d')
ax1.plot_surface( xx, yy, u1, color='white', shade=False, edgecolor='black')
ax1.set_xlabel('x'); ax1.set_ylabel('y'); ax1.set_zlabel('u')
ax1.view_init(azim=-45, elev=25)
ax1.set_title( 'Iteration 0' )
plt.show()

# ***** Iterate the specified number of times, displaying
# every f-th iterate on the screen *****
for itcnt in range(1, maxits+1):
    for i in range(1,N):
        for j in range(1,N):
            u[i,j] = 0.25*(u[i+1,j] + u[i-1,j] + u[i,j+1] + u[i,j-1])
    for i in range(1,N):
        u[i,N] = 0.25*(2.0*u[i,N-1] + u[i+1,N] + u[i-1,N])
    if f*int(itcnt/f) == itcnt:
        print( 'nIteration # ' + str(itcnt) ); print(np.around(u,3))
15.D.7 Listing of fdmlap2d_anim.py

# ***** Command file fdmlap2d_anim.py *****

# ***** Execute
# import numpy as np
# import matplotlib.pyplot as plt
# from mpl_toolkits.mplot3d import Axes3D
# ***** before running this file *****

# ***** Acquire controlling parameters *****

N = input( 'Enter number of segments (N): ' ); N = int(N)
L = input( 'Enter length of side(L): ' ); L = float(L)
maxits = input( 'Maximum number of iterations (maxits): ' )
maxits = int( maxits )
f = input( 'Display frequency (f): ' ); f=int(f)
delay = input( 'Delay between plots: ' ); delay = float(delay)

# ***** Calculate grid spacing, values of x and y at grid points
# and create 2D grids for plotting *****

dx = L/N; x = np.linspace(0.0, L, N+1); y = x.copy()
xx, yy = np.meshgrid(x, y)

# ***** Create and initialize array for solution and
# Display solution on the screen *****

fig, ax1 = plt.subplots()

u = np.zeros( (N+1, N+1) )
u[0, :] = 100.0
u[:, 0] = 100.0 - 100.0*x/L
u1 = np.flip(u, axis=0)
plt.clf()
ax1 = plt.axes(projection='3d')
ax1.plot_surface(xx, yy, u1, color='white', shade=False, edgecolor='black')
ax1.set_xlabel('x'); ax1.set_ylabel('y'); ax1.set_zlabel('u')
ax1.view_init(azim=-45, elev=25)
ax1.set_title( 'Iteration 0' )
plt.pause(delay)

# ***** Iterate the specified number of times, displaying
# every f-th iterate on the screen *****

```python
for itcnt in range(1, maxits+1):
    for i in range(1,N):
        for j in range(1,N):
            u[i,j] = 0.25*(u[i+1,j] + u[i-1,j] + u[i,j+1] + u[i,j-1])
    for i in range(1,N):
        u[i,N] = 0.25*(2.0*u[i,N-1] + u[i+1,N] + u[i-1,N])
    if f*int(itcnt/f) == itcnt:
        u1 = np.flip(u, axis=0)
        plt.clf()
        ax1 = plt.axes(projection='3d')
        ax1.plot_surface( xx, yy, u1, color='white', shade=False, edgecolor='black')
        ax1.set_xlabel('x'); ax1.set_ylabel('y'); ax1.set_zlabel('u')
        ax1.view_init(azim=-45, elev=25)
        ax1.set_title( 'Iteration '+str(itcnt) )
        plt.pause(delay)
```

```python
if int(sys.version[0])==2: plt.show()
```

### 15.E A Useful Integral

In this appendix, we deduce a two-dimensional analog to the familiar one-dimensional formula for integration by parts. Consider the integral

\[
I = \int \int_\Omega \psi \left( \frac{\partial V_x}{\partial x} + \frac{\partial V_y}{\partial y} \right) \, dx \, dy
\]  

(15.225)

where \(\psi\), \(V_x\), and \(V_y\) are functions of \(x\) and \(y\), and \(\Omega\) is the region in the \(xy\) plane over which the 2D integral is to be evaluated. Recast the integral in the form

\[
I = \int \int_\Omega \left( \frac{\partial (\psi V_x)}{\partial x} + \frac{\partial (\psi V_y)}{\partial y} \right) \, dx \, dy - \int \int_\Omega \left( V_x \frac{\partial \psi}{\partial x} + V_y \frac{\partial \psi}{\partial y} \right) \, dx \, dy
\]  

(15.226)

Now, introduce the vector \(Q\) whose \(x\) and \(y\) components are \(-\psi V_y\) and \(\psi V_x\), respectively. Since

\[
\frac{\partial (\psi V_x)}{\partial x} + \frac{\partial (\psi V_y)}{\partial y} = \frac{\partial Q_y}{\partial x} - \frac{\partial Q_x}{\partial y} = (\nabla \times Q)_z = (\nabla \times Q) \cdot \hat{k}
\]  

(15.227)

the integral of concern then can be written in the form

\[
I = \int \int_\Omega (\nabla \times Q) \cdot \hat{k} \, dx \, dy - \int \int_\Omega \left( V_x \frac{\partial \psi}{\partial x} + V_y \frac{\partial \psi}{\partial y} \right) \, dx \, dy
\]  

(15.228)

which, upon invoking Stokes’ theorem, we can rewrite further in the form

\[
I = \oint_\Gamma Q \cdot \hat{t} \, dl - \int \int_\Omega \left( V_x \frac{\partial \psi}{\partial x} + V_y \frac{\partial \psi}{\partial y} \right) \, dx \, dy
\]  

(15.229)

where \(\Gamma\) is the path in the \(xy\) plane bounding the region \(\Omega\) and \(\hat{t}\) is a unit vector tangent to that path and pointing in the direction of the thumb of the right hand when the fingers grasp the path while piercing the region \(\Omega\) in the \(z\) direction. The one-dimensional integral in this result is the analog of the unintegrated term in the more familiar one-dimensional formula for integration by parts.
That one dimensional integral involves the component of the vector \( \mathbf{Q} \) tangent to the path \( \Gamma \). We are actually better served by casting things in terms of components normal to that curve. Thus, we introduce the unit vector \( \mathbf{n} \) normal to the curve and pointing outward from the perspective of a viewer in the region \( \Omega \). Since \( \mathbf{t} = \mathbf{k} \times \mathbf{n} \), we find that

\[
\mathbf{t} = \mathbf{k} \times \mathbf{n} = \mathbf{k} \times (n_x \mathbf{i} + n_y \mathbf{j}) = -n_y \mathbf{i} + n_x \mathbf{j}
\]  

(15.230)

Thus,

\[
\mathbf{Q} \cdot \mathbf{t} = n_x Q_y - n_y Q_x = n_x \psi V_x + n_y \psi V_y = (\psi V_x \mathbf{i} + \psi V_y \mathbf{j}) \cdot \mathbf{n} = \psi (V_x \mathbf{i} + V_y \mathbf{j}) \cdot \mathbf{n}
\]  

(15.231)

In other words, the tangential component of \( \mathbf{Q} \) can be recast as the normal component of the vector from which \( \mathbf{Q} \) was originally derived. Thus, we find that the original integral in Eq. (15.225) can alternatively be evaluated as

\[
I = -\int \int_\Omega \left( V_x \frac{\partial \psi}{\partial x} + V_y \frac{\partial \psi}{\partial y} \right) \, dx \, dy + \oint_\Gamma (\psi V_x \mathbf{i} + \psi V_y \mathbf{j}) \cdot \mathbf{n} \, dl
\]  

(15.232)

### 15.F Evaluating Integrals for 2D FEM Problem

A symbol manipulating program can usefully be invoked to evaluate the several integrals that appear in the solution of the example problem introduced in Section 15.23.

#### 15.F.1 ... using MAXIMA

An appropriate batch file to use MAXIMA to evaluate the necessary integrals when array indices start at 0 is named \texttt{FEM2DCalcs0.mac}, can be copied from the directory \texttt{\$HEAD/maxima} and, with the directory containing this file in MAXIMA’s search path,\(^{70}\) invoked within MAXIMA with the statement \texttt{batch( "FEM2DCalcs0.mac" )} issued at MAXIMA’s prompt for commands. This batch file includes the statements

\[
/* FEM2DCalcs0.mac */
/* This batch file evaluates several integrals that appear in */
/* setting up coding to solve the 2D FEM problem in CPSUP */
/* for programs whose array indices start at 0. */
/* Create interpolation functions and their derivatives */

\[
r : [1, x, y]$
\]

/* Create rows */

\[
r0 : [1,x0,y0]$
\]

\[
r1 : [1,x1,y1]$
\]

\[
r2 : [1,x2,y2]$
\]

\[
delta : \text{determinant( matrix( r0,r1,r2 ) )}$  
/* Evaluate denominator */
\]

\[
n0 : \text{determinant( matrix( r,r1,r2 ) )}/\delta$
\]

/* Evaluate functions */

\[
n1 : \text{determinant( matrix( r0,r,r2 ) )}/\delta$
\]

\[
n2 : \text{determinant( matrix( r0,r1,r ) )}/\delta$
\]

\[
n : [n0, n1, n2]$
\]

\(^{70}\)Alternatively, the full path to the file can be included within the quotation marks in the indicated command.
/ * Simplify functions and evaluate derivatives */
/* for odd-numbered elements */

(x1 : x0 + dx, x2 : x0 + dx, y1 : y0, y2 : y0 + dx )$
nodd : expand( ev(n) )$
dnodd : diff( nodd, x )$
dnodd : diff( nodd, y )$

/* Simplify functions and evaluate derivatives */
/* for even-numbered elements */

(x1 : x0 + dx, x2 : x0, y1 : y0 + dx, y2 : y0 + dx )$
neven : expand( ev(n) )$
dneven : diff( neven, x )$
dneven : diff( neven, y )$

remvalue( x1, x2, y1, y2 )$

/* Create integrand for Kodd, then integrate */

Kodd : ax*transpose(dnodd).dnodd + ay*transpose(dnodd).dnodd +
b*transpose(nodd).nodd$
Kodd : integrate( integrate(Kodd, y, y0, y0+x-x0 ), x, x0, x0+dx )$
Kodd : expand( Kodd )$

/* Create integrand for Keven, then integrate */

Keven : ax*transpose(dneven).dneven + ay*transpose(dneven).dneven +
b*transpose(neven).neven$
Keven : integrate( integrate(Keven, y, y0+x-x0, y0+dx ), x, x0, x0+dx )$
Keven : expand( Keven )$

/* Evaluate bodd and beven */

bodd : integrate( integrate( nodd*f, y, y0, y0+dx-x0 ), x, x0, x0+dx )$
bodd : expand( bodd )$

beven : integrate( integrate( neven*f, y, y0+dx-x0, y0+dx ), x, x0, x0+dx )$
beven : expand( beven )$

/* Evaluate needed g’s */

g : integrate( nodd*q, y, y0, y0+dx )$
g : expand( subst( x0+dx, x, g ) )$

An appropriate batch file to use MAXIMA to evaluate the necessary integrals when array indices
start at 1 is named FEM2DCalcs1.mac, can be copied from the directory $HEAD/maxima and, with
the directory containing this file in MAXIMA’s search path,71 invoked within MAXIMA with the
statement batch( "FEM2DCalcs1.mac" ) issued at MAXIMA’s prompt for commands. This batch
file includes the statements

/* FEM2DCalcs1.mac */

71 Alternatively, the full path to the file can be included within the quotation marks in the indicated command.
/* This batch file evaluates several integrals that appear in */
/* setting up coding to solve the 2D FEM problem in CPSUP */
/* for programs whose array indices start at 1. */

/* Create interpolation functions and their derivatives */

r : [1, x, y]$ /* Create rows */
r1 : [1,x1,y1]$
rm2 : [1,x2,y2]$
r3 : [1,x3,y3]$

delta : determinant( matrix( r1,r2,r3 ) )$ /* Evaluate denominator */

n1 : determinant( matrix( r,r2,r3 ) )/delta$ /* Evaluate functions */
n2 : determinant( matrix( r1,r,r3 ) )/delta$
n3 : determinant( matrix( r1,r2,r ) )/delta$
n : [ n1, n2, n3 ]$

/* Simplify functions and evaluate derivatives */
/* for even-numbered elements */
(x2 : x1 + dx, x3 : x1 + dx, y2 : y1, y3 : y1 + dx )$
neven : expand( ev(n) )$
dnevenx : diff( neven, x )$
dneveny : diff( neven, y )$

/* Simplify functions and evaluate derivatives */
/* for odd-numbered elements */
(x2 : x1 + dx, x3 : x1, y2 : y1 + dx, y3 : y1 + dx )$
nodd : expand( ev(n) )$
dnoddx : diff( nodd, x )$
dnoddy : diff( nodd, y )$

remvalue( x1, x2, y1, y2 )$

/* Create integrand for Keven, then integrate */
Keven : ax*transpose(dnevenx).dnevenx + ay*transpose(dneveny).dneveny +
b*transpose(neven).neven$
Keven : integrate( integrate(Keven, y, y1, y1+x-x1 ), x, x1, x1+dx )$
Keven : expand( Keven );

/* Create integrand for Kodd, then integrate */
Kodd : ax*transpose(dnoddx).dnoddx + ay*transpose(dnoddy).dnoddy +
b*transpose(nodd).nodd$
Kodd : integrate( integrate(Kodd, y, y1+x-x1, y1+dx ), x, x1, x1+dx )$
Kodd : expand( Kodd );

/* Evaluate bodd and beven */
beven : integrate( integrate( neven*f, y, y1, y1+x-x1 ), x, x1, x1+dx )$
beven : expand( beven );

bodd : integrate( integrate( nodd*f, y, y1+x-x1, y1+dx ), x, x1, x1+dx )$
      bodd : expand( bodd );
/* Evaluate needed g's */

    g : integrate( neven*q, y, y1, y1+dx )$
       g : expand( subst( x1+dx, x, g ) );
15.G  Program(s) for FEM Approach to 2D Problems

15.G.4  Listing of fem2d.py (PYTHON)

# ***** Command file fem2d.py *****

# ***** Note that, when this command file has completed executing,
# all variables to which it assigns values---and in
# particular x and phi---will be accessible at IDL's main
# command level.

# Import needed modules

import numpy as np

# ***** Request input of necessary parameters, assure that each is
# stored with the proper data type, and calculate the number
# of elements, the number of nodes, and the segment size.

L    = input( 'Enter length of side (L):' ); L = float(L);  
d    = input( 'Enter number of segments (d):' ); d = int(d);  
alpha_x = input( 'Enter alpha_x: ' ); alpha_x=float(alpha_x);  
alpha_y = input( 'Enter alpha_y: ' ); alpha_y=float(alpha_y);  
beta   = input( 'Enter beta: ' ); beta=float(beta);  
f    = input( 'Enter f: ' ); f=float(f);  
p1    = input( 'Enter value for top edge (p1): ' ); p1=float(p1)  
p2    = input( 'Enter value for bottom edge (p2): ' ); p2=float(p2)  
q    = input( 'Enter q: ' ); q=float(q)

M = 2*d**2          # Calculate number of elements
N = (d+1)**2        # Calculate number of nodes
dx = L/d            # Calculate segment size

# Determine x and y coordinates of nodes.

x = np.zeros(N)       # Create array to store x values
y = np.zeros(N)       # Create array to store y values
cr = -1               # Initialize a counter variable
for i in range(d+1):  # Start row number loop
    for j in range(d+1):  # Start column number loop
        ct=ct+1           # Increment counter
        x[ct] = i*dx       # Find x coordinate
        y[ct] = L - j*dx   # Find y coordinate

# Create connectivity matrix cm

cm = np.zeros([3,M], dtype=int)   # Create 3 by M null array
for e in range(M):                # Loop through all elements
    vl = int(e/(2*d))            # Find index of vertical line
    cm[0,e] = int(e/2) + 1 + vl  # Find global number of node 0
    if 2*int(e/2) == e:          # If e is even
        cm[1,e] = cm[0,e] + d    # find global number of node 1
cm[2,e] = cm[0,e] - 1  # find global number of node 2
else:  # If e is odd
    cm[1,e] = cm[0,e] + d + 1  # find global number of node 1
    cm[2,e] = cm[0,e] + d  # find global number of node 2

# Construct the stiffness matrix K
Kodd = np.zeros([3,3])  # Create two 3 by 3 arrays
Keven = np.zeros([3,3])  # Evaluate a common quantity
bx = beta*dx**2
Kodd[0,0] = (bx + 6*alpha_x)/12  # Assign the appropriate value to
Kodd[1,1] = (bx + 6*alpha_x + 6*alpha_y)/12  # each K(i,j). Note that the array
Kodd[2,2] = (bx + 6*alpha_y)/12  # are symmetric, and that Keven
Kodd[0,1] = (bx - 12*alpha_x)/24  # includes all of the same values as
Kodd[1,0] = Kodd[0,1]  # Kodd, but in different locations.
Kodd[1,2] = (bx - 12*alpha_y)/24
Kodd[2,1] = Kodd[1,2]
Kodd[0,2] = bx/24
Kodd[2,0] = bx/24

Keven[0,0] = Kodd[2,2]
Keven[1,1] = Kodd[0,0]
Keven[2,2] = Kodd[1,1]
Keven[0,1] = Kodd[0,2]
Keven[1,0] = Keven[0,1]
Keven[1,2] = Kodd[0,1]
Keven[2,1] = Keven[1,2]
Keven[0,2] = Kodd[1,2]
Keven[2,0] = Keven[0,2]

# Assemble full stiffness matrix
K = np.zeros([N,N]);  # Create arrays to store values of
b = np.zeros([N,1]);  # K(i,j) and b(i)
for e in range(M):
    for i in range(3):
        for j in range(3):
            if (2*int(e/2) != e):
                K[cm[i,e],cm[j,e]] = K[cm[i,e],cm[j,e]] + Kodd[i,j]
            else:
                K[cm[i,e],cm[j,e]] = K[cm[i,e],cm[j,e]] + Keven[i,j]

# Impose boundary conditions
for i in range(d+1):
    u = i  # Nodes on the left boundary
    s = i*(d+1)  # Nodes on the top boundary
    t = (i+1)*(d+1) - 1  # Nodes on the bottom boundary
    p3 = (p1-p2)/L * y[u] + p2  # Find values of phi on left boundary
    for j in range(N):
K[j,s] = 0  # Set rows in K to zero where value
K[j,t] = 0  # of phi is known
K[j,u] = 0
b[s] = p1  # Set values of b
b[t] = p2
b[u] = p3
if (j != s):  # Reflect influence
    b[j] = b[j] - K[s,j]*p1  # of known values of
K[s,j] = 0  # equations
if (j != t):
    b[j] = b[j] - K[t,j]*p2
K[t,j] = 0
if (j != u):
    b[j] = b[j] - K[u,j]*p3
K[u,j] = 0
K[s,s] = 1  # Set the appropriate entry to 1 in
K[t,t] = 1  # the rows where phi is known
K[u,u] = 1

# Calculate g vector and vector of inhomogeneities

g = np.zeros([N,1])  # Create vector to store g values
for i in range(d):
    nd = d*(d+1)+i  # Nodes on right boundary
    g[nd] = q * dx
b = b+g  # Store the vector \{b+g\} in \{b\}

# Solve the system of equations

phi = np.linalg.solve( K, b )  # Solve the equation K*phi = b

# Store solution and node locations in an array

A = np.zeros([d+1,d+1])  # Create a d+1 by d+1 array
cnt=0  # Initialize a counter
for i in range(d+1):  # Use nested for loops to write all
    for j in range(d+1):  # entries in A
        A[i,j] = phi[cnt]
cnt=cnt+1  # Increment counter
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\(^1\) 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\(^2\) 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,\(^3\) 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 w\texttt{y}si\texttt{w}yg,\(^4\) 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 \texttt{User’s Guide and Reference Manual} (which we will refer to as \texttt{The \LaTeX Manual}), the \LaTeX \texttt{Companion}, and/or to any of the several other \TeX and \LaTeX books.\(^5\) The wisdom of taking fifteen minutes every now and then to browse in these publications cannot be overstressed; as you become more skilled, the fine print and other subtleties will gradually be more and more meaningful.\(^6\)

\(^1\)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’.

\(^2\)Here, individuals disagree on the pronunciation of the ‘la’, and Leslie Lamport offers no guidance. Some say lah\TeX while others say lay\TeX, with the emphasis on the first syllable in both cases. We simply must become accustomed to each other’s preferences.

\(^3\)The primary site for information (history, current plans, downloads, ... ) for \TeX, \LaTeX, and numerous other publicly available components of \TeX and its derivatives is the web site of the \TeX Users’ Group (TUG), \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.

\(^4\)what you see is what you get.

\(^5\)See Section A.18 for more detailed references.

\(^6\)You may also find the web pages of TUG at the URL \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.
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 (\texttt{xemacs}, \texttt{notepad}, \texttt{wordpad}, \texttt{winedt}, \texttt{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 \LaTeX2ε and will not work with the previous version (\LaTeX 2.09). \LaTeX2ε, 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} \text{ 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 \LaTeX. Other available classes may be described in the \textit{Local Guide}.

\begin{itemize}
\item \textbf{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 \textbf{report} This class is generally used for longer documents. See Sections 2.2.2 and C.5.1 in \textit{The \LaTeX Manual}.
\item \textbf{book} This class is meant for actual books. See Sections 5.1 and C.5.1 in \textit{The \LaTeX Manual}.
\item \textbf{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 \textbf{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 11pt or 12pt} (default = 10pt) Specifies default type size.
\item \texttt{letterpaper or legalpaper or a4paper or ...} (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 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 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 filename}  
   \hspace{1cm} (Default extension for the input file is .tex.)

   where \texttt{filename} specifies the input file to be processed, or (2) by selecting an item from a menu in a graphical user interface (GUI).\footnote{See the \textit{Local Guide} for the precise command at your site.} However \LaTeX is invoked, it will create several output files, all of which will have the same name as the input file except for the extension: (1) a binary output file, with extension .dvi, which contains the translation of the original file into \TeX's \textit{device} independent language; (2) an ASCII log file, with extension .log, which contains an expanded log of the messages that appear on the screen as \texttt{latex} works its magic on the input file; and (3) an auxiliary ASCII file, with extension .aux, which contains information that is important only if the code makes use of \LaTeX's capabilities for generating internal cross references.\footnote{These capabilities will be described in later sections of this Appendix. For now, we may ignore messages relating to the .aux file, noting only that, when we ultimately do make use of internal references, the \textit{first} pass of the document through \LaTeX writes the .aux file and a \textit{second}—and occasionally a third—pass is necessary to incorporate that information in the final formatting of the pages.} The resulting .dvi file can be directly displayed on the screen (see Section A.1.4).\footnote{Remember that we have limited the present discussion to documents containing no figures. See Section A.7 for the adjustments to be made if PostScript or PDF figures are included.}

2. Second, we must translate the .dvi file into a file that can be understood by the printer on which the document is to be printed. Most often, the .dvi file will be translated into
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\footnote{Additional information about dvips can be found by typing the command 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 dvips, accessed in many systems by typing the statement texdoc dvips at a Shell prompt or, in the UNIX and LINUX operating systems, the statement man dvips at a Shell prompt.}
\begin{verbatim}
dvips -o filename.ps -t letter filename
\end{verbatim}
which translates the entire file into PostScript, stores it in a file, and (with the \texttt{-o} option here used) gives the stored file the same name as the input file but with extension \texttt{.ps} and with the \texttt{-t} option forces use of letter size paper\footnote{If the explicit specification of the output file is omitted, 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.} or (2) by selecting an item from a menu.\footnote{This option is often the default. The option \texttt{-t landscape} will override the default portrait orientation.}

To be more explicit, the file texsample1.tex, contains the \LaTeX{} code in Table A.3. Once this file has been copied to the default directory, we would produce the \texttt{.dvi} and \texttt{.ps} files with statements like
\begin{verbatim}
latex texsample1
dvips -o texsample1.ps -t letter texsample1
\end{verbatim}
or equivalent selections from a menu.

\subsection*{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\footnote{Check the Local Guide for more details. In particular, some sites may have implemented the shorthand command \texttt{dvip filename} for the command dvips \texttt{-o filename.ps filename}.}
\begin{verbatim}
pdflatex filename
\end{verbatim}
or, for the sample above,
\begin{verbatim}
pdflatex texsample1
\end{verbatim}

\footnote{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                  (No default file type)
\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 \texttt{L\LaTeX} on the screen exist. Among the more common are the following:

- (for displaying a `.dvi` file) \texttt{xdvi} for UNIX and LINUX platforms and \texttt{yap} (yet another previewer) for windows platforms. These programs are described more fully in Section A.14.
- (for displaying a PostScript file) \texttt{ghostview}, which is available for almost all platforms.
- (for displaying a PDF file) Adobe \texttt{acroread}, which is available for almost all platforms.

These programs are invoked by double-clicking ML on an icon for the program and then opening the desired file, by double-clicking ML on an icon for the file to be displayed, by executing an appropriate statement as a command in a \textit{Shell} window, or by double-clicking ML on the name of the file in a directory displayed on the screen.\footnote{See the \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.

\footnote{See the \textit{Local Guide} for specifics at your site.}
\footnote{Printers without this capability are rare.}
\footnote{Again, check the \textit{Local Guide} for details on how to print a file.}
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 (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) 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

\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

---

20The point, which is a standard printer’s measure of length, is 1/72” (72 points per inch).

21The 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.

22By 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 \LaTeX 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.
\textheight (default 7.375”) Specifies the vertical dimension of the region of the page occupied by text, excluding the head and the foot of the page. Its value is commonly 9” for an 8.5”×11” page.
\oddsidemargin (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 $\$HEAD/tex$.

lu_article.template Specifies a general article style, including a 6”×9” text area centered on an 8.5”×11” page.
lu_cpl.template Specifies the style of CPL publications.
lu_sloan.template 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.
lu_letter.template Supplies a starting format for business letters.
lu.memo.template Supplies a starting format for memos.
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—L\TeX 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 L\TeX’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 L\TeX, 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>\texttt{\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.

---

23 The default is medium series, Roman family, upright shape—the type style used for this book.

24 Because it is sometimes difficult to remember which “commands” are commands and which are actually declarations, we index both as commands.
Note also in Table A.7 that emphasized text can be achieved not only with the declaration \em but also with the command \textit. 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 \textbackslash inserted immediately after the last italicized or emphasized letter instructs \LaTeX{} to insert the last letter’s italic correction; use of this correction will usually improve the legibility of the final copy. Thus, the careful way to specify an emphasized word is, for example, \{	extit and \textbackslash}. (The italic correction is not necessary in all situations, e.g., if the following character is a period or a comma, which explains why it cannot be automatically and always inserted.)

In using one or another of these simple specifications, we are accepting particular combinations of series, family, and shape. \LaTeX{} provides two ways to take advantage of the full flexibility with separate specification of these three characteristics. For example, we might use declarations to specify bold, sans serif, upright type with the construction

\{	extbfseries\textsf\textup 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

\textbackslash \textmd{...}, \textbf{...}

to specify medium and bold series, respectively,

\textbackslash \textfamily, \textbf{...}

to specify Roman, sans serif, and typewriter families, respectively, and

\textbackslash \textup, \textit, \textsl, \textsc

\textup{...}, \textit{...}, \textsl{...}, \textsc{...}

to specify upright, italic, slanted, and small-cap shapes. A quick return to the default specified in the originally invoked document class is accomplished with the declaration \textbackslash \textnormalfont or the command \textbackslash \textnormal.

\textbf{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., \{	exttt 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</td>
<td>Sets text in tiny type.</td>
<td>Sample</td>
</tr>
<tr>
<td>\scriptsize \text</td>
<td>Sets text in scriptsize type.</td>
<td>Sample</td>
</tr>
<tr>
<td>\footnotesize \text</td>
<td>Sets text in footnotesize type.</td>
<td>Sample</td>
</tr>
<tr>
<td>\small \text</td>
<td>Sets text in small type.</td>
<td>Sample</td>
</tr>
<tr>
<td>\normalsize \text</td>
<td>Sets text in normalsize type.</td>
<td>Sample</td>
</tr>
<tr>
<td>\large \text</td>
<td>Sets text in large type.</td>
<td>Sample</td>
</tr>
<tr>
<td>\Large \text</td>
<td>Sets text in Large type.</td>
<td>Sample</td>
</tr>
<tr>
<td>\LARGE \text</td>
<td>Sets text in LARGE type.</td>
<td>Sample</td>
</tr>
<tr>
<td>\huge \text</td>
<td>Sets text in huge type.</td>
<td>Sample</td>
</tr>
<tr>
<td>\Huge \text</td>
<td>Sets text in Huge type.</td>
<td>Sample</td>
</tr>
</tbody>
</table>

A.3.3 White Space

Some aspects of the way the components (text, tables, graphs, ...) of a document are placed in the available space are controlled by the length parameters set in the preamble and discussed in Section A.2. Additional features of \LaTeX\ that give control over this feature of a document include:

- The command \noindent to suppress paragraph indentation for a single paragraph.
- The commands \newpage and \clearpage to force a new page. The first of these commands terminates the current page and starts a new page. The second also flushes out any accumulated tables and figures that have not yet been output.
- The length parameter \baselineskip to specify the linespacing. This parameter is set with the command \setlength described in Section A.2. The default value is 12 points, which yields single spacing. The value 18 points specifies space and a half, and the value 24 points specifies double spacing. This specification must be placed after the command \begin{document}, and it can be changed whenever appropriate to the text. See, however, Section A.3.4 for a comment about limitations of this approach to setting the linespacing.
- The commands \quad and \qquad, which insert an en space and an em space, respectively. (An en space is about the width of the letter x, and an em space is about the width of the letter M, both in the current font.)
- The command \vspace{Value} to generate vertical space. Here, Value can be any defined length parameter, e.g., \vspace{\parskip}, or a specific length, e.g., \vspace{2.0truein}. If the space happens to occur at the top of a new page, it will be omitted, but the command \vspace*{Value} will force the space to be included even if it is at the top of a page.
- The command \hspace{Value} to generate horizontal space. Here, Value can be any defined length parameter, e.g., \hspace{\parindent}, or a specific length, e.g., \hspace{36.0pt}. If the space happens to occur at the end of a line, it will be omitted, but the command \hspace*{Value} will force the space to be included even if it is at the end of a line.
- The commands \settowidth, \settoheight, and \settodepth, which return lengths for the corresponding dimensions of the box that will enclose whatever is the argument of the
command. No printed output is produced by these commands. We might, for example, define a new length parameter \texttt{dblast} as the width of the box accommodating two asterisks with the two commands

\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 \LaTeX\ Manual}.

\subsection{Environments}

Sometimes, the formatting to be accomplished requires a more sophisticated definition of a change than is possible within the framework of a simple declaration or command with a few arguments. \LaTeX's \textit{environments} are more versatile and flexible than declarations and commands. For example, to present text in a narrower paragraph, as is conventionally done with extended quotations, we might invoke one of the constructions

\begin{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 \hfill Second line \hfill Third line \hfill ...\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{\hfill}, 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} ... \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
individual users can supplement the standard set by defining additional environments suited to the circumstances of the document. Individual exercises in this book, for example, are formatted by a user-defined environment.

The length parameter `\baselineskip` interacts in interesting ways with the formatting of text contained within environments. Changing `\baselineskip` will affect only the spacing in the main text of the document, including text in `itemize`, `enumerate`, and `verbatim` environments. Footnotes, material in `tabular` environments, table and figure captions, and perhaps other components will remain single spaced unless—as described in Section C.3.2 in *The \LaTeX\ Manual*—the parameter `\baselinestretch` is also changed with the command `\renewcommand`. This additional change, however, has interesting side effects. (For example, the spacing of lines within footnotes will be changed but the spacing between separate footnotes on the same page remains single!) Tampering with global style can at times have unintended (and unwanted) consequences. *Beware!*

### A.3.5 \LaTeX{} Packages

As a publishing system, \LaTeX{} is infinitely extensible. Rather than incorporate all manner of special tools within the basic program, its designers provided a means by which additional features could be added through the use of packages, the features of which can be made available in any specific code by placing the command

\begin{verbatim}
\usepackage{ PackageName }
\end{verbatim}

in the preamble *The \LaTeX{} Companion* describes numerous packages, including `amstex`, `babel`, `color`, `graphics`, `graphicx`, `graphpap`, `ifthen`, `latexsym`, `imakeidx`, `verbatim`, `pict2e`, and `showidx`. A brief description of many of these packages is included in Section C.5.2 of *The \LaTeX{} Manual*. The statement `texdoc PackageName` at a `Shell` prompt also will bring up documentation on several of these packages.

In addition, some programs, especially those with notebook capabilities, have the ability to write the contents of the notebooks into \LaTeX{} source files. Usually, those programs make use of their own special \LaTeX{} commands and supply program-specific \LaTeX{} packages which must be accessible, either at the proper point in the \LaTeX{} directory structure or in the directory from which you run \LaTeX{} when processing a \LaTeX{} file produced by the programs. Details on those additional packages will be found in the manuals provided by the vendors of the programs.

### A.3.6 Miscellaneous Other Capabilities

Several additional capabilities merit particular comment:

- To force \LaTeX{} to keep words together on a single line and/or to prevent \LaTeX{} from inserting additional space as it justifies a line, use a tilde `~` instead of a space between the words. The tilde should also be used after a period that does not end a sentence. For example, we should type `Mr.~Cook` rather than `Mr.␣Cook` in our file so as to produce `Mr. Cook` rather than `Mr. Cook` in our output. In the second form, there is more space between the abbreviation and the name because \LaTeX{} automatically inserts extra space after each period (which is assumed to mark the end of a sentence). \LaTeX{} may add even more space in the second case as it justifies the line between the margins.

- To generate a dash, use ``, `--`, or `---`, depending on the length of the required dash. One hyphen produces a hyphen, as in two-column; two hyphens in a row produce a slightly longer

---

The symbol `⊔` underscores the presence of a space at the indicated point; it is *not* a character explicitly present in the code.
dash (an en dash in printers' terminology), conventionally used to indicate ranges, e.g., 10–12; three hyphens in a row produce a still longer line (—, an em dash in printers' terminology), sometimes called a punctuation dash and conventionally used in pairs as an alternative to parentheses. Note that none of these constructions is a minus sign —, which has yet a different length and is produced only in math mode. (See Section A.4.)

- To generate opening and closing (double) quotation marks, use ‘‘ and ’’, i.e., two opening or closing (single) tics in a row, respectively. Do not use the symbol * for either of these punctuation marks.

- To specify a footnote, use the command \footnote{Text of footnote.}. The specified text will be placed at the bottom of the page and keyed to the text with an automatically generated number that starts with 1 at the beginning of the document.

- To start a new section, use the command \section{Title of Section}. In the \texttt{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 \texttt{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 \texttt{LaTeX} is permitted to insert a hyphen as it fills and justifies lines. \texttt{LaTeX} has its own rules for hyphenation, but they are not infallible. Sometimes \texttt{LaTeX} needs help. Note that words containing even one indication of an optional hyphen will not be hyphenated at any other point(s) in the word.

- To suppress hyphenation of a single word altogether, place the word as the argument of an \mbox command, e.g., \mbox{customize}.

## A.4 Including Equations

Most scientific documents will have equations, all of which must be specified in \texttt{LaTeX}'s math mode. Shifting from the default text mode to math mode can be accomplished in several ways. For short, unnumbered equations or mathematical symbols that appear in a line of text, the equation or symbol must be enclosed in one of the two constructions \( \ldots \) or $\ldots$, which are equivalent. If we want one or more \textit{displayed} equations, with or without automatically generated equation numbers, we need to use the form \begin{..}\end. Three standard environments use this construction. The \texttt{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 \texttt{equation} environment centers a single equation on a line by itself and numbers the equation automatically. The \texttt{eqnarray} environment—see \texttt{The \LaTeX\ Manual}—is used for a string of equations and for long equations that will not fit on one line; its syntax allows vertical alignment of two or more displayed and numbered equations. To facilitate references to numbered equations produced in the \texttt{equation} and \texttt{eqnarray} environments, \texttt{LaTeX} provides the command \label{ReferenceName} for defining a symbolic label within the environment and the command \ref{ReferenceName} to permit referencing the equation by its number in the text. Further, the command \pageref{ReferenceName} permits referencing the page on which the equation occurs in the text.\footnote{When these references are used, the information written into the \texttt{.aux} file becomes important and the code must be processed by \texttt{latex} or \texttt{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 \texttt{table} and \texttt{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 \texttt{part}, \texttt{chapter}, \texttt{section}, \texttt{subsection}, \texttt{subsubsection}, and \texttt{footnote} and in the text following an \item command in an \texttt{enumerate} environment to facilitate reference to these components of the document.} (Warning: The \texttt{displaymath}, \texttt{equation}, and \texttt{eqnarray} environments will generate error messages if they contain blank lines.)


\begin{itemize}
  \item This is the first item.
  \item This is the second item.
\end{itemize}

For a complete list of all symbols and the commands that create them, see Section 3.3 on Mathematical Formulas in The \texttt{\LaTeX} Manual.

In math mode, Greek letters can be produced by commands that simply name the letter, e.g., \texttt{\Theta} for $\Theta$ and \texttt{\omega} for $\omega$. Only the initial letter of the name is capitalized to produce an upper-case letter. Note, however, that commands for letters that are identical to Arabic letters, e.g., \texttt{\Kappa}, do not exist.

Technically and officially, the differential element $dx$ should be written with a Roman d and an italic $x$, i.e., $dx$, but this convention is rarely followed.

See Table 3.9 in The \texttt{\LaTeX} Manual for a complete listing.
\begin{...}
\item Text of first item.
\item Text of second item.
\item Text of third item.
\end{...}

The bullets and the numbers are, of course, generated automatically and, within the \texttt{enumerate} environment, the numbers are automatically incremented. Details will be found in Sections 2.2.4, 6.6, C.6.2, and C.6.3 in \textit{The \TeX\ Manual}. Note in particular

- The optional argument for the command \texttt{\textbackslash item}, which provides a means to override the automatic label for that item and replace it with an explicit stipulated label. For example, the command \texttt{\textbackslash item[DMC]} will cause the item to be labeled ‘DMC’ and the command \texttt{\textbackslash item[]} will suppress the label altogether. Note that, if the text of the item itself begins with something enclosed in square brackets, the command introducing that item must be written \texttt{\textbackslash item{}} to prevent the text in square brackets from being interpreted as the desired label.

- The length parameters \texttt{\textbackslash itemsep}, \texttt{\textbackslash parskip}, and \texttt{\textbackslash parsep}, which provide a means to override default spacings within the environment.

- The way to change the “bullets” in the \texttt{itemize} environment. The symbols used to label the “bulleted” items in the various levels of nested \texttt{itemize} environments are created by the commands \texttt{\textbackslash labelitemi}, \texttt{\textbackslash labelitemii}, \texttt{\textbackslash labelitemiii}, and \texttt{\textbackslash labelitemiv}. The default sequence for marking items at each level is •, – *, and ·. Each of these symbols can, however, be changed by redefining the corresponding command. For example, the command

\begin{verbatim}
\renewcommand{\labelitemii}{$\circ$}
\end{verbatim}

executed in the preamble will change the symbol – to ◦ for the second level in nested \texttt{itemize} environments.

- The way to change the form of the labels for each item in the \texttt{enumerate} environment. The labels at the various levels of nested \texttt{enumerate} environments are determined by invoking one of the commands \texttt{\textbackslash theenumi}, \texttt{\textbackslash theenumii}, \texttt{\textbackslash theenumiii}, and \texttt{\textbackslash theenumiv}, whose action is in turn determined from the value of the corresponding one of the counters \texttt{enumi}, \texttt{enumii}, \texttt{enumiii}, and \texttt{enumiv}. The default sequence for marking items at each level is ‘1.’, ‘(a)’, ‘i.’, ‘A.’, i.e., arabic number with period, lower-case letter in parentheses, lower-case Roman number with period, and upper-case letter with period. These defaults, however, can be changed with a command like

\begin{verbatim}
\renewcommand{\theenumi}{(\Alph{enumi})}
\end{verbatim}

which will change the labels used at the first level in \texttt{enumerate} environments to upper-case arabic letters in parentheses. The command \texttt{\Alph} could be replaced with \texttt{\alph}, \texttt{\roman}, \texttt{\Roman}, or \texttt{\arabic} to translate the underlying counter into lower-case arabic letters, lower-case Roman numbers, upper-case Roman numbers, or arabic numbers, respectively. As it turns out, whatever is specified in the argument of the commands translating the counters, a period will be appended by \TeX\ at the first, third, and fourth levels, and enclosing parentheses will be supplied at the second level.

\section{Including Tables}

Creation of columnar arrangements is facilitated by \TeX’s \texttt{tabular} environment, the details of which are involved and are fully described in \textit{The \TeX\ Manual}. The \texttt{tabular} environment can
be invoked anywhere in the code and the resulting table will be placed at the point at which the environment appears, perhaps even in the middle of a line of text. Conventionally, however, tables are placed at the top or bottom of the page,\footnote{Actually, the command \texttt{\begin{table}} has an optional argument (\texttt{\begin{table}[OptArg]}, which can assume any of the values \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.} and \LaTeX{} provides the table environment to facilitate proper placement of a table,\footnote{\LaTeX{} automatically uses the specified caption as an entry for a list of tables. There is, however, a limit to the length of such entries. Especially long captions may generate error messages. To avoid this problem, we can—and sometimes must—exploit an optional argument to the command \texttt{\caption{...}} 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[]{...}}, though we shall here not follow that recommendation. See \textit{The \LaTeX{} Manual} for details.} though the bare table environment contains nothing other than its name that suggests its use for tables. Any text at all can be incorporated in the table environment and will be placed on the page as would a table. Most commonly, however, the table environment will embrace a tabular environment defining the table itself and will use the command \texttt{\caption} to specify the caption of the table and the command \texttt{\label} to specify a symbolic label for use in referring to the table within the document.\footnote{The \texttt{\caption} and \texttt{\label} commands can be placed anywhere within the table environment. Captions will commonly be placed either above or below the captioned component. In some contexts (see the warning at the end of Section A.10), placing the caption above the item is preferable.} Frequently, the tabular environment will itself be embraced in a center environment to control the horizontal placement of the table on the page. Thus, the common structure for the code defining a table would be of the form shown in Table A.9.\footnote{See Sections 3.6 and C.10 in \textit{The \LaTeX{} Manual} for much more detail on ways to line things up in columns. Note particularly (1) the command \texttt{\multicolumn}, which provides for entries that span more than one column, and (2) ways to create horizontal and vertical rulings in the table.}

Here, individual elements in a row are separated from one another by ampersands, and the end of each row—except the last—is marked with the command \texttt{\\}. In this example, the caption, including an automatically generated phrase and number of the form ‘Table 3:’, will appear above the table. Positioning the commands \texttt{\caption} and \texttt{\label} after the center environment would place the caption below the table. Whichever position is adopted, the command \texttt{\label} must appear after the command \texttt{\caption}. Wherever the command \texttt{\label} is positioned, it will place an entry in the .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 tabular environment requires a bit more explanation.
In general, this argument will be a string of c's, l's and r's, one for each column in the table. Each specifies the position (centered, left justified, right justified), respectively, of the entry in the corresponding column. In the example, there are three columns, with entries in the first column centered, entries in the second column left justified, and entries in the third column right justified. The argument is mandatory but will, of course, be a string appropriate to the table being constructed rather than the specific string clr.

By default, tables will be produced with no vertical or horizontal rulings. See Section C.10.2 in The \LaTeX Manual for a description of the means to add these rulings.

A.7 Including Illustrations

Graphics displays frequently appear in technical documents. Within \LaTeX, means to incorporate graphics include:

- A cut and paste method (Section A.7.1).
- A method for incorporating a properly constructed PostScript or PDF file that invokes the command \texttt{\includegraphics} from the package graphicx (Section A.7.2).
- A method that exploits the \texttt{tikz} package of macros that facilitate describing the graphical display and its formatting in the same way that \LaTeX itself facilitates describing the text and its formatting (Section A.7.3).
- The built-in \texttt{picture} environment (Section A.7.4), which has limited capability but is sometimes just the ticket for simple displays.

Most often, commands incorporating figures will be bracketed in a \texttt{figure} environment, which facilitates locating the illustration at the top or bottom of the current (or a following) page. In most cases, we need to know the final vertical extent of the figure in order to specify the size of an appropriate space. Further, the commands \texttt{\caption} and \texttt{\label} are available to caption the figure and to place appropriate entries in the .aux file so that the commands \texttt{\ref} and \texttt{\pageref} will function here as described in the first paragraph of Section A.4.

A.7.1 Using Cut and Paste

The cut and paste method requires only that \LaTeX be instructed to set aside appropriately sized and positioned space into which the illustration will be placed after the document has been printed. The segment of the code that creates and labels space for a figure will have the general form

\begin{verbatim}
\begin{figure}
 \caption{ ... }
 \label{ ... }
 \vspace{ ???.??truein }
\end{figure}
\end{verbatim}

As illustrated here, the caption, including an automatically generated phrase and number of the form ‘Figure 3:’, will appear above the space left for the figure. Positioning the commands \texttt{\caption} and \texttt{\label} after the command \texttt{\vspace} would place the caption below the figure. Note also that,

\footnote{We elect here to discuss only a very few of the numerous available packages, choosing those of the broadest applicability. Packages for drawing Feynman diagrams, organic molecules, musical scores, and many other displays are described in The \LaTeX Graphics Companion.}

\footnote{The \texttt{figure} environment also admits the optional argument described in footnote 31.}
whichever position is adopted, the command `\label` must appear *after* the command `\caption`. For this method, you need only have printed copies of your figures.\footnote{\textit{Appropriately translated, the substance of footnote 34 applies here to the entry placed automatically in the list of 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 `graphicx` package,\footnote{\textit{The graphics package might also be used, but the syntax of the commands it defines is marginally less convenient than the syntax of the parallel commands in the graphicx package, i.e. the \textit{extended graphics} 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{\textit{More generally, this statement admits an optional argument to the desired graphics driver explicitly. Frequently, that driver will be dvips and the statement would be `\usepackage[dvips]{graphicx}`, but numerous other drivers exist, and many may be installed at your site. See the \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 `\usepackage{graphicx}` in our preamble, we incorporate the graphical display itself by placing a statement like

\begin{verbatim}
\includegraphics[height=\textit{dimension}]{\textit{filename}}
\end{verbatim}

at the point where the PostScript or PDF figure defined in by the file \textit{filename} is to be inserted.\footnote{\textit{If you use latex, dvips, and \textit{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 \textit{.ps} or \textit{.eps} files, and—even if the file type is omitted in the `\includegraphics` command—the processing will correctly identify and incorporate the figures. If you seek to produce a PDF document directly with pdflatex as described in item 1 in Section A.1.3, then—even if the file type \textit{.pdf} is omitted in the `\includegraphics` command—the processing will correctly identify and incorporate the figures. If your figures are PostScript, you must use the first route to the final document; if your figures are PDF, you must use the second route. Software to convert PostScript figures to PDF figures and \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 `\includegraphics` command, then you can use either \textit{latex} or \textit{pdflatex} for the processing.}}

In general, the image would be centered horizontally on the page by inserting this command in a `center` environment, e.g.,

\begin{verbatim}
\begin{center}
\includegraphics[height=\textit{dimension}]{\textit{filename}}
\end{center}
\end{verbatim}

For example, the code\footnote{\textit{Appropriately translated, the substance of footnote 34 applies here to the entry in the list of figures.}}
Figure A.1: Mesh surface representation of the irradiance produced by a square aperture.

\begin{figure}
  \caption{Mesh surface representation of the irradiance produced by a square aperture.}
  \label{LATEX:irrad}
  \centering
  \includegraphics[height=3.0truein]{diffract}
  \caption*{Figure A.1 if the file diffract is stored in the current default directory.43,44 Here, the specification of the parameter height in the optional argument of the command includegraphics causes the figure to be scaled to the specified dimension vertically and scaled by the same fraction horizontally to preserve its aspect ratio. As in the example in Section A.7.1, the figure here has been inserted before the command \caption, so that the caption will appear above the figure. Further, the command \label, which must follow the command \caption, defines a symbolic label that can be used within the document to refer to the figure.

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 .dvi file but those produced by the qt toolkit may not then translate with dvips to the subsequent PostScript file. When the latter is the objective, use gnuplot (or maybe fltk). A bit of testing may be necessary.

The command includegraphics admits several optional arguments. The argument width can also be included. If width is specified instead of height, the height will be scaled to preserve

---

43Omission of the file type in the statement including the graph prepares the way to use either latex-dvips to produce a PostScript file using latex and dvips (Section A.1.2), in which case the file diffract.ps or diffract.eps will be used, or pdflatex to produce a PDF file (Section A.1.3), in which case the file diffract.pdf will be used. The appropriate file must, of course, be accessible in either case.

44The 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 \texttt{height} and \texttt{width} are specified, each will be respected and the aspect ratio of the figure may be distorted; specification of \texttt{neither} will cause the display to be produced in its original size. Arguments that allow overriding of the bounding box specified in the PostScript file and arguments that permit clipping a portion of a fuller figure are described in\textit{ The \LaTeX\ Graphics Companion}.

\subsection*{A.7.3 Using the \texttt{tikz} Package}

The \texttt{tikz} package, which is routinely included in present-day \LaTeX\ distributions, provides an assortment of \LaTeX\ macros to facilitate the drawing of simple—and even complicated—figures. Basically, the statement \texttt{\usepackage{tikz}} in the preamble followed in the document itself by statements embedded in a \texttt{tikzpicture} environment will specify the desired graph and incorporate the graph in a PDF or Postscript file created as described in Sections A.1.2 and A.1.3.\footnote{Note that, when displayed on the screen with \texttt{xdvi} or \texttt{yap}—see Section A.1.4—the intermediate \texttt{.dvi} file produced in Section A.1.2 will \textit{not} contain the correct figures.} 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 \textit{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 \texttt{Shell} window to your operating system will probably bring up links to a number of manuals, including the main—and voluminous (1100-plus pages!)—manual in the file \texttt{pgfmanual.pdf}. Further, googling \texttt{tikz} will bring up links to numerous documents, including in particular a link to \texttt{pgfmanual.pdf} and a link to the much more compact tutorial-style introduction in the file \texttt{minimaltikz.pdf}. Once you have selected any of these items, you may in some operating systems have to look in your \texttt{DOWNLOADS} folder to access the item.

This section quickly orients you to the general features of \texttt{tikz} without in any way pretending to be complete. To that end, having included the statement\footnote{Some installations may also require inclusion of the packages \texttt{pgf} and \texttt{xcolor}.}

\texttt{\usepackage{tikz}}

in the preamble of a document, we produce Fig. A.2 by placing the coding in Table A.10 at the appropriate point in the document itself. This coding briefly illustrates some of the more elementary capabilities of \texttt{tikz}. Note specifically the following:

\begin{itemize}
  \item Optional arguments to any \texttt{tikz} command are included in (square) brackets.
  \item By default, specified coordinates are expressed in centimeters, though any recognized \LaTeX\ unit of length can be specifically stipulated. The optional argument \texttt{[x=1.0in,y=1.0in]} following \texttt{\begin{tikzpicture}} changes the default to inches. Once a figure has been defined, its size can be easily altered simply by changing this optional argument.
\end{itemize}
• Every statement must end with a semicolon. Any statement can be spread over several lines with only the last line so terminated.

• The statement \draw [fill=black] (0,0) circle [radius=0.1]; draws a black-filled circle centered at the origin and having radius 0.1 inches, thus marking the origin of the coordinate system.\footnote{Note that the default unit—here inches—for radius can be overridden with an explicit stipulation, e.g., [radius=0.1cm].} Available colors include red, green, blue, cyan, magenta, yellow, and many others. You can also define your own color using a command like\footnote{The command \definecolor here used is defined within the \tikzpicture environment and does not require use of the \LaTeX package \color.}

\\definecolor{ColorName}{rgb}{r,g,b}

where \textit{ColorName} is the name you assign to the color and \textit{r}, \textit{g}, and \textit{b} specify the \textit{r}, \textit{g}, and \textit{b} components that make up the color. Available shapes include \texttt{rectangle}, \texttt{ellipse}, and \texttt{arc}.\footnote{See the manuals for the details of how to specify the dimensions of these shapes.}

• The statement \draw [\texttt{<--}, 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 \texttt{<--} places an arrow at both ends of the line,\footnote{The separate arguments \texttt{->} and \texttt{<} will place an arrow at the end or the beginning of the line, respectively.} and the argument \texttt{line width} species the weight of the line, by default, in points.\footnote{The argument \texttt{line width=2} could be replaced, among others, by \texttt{ultra thin}, \texttt{very thin}, \texttt{thin}, \texttt{thick}, \texttt{very thick}, and \texttt{ultra thick}, and the default unit could be overridden with an argument like \texttt{line width=0.1cm}.}

• Text is placed where specified with the statements \node at (0,1.3) \{\texttt{Large\bf y}\}; and \node at (2.4,0.0) \{\texttt{Large\bf x}\};. Note that \LaTeX stipulations of font size and style are recognized by \texttt{tikz}. Note also that the text is, by default, \textit{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 \texttt{tikz}. Note also the availability of the irrational number \textit{\pi} with the simple coding \texttt{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 \texttt{\draw} command.}

• The construction

\begin{verbatim}
\foreach \x in {0.0,1.0,2.0} {
  \draw (\x,0.1 )--(\x,-0.1);
  \node at (\x, -0.4){\x};;
}\end{verbatim}

creates a loop executing the statements enclosed in \{\ldots\} for each value in the list following the keyword \texttt{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 \texttt{\draw} is a versatile command that admits numerous embellishments through the use of keywords and/or optional arguments. Conveniently, \texttt{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 \texttt{tikz} coding.

When features of the package \texttt{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) {\Large\bf y};
\node at (2.4,0.0) {\Large\bf 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}

Table A.10: Coding to produce Fig. A.2.

\begin{verbatim}
\begin{figure}
\caption{An illustrative \texttt{tikz} figure.}
\label{LATEX:tigzfig}
\begin{center}
\begin{tikzpicture}[x=1.0in,y=1.0in]
\draw [fill=black] (0,0) circle [radius=0.1];
\draw [<->, line width=2] (0,1.2)--(0,0)--(2.3,0);
\node at (0,1.3) {\Large\bf y};
\node at (2.4,0.0) {\Large\bf 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}
\end{verbatim}

\texttt{pdflatex filename} \hspace{1cm} (Default file type \texttt{.tex})

will produce a PDF file and the statements

\texttt{latex filename} \hspace{1cm} (Default file type \texttt{.tex})
\texttt{dvips \text{-O filename.ps \text{-t letter filename}} \hspace{1cm} (Default file type \texttt{.dvi})
\texttt{ps2pdf filename.ps}

will produce a PostScript and then a PDF file, though figures defined by \texttt{tikz} will not be properly
rendered in the intermediate .dvi file. If a table of contents or internal references or both are involved, two—and maybe three—passes through latex or pdflatex will be necessary. If there are PostScript figures and no PDF figures to be incorporated, only latex will work; if there are PDF figures (and no PostScript figures) or hyperlinks to be incorporated, only pdflatex will work. Finally, if there are both PostScript and PDF files defining figures, files of one type will have to be converted to the other type before either of these routes to a finished document will work.\footnote{See Section A.11 for details about that conversion.}

If you wish to short circuit learning detailed tikz code, you might wish to explore a WYSIWYG editor TikzEdt that provides a graphical interface for creating figures and translates that figure into the corresponding tikz code. This program, which is free, can be downloaded from the web site www.tikzedt.org. Documentation is also available at that site.

This section provides a woefully incomplete introduction intended more to wet your appetite than to make you an expert. The effort invested to study the manuals identified in the second paragraph of this section will be richly rewarded.

A.7.4 Using the picture Environment and pict2e Package

Finally, we merely mention the picture environment, described in Sections 7.1 and C.14.1 of The \LaTeX Manual, and the supplementary pict2e package, described in 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 \tableofcontents, \listoffigures, and \listoftables. Each

- results in the output of an auxiliary ASCII file of information with file type .toc, .lof, and .lot, respectively,
- requires a second pass through latex or pdflatex to incorporate the list in the finished document, 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 \part, \chapter, \section, \subsection, and \subsubsection will automatically generate a line in the .toc file;\footnote{Note that not all of these options are available in all document classes. For example, \part and \chapter are available only in the book class. Note also the counters \seccntdepth and \tocdepth that control the depth to which sections are numbered and the book to which sections are catalogued in the table of contents. The default values of these counters can be overridden using \setcounter in the preamble.} each use of the \caption command in a figure or table environment will automatically generate a line in the .lof or .lot file. Then, in the second pass—which will always be necessary—through latex or pdflatex, these files are read and each line results in an entry in the corresponding list.

Two commands allow explicit entry of information into one or another of these files at the point where the command is inserted in the \LaTeX source file, specifically

- the command \addcontentsline{file}{unit}{entry}
- the command \addtocontents{file}{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 \caption and the several sectioning commands (\chapter, \section, ...) which provide control over the entry each command places in the table of contents, list of figures, and list of tables.

A.9 Including an Index

Among many capabilities, \LaTeX\ is able to generate an index, though not automatically. In essence, we must place the commands\footnote{Earlier versions of \LaTeX\ used the package makeidx, which is still available. We here recommend using imakeidx because it makes available a few more features than are available in the previous package and it eliminates the need to run an auxiliary program to format the index.}

\usepackage{imakeidx}
\makeindex

in the preamble and the command

\printindex

at the point in the code at which the index is to be printed—usually the very end. Then, we indicate what items are to be indexed by placing commands like

\index{Item to be indexed}

at appropriate points throughout the document. (The detailed structure of the argument to the command \index is described in Section 4.5 and Appendix A of The \LaTeX\ Manual.) When the file filename.tex is processed by latex or pdflatex, the presence of these commands results in the addition of three auxiliary ASCII files named

- filename.idx containing one line for each index entry,
- filename.ilg containing a log of messages created when, behind the scenes, the idx file is automatically processed through makeindex, and

\footnote{If you are creating a linked document, this entry in the table of contents will also appear in the navigation panel of that document.}
• filename.ind produced by makeindex and containing the formatted index that is then read by \printindex to create the actual index in the document.

With \imakeidx, the index is automatically created and incorporated in the document because an execution of makeindex is inserted automatically at the appropriate point when latex or pdflatex is run. Errors and warnings in that step are compiled in the .ilg file (which should be examined because on-screen display of those glitches may well scroll by too quickly to be comprehended).\footnote{With makeidx, a separate explicit processing of the .idx file with makeindex to produce the .ind file and a further pass through latex or pdflatex was necessary to include the index in the document.}

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

\makeindex [options=-s ind_style, intoc]

stipulates that the style defined by the file ind_style.ist should be invoked and the index should be provided—argument intoc—with an entry in the table of contents. The construction of the style file is fully described in Section 12.4 in The \LaTeX Companion as identified in Section A.18.

To assist in constructing the index, an auxiliary package called showidx, which works both with latex and with pdflatex, can be invoked to print the index entries on each page as marginal notes. This package is invoked simply by including the command

\usepackage{showidx}

in the preamble, though it appears as if this command must be inserted after the \makeindex command described above. Unfortunately, when printing in a 6" × 9" area on 8.5" × 11" paper (letterpaper), the marginal insertions of index entries will extend outside the edges of the page and, in .ps and .pdf files, the portions that are off the page will not be displayed in ghostview or acroread. Conveniently, if the .dvi file is displayed on the screen with xdvi (UNIX) or yap (Windows), those marginal notes will be visible in their entirety, even if they extend beyond the boundaries of the page, though that route is available only if figures are provided in .ps or .eps format. For the purposes of checking the index entries, one workaround is to shrink the area of the page in which printing of the text is allowed. For example, for this document, replacing the commands

\setlength{\textwidth}{6.0truein}
\setlength{\oddsidemargin}{0.5truein}
\setlength{\evensidemargin}{0.0truein}

in the preamble with the commands

\setlength{\textwidth}{4.0truein}
\setlength{\oddsidemargin}{1.0truein}
\setlength{\evensidemargin}{1.0truein}

will shrink the text width and reset the margins so there will be space for the marginal notes. To be sure, the pagination of the text will also change, but at least the marginal notes will be visible, even in the .ps and .pdf files. In different situations, tampering with the above illustrated length parameters and, perhaps also, tampering with the length parameters \marginparwidth (which sets the width of the marginal boxes) and \marginparsep (which sets the space between the text and the boxes) as well as seeking ways to shrink the entire output including marginal notes on each page as a unit may yield fruit.\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 \LaTeX source file can be used to create a PDF document that includes a navigation panel and hyperlinked references for easy viewing with a compatible PDF viewer. Only a few changes need be made to the source files so far described to turn entries in the table of contents, internal references in the body of a document, and page references in an index into hyperlinks to the appropriate points in the document, though these features will appear only in PDF files created via the route described in item 1 in Section A.1.3 above. Specifically,

- the \texttt{\documentclass} statement at the beginning of the file must not specify any particular driver, i.e. should be of the form \texttt{\documentclass\{Class\}}.

- if used at all, the \texttt{\usepackage\{graphicx\}} command in the preamble must not include any options specifying a graphics driver.

- figures must be provided as PDF files or defined using the package \texttt{tikz}. PostScript descriptions of figures will not be properly rendered.

- the command \texttt{\usepackage\{hyperref\}}, perhaps with some optional arguments, must be added in the preamble and, to make sure that other included packages do not overwrite redefinitions made by \texttt{hyperref}, should be the last package included.

This edited source file is then processed by \texttt{pdflatex} (Section A.1.3) at least twice to include hyperlinks, and tables of contents, figures, and tables. With these simple additions, a navigation panel in the PDF file containing chapter, section, subsection, and ... subdivisions of the document will be created. Further, all entries in the tables of contents, list of figures, and list of tables, all internal references created with \texttt{\ref} commands, and all index entries will be converted into hot links within the document.

The command \texttt{\usepackage\{hyperref\}} in the preamble is the minimum necessary to create a hyperlinked PDF file. By default, hot links in the document will be displayed in a red box. To override that default, one can exploit options in the \texttt{\usepackage} command. For example, the statements,

\begin{verbatim}
\usepackage\{color\}
\definecolor\{MyPurple\}\{rgb\}{0.577,0.000,1.000}
\usepackage\{colorlinks=true,\%
linkcolor=\{MyPurple\},bookmarksnumbered=true,linktocpage=true\}\{hyperref\}
\end{verbatim}

in the preamble of the \LaTeX source file will set the stage for automatic creation of hyperlinks for all references, setting the color of those hyperlinks to \texttt{MyPurple} as specified by the RGB values in the \texttt{\definecolor} statement and removing the enclosing box, stipulating that the bookmarks in the navigation panels should include chapter and section numbers, and stipulating that page numbers rather than section titles should be the linked entries in the Table of Contents.\textsuperscript{60} The final PDF file is then produced by processing the source file with \texttt{pdflatex} (Section A.1.3)—at least twice and perhaps three times. To avoid error messages, all auxiliary files hanging over from a previous processing of the source file without hyperlinks through \texttt{latex} or \texttt{pdflatex} should be deleted before processing the file with hyperlinks.\textsuperscript{61}

The files created in the user’s directory when a properly constructed \texttt{.tex} file with hyperlinks, index, and table of contents is processed through \texttt{pdflatex} to produce a linked PDF file are\textsuperscript{62}

\textsuperscript{60}The default value of the \texttt{bookmarksnumbered} and \texttt{linktocpage} options is \texttt{false}.

\textsuperscript{61}See item 25 in Section A.17 for ways to achieve this deletion.

\textsuperscript{62}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}. 
(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 WARNING: Links to figures and tables will point to the line containing the caption for the figure or table. Captions for these items are perhaps better placed above the item rather than below the item.

A CONVENIENCE: If the commands \texttt{\documentclass} and \texttt{\usepackage{graphicx}} are phrased without options and the files in all \texttt{\includegraphics} commands are presented without file type, then the file processed with \texttt{latex} will look for .ps and .eps graphics files and the file processed with \texttt{pdflatex} will look for .pdf files. All files to be sought must, of course, exist.\footnote{See Section A.11 for a means to convert .ps and .eps files to .pdf and vice versa.} This feature makes it easy to process the same source file both with \texttt{latex} and with \texttt{pdflatex}.

### A.11 Converting .eps and .ps Files to .pdf

When files describing figures are created, it is often easier to output those files from the creating program as .eps or .ps files rather than as .pdf files. Unfortunately, if you desire to produce the final output with \texttt{pdflatex}, files describing pictures need to be .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} \texttt{ps2pdf figure.ps} \hspace{1cm} (which will yield \texttt{figure.eps.pdf})
2. \texttt{pdfcrop figure.eps.pdf} \hspace{1cm} \texttt{pdfcrop figure.pdf} \hspace{1cm} (which will yield \texttt{figure.eps-crop.pdf} or \texttt{figure.pdf-crop.pdf})

Here, step 1 creates the .pdf file and step 2 removes extraneous white space around the perimeter of the first-created .pdf file. Second arguments to both \texttt{ps2pdf} and \texttt{pdfcrop}, as in

\begin{verbatim}
ps2pdf figure.eps figure.pdf
pdfcrop figure.pdf figurecrop.pdf
\end{verbatim}

can be used to relieve the awkwardness of the file names. In addition, \texttt{pdfcrop} has a number of options. The statement \texttt{pdfcrop --help} will provide a list of those options on the screen. Sometimes the tight cropping provided by the illustrated commands will clip the edges a bit too closely. The option \texttt{--margins ...}, as in the statement

\begin{verbatim}
pdfcrop --margins 10 figure.pdf figurecrop.pdf
\end{verbatim}
A.11. CONVERTING .EPS AND .PS FILES TO .PDF

will provide a 10-pixel margin of white space around the image. See the help message for further details.

To simplify the process, the MS DOS batch files listed in Section A.A.1.1, respectively, and the UNIX Shell scripts listed in Section A.A.2.1, respectively, exploit the second argument for both ps2pdf and pdfcrop to control the file names and, when executed with statements like

![MS DOS](https://example.com)

![UNIX](https://example.com)

will (1) leave in the involved directory a cropped PDF file whose name `figure.pdf` differs from that of the original PostScript in only the file type and (2) remove any temporary files created along the way.

The above procedure is a bit tedious if you have more than a few files to convert. If you work in a Command window (Windows) or a Shell window (UNIX), the steps

- Create a temporary directory and create the several files listed in Section A.A.1 (Windows) or A.A.2 (UNIX) and the file listed in Section A.A.3 (Windows and UNIX) into that directory. These files can be created by direct typing or they can be copied from the directory `$HEAD/tex`.
- Copy all `.ps` or all `.eps` files to be converted into that temporary directory. *This step guards against disaster should the process to come be run in the initial directory and fail.*
- Create a file containing a list of the names of the `.eps` or `.ps` files in the directory. The statements

  ![MS DOS](https://example.com)

  ![UNIX](https://example.com)

  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 `ExtractFileName.py` (Section A.A.3.1). Here the same code works in both Windows and UNIX. This action will create the file `nameonly.txt` containing only the names of the files to be converted.
- Effect the conversions by running the script `rdfile` with statements like

  ![MS DOS](https://example.com)

  ![UNIX](https://example.com)

  This action will convert each file in turn from `.eps` or `.ps` to `.pdf` and leave no intermediate files in the directory.
- Delete the `*.eps` or `*.ps` files in the temporary directory.
- Move the `*.pdf` files to the directory from which the `.eps` and `.ps` files were moved temporarily, leaving both the original `.ps` and `.eps` files intact but adding the `.pdf` files, so either `latex` or `pdflatex` can then be used to create the final document.

will accomplish that conversion.

The reverse process of converting a `.pdf` file to `.ps` or `.eps` format is easier. Specifically, the statements
APPENDIX A. INTRODUCTION TO LATEX

pdftops FileName.pdf FileName.ps
dfptops -eps FileName.pdf FileName.eps

will effect this conversion. The task of creating scripts to automate this process when many files are to be converted is left to the reader.

A.12 Using Conditional Expressions in LATEX

The LATEX package ifthen provides a capacity to include text and LATEX commands conditionally, i.e., depending on the state of a Boolean flag. If the package is to be used, the preamble of the LATEX source file must contain the command

\usepackage{ifthen}

Once that command has been processed, we must—also in the preamble—then define one or more Boolean flags and set their values with commands like

\newboolean{FlagName}
\setboolean{FlagName}{FlagValue}

where FlagName can be any name that does not conflict with names already used and FlagValue will be either true or false. Finally, at the point in the source file where some text is to be included conditionally, we would place the command

\ifthenelse{\boolean{FlagName}}{Text to be inserted if FlagValue is true.}{Text to be inserted if FlagValue is false.}

Either text can be null, in which case the opening and closing braces should still appear side by side ({}), and either text can include LATEX commands—which will be executed—and can spread over several lines. In particular, the text can include commands to input additional files. Indeed, the customization in 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 CPSUP, the command

\usepackage{ifthen}

appears and several flags are defined and set with commands like, for example,

\newboolean{LATEX} \setboolean{LATEX}{true}

Finally, the command

\ifthenelse{\boolean{LATEX}}{\input{FileName}}{ }

is placed at the point where the file containing the LATEX portions of this book would be inserted. If the file identified is in the working directory, only its name need be included; otherwise, a full path—absolute or relative—must be specified. At this point, the LATEX Appendix will be included if the flag is true and omitted if the flag is false.

More complicated logical operations can be constructed with the operators \and, \or, and \not. For example, if the Boolean flags NUMREC and FORTRAN are defined and set, then the composite statement
A.13. ERROR MESSAGES GENERATED BY $\LaTeX$

\texttt{\ifthenelse{\boolean{NUMREC} \and \boolean{FORTRAN} } { \input{FileName} } { }}

would include $\texttt{FileName}$ only if both \texttt{NUMREC} and \texttt{FORTRAN} are true.

In the effort to render files processable with either latex or pdflatex and also to create easily either a printable document or a linked document, the package hyperref must be invoked only when a linked document is desired. In order to avoid manual editing of the source file, one can insert in the preamble the statements$^{64}$

\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 hyperref, but only if \texttt{PRINT} is false.

One further conditional command included in latex and pdflatex without adding a package facilitates testing whether a file exists. For example, the statement

\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 latex or pdflatex and then read in a subsequent pass. The “false” clause could alternatively be used to print an error message and terminate execution, e.g.,

\begin{verbatim}
\IfFileExists{test.tex}{\input{test.tex}{\typeout{File not found} \end{document}}}
\end{verbatim}

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

\begin{verbatim}
! error identification
1. number text read  text not read
?
\end{verbatim}

$^{64}$Inclusion of the package \texttt{ifthen} can be omitted if it had been entered previously for other reasons.
The exclamation point is followed by a message that explains the nature of the error. The lower case l (el) is followed (1) by the number of the line in the code at which the problem is detected and (2) the text surrounding the problem. Finally, the question mark prompts for user input. A simple ⟨RETURN⟩ at the question mark will instruct L\TeX to continue to read the file unless a fatal error has occurred (though L\TeX 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 L\TeX will produce the desired finished product. A full description of the messages that may be produced as well as an enumeration of other user inputs at the ? will be found in Chapter 8 in \textit{The L\TeX Manual}.

\section{The Page Previewer for .dvi Files}

Since L\TeX is not wysiwyg, we must use a page previewer to see exactly what our document looks like without printing it out many, many times. The \textit{Local Guide} describes the page previewer available at your site. UNIX systems usually provide a version of 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 L\TeX package. To use this previewer, we first create the .dvi file by running our code through L\TeX 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 .dvi.) or select an appropriate item from a menu (as described in \textit{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 \texttt{man} page.

In some environments, a GUI may provide an easy way to use the mouse to work on L\TeX code, process it repeatedly with L\TeX, 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, L\TeX, screen previewer, printer, ...), one convenient strategy for using a text editor and a page previewer simultaneously involves invoking the previewer with a command like

\begin{verbatim}
xdvi filename or yap filename
\end{verbatim}

\footnote{The point at which L\TeX encounters difficulty is conveyed by the downward displacement of the text after that point, though that point is not always where the offense actually occurs.}

\footnote{Yet Another Previewer}

\footnote{Additional information about xdvi on UNIX systems can be found in the on-line help, accessed in many systems by typing the command \texttt{man xdvi}.}

\footnote{If explicit specification of the input file is omitted, some implementations of xdvi will bring up a browser in which the desired .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 \textit{File} menu.}
Table A.11: Some of the features available through selection from menus (first column) in some versions of \texttt{xdvi}, through clicking ML on buttons in other versions of \texttt{xdvi}. The features actually available by these means may vary from version to version of \texttt{xdvi}.

<table>
<thead>
<tr>
<th>Menu Button</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>File</strong>→Reload</td>
<td>Reread</td>
</tr>
<tr>
<td><strong>File</strong>→Quit</td>
<td>Quit</td>
</tr>
<tr>
<td><strong>Zoom</strong>→Shrink by 1</td>
<td>100% Displays text at full size (quite large).</td>
</tr>
<tr>
<td><strong>Zoom</strong>→Shrink by 3</td>
<td>33% Displays text at 33% of full size.</td>
</tr>
<tr>
<td><strong>Zoom</strong>→Shrink by 4</td>
<td>25% Displays text at 25% of full size.</td>
</tr>
<tr>
<td><strong>Zoom</strong>→Shrink by 6</td>
<td>17% Displays text at 17% of full size.</td>
</tr>
<tr>
<td><strong>Navigate</strong>→First Page</td>
<td>First Moves to first page.</td>
</tr>
<tr>
<td><strong>Navigate</strong>→Page-10</td>
<td>Page-10 Moves to the tenth page before the one displayed.</td>
</tr>
<tr>
<td><strong>Navigate</strong>→Page-5</td>
<td>Page-5 Moves to the fifth page before the one displayed.</td>
</tr>
<tr>
<td><strong>Navigate</strong>→Prev</td>
<td>Prev Moves to previous page</td>
</tr>
<tr>
<td><strong>Navigate</strong>→Next</td>
<td>Next Moves to next page.</td>
</tr>
<tr>
<td><strong>Navigate</strong>→Page+5</td>
<td>Page+5 Moves to the fifth page after the one displayed.</td>
</tr>
<tr>
<td><strong>Navigate</strong>→Page+10</td>
<td>Page+10 Moves to the tenth page after the one displayed.</td>
</tr>
<tr>
<td><strong>Navigate</strong>→Last Page</td>
<td>Last Moves to last page.</td>
</tr>
<tr>
<td><strong>Options</strong>→PostScript</td>
<td>View Page Allows selection among two or more options, including displaying figures (the initial state) and replacing them with properly sized rectangles.</td>
</tr>
<tr>
<td><strong>File</strong>→Open</td>
<td>File Brings up a browser for selection of new \texttt{.dvi} file.</td>
</tr>
</tbody>
</table>

Table A.12: Keystrokes for controlling \texttt{xdvi}.

<table>
<thead>
<tr>
<th>Key</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>s</td>
<td>Size text to fit window.</td>
</tr>
<tr>
<td>1s</td>
<td>(one-s) Largest text.</td>
</tr>
<tr>
<td>ns</td>
<td>Apply shrink factor (n).</td>
</tr>
<tr>
<td>g</td>
<td>Move to last page.</td>
</tr>
<tr>
<td>1g</td>
<td>(one-g) Move to first page.</td>
</tr>
<tr>
<td>ng</td>
<td>Move to (absolute) page (n).</td>
</tr>
<tr>
<td>n</td>
<td>Move to next page.</td>
</tr>
<tr>
<td>p</td>
<td>Move to previous page.</td>
</tr>
<tr>
<td>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>

\texttt{xdvi filename &} (UNIX) or \texttt{yap filename &} (Windows)

which will start \texttt{xdvi} or \texttt{yap} but detach it from the launching \texttt{Shell} window, leaving the \texttt{Shell} window free for other uses. Then, with the \texttt{.tex} file in a text editor detached from the launching \texttt{Shell} window, we can edit the text, save the edited version, use the \texttt{Shell} window to invoke \texttt{\LaTeX} on the edited file, and then simply click ML at an appropriate point in the displayed text to instruct \texttt{xdvi} or \texttt{yap} to reread the \texttt{.dvi} file.\footnote{In some environments, rereading an updated file will happen automatically without an explicit user request.} Thus, the effect of each new set of edits can be examined quickly without repeatedly starting and exiting from the previewer.

\section{A.15 The Spell Checker in UNIX\footnote{To the author’s knowledge, no comparable stand-alone spell checker exists for Windows.}}

The programs \texttt{ispell} and \texttt{aspell}, either or both of which may be part of your \texttt{\LaTeX} distribution, are common spell checkers. If either is installed at your site (see the \texttt{Local Guide}), it can be invoked
with a command like\footnote{Additional information about ispell or aspell can be found by typing the command ispell with no arguments or the command aspell help. Even more detailed information may be available in the on-line help, accessed in many systems by typing the command man ispell or the command man aspell. Googling ispell or aspell will surely also provide links to detailed descriptions.}

\texttt{ispell filename} \quad \text{or} \quad \texttt{aspell check filename}

or perhaps with a mouse click in a GUI. For files with extension .tex, 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 ignored by ispell and aspell; their correctness or legitimacy as commands to \LaTeX{} is 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 \textit{comments} following the \% sign can be included in the code but have no effect on the output produced.)

\begin{verbatim}
\documentclass{article} % Mandatory command; select
% default 10 point type

\setlength{\textheight}{9truein} % Set height of text area
\setlength{\topmargin}{-0.5truein} % Center text on 11'' height
\setlength{\textheight}{9truein} % 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\}/: J.~Q.~Student \ \\
% \\ forces a new line
% Note italic correction
% Tilde prevents extra space

{\em Date\}/: \today
\end{flushleft}

In this document we have used the \verb+\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 \verb+$\backslash$\tt LaTeX+}
provided for the display of the \LaTeX{} ‘‘logo’’. The title could have been generated using the three commands
\verb+\title{+\ldots\verb+}+, \verb+\author{+\ldots\verb+}+, and
\verb+\date{+\ldots\verb+}+ before the \verb+\begin{document}+
\end{verbatim}
The command---i.e., in the preamble---followed by the command
\verb*+\maketitle+ after the beginning of the document. The command
\verb+?\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 \em{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,
\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\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 \emph{twice} by \LaTeX{}. The file itself is named \texttt{tексsample2.tex} and can be copied from the
directory \$\HEAD/tex$.

\section{Miscellaneous Other Features}

\LaTeX{} has an enormous number of additional features beyond those enumerated above. In particular,\be aware of

1. All of the environments listed in Section A.3.4.
2. All of the packages listed in Section A.3.5.
3. The command \verb+\today+, which returns today's date in the form “month date, year”.
4. The command \verb+\newcommand+, which permits us to define commands supplementing the standard
commands. For example, the commands
\begin{verbatim}
\newcommand{\beq}{\begin{equation}}
\newcommand{\eeq}{\end{equation}}
\end{verbatim}
Table A.13: Output produced by the code in Section A.16, except that the equations here are numbered (A.1) and (A.2) rather than (1) and (2) and the footnote is labeled a rather than 1. Further, the page number that would appear has been deleted in this display.

A Sample Document for Your Perusal

Author: J. Q. Student
Date: 14 February 2023

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" ×11" page. The title could have been generated using the three commands \texttt{title{...}}, \texttt{author{...}}, and \texttt{date{...}} before the \texttt{begin{document}} command—i.e., in the preamble—followed by the command \texttt{maketitle} after the beginning of the document. The command \texttt{verb*+...+} will cause \LaTeX to print whatever is between the plus signs (which could be virtually any specified character) exactly as it is. The asterisk is optional, and it makes \LaTeX highlight the spaces that occur inside the \texttt{verbatim} environment.

Notice that a blank line creates a new paragraph. The \texttt{noindent} command suppresses the standard paragraph indentation. In this special space we will also demonstrate some math environment operations. Consider the equation
\[ \int_{0}^{\infty} e^{-x} \, dx = 1, \] (A.1)
where a medium sized space, specified by the command \texttt{\textbackslash ;}, is put between the integral sign \texttt{\textbackslash \int_{0}^{\infty}}, and the integrand and a small space, specified by the command \texttt{\textbackslash ,}, is inserted between the integrand and the \texttt{dx} in Eq. (A.1). The equation
\[ \frac{\partial e^{x_0 y^2}}{\partial x_0} = y^2 e^{x_0 y^2} \] (A.2)
demonstrates partial derivatives and fractions. Note the automatic generation and placement of equation numbers in the output. Also note that pages are automatically numbered, though the actual printing of page numbers can be suppressed with the commands \texttt{\pagestyle{...}} and \texttt{\thispagestyle{...}}.

in the preamble will permit us to type the commands \texttt{\beq} and \texttt{\eeq} rather than the longer forms, measurably simplifying the typing of a document containing many displayed equations. Details will be found in Sections 3.4.1 and C.8.1 in \textit{The \LaTeX Manual}.

5. The command \texttt{\renewcommand} for changing commands that already exist. When a command already exists (as, for example, when a command is defined by the selected document class), \texttt{\newcommand} will fail. In those circumstances (as we have seen already in Section A.5), we need the command \texttt{\renewcommand}. As one particular example, the built-in command \texttt{\today} which, by default, formats the date in American style (e.g., April 3, 1938) can be changed to format the date in European style (3 April 1938) with the command
\begin{verbatim}
\renewcommand{\today}{\number\day \space \ifcase\month \or January\space
\or February\space \or March\space \or April\space \or May\space \or June\space \or July\space \or August\space
\or September\space \or October\space \or November\space \or December\fi \space \number\year}
\end{verbatim}

The percent signs at the end of the first two lines are not superfluous; they guarantee that \BTeX will see this command as a single line and hence that extraneous spaces will not appear in
the output for occasional dates. Note also that this modification invokes \TeX’s `case` structure, which is introduced by the \texttt{\ifcase} command and terminated by the \texttt{\fi} command.

6. The command \texttt{\pagestyle{Style}}. Placed in the preamble, this command selects a particular (global) page style for the entire document. Recognized styles include \texttt{plain}, \texttt{empty}, \texttt{headings}, and \texttt{myheadings}. Details will be found in Sections 6.1.2 and C.5.3 in \textit{The \LaTeX\ Manual}.

7. The command \texttt{\thispagestyle{Style}}. Placed at any point, this command selects a particular page style for the current page, overriding the global specification for that page alone. Recognized styles are the same as for the command \texttt{\pagestyle}. This command in the form \texttt{\thispagestyle{empty}} is commonly used to suppress page numbering on the first page of a several-page document. Details will be found in Sections 6.1.2 and C.5.3 in \textit{The \LaTeX\ 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 \REVTeX\ Input Guide} prepared by the AIP, the APS, and the OAS.\footnote{Version 4.1, which was released in final form on 11 August 2010, is compatible with \LaTeX\2\epsilon. The APS website publish.aps.org/revtex4 (no www) contains up-to-date information about \REVTeX4, links to an assortment of manuals (including one titled \textit{Rev\TeX\4.1 Author’s Guide}, and a link from which that version can be downloaded. \REVTeX is automatically included in many standard \LaTeX\ distributions.}

9. The command \texttt{\input}, which simply inputs the file specified in its argument. Details are described in Section 4.4 of \textit{The \LaTeX\ 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 \LaTeX\ Manual}.

11. Methods for placing accents over characters. For example, the command \texttt{"{o}} will produce ¨ o, the command \texttt{\~{n}} will produce ˜ n, and the command \texttt{\c{c}} will produce ¸ c. A full listing of the possibilities will be found in Section 3.2.1 and Table 3.1 of \textit{The \LaTeX\ 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 \LaTeX\ 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 \LaTeX\ commands

\begin{verbatim} \input{FileName} \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 \LaTeX\ 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 \LaTeX\ file.
Actually, there is at least one situation in which the alltt environment isn’t quite up to the task. If we want to read in a C program that specifies newline characters with `\n`, \LaTeX\ 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 \LaTeX.\footnote{The verbatim package also redefines the verbatim environment in ways, however, of little consequence unless the text in the environment is very extensive. If the verbatim package is invoked, one particularly significant change in the verbatim environment results in the complete ignoring of any characters following the statement `\end{verbatim}` in the same line. It is best always to place the statement `\end{verbatim}` on a line by itself.}

13. The ten characters (`$, &; {, }, %, #, \, ~, ^, \`), which have special meanings to \LaTeX\ and will not normally be printed as characters. The first seven of these characters can be printed as characters by preceding the character with a backslash, e.g., to print `$`, 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 \verb$+$.\footnote{The verbatim package also redefines the verbatim environment in ways, however, of little consequence unless the text in the environment is very extensive. If the verbatim package is invoked, one particularly significant change in the verbatim environment results in the complete ignoring of any characters following the statement `\end{verbatim}` in the same line. It is best always to place the statement `\end{verbatim}` on a line by itself.}

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 \LaTeX\) 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 `\verb$\`$.

14. The commands that allow us to create boxes as described in Sections 6.4.3 and C.13.3 in The \LaTeX\ 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 \LaTeX\ 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 \LaTeX, 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 \LaTeX\ 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 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 \texttt{enumerate} environment, the counter \texttt{enumi}, which is used behind the scenes to keep track of the number of items, is initialized to zero. It is then incremented by 1 with each new item. At the end of the environment, \texttt{enumi} stores the number of the last item. The command \texttt{\setcounter} in the first environment saves the current value of \texttt{enumi} in the counter \texttt{hold}, which survives the exit from the environment. Thus, immediately on entry to the second environment, we can use a “reflection” of the first use of \texttt{\setcounter} to restore the value that \texttt{enumi} had reached at the end of the first environment.

16. Parameters influencing placement of “floats” (figures and tables). Sometimes \LaTeX seems to have a mind of its own with regard to placement of floating objects on the current or later pages. Many of these “problems” can be cleared up by tampering with the default values of the parameters that limit the number of floats that can be put at the top or bottom of the page or, even more, by changing the parameters that stipulate the maximum fraction of a page that can be devoted to floats or the minimum amount of text that must appear on a page. These parameters are enumerated at the end of Section C9.1 in \textit{The \LaTeX Manual}. For single column presentations, the most important of these parameters are \texttt{\topfraction}, \texttt{\bottomfraction}, \texttt{\textfraction}, and \texttt{\floatpagefraction}. The command \texttt{\renewcommand} must be used to change the values of these parameters.

17. The way to change section, page, figure, table, and footnote “numbering”. By default in the \texttt{article} class, sections, subsections, subsubsections, pages, figures, and footnotes are labeled with Arabic numbers, e.g., Section 3.1.2. The format of that label as determined from the underlying counters \texttt{section}, \texttt{subsection}, \texttt{subsubsection}, \texttt{page}, \texttt{figure}, \texttt{table}, and \texttt{footnote} can be changed. If, for example, we wished the sections to be labeled with upper-case letters, the subsections with lower-case letters, and the subsubsections with arabic numbers (the default), we would simply execute the instructions

\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}
\twocolumn
[ \begin{center}
  \{\textbf{Title}\} \quad [12pt] \% Bold title; extra space
  \{\textbf{Author}\} \quad [12pt] \% Bold author; extra space
\end{center}
\begin{quotation}
  \noindent ... Text of abstract.
\end{quotation}
]

... Text of paper
\end{document}

will change table “numbering” to upper-case Roman numbers and page “numbering” to lower-case Roman numbers. Essentially, the “numbers” on these components of a document are generated when \LaTeX{} executes the command \texttt{\the...}, where \texttt{...} stands for the name of the appropriate counter. Redefining this command changes the translation \LaTeX{} applies to the counter when generating the “number”.

18. The way to create a full-width heading and abstract above (and on the same page as) a two-column presentation of text, which exploits an optional argument to the \texttt{\twocolumn} (not the optional argument \texttt{twocolumn} to the command \texttt{\documentclass}). One format that achieves this end uses the code is shown in Table A.14.\footnote{In more recent versions of \LaTeX{}, the optional argument \texttt{[12pt]} to put a bit of extra space after the title and author appears to generate error messages. The “fix” is to replace \texttt{[12pt]} with a full command, specifically \texttt{vspace[12pt]}.} If this pattern is to work, there can be no commands that produce output between the command \texttt{\begin{document}} and the command \texttt{\twocolumn}.

19. The means to change the label in the caption of a figure. By default, \LaTeX{} introduces the caption of the first figure with the label ‘Figure 1:’, incrementing the number automatically with each subsequent figure. The word ‘Figure’ in this caption is the default value of the command \texttt{\figurename}, but that portion of the label can be changed with a simple command like

\begin{verbatim}
\renewcommand{\figurename}{Fig.}
\end{verbatim}

in the preamble. Changing the colon after the number is harder, since the specification of that punctuation is embedded in the class file in use and is not brought to the outside in a simple command. To change the colon to a period, for example, we need to find the file \texttt{article.cls}, search through the file for colons, replace the critical ones with periods, save the file in the default directory with a new name, and then specify it rather than the standard file in the \texttt{\documentclass} command at the beginning of the \LaTeX{} source file.\footnote{The file \texttt{article.cls} in the version of \LaTeX{} installed at Lawrence University has only two colons that are not in comments; both should be changed.}
20. The means to change the label in the caption of a table. By default, \LaTeX\ introduces the caption of the first table with the label ‘Table 1:’, incrementing the number automatically with each subsequent table. The word ‘Table’ is supplied by the command \tablename\ and can be changed in the way described for \figurename\ at item 19 above. The colon can also be changed as described in the previous item, though changing the colon for figures will change it in the same way for tables.

21. The means to change the label in the title of a chapter. By default, \LaTeX\ introduces the title of the first chapter with the label ‘Chapter 1’ on a line by itself, incrementing the number automatically with each subsequent chapter. The word ‘Chapter’ is supplied by the command \chaptername\ and can be changed in the way described for \figurename\ at item 19 above.

22. The means to change the title above the table of contents and above the index. By default, \LaTeX\ labels the table of contents ‘Contents’ and the index ‘Index’. The word ‘Contents’ is supplied by the command \contentsname\ and the word ‘Index’ is supplied by the command \indexname\. Each can be changed in the way described for \figurename\ at item 19 above.

23. Using the array environment to create matrices in math mode. As with the tabular environment (see Section A.6), an optional argument specifies the positioning of entries in the columns, ampersands separate entries in each row, and the command \\ marks the end of each row. Thus, for example, the \LaTeX\ code

\[
\begin{array}{cccc}
1 & 2 & 3 & 4 \\
5 & 6 & 7 & 8 \\
9 & 10 & 11 & 12
\end{array}
\]

will produce the display

\[
\begin{array}{cccc}
1 & 2 & 3 & 4 \\
5 & 6 & 7 & 8 \\
9 & 10 & 11 & 12
\end{array}
\]

The entries in each cell can, of course, be much more elaborate than the simple choices made here.

24. The \LaTeX\ package needspace, which facilitates avoiding awkward page breaks conditionally. For example, when a section title occurs close to the bottom of a page, \LaTeX\ may place the title and only a single line of the text before turning the page. Sensible styles would dictate that the section be started on a fresh page. The command \newpage\ will, of course, achieve that objective. As a document experiences further edits, however, that page break may not any longer be appropriate. Placing the command

\usepackage{needspace}

in the preamble and then, at the point where a conditional page break might be wise, inserting one of the commands

\needspace{5\baselineskip} \quad \text{or} \quad \Needspace{5\baselineskip}

will result in turning the page, but only if five or fewer lines remain at the bottom of the page. The amount of space left by \needspace\ will depend some on what penalties are in effect but will usually be close enough to be acceptable; further, \needspace\ will leave a ragged bottom even if \flushbottom\ is in effect. The command \Needspace\ will leave the requested space, will take longer to execute, and will leave a ragged bottom; the command \Needspace*\ will produce a flush bottom if \flushbottom\ is in effect. The unit of measure is the size of the length parameter \baselineskip. The 5 in this example can, of course, be whatever number seems appropriate. Note that there is no multiplication sign after the number.
25. The Windows batch command `cleanTEX` defined by the batch file

```bash
REM delete all TEX intermediate files
```

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/LaTeX 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/LaTeX 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/LaTeX auxiliary files.

26. The Windows batch command `cleanbak` defined by the batch file

```bash
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.

### 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](http://www.tug.org) will yield valuable results. Further, when \LaTeX{} is installed in UNIX and UNIX-based operating systems (which includes Mac computers), the Shell command `man`, e.g. `man hyperref`, may bring up a

---

\(^{77}\)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 `~`. 
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:


* **Math Into \textsc{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 \textsc{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 \textsc{LATeX} code presented in Section A.16 and the resulting output in Table A.13, making sure you understand both the syntax of each command and the effect it produces in the output.

**A.2.** Use \textsc{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, ...) 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. 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{verbatim}
M E M O R A N D U M

Current date

TO: Insert name of recipient here.
FROM: Insert name of sender here.
SUBJECT: Insert topic here.

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. Suggestion: You may want to save this template in a file, say 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 539. 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>$xxx,xxx</td>
<td></td>
</tr>
<tr>
<td>Expenses in current period:</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Salaries</td>
<td>$xxx,xxx</td>
<td></td>
</tr>
<tr>
<td>Fringe benefits</td>
<td>$xxx,xxx</td>
<td></td>
</tr>
<tr>
<td>Equipment</td>
<td>$xxx,xxx</td>
<td></td>
</tr>
<tr>
<td>Supplies</td>
<td>$xxx,xxx</td>
<td></td>
</tr>
<tr>
<td>Travel</td>
<td>$xxx,xxx</td>
<td></td>
</tr>
<tr>
<td>Page charges</td>
<td>$xxx,xxx</td>
<td></td>
</tr>
<tr>
<td>Other: Identify</td>
<td>$xxx,xxx</td>
<td></td>
</tr>
<tr>
<td>Total for current period</td>
<td>$xxx,xxx</td>
<td></td>
</tr>
<tr>
<td>AVAILABLE FUNDS FOR REMAINDER OF PROJECT</td>
<td></td>
<td>$xxx,xxx</td>
</tr>
</tbody>
</table>

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

A.A.1.1 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

A.A.1.2 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

A.A.1.3 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

A.A.1.4 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

A.A.2.1 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

A.A.2.2 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

A.A.2.3 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

A.A.2.4 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

A.A.3.1 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 -i
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.

LAPACK

LAPACK is a large FORTRAN 90 package which implements numerous algorithms to accomplish various tasks in linear algebra. Full information is available at www.netlib.org/lapack. The source code is in the public domain. Instructions for downloading the package are included at the referenced website. Those downloads include the routines in BLAS (Basic Linear Algebra Subprograms), which are invoked by routines in LAPACK. The development of the package was supported by the National Science Foundation and the Department of Energy, and maintenance has been supported for many years by MathWorks and Intel. (Netlib is a large repository of programs maintained at Oak Ridge National Laboratory.)

\LaTeX{} – see \TeX{}

LSODE

The ODE solver LSODE is one component in a large package of ODE solvers originating in the Computing and Mathematics Research Division of the Lawrence Livermore National Laboratory. The full package is called ODEPACK, public-domain software written in FORTRAN by Alan C. Hindmarsh and others. Information about compiled solvers, numerous example programs, and source code for the package are available for download from links at computing.llnl.gov/projects/odepack and computing.llnl.gov/projects/odepack/software. Additional information and downloads may be found from links at www.netlib.org/odepack. In particular, the two text files opkd-sum and
opks-sum at this URL provide a description of the single- and double-precision components in the package. (Netlib is a large repository of programs maintained at Oak Ridge National Laboratory.)

**Mac\TeX**

Mac\TeX is a shareware implementation of the \TeX/\LaTeX system for Macintosh computers. Information is available at the URL [www.tug.org/mactex](http://www.tug.org/mactex).

**MAXIMA**

Originally called MACSYMA, this first of the computer algebra systems was developed at MIT and supported from its origin in 1968 until 1982 by MIT, NASA, ONR, and DOE. In 1982, MACSYMA became a commercial product that was further developed and remained available until about 1999. The 1982 MIT version remained available as DOE MACSYMA but was released in 1999 to a group that continues to develop and maintain the program, changing its name to MAXIMA, whose website is at the URL [http://maxima.sourceforge.net/](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](http://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](http://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](http://www.mathworks.com)

**MiK\TeX**

MiK\TeX is an implementation of the \TeX/\LaTeX system for Windows computers. Information is available at the URL [miktex.org](http://miktex.org). MiK\TeX is also available from CTAN, the \TeX/\LaTeX distribution network accessible from links at [www.tug.org](http://www.tug.org).

**MUDPACK**

The freely-available package of FORTRAN PDE solvers called MUDPACK, written by John C. Adams and others, originated at the National Center for Atmospheric Research (NCAR). The main web page for information about this FORTRAN package, and many others, is [www2.cisl.ucar.edu/research-software/software](http://www2.cisl.ucar.edu/research-software/software). MUDPACK can be downloaded from the webpage [github.com/NCAR/NCAR-Classic-Libraries-for-Geophysics](https://github.com/NCAR/NCAR-Classic-Libraries-for-Geophysics). While this package is no longer under active development, it remains available and useful.

**Numerical Algorithms Library**

The Numerical Algorithms Library (NAG library) is a large commercially available library of C and Fortran subroutines/subprograms implementing a wide assortment of numerical and statistical algorithms.

Numerical Algorithms Group, Inc.
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](http://www.nag.com)
NUMERICAL RECIPES

The vendor of Numerical Recipes is reluctant to provide detailed contact information, preferring that potential customers deal with them through forms on their website at www.numerical.recipes (without a '.com'). The current version and many of the past versions, some of which include languages no longer being updated, are available for download from this site.

OCTAVE

Available under the terms of the GNU General Public License as published by the Free Software Foundation, OCTAVE is an array processing program whose syntax is similar to that of MATLAB. Information about the program can be found at the site octave.org.

ODEPACK – see LSODE

OzTeX

OzTeX is a shareware implementation of the TeX/LaTeX system for Macintosh computers. Information is available at the URL www.trevorrow.com/ozte. OzTeX is also available from CTAN, the TeX/LaTeX distribution network accessible from links at www.tug.org.

PYTHON/NUMPY/MATPLOTLIB/PLOTLY/MAYAVI/SCIKIT-IMAGE

PYTHON, a high-level programming language created by Guido van Rossum, was first released in 1991. In execution, PYTHON programs are interpreted, not compiled. Information and free downloads are available at the URL www.python.org. Information about various add-on modules can be found as follows:

- The module numpy, which adds numerous mathematical capabilities (arrays, matrices, mathematical functions, ...) to PYTHON, is described at the URL www.numpy.org.
- The module matplotlib, which adds numerous 2D plotting capabilities and a few 3D plotting capabilities to PYTHON, is described at the URL matplotlib.org.
- The module plotly, which provides several 3D plotting capabilities, is described at the URL plot.ly.
- The module mayavi, which provides numerous capabilities for 3D data visualisation and plotting, is described at the URL docs.enthought.com/mayavi/mayavi.
- The module scikit-image, which includes scimage and provides image processing capabilities, is described at the URL scikit-image.org.

There are numerous distributions of PYTHON. A popular distribution that automatically includes numerous modules that might not be included in many other distributions is described at www.anaconda.com.

\TeX/La\TeX/dvips/graphicx/makeindex/xdvi/dvips/…

The primary site for information (history, current plans, downloads, ...) for \TeX, \LaTeX, and numerous other publicly available components of \TeX and its derivatives is the web site of the \TeX Users’ Group (TUG), www.tug.org. This organization maintains CTAN (the Comprehensive \TeX Archive Network), which has a handful of backbone machines around the world and a number of mirror sites, from any of which an enormous number of files associated with the \TeX/La\TeX system can be downloaded.

\TeXlive

\TeXlive is among the newer implementations of the \TeX/La\TeX system for all platforms. Information about this distribution and instructions for downloading and installing it are available from the URL www.tug.org/texlive.
TGIF

TGIF (pronounced T-G-I-F) is a versatile program for creating two-dimensional drawings. The program is Xlib-based and interactive, and it runs under X11 on LINUX and UNIX platforms (including MAC OS X and cygwin on Windows). Information (brief history, licensing understandings, instructions for downloading, ...) about TGIF is available from the URL bourbon.usc.edu/tgif/.

Winedt

Winedt is an inexpensive ($40.00 per student user, $60 per educational user) editor for Intel-based machines running one or another version of Microsoft Windows. It has particular features that make it especially suitable for creating source files for \TeX/\LaTeX. Information about licensing and downloads can be found at the URL www.winedt.com. Winedt is also available from CTAN, the \TeX/\LaTeX distribution network accessible from links at www.tug.org.

Xemacs

This flexible, open source text editor is available from a variety of sources and is protected under the terms of the GNU Public License. Downloads and information about the program can be found at the URL xemacs.sourceforge.net.

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

$\text{HEAD}$, 11, 15, 40, 56, 143, 193–195, 198, 200, 250, 272, 297, 313, 353, 402, 495
$\text{IDLHEAD}$, 11
$\text{MAXHEAD}$, 117
$\text{NRHEAD}$, 11
$\text{PYTHONHEAD}$, 15

A
aboring execution
  in MAXIMA, 116
  in PYTHON, 16, 193
accuracy
  in evaluating integrals, 308–309, 313–316
  in finding roots, 352–353
  in solutions via FEM, 465–467
  in solving ODEs, 247–249, 258, 261
    with conservation laws, 249, 261–263
  in solving PDEs, 417–419, 432–433,
    438–440, 447–448
acropread, 500
action statements, 175, 178
Adams/Bashforth/Moulton (ABM) methods, 250
adaptive methods
  for integrals, 310
  for ODEs, 249
adding text to graphics, see placing text in graphics
algorithms
  definition, 175
  for $a^2 + b^2 = c^2$, 203
  for adding, 203
  for bowling, 202
  for controlling loops, 184
  for counting, 184, 185
  for exchange, 184
  for finding extremes, 185, 186
  for integration, 303–313
  for Laplace’s equation, 186
  for ODEs, 241–250
  for parking lot, 201
  for root finding, 350–353
  for searching, 186, 202
  for sorting, 203
  for summing, 185
animation
  in MAXIMA, 155
  in PYTHON, 80–87, 431–432, 437–438,
    446–447
arithmetic operators
  in MAXIMA, 120
  in PYTHON, 21
array indices
  in MAXIMA, 136
  in pseudocode, 177
  in PYTHON, 23–27
arrays
  for data storage, 188
  in MAXIMA, 119, 135–137
  in pseudocode, 177
  in PYTHON, 24–27
scalar, see scalar fields
vector, see vector fields
ASCII character codes, 5
ASCII text files, 8
aspell, 527–528
assigning values to variables
  in MAXIMA, 121–122
  in pseudocode, 178
  in PYTHON, 18

B
backsubstitution, 369
backward difference, 398
Bessel functions, 334, 335, 377
binary files, 8
binary number system, 4–5
bisection, see root finding
bit, 4–5
black body radiation, 167, 333
blocking statements, see also control structures
  in MAXIMA, 140, 365
  in pseudocode, 180, 190
  in PYTHON, 35
Boolean values, see logical expressions
Boolean variables, see logical expressions
boundary conditions
  Dirichlet, 383, 397, 399–400
  mixed, 383, 397, 399–400
  Neumann, 383, 397
  of first kind, see boundary conditions,
  Dirichlet
  of second kind, see boundary conditions,
  Neumann
  of third kind, see boundary conditions,
  mixed
boundary value problems, 215, 226, 380, 381, 383
byte, 5–6
C
  C, 1
  cardioid, 52, 331
  carrying capacity, 220, 232–234
  Cartesian coordinates
    in MAXIMA, 172
  case sensitivity
    in MAXIMA, 116
    in pseudocode, 176
    in PYTHON, 16–18
  center of mass
    definition of, 286
    of cardioid, 331
    of semicircular plate, 287, 295
  central difference, 398
  central processing unit, 3
  chaos
    Lorenz system, 281
  character codes, 4–5
  character strings
    in PYTHON, 18
  characteristic equation, 340
  charging capacitor, 101
  classification
    of boundary conditions, 383–384
    of integrals, 285
    of ODEs, 215
    of PDEs, 383
  color
    in PYTHON, see also PYTHON keywords:
      color, edgecolor, colors, ..., 51, 52, 91–92
  combining statements on one line
    in MAXIMA, 116
    in pseudocode, 180
    in PYTHON, 16, 34, 35, 54
  command files
    in MAXIMA, see MAXIMA command files
    in PYTHON, see PYTHON scripts
  command line editing
    in MAXIMA, 122
    in PYTHON, 16
  commands
    in MAXIMA, see MAXIMA commands
    in pseudocode, see pseudocommands
    in PYTHON, see PYTHON commands
  comments, see also documentation
    in LATEX, 528
    in MAXIMA, 143
    in pseudocode, 181, 184
    in PYTHON, 16, 41, 47
  common storage
    in PYTHON, see global variables
  compiler directives, 177
    in pseudocode, see pseudocode directives
  compiling source file, see Local Guide, 176
  complex numbers
    in MAXIMA, 126–127
    in PYTHON, 28–29
  Compton scattering, 103
  conditions, see logical conditions
  conservation of mass, 392–393
  constants
    in MAXIMA, see MAXIMA special symbols
    in PYTHON, see PYTHON constants
    contact information, 543–546
  continuing statement on second line
    in MAXIMA, 116
    in pseudocode, 180
    in PYTHON, 16
  control characters, 4
  control structures
    in MAXIMA, see MAXIMA control structures
    in pseudocode, see pseudocode control structures
    in PYTHON, see PYTHON control structures
  convergence criteria, 188, 247–249, 308–309, 352–353
  convergence error, 400–401, 427
  convert .eps/.ps to .pdf, 522–523
  convert .pdf to .eps/.ps, 523
  Cornu spiral, 290, 321–322
  coupled oscillators, 222–223, 236–237, 276
  CPU, see central processing unit
  Cramer’s rule, 369
  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 MAXIMA, 143–144, 174
  in PYTHON, 26, 100
current directory, see also default directory
  in PYTHON, 40, 45, 56, 59, 81, 83, 93–94
curve fitting, see least squares fitting
to parabola, 171, 379
customizing
  MAXIMA, 161
  PYTHON, 89–91, 94–96
cylindrical coordinates
  in MAXIMA, 173

D
data analysis, 289
data files
  freefall.dat, 379, 380
  irrad_python.dat, 194, 200–201, 208, 214
  laplace_c.dat, 193
  laplace_py.dat, 192
  ldrum.dat, 82–83, 111–112
  pdens_python.dat, 195, 209–210
  quadpole_python.dat, 200, 212–213
  radio.dat, 40–41, 109
  rcdata.dat, 55, 56, 109–110
  ruchardt.dat, 381
  wavegd_python.dat, 198, 211
data types
  floating point, 6–7
  integer, 6–7
  string, 6–7
data typing
  explicit, 177
  implicit, 177
  in pseudocode, 176–177, 179–180
  in PYTHON, 17
default directory, see also current directory, 9
  in MAXIMA, 159–160
  in PYTHON, 39, 56, 85, 256, 264
defining ODEs
  for MAXIMA, 132, 230, 232, 235, 237, 239, 274
  in PYTHON, 254–255
deleting functions
  in MAXIMA, 163
deleting variables
  in MAXIMA, 122, 129, 163
  in PYTHON, 24, 34
demos
  in MAXIMA, 118
diffraction patterns, see also interference
  patterns, 193, 335, 377, 378
  of square aperture, 59–68
diffusion equation
  in 1D, 388–389
  in PYTHON, 434–440
  in 2D, 422–424
  in 3D, 389
dimensionless casting
  for plotting, 103
  of integrals, 287–289, 291, 297, 330, 333
  of ODEs, 134–135, 216–221, 223–225, 233–234, 339, 341
direct algorithm, 400
  discretization
    of Laplace’s equation, 187
    of ODEs, 398–400
    of PDEs, 384, 405
  discretization error, 400–401, 405, 427
displaying data on screen
  in MAXIMA, 116
  in pseudocode, 178
  in PYTHON, 16, 20–24
dividing statement between lines, see continuing
  statement on second line
documentation, see also comments
  in MAXIMA workbooks, 155–156
  in PYTHON py-files, 47
dot product
  in MAXIMA, 143–144, 174
  in PYTHON, 26, 99
double factorial, 12
drawing space curves
  with MAXIMA, 162
  with PYTHON, 87–88
drumhead, 82–83
dvipdfm, 500
dvis, 499, 517
dynamic allocation of memory, 177

E
elastic swing, 282
electric field, 290
  in wave guide, 75–78, 196–198
  of dipole, 206
  of quadrupole, 199–200
  of relativistic particle, 103
electrostatic potential, 290
  of charged disks, 101, 102, 330
  of charged rings, 290–291, 324, 328–329, 335
  of finite line charge, 296–297
  of spherically symmetric distribution, 331
elemental equation in finite element analysis, 410
elliptic integrals, 171, 289, 334
e s t u n d e g, 508
  em dash, 508
  em space, 505
  en dash, 508
  en space, 505
energy levels
  in quantum oscillator, see harmonic
  oscillator
  in quantum well, 238–239, 343, 360–361, 366–368, 377
equation of continuity, 392–393
  equilibrium points, 338, 346
  equivalent resistance, 379
error function, 290, 292, 313, 320–321, 327–328, 333
error messages
in \LaTeX, 525–526
Euler’s method, 242–243, 251–253
improved, see improved Euler method
evaluating integrals
with MAXIMA, 130, 293–303, 325–329
with PYTHON, 313–324
exchange algorithm, 184
executing program, see Local Guide
exiting
from MAXIMA, 116
from PYTHON, 16
expressing infinity
in MAXIMA, 294
expressions
in MAXIMA, 119–121

\section{F}

Fabry-Perot interferometer, 102
factorial, 12, 21, 123
double, 123
Fibonacci numbers, 100
field mapping, 105, 106
finding eigenvalues/eigenvectors
with MAXIMA, 136–137, 169
with PYTHON, 36–38, 100, 101
finding roots, see root finding
finite difference approximations
for derivatives, 203
for PDEs, 384
finite difference methods
error in, 400–401
in 1D, 398–405
in 2D, 419–448
with PYTHON, 401–405, 427–448
finite element approximations
for PDEs, 384
finite element methods
assembling equations, 406
in 1D, 405–419
in 2D, 448–467
incorporating boundary conditions, 406
with PYTHON, 414–419, 457–467
fitting data, see curve fitting, least squares fitting
floating point number
double precision, 7
single precision, 6
fluid dynamics, 391
fonts, see also Greek letters
formatting data files, 188–189
formatting output
in pseudocode, 178
in PYTHON, 43, 191, 251
FORTRAN, 1
forward difference, 398
four-slit interference pattern, 103
Fourier series, 293, 302–303, 331
Fraunhofer diffraction, 193–194
Fresnel integrals, see also Cornu spiral, 290, 335
functions
in MAXIMA, see MAXIMA functions
in PYTHON, see PYTHON functions,
PYTHON scripts
typical available, 178

\section{G}

Galerkin method, 405, 408–410
gamma function, 12–13
Gauss-Jordan elimination, 406
Gaussian distribution, 289, 333
Gaussian elimination, 369–370, 406
Gaussian integration, see quadrature
gedit, 1
ghostview, 500
global positioning, 378
global truncation error, see truncation error,
global
global variables
in PYTHON, 46–47, 322
gnuplot command
set, 149–152
gnuplot keywords
border, 151–152
cntrparam, 149–152
cycle, 151–152
discrete, 150
incremental, 150
lc, 151–152
levels, 149–152
lincolor, 151–152
linetype, 151–152
linewidth, 151–152
lw, 151–152
ratio, 150–152
size, 150–152
golden ratio, 375
gravitational force, 224
Greek letters, see also fonts
in \LaTeX, 508–509
in MAXIMA, 119
in PYTHON, 62–63, 87

\section{H}

hard copy
from MAXIMA, 152–154
from PYTHON, 58–59
harmonic oscillator, 132–135, 165, 168, 220–221,
234, 258–260, 266–271, 276, 278, 280,
402–405
Helmholtz coil, 297–301
Helmholtz equation, 390
help, see on-line help
hexadecimal number system, 5
high-pass filter, 55–58  
high-resolution graphing  
in MAXIMA, 145–152  
in PYTHON, 31–33, 48–79  
Hooke’s law, 339  
hyperlinks, 521–522

I

IDL, 1  
IDL contact information, 543  
improved Euler method, 244, 253  
indices in \LaTeX, 519  
indicial equation, 239  
inert commands  
in MAXIMA, see suppressing evaluation in MAXIMA  
inhomogenieties in linear equations, 369  
initial value problems, 215, 383  
initialization files  
in MAXIMA, 161  
in PYTHON, 94–95  
integer  
signed, 6  
unsigned, 6  
interactive plotting in PYTHON, 92–93  
interference patterns, see also diffraction patterns  
Fabry-Perot, 102  
from four slits, 103  
internal storage of expressions  
in MAXIMA, 120–121  
interpolation functions, see shape functions  
interpolation points, 306, 307  
interpreting source code, see also Local Guide, 176  
interrupt  
in MAXIMA, 116  
in PYTHON, 16  
ispell, 527–528  
iterative algorithm, 401

K

Kaprekar Constant, 108  
keywords  
in PYTHON, see PYTHON keywords  
Kirchoff’s laws, 379  
Knuth, Donald, 495  
Kronecker delta, 292

L

Lagrange equations  
for 2D harmonic oscillator, 170  
for compound pendulum, 170  
Laguerre polynomials, 168  
Laguerre’s method, see root finding  
Lamport, Leslie, 495  
LAPACK, 1  
contact information, 543  
Laplace transforms, 13–14, 227–228, 278  
definition, 13, 131, 227  
for ODEs, 227  
in MAXIMA, 131–132, 171  
Laplace’s equation, 381, 389, 390  
in 2D, 424–427  
in PYTHON, 441–448  
in pseudocode, 186–188  
in PYTHON, 191–193  
\LaTeX, 1  
accents in, 531  
counters in, 532  
dashes in, 507  
math mode in, 508–509  
quotation marks in, 508  
special characters in, 532  
\latex, 498, 499, 517  
\LaTeX commands  
\textit (italic correction), 504  
\textbackslash (new line), 506, 511  
\textbackslash！ (negative thin space), 509  
\textbackslash (thin space), 509  
\textbackslash- (allow hyphen), 508  
\textbackslash: (medium space), 509  
\textbackslash; (thick space), 509  
\addcontentsline, 518–519  
\addtocontents, 518–519  
\Alph, 510, 533  
\alph, 510, 533  
\and, 524  
\arabic, 510  
\author, 528  
\bf, 528  
\bfseries, 504  
\boldmath, 509  
\cal, 531  
\caption, 511, 512, 514, 517  
\centerline, 514  
\chapter, 508  
\chaptername, 535  
\char, 532  
\clearpage, 505  
\contentsname, 535  
\date, 528  
\definecolor, 521, 525  
\documentclass, 496, 498, 501, 522, 528  
\documentstyle, 496  
\fbox, 532  
\figurename, 534  
\footnote, 508, 528  
\footnotecite, 505  
\frac, 528  
\framebox, 532  
\hspace, 505, 506  
\hspace, 505  
\huge, 505  
\IfFileExists, 525
<table>
<thead>
<tr>
<th>command</th>
<th>page(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>article</td>
<td>501, 508</td>
</tr>
<tr>
<td>book</td>
<td>501, 508</td>
</tr>
<tr>
<td>report</td>
<td>501</td>
</tr>
<tr>
<td>slides</td>
<td>501</td>
</tr>
<tr>
<td>\LaTeX\ environments</td>
<td>506–507</td>
</tr>
<tr>
<td>(\ldots) \ (in-line equation)</td>
<td>508</td>
</tr>
<tr>
<td>[\ldots] \ (displayed equation)</td>
<td>508</td>
</tr>
<tr>
<td>$\ldots$ \ (in-line equation)</td>
<td>508</td>
</tr>
<tr>
<td>array</td>
<td>535</td>
</tr>
<tr>
<td>center</td>
<td>506, 511, 513, 517, 528</td>
</tr>
<tr>
<td>comment</td>
<td>532</td>
</tr>
<tr>
<td>displaymath</td>
<td>508</td>
</tr>
<tr>
<td>document</td>
<td>497, 498, 528</td>
</tr>
<tr>
<td>enumerate</td>
<td>509–510, 532, 533</td>
</tr>
<tr>
<td>eqnarray</td>
<td>508</td>
</tr>
<tr>
<td>equation</td>
<td>508, 528</td>
</tr>
<tr>
<td>figure</td>
<td>512–515, 517</td>
</tr>
<tr>
<td>flushleft</td>
<td>528</td>
</tr>
<tr>
<td>itemize</td>
<td>509–510</td>
</tr>
<tr>
<td>minipage</td>
<td>532</td>
</tr>
<tr>
<td>pict2e</td>
<td>518</td>
</tr>
<tr>
<td>picture</td>
<td>512, 518</td>
</tr>
<tr>
<td>quotation</td>
<td>506</td>
</tr>
<tr>
<td>quote</td>
<td>506</td>
</tr>
<tr>
<td>tabular</td>
<td>510–512</td>
</tr>
<tr>
<td>tabular</td>
<td>510–512</td>
</tr>
<tr>
<td>tikzpicture</td>
<td>515–518</td>
</tr>
<tr>
<td>\LaTeX\ length parameters</td>
<td>502</td>
</tr>
<tr>
<td>\baselineskip</td>
<td>505, 507</td>
</tr>
<tr>
<td>\baselinestretch</td>
<td>507</td>
</tr>
<tr>
<td>\bottomfraction</td>
<td>533</td>
</tr>
<tr>
<td>\evensidemargin</td>
<td>501, 520</td>
</tr>
<tr>
<td>\fleqn</td>
<td>533</td>
</tr>
<tr>
<td>\itemsep</td>
<td>510</td>
</tr>
<tr>
<td>\marginparsep</td>
<td>520</td>
</tr>
<tr>
<td>\marginparwidth</td>
<td>520</td>
</tr>
<tr>
<td>\oddsidemargin</td>
<td>501, 528</td>
</tr>
<tr>
<td>\parindent</td>
<td>501, 528</td>
</tr>
<tr>
<td>\parskip</td>
<td>510</td>
</tr>
<tr>
<td>\parsep</td>
<td>510</td>
</tr>
<tr>
<td>\parskip</td>
<td>501, 510, 528</td>
</tr>
<tr>
<td>\textfraction</td>
<td>533</td>
</tr>
<tr>
<td>\textheight</td>
<td>501, 528</td>
</tr>
<tr>
<td>\textheight</td>
<td>501, 528</td>
</tr>
<tr>
<td>\topfraction</td>
<td>533</td>
</tr>
<tr>
<td>\topmargin</td>
<td>501, 528</td>
</tr>
</tbody>
</table>

**LaTeX Manual**: 495

**LaTeX output**

from MAXIMA, 154–155

**LaTeX packages**: 507

alltt, 531

color, 521, 525

graphics, 512–515, 521–522

hyperref, 521–522, 525

ifthen, 524–525

makeidx, 519–520

makeidx, 519–520

needspace, 535

showidx, 520

tikz, 512, 515–518

**verbatim**: 531–532

**LaTeX page previewers, see also xdvi, yap**: 526–527

**LaTeX preamble**: 501–503

**LaTeX sample**: 497–500, 528–529

**LaTeX special symbols**

" (closing double quotation), 508

--- (em dash), 507

-- (en dash), 507

( (dash), 507

( (minus sign), 508

( (hard space), 507

" (opening double quotation), 508

**LaTeX templates**: 502

**LaTeX type size**: 504

**LaTeX type styles**: 503–504

lawn mower, 374

least squares fitting, 379–381

Legendre polynomials, 165, 166, 312, 332, 376

Lennard-Jones potential, 378

licensing of software, 14

line labels

in MAXIMA, 116

linking object code, see *Local Guide*, 176

lists

in MAXIMA, 119–121, 135–137

in PYTHON, 22–24


logical conditions, see *logical expressions*

**logical expressions**

in **LaTeX**, 524–525

in MAXIMA, 141

in pseudocode, 179–180

in PYTHON, 35–36

**logical operators, see *logical expressions***

long integer

signed, 6

unsigned, 6

loops

in MAXIMA, 124, 128, 129, 139–141

in pseudocode, 181–183

in PYTHON, 34–36

Lorentz distribution, 101, 330, 334

Lorentz lineshape, 45, 46

1p, see UNIX commands

LSODE, 1

LSODE contact information, 543

LU decomposition, 99, 370–371, 406

**M**

machine code, 176

Macintosh OS, 1

**Mac\TeX**

contact information, 544
magnetic field, 290
  in wave guide, 75–78, 106, 196–198
  of circular loops, 50, 104, 203, 297, 335
  of Helmholtz coil, 297–301
  of solenoid, 103
magnetic vector potential, 290
MAPLE, 1
MAPLE contact information, 544
MATHEMATICA, 1
MATHEMATICA contact information, 544
mathematical functions
  in MAXIMA, see MAXIMA functions
  in PYTHON, see PYTHON functions
MATLAB, 1
MATLAB contact information, 544
matrices
  in MAXIMA, 135–137
  in PYTHON, 27–28
matrix multiplication
  in MAXIMA, 136
  in PYTHON, 28
max-min problems, 338
MAXIMA, 1
MAXIMA command files, see also MAXIMA commands, MAXIMA functions,
  141–145
crossdot, 143, 145, 174, 297
MAVectcart, 137–139, 172
MAVectcyl, 173
MAVectsph, 173
testmultiply, 142–143
MAXIMA commands, see also MAXIMA command files, MAXIMA functions
  abs, 126
  algsys, 374
  allroots, 362, 363
  append, 160
  appendfile, 153
  apply, 162
  apply1, 157, 158
  apply1b, 157
  apply2, 157
  apropos, 163
  assume, 163, 294–296, 298, 302
  atvalue, 229, 235, 236
  batch, 141–145
  batchload, 141–145
  bc2, 229, 237
  bf_find_root, 363
  bfallroots, 362, 363
  bffloat, 123
  cabs, 126
carg, 126
closefile, 153
  coefmatrix, 237
  collectterms, 230–231, 239, 241
  combine, 164
  compile, 144, 328, 329
  compile_file, 144
INDEX

limit, 233, 235, 294, 349
linsolve, 371
loadfile, 163
logcontract, 233, 297
makelist, 128, 275, 328, 329, 365, 367
map, 163, 346, 347
matchdeclare, 156–159
matrix, 135, 136
mode DECLARE, 145, 329
multithru, 164
newton, 363–365, 374
nroots, 362–364
num, 127
ode2, 133, 135, 228, 230, 232, 237
partfrac, 125, 171
pickapart, 164
playback, 153
plot2d, 146–148, 161, 233, 234, 236, 275, 299, 304, 328, 329, 347, 365, 368
plot3d, 148–149, 162
polarform, 127
print, 124, 129, 240
product, 163
quad_qag, 325–329
quad_qagi, 325
quad_qags, 325
radcan, 164, 170, 233
ratsimp, 130, 164, 238
realpart, 126, 346
realroots, 362–364
rectform, 127
remfunction, 122, 163
remove, 163
remvalue, 122, 129, 163
reset, 117, 123–125, 127, 128, 130–132, 135, 137, 152, 158
rest, 367
rhs, 230, 231, 233, 235, 237, 238, 346, 347
rk, 274–275
save, 153, 163
set_plot_option, 145
stringout, 153, 163
subst, 125, 133–134, 164, 230–231, 233, 294, 297, 298, 301
subpart, 125, 164
sum, 163, 239, 304
taylor, 131, 232, 241, 294, 297, 299, 301, 302, 348, 349
tellsimp, 158
tellsimpafter, 158
tex, 154–155
tracematrix, 136
translate, 144
translate_file, 144
transpose, 136
trigexpand, 127, 128
trigreduce, 127
trigsimp, 128, 296, 298
ueivectors, 137
uniteigenvectors, 137
writefile, 153
wxplot2d, 146
wxplot3d, 148–149
xthru, 164
zeromatrix, 170
MAXIMA contact information, 544
MAXIMA control structures
block(...), 140, 365
for/step/unless/do, 139–140
for/step/while/do, 139–140
if/then/elseif/.../else, 140–141
unless/do, 141
while/do, 141
MAXIMA examples, 118
MAXIMA functions, see also MAXIMA commands, MAXIMA command files, 119–120, 122
asin, 296
atan2, 126, 127
cos, 120, 121, 127, 128, 134, 135, 145, 157, 158, 162, 296, 298, 329, 347, 365
cosh, 126
cot, 367
exp, 127, 130, 131, 145, 233, 327, 347, 363, 365
factorial, 123
map, 233
sqrt, 123, 129, 133, 294–296, 327, 329, 347, 367
tan, 135
MAXIMA keywords, 119
epsabs, 326
epsrel, 326
limit, 326
MAXIMA logical expressions, see logical expressions in MAXIMA
MAXIMA manuals, 115
MAXIMA notebooks, see MAXIMA workbooks
MAXIMA option variables, 119
algebraic, 164
file_search_demo, 118
file_search_lisp, 141–142, 160–161
file_search_maxima, 141–142, 160–161
file_type_lisp, 141–142
file_type_maxima, 141–142
fpprec, 123
fpprintprec, 123
maxima_tempdir, 145, 153, 154, 159–160
INDEX

in PYTHON, see global variables
natural frequencies
  of spherical cavity, 376
  of stiff bar, 376
  of three-mass system, 375
Navier-Stokes’ equations, 391
Newton’s method, see root finding
Newton’s second law, 285, 286, 339, 341
  in fluids, 393–395
Newtonian fluid, 395
node numbers
  global, 406
  local, 406
nodes, 383
non-viscous fluid flow, 395
normal distribution, see Gaussian distribution
normal modes
  of membrane, 59–68
  of string, 107
normal stress
  in fluids, 393–395
numerical approximation
  for first derivative, 398
  for second derivative, 399
numerical integration, see quadrature
Numerical Recipes, 1
NUMERICAL RECIPES contact information, 545
nybble, 5

O

object code, 176
object-image relations, 102
octal number system, 5
OCTAVE, 1
OCTAVE contact information, 545
on-line help
  in MAXIMA, 117–118
  in PYTHON, 48
ordinary differential equations
  with PYTHON, 250–274
orthogonal functions, 292–293
OzTeX contact information, 545

P

page previewer
  in \LaTeX, 526–527
partial fractions, 125, 171, 277
passwords, see Local Guide
path (in directory tree), 8
path specification
  in MAXIMA, 159
  in PYTHON, 16
PDEs
  classification of, 390–391
PDF files, 499–500
pdfcrop, 522–523
pdfLaTeX, 499, 516, 521–522
pdftops, 523
pendulum, simple, see simple pendulum
pivotal condensation, 370
pivoting, 370
placing text in graphics
  in MAXIMA, see MAXIMA plot options, label
  in PYTHON, see PYTHON commands, matplotlib.pyplot.text
Planck radiation law, 103
Poisson’s equation, 390
polar coordinates
  in MAXIMA, 127, 139
  in PYTHON, 52–53, 68–69
population biology, 220, 232–234, 282
preallocation of arrays
  in PYTHON, 34, 71
predictor-corrector methods, 244
procedure, 184
program, 175
projectile motion, see trajectories
properties
  in PYTHON, see PYTHON keywords
proprietary software, 14
protected variables
  in MAXIMA, see MAXIMA special symbols, MAXIMA system variables
  in PYTHON, see PYTHON special symbols, PYTHON constants
ps2pdf, 500, 517, 522–523
pseudocode comments, 181, 184
pseudocode control structures, 175, 180–184
CASE-END CASE, 181
EXECUTE, 183
FOR-THRU-DO, 182–183, 188–190
FOR-THRU-DO-STEP, 182
IF-THEN-ELSE-END IF, 3, 4, 181, 185, 186
LOOP-END LOOP, 3, 4, 181, 184–186
REPEAT-UNTIL, 182
WHILE-DO, 182
pseudocode directives
  DIMENSION, 177, 188, 190
pseudocommands
  ATTACH, 190
  CLOSE, 190
  READ, 178
  WRITE, 178, 188, 190
py-files, see PYTHON scripts, PYTHON functions
PYTHON, 1
python, 16, 39, 82, 83
PYTHON attributes, 30
  add_subplot, 83, 88, 91
  append, 35, 40, 82, 94
  axis, 90
  clear, 83, 86
vpython
    box, 84
    rate, 85
    sphere, 84
    text, 84
    vector, 84, 85
    zip, 73

PYTHON constants
    False, 17, 35
    math
        e, 22
        pi (π), 22
    None, 98
    numpy
        pi (π), 53, 60, 61, 70, 76, 90, 91, 194,
        198, 313, 314, 317, 321, 324, 356, 358
    sys.path, 94
    True, 17, 35

PYTHON control structures
    def/return, 44–47, 81, 83, 255–257, 259,
        261, 264, 268, 318, 319, 322–324, 354,
        355, 357, 360
    for, 34–35, 40, 71, 82, 86, 94, 99, 192, 194,
        198, 251–252, 263, 266, 268, 270,
        313–316, 322, 324, 356, 358, 361, 401,
        404, 415–416, 418, 428, 429, 435, 440,
        442, 458–460, 462–464
    if/elif/else, 36, 428, 430, 434, 436, 460,
        462–463
    try/except, 98
    while, 35, 39, 82, 85, 98

PYTHON current directory, see current directory in PYTHON

PYTHON data attributes, see also PYTHON attributes, 30

PYTHON data types
    bool, 35
    complex, 28
    dict, 24
    float, 17–18
    int, 17–18
    int32, 25, 30
    list, 17–18, 22–24
    numpy.ndarray, 25
    set, 17–18, 22–24
    str, 17–18, 22–24
    tuple, 17–18, 22–24

PYTHON default directory, see current directory in PYTHON

PYTHON Edit Window, 38–40

PYTHON environment variables
    MATPLOTLIBRC, 96
    PYTHONPATH, 94
    PYTHONSTART, 94–95

PYTHON functions, see also PYTHON commands, PYTHON scripts, 44–47
    cos, 324
    float, 40, 414
    int, 414
    math
        acos, 21
        asin, 21
        atan, 21
        atan2, 21
        cos, 21
        acosh, 21
        erf, 21
        exp, 21
        fabs, 21
        factorial, 21
        gamma, 21
        radians, 21
        sin, 21
        asinh, 21
        sqrt, 21, 22
        tan, 21
        atanh, 21
        max, 404
        min, 404
    numpy
        cos, 45, 50, 53, 54, 68, 76, 81, 87, 88, 90,
            198, 321, 358, 361, 435
        cosh, 32, 54
        exp, 52, 57, 68, 70, 313–317, 357, 358
        max, 418
        min, 418
        sin, 49, 50, 52, 54, 60, 61, 76, 81, 86–88,
            90, 91, 194, 198, 322, 357, 360, 361
        sinh, 54
        sqrt, 44, 57, 70, 262, 313–315, 317, 319,
            361
        pow (exponentiation), 20
        quad, 44
        rings, 324
        round, 20
        sqrt, 324
        user-defined, 44–48

PYTHON instance attributes, see also PYTHON attributes, 30

PYTHON keywords, 18–19, 33
    align, 84
    args, 257, 259, 263, 264, 267, 322–325
    atol, 271
    axis, 37, 82, 361, 445
    azim, 63
    block, 32
    both, 361
    bottom, 361
    capsize, 57
    cmap, 67, 71
    color, 33, 41, 49, 50, 52, 53, 58, 62, 63, 69,
        71, 78, 80, 83, 84, 87, 88, 90, 91, 251,
        256, 257, 263, 264, 268, 320, 322, 324,
        356, 358, 361, 403, 417, 428–429, 433,
        435–436, 439, 465
    colormap, 73, 74
    colors, 64–65, 77, 91
    cstride, 62
INDEX

henon.py, 273
irrad.py, 193–194, 208
laplace.py, 192–193
largamp.py, 273
lineshape.py, 45, 46
lineshape.py, 47
lorenz.py, 273
luplot.py, 45–47
moment.py, 319
onedshm.py, 273
pdens.py, 195, 209–210
planet.py, 260–263, 273
plotradio.py, 41, 109
proj.py, 356, 357
projectile.py, 84–85, 113
qmlower.py, 360
qmshm.py, 266–270, 273
qmupper.py, 360
quadpole.py, 199–200, 212–213
rcfilter.py, 56, 109–110
read_irrad.py, 214
rossler.py, 273
spiralc.py, 321
spirals.py, 322
stdwaves.py, 263–266, 273
stringanim.py, 113–114
stwave.py, 110–111
turnpts.py, 354–355
turnpts.py, 355
twodshm.py, 273
twooscil.py, 273
vandpol.py, 273
wavegd.py, 197–198, 211

PYTHON search path, 94

PYTHON special symbols
\n (new line), 442, 444
'..' (enclose strings), 18, 22
'...' (enclose strings), 18
'''...''' (comments), 16, 47
* (multiplication), 20, 21
** (exponentiation), 20, 21, 50, 60, 61, 70,
106, 262, 401, 427, 434, 458
+ (addition), 20, 21
- (subtraction), 20, 21
/ (division), 20, 21
// (integer division), 20
: (range), 26, 41, 57, 71, 262, 361
; (combine statements), 16, 54
= (assignment), 18
== (equal to), 98, 430, 436, 444
# (comments), 16, 47
\ (continue line), 16, 50, 52
f (in format statements), 191
for expressing conditions, 35
j (complex number), 28–29
\n (end of line), 41, 43, 94, 192, 194
nan (not a number), 29

PYTHON symbol codes, 56

PYTHON system variables, see PYTHON
constants

PYTHON variables

vpython.scene
autoscale, 84
background, 84
height, 84
title, 84
width, 84

Q

Q of resonant circuit, 104

quadratic formula, 44, 129, 344, 345, 374, 375

quadrature, 303–313

adaptive algorithms, 310
assessing error in, 308–309, 313–316
Gauss-Legendre, 313
Gaussian, 310–313, 315–316, 333
iterative algorithms, 310
midpoint rule, 172, 305
Newton-Cotes, 303–309
recursive trapezoidal rule, 307
Romberg integration, 308
roundoff error, see roundoff error
Simpson’s rule, 171, 172, 306, 308, 312, 314,
332

trapezoidal rule, 305, 313
truncation error, see truncation error
with MAXIMA, 325–329
with PYTHON, 313–324

quality factor, see Q of resonant circuit
quantum energy levels, see energy levels
quantum probabilities, 292
in hydrogen atom, 70, 104, 195, 205, 330,
334, 335
of harmonic oscillator, 292, 320, 327, 333

quantum scattering, 167
quantum wave function, see wave function

R

radioactive decay, 40–41, 219, 234–236, 242–244,
251–253, 255–258, 280

radius of gyration
definition of, 287
of cardioid, 331
of semicircular plate, 295
railroad problem, 377
range of projectiles, see trajectories
Rayleigh criterion, 378
reaction kinetics, 282

reading data
from files
in pseudocode, 191
in PYTHON, 40–41, 200–201
from keyboard
in pseudocode, 178
in PYTHON, 97–98
INDEX

recasting
algorithms, 202
control structures, 202
recursion relationship, 239
reflection at dielectric film, 102
relativity
mass/momentum/energy, 102
motion under constant force, 294
relaxation, 187–188
residual, 352
resonance
in LRC circuit, 104, 222, 278
restoring initial state
in MAXIMA, 117, 161
in PYTHON, 96–97
returning to command level
in MAXIMA, 116
in PYTHON, 16
returning to operating system
from MAXIMA, 116
from PYTHON, 16
ReVTeX, 531
Richardson extrapolation, 307, 314
Ritz variational method, 405
Rodriques’ formula, 165, 168
Romberg integration, see quadrature
root finding
accuracy in, 352–353
by bisection, 350, 353–354
by Laguerre’s method, 352
by Müller’s method, 351
by Newton’s method, 351, 354–355, 373–374, 376
by secant method, 351, 354–355
by trial and error, 373
in MAXIMA, 345–349, 361–368, 374
in PYTHON, 353–361, 371, 374
roundoff error, 400–401, 405, 427
in numerical quadrature, 308
in ODEs, 247
rubber lengths in \( \LaTeX \), 501
Ruchardt’s experiment, 381
Runge-Kutta methods, 245–247
running program, see Local Guide

S

saving files, see writing data to files
scalar fields
three-dimensional, 195
in PYTHON, 195
two-dimensional, 193–194
in PYTHON, 193–194
Schrödinger equation, 226–227, 238–239, 266–271, 343, 360–361, 390
search path
in MAXIMA, 142, 160–161
in PYTHON, 94
secant method, see root finding

self-adjoint ODE, 397
separating statements on one line
in MAXIMA, 116
in pseudocode, 180
in PYTHON, 16, 34, 35, 54
series solutions to ODEs, 239–241
sets
in PYTHON, 22–24
shape functions, 405–408
shear stress
in fluids, 393–395
shooting method, 263
sigmoid curve, 220, 235
simple pendulum, 280, 288, 301–302, 334
Simpson’s rule, see quadrature
simultaneous equations
by direct methods, 371
by iterative methods, 371
linear, 369–371
nonlinear, 372–374
singular equations, 369
singular value decomposition, 371
small angle approximation
in MAXIMA, 157–158
solving Laplace’s equation, see Laplace’s equation
solving linear equations, 369–371
in MAXIMA, 129
in PYTHON, 29–30
solving ODEs
with MAXIMA, 228–241, 274–275
with PYTHON, 250–274
sound waves in fluids, 396–397
source code, 176
special symbols
in MAXIMA, see MAXIMA special symbols
in PYTHON, see PYTHON special symbols
speed of sound, 396
spherical coordinates
in MAXIMA, 173
spurious roots, 337, 343, 361, 367
stable equilibrium, 338
standing waves
in cube, 105, 206
in string, 225–226, 237–238, 263–266, 279
Stark effect, 101, 169, 375
startup files, see initialization files
structure of data files, 188–189
subroutines
in MAXIMA, see MAXIMA functions
in PYTHON, see PYTHON functions,
PYTHON scripts
suppressing evaluation
in MAXIMA, 133–134
suppressing output
in MAXIMA, 116
in PYTHON, 22

T
INDEX

tables
  in \LaTeX, 510–512
Taylor series
  for solving ODEs, 245–249
  for solving simultaneous equations, 373
  in MAXIMA, 131, 165, 171
termination of statements
  in MAXIMA, 116
  in pseudocode, 180
  in PYTHON, 16
\LaTeX, 495, 503
\LaTeX contact information, 495, 545
\LaTeX output
  from MAXIMA, see \LaTeX output
texdoc, 515, 536
\TeXlive, 1
text editors, see Local Guide
text in graphics, see placing text in graphics
TGIF, 1
TGIF contact information, 546
thermal equilibrium, 104
thin lens equation, 102
tikz, 515–518
tikz commands
  \definecolor, 516
  \draw, 516–517
  \foreach, 516, 517
  \node, 516, 517
  \x, 516, 517
trajectories, 285
double pendulum, 277
earth falling into sun, 330, 333
exponentially decaying force, 330
in central forces, 223–225, 260–263, 280, 281
in constant magnetic field, 87, 162, 204
in Coulomb field, 278
in crossed E and B, 281
in viscous medium, 216–219, 230–232, 278,
  374
simple harmonic oscillator, 329
under constant force, 294, 329
transmission at dielectric film, 102
transpose
  in MAXIMA, 136, 143
  in PYTHON, 28
trapezoidal rule, see quadrature
tree-structured directories, 8
tridiagonal systems, 371, 380
truncation error
  global, 249
  in midpoint formula, 309
  in numerical quadrature, 308
  in ODEs, 247, 252–253
  in Simpson’s rule, 309
  in trapezoidal rule, 309, 313
  per step, 248
truth table, 179
tuples
  in PYTHON, 22–24
turning points, 292, 320, 333, 338, 345–346,
  354–355, 364, 377, 378
two’s complement notation, 6
two-way selection
  in MAXIMA, 140–141
  in pseudocode, 181
  in PYTHON, 36
U
UNIX, 1
UNIX commands
  aspell, 527–528
  cleanbak, 536
  cleanTEX, 536
  dvipdfm, 500
  dvips, 499
  ispell, 527–528
  latex, 498, 499
  lp, 500
  ls, 523
  pdfcrop, 522–523
  pdflatex, 499
  pdftops, 523
  ps2pdf, 500
  python, 16
  setenv, 95
  texdoc, 515
  xdvi, 500
UNIX shell scripts
  ceps2pdf, 523, 541
  cleanbak, 536
  cleanTEX, 536
  cps2pdf, 523, 541
  rdfileeps, 523, 541
  rdfileps, 523, 541
UNIX tutorial, see Local Guide
unstable equilibrium, 338
user accounts, see Local Guide
user-defined commands, see command files
user-defined functions
  in MAXIMA, see MAXIMA functions
  in PYTHON, see PYTHON functions
V
Van der Pol oscillator, 280
variable names
  in MAXIMA, 119
  in pseudocode, 176
  in PYTHON, 17–18
vector arrays
  three-dimensional, 199–200
  two-dimensional, 196–198
vector calculus
  with MAXIMA, 137–139
vector fields
three-dimensional
  in PYTHON, 199–200
two-dimensional
  in PYTHON, 197–198
vector identities, 174
velocity field
  of rotating object, 206
vertices, 383
vibrating string, 80–82
vim, 1
viscosity, 395
viscous damping, 217, 220

W

wave equation
  in 1D, 385–387
  in PYTHON, 427–433
  in 2D, 419–422
  in 3D, 387–388
wave function, 226–227, 266–271, 292, 331, 333
weighting functions, 408
Wheatstone bridge, 166
Windows, 1
Windows batch files
ceps2pdf.bat, 523, 540
cleanbak.bat, 536
cleanTEX.bat, 536
cps2pdf.bat, 523, 540
dfileeps.bat, 523, 540
dfileps.bat, 523, 540
Windows commands
cleanbak, 536
cleanTEX, 536
dir, 523
dvipdfm, 500
dvips, 499
latex, 498, 499
pdfcrop, 522–523
pdflatex, 499
pdfops, 523
print, 500
ps2pdf, 500
python, 16
set, 95
setx, 95
texdoc, 515
yap, 500
Winedt, 1
Winedt contact information, 546
writing data to files, 203–206
  in pseudocode, 188–191
  in PYTHON, 42–43, 193–195, 197–200
writing data to screen
  in pseudocode, 178
  in PYTHON, 20–24

X

xdvi, 500, 526–527
Xemacs, 1
Xemacs contact information, 546
Xv contact information, 546

Y

yap, 500, 526–527