A First Course in Scientific Computing

Symbolic, Graphic, and Numeric Modeling Using Maple, Java, Mathematica, and Fortran90

Fortran Version

RUBIN H. LANDAU

Fortran Coauthors:

KYLE AUGUSTSON
SALLY D. HAERER

PRINCETON UNIVERSITY PRESS
PRINCETON AND OXFORD

Contents

List of Figures

Preface

This book contains an introduction to scientific computing appropriate for all lower-division college students. Its goal is to make students comfortable using computers to do science and to provide them with tools and knowledge they can utilize throughout their college careers. Its approach is to introduce the requisite mathematics and computer science in the course of solving realistic problems. On that account care is given to indicate how each discipline uses its own language to describe the same concept, how their tools are useful to us, and how computations are often concrete examples of abstract ideas.

This is easier said than done. On the one hand, lower-division students are simultaneously learning elementary mathematics and physics, and so this may be the first place they encounter the science and mathematics used in the problems. On the other hand, in order for the tools and techniques to be useful for more than the assigned problem, we give *more* than an introduction (the original title of this book) to the computational tools. We address the first issue in our teaching by reminding the students that our focus is on having them learn the techniques in the proper context, and that any new science and mathematics they become familiar with will make it easier for them in their other courses. We address the second issue by placing an asterisk * in the title of chapters and sections containing optional materials and by reminding the students of which sections are most appropriate for the problem at hand.

This book covers some of the basics of computation, numerical analysis, and programming from a computational science point of view. We want the reader to acquire some ideas of what is possible with computers, what type of tools there are for it, and how to go about getting all the pieces to work together. After that, it is easy to use on-line help or the references to get more details. As a result, our presentation is more practical and more focused on mathematics and science than an introductory programming or computer science text, with minimal discussion of computer science theory. The book follows our own personal preference for "just enough" computer information in that it avoids going through every option for every command and instead presents realistic examples.

We follow the dictum that science and engineering students learn computing best while sitting down at a computer in a trial-and-error mode. Hence, we adopt

a tutorial approach in which readers work along with us in solving a problem, learn by doing, and then work on their own version of the problem (there are also additional exercises at the end of each chapter). We use the command-line mode for the compiled languages that makes the tutorial as universal as possible, and, we believe, is better pedagogically. It then follows that this book is closer to a workbook than a reference book. Yet because one always comes back to find worked examples of commands, it should be valuable for reference as well.

A problem solving environment such as *Maple* or *Mathematica* is probably the easiest way to start scientific computing, is natural to use with trial and error, and is what we do in Part 1. Its graphical interface is friendly, it shows the user a wide spectrum of what can be done with modern computation, such as symbolic manipulations, 2-D, 3-D visualization and linear algebra, and is immediately useful in other courses and for writing technical reports. After the first week with Maple or Mathematica, students with computer fright usually feel better. These environments also demonstrate how computers can give an immediate response in beautiful mathematical notation, or fail at what should be the simplest of tasks.

Part 2 of the text is on Java (or Fortran). We believe that learning to program in a compiled language teaches more of the basics of computation, gets closer to the actual algorithms used, teaches better the importance of logic, and opens up a broader range of technical opportunities (jobs) for the students than the use of a problem solving environment. In addition, compiled languages also tend to be more powerful and flexible for numerically intensive tasks, and students naturally move to them for specialized projects. Likewise, after covering Parts 1 and 2 of the text it may make sense for a student to use environments like *Matlab*, which combine elements of both compiled languages and problem solving environments.

Many students may find that the logic and the precision of language required in programming is more challenging than anything they have ever faced before. Others find programming satisfying and a natural complement to their study of mathematics and foreign languages. We try to decrease the slope of the learning curve by starting the neophytes with sample programs to run and modify, rather than requiring them to write all their own programs from scratch. This process is more exciting, saves a great deal of time otherwise spent in frustrating debugging, and helps students learn by example.

Even though it might not be evident from all the hype about Java and Web computing, Java is actually a good language for beginning science students. It demands proper syntax, produces useful error messages, is consistent and intelligent in handling precision, goes a long way towards being computer-system independent, has Sun Microsystems providing free program-development environments [SunJ], and runs fast enough for nonindustrial purposes (its speed is increasing, as are the number of scientific subroutine libraries being developed in Java).

Part 3 of the text provides a short LaTeX survival guide. A number of colleagues have suggested the need for such materials, and since using LaTeX is quite similar

to compiling a code, it does make a useful extension of the text. Even though we do not try to reveal the full power and complexity of LaTeX, we do give enough of its basic elements for the reader to write beautiful-looking scientific documents.

Depending on how many chapters and modules are used, this book contains enough materials for a one- or two-semester course. Our course has one lecture and two labs every week, with roughly one instructor for every 10 students in lab. Attending lectures and reading the materials before lab are important in acquainting the students with the general concepts behind the exercises and in providing a broad picture of what we are trying to do. The supervised lab is where the real learning occurs.

We believe that a modern student should be acquainted with several approaches to scientific computing. Notwithstanding our avowed claim that there are multiple paths leading to good scientific computing, we have had to make some choices as to what to place in the printed version of this book and what to place on the CD. The basic ideas behind scientific computing are language independent, yet the details are not. For all these reasons we have decided to cover Maple, Java, and LaTeX in the printed version of this book, but to place other languages on the CD that comes with the text.

The CD is platform independent and has been tested on Windows, Macs, and Unix. The Java and Fortran programs are pure text. The Maple and Mathematica files, however, require the respective programs to execute them. The pdf files will require Abode Acrobat, which is free. Any difficulties with the CD should be reported to rubin@physics.orst.edu. Additions and corrections to the CD are found on our Web pages (http://www.physics.orst.edu/~rubin/IntroBook) or through Princeton University Press Web pages (http://pup.princeton.edu/titles/7916.html).

Specifically, the CD contains Java programs, LaTeX files, data files, and various supplementary materials. As indicated, it also contains Maple worksheets with essentially identical materials as in the Maple section of the text but in an interactive format that we recommend over reading the paper version. Furthermore, the CD contains essentially identical materials to the Maple tutorials as Mathematica notebooks. These can be read with Mathematica or printed out as an alternative version of the text. Likewise, the CD also contains the Java materials of the text converted over to Fortran90, as well as the appropriate Fortran programs. Though we do not recommend trying to learn two languages simultaneously, having alternative versions of the text does present some interesting teaching possibilities. Additions and corrections to the CD are found on our Web pages.

Acknowledgements

This book was developed on seven year's worth of students in the introductory scientific computing class at Oregon State University. The course was motivated by the pioneering text by Zachary [Zach 96] and encouraged by [UCES]. The

course has been taught by Albert Stetz, David McIntyre, and the author; I am proud to acknowledge their friendship and the inclusion of their materials in the text. I have been blessed with some excellent student assistants without whose efforts the course could not be taught and the materials developed; these include Matt Des Voigne, Robyn Wangberg, Kyle Augustson, Connelly Barnes, and Juan Vanegas. Valuable materials and invaluable friendships have also been contributed by Manuel Páez and Cristian Bordeianu, coauthors of our *Computational Physics* text, and Sally Haerer. They have helped make this book more than the introduction it would have been otherwise. I look forward to their continued collaborations.

Financial support for developing our degree program in computational physics and associated curricular materials comes from the National Science Foundation's Course, Curriculum, and Laboratory Improvement program directed by Duncan McBride, and the Education, Outreach and Training Thrust area of the National Partnership for Computational Infrastructure, under the leadership of Gregory Moses and Ann Redelfs. The courses and materials have benefited from formative and summative assessments by Julie Foertsch of the LEAD Center of the University of Wisconsin. This work has also benefited from formative comments by various reviewers and colleagues, in particular Jan Tobochnik and David Cook. Thanks also goes to my editor, Vickie Kearn at Princeton University Press, who has been particularly insightful, encouraging, and courageous in helping me develop this multidisciplinary text and having it turn out so well, and to Ellen Foos for her caring production of the book. And Jan Landau, always.

RHL

Chapter One

Introduction

1.1 NATURE OF SCIENTIFIC COMPUTING

> *Computational scientists solve tomorrow's problems with yesterday's computers; computer scientists seem to do it the other way around.*
>
> —anonymous

The goal of scientific computing is problem solving. The computer is needed for this, because real-world problems are often too difficult or complex for analytic or human solution, yet workable with the computer. When done right, the use of a computer does not replace our intellect, but rather leverages it by providing a super-calculating machine or a virtual laboratory so that we can do things that were heretofore impossible.

The mathematical modelling and problem-solving orientation of scientific computing places it in the discipline of *computational science*. In contrast, computer science, which studies computers for their own intrinsic interest, provides the underpinning for the development of the hardware and software tools that computational scientists use. As illustrated on the left of Figure 1.1, computational science is a *multidisciplinary* field that combines a traditional discipline, such as physics or finance, with computer science and mathematics, without ignoring the rigor of each. Books such as this, which employs materials from multiple fields, aim to be the central bridge in this figure, connecting and drawing together the three fields.

Studying a multidisciplinary field is challenging. Not only must you learn more than one discipline, you must also work with the separate languages and styles of the different disciplines. To illustrate, a computational scientist may be pleased with a particular solution because it is reliable, self-explanatory, and easy to run on different computers without modification. A computer scientist may view this same solution as lengthy, inelegant, and old-fashioned. Each may be right in the sense that they are making judgments based on the differing values of different disciplines.

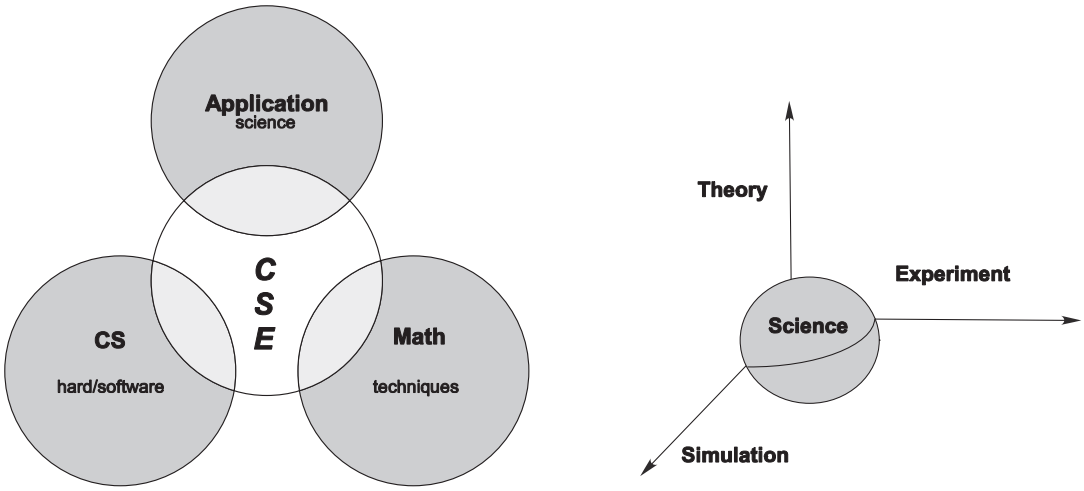


Figure 1.1 *Left:* Computational science is a multidisciplinary field that combines science with computer science and mathematics. *Right:* A new paradigm for science in which simulation plays as essential a role as does experiment and theory.

Another, possibly more fundamental, view of how computation is playing an increasingly important role in science is illustrated on the right of illustrated in Figure 1.1. This symbolizes a paradigm shift in which science's traditional foundation in theory and experiment is extended to include computer simulation. While one may argue that the future will see the CSE on the left of Figure 1.1 get absorbed into the individual disciplines, we think all would agree that the simulation on the right of Figure 1.1 will play an increasing role in science.

1.2 TALKING TO COMPUTERS

As anthropomorphic as your view of computers may be, it is good to keep in mind that a computer always does exactly as told. This means that you should not take the computer's response personally, and also that you must tell the computer exactly what you want it to do. Of course programs can be so complicated that you may not care to figure out what they will do in detail, but it is always possible in principle. Thus it follows that a basic goal of this book is to provide you with enough understanding so that you feel well enough in control, no matter how illusionary, to figure out what the computer is doing.

Before you tell the computer to obey your orders, you need to understand that life is not simple for computers. The instructions they understand are in a *basic machine language*[1] that tells the hardware to do things like move a number stored in one memory location to another location, or to do some simple, binary

[1]The "BASIC" (Beginner's All-purpose Symbolic Instruction Code) programming language should not be confused with basic machine language.

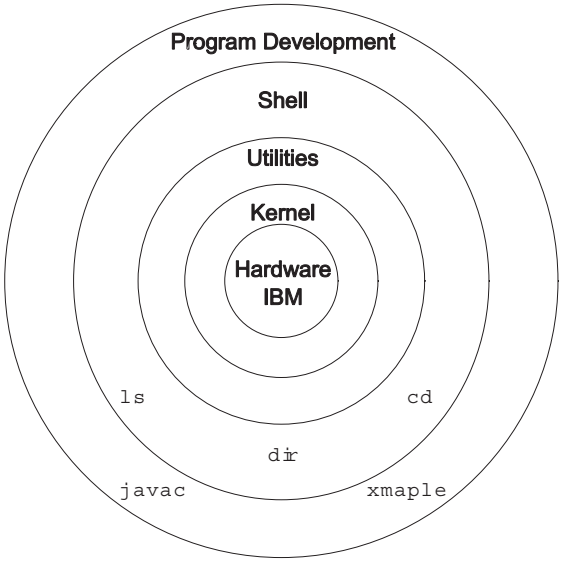


Figure 1.2 A schematic view of a computer's kernel and shells.

arithmetic. Hardly any computational scientist really talks to a computer in a language it can understand. Instead, when writing and running programs, we usually talk to the computer through a *shell* or in a *high-level language*. Eventually these commands or programs all get translated to the basic machine language.

A *shell* is a name for a command-line interpreter, that is, a place where you enter a command for the computer to obey. It is a set of medium-level commands or small programs, run by the computer. As illustrated in Figure 1.2, it is helpful to think of these shells as the outer layers of the computer's *operating system*. While every general-purpose computer has some type of shell, usually each computer has its own set of commands that constitute its shell. It is the job of the shell to run various programs, compilers, and utilities, as well as the programs of the users. There can be different types of shells on a single computer, or multiple copies of the same shell running at the same time for different users. The nucleus of the operating system is called, appropriately, the *kernel*. The user seldom interacts directly with the kernel, but the kernel interacts directly with the hardware.

The *operating system* is a group of instructions used by the computer to communicate with users and devices, to store and read data, and to execute programs. The operating system itself is a group of programs that tells the computer what to do in an elementary way. It views you, other devices, and programs as input data for it to process; in many ways it is the indispensable office manager. While all this may seem unnecessarily complicated, its purpose is to make life easier for you by letting the computer do much of the nitty-gritty work that enables you to think

higher-level thoughts and communicate with the computer in something closer to your normal, everyday language. Operating systems have names such as *Unix, OSX, DOS,* and *Windows*.

In Part 1 of this book we will use a high-level *interpreted* language, either Maple or Mathematica. In Part 2 we will use a high-level *compiled* language, either Java or Fortran90. In an interpreted language the computer translates and executes one statement at a time into basic machine instructions. In a compiled language the computer translates an entire program unit all at once before executing the statements. Compiled languages usually lead to faster programs, permit the use of vast libraries of subprograms, and tend to be portable and re-usable. Interpreted languages appear to be more responsive, interactive, and, consequently, more "user friendly."

When you submit a program to your computer in a compiled language, the computer uses a *compiler* to process it. The compiler is another program that treats your program as a foreign language and uses a built-in dictionary and set of rules to translate it into basic machine language. As you can imagine, the final set of instructions is quite detailed and long, especially after the compiler has made several passes through your program to translate your convoluted logic into fast code. The translated statements ultimately form an *executable* code that runs when loaded into the computer's memory.

1.3 INSTRUCTIONAL GUIDE

Landau's Rules of Education: Much of the educational philosophy applied in this book is summarized by these three rules:

1. Most of education is learning what the words mean; the concepts are usually quite simple once you understand what you are being told.
2. Confusion is the first step to understanding.
3. Traumatic experiences tend to be the most educational ones.

This book has an attitude, and we hope you will develop one too! We enjoy computing and relish the increased creativity and productivity resulting from powerful computing tools. We believe that computing has become part of the fabric of science and that an introductory scientific computing course should be part of every lower-division university student's education. Hence we deliberately mix the languages of mathematics, science, and computer science. This mix of languages is how modern scientists think about things, and since ideas must be communicated with these same words and ideas, this is how the book is written. However, we are sensitive to the confusion multiple definitions may reap and do provide a section at the end of each chapter indicating some key words and concepts, and a

glossary at the end of the book defining many technical terms and jargon.

Because we aim to give an introduction to computational science, we cover some basics of numerical analysis, information about how data are stored, and the concordant limits of computation. However, we present very little discussion of hardware, computer architecture, and operating systems. That is not to say that these are not interesting and important topics, but rather that we want to get the reader busy computing and acquiring familiarity with concrete examples. In our experience, science and engineering students need this practical experience before they can appreciate the more abstract principles of computer science.

We have aimed our presentation at first- and second-year college students. We believe that the chapters and sections not marked optional with an asterisk * provide a good introductory course for them. The inclusion of the optional chapters would raise of the level of the presentation and lead to a longer course. To maintain the logical organization of materials, we have intermixed the optional chapters with the others. However, there should be nothing in the optional chapters that is required in order to understand the chapters that follow.

It is hard to keep up with the rapid changes in computer technology and with the knowledge of how to use them. That is not such a big issue with scientific computing, because it is the basic principles of mathematics, logic, science, and computing that are important, and not the details of hardware and software. Nevertheless, students and faculty often do not enjoy computing because of the frustration and helplessness they feel when computers do not work the way they should. Although we commiserate with those feelings, one of the things we try to teach is that it is not unusual to have new things not work quite right, but that if you relax and follow a trial-and-error approach, then you usually find success working around them.

Be warned, there are some exercises in this book that give the wrong answer, that lead to error messages, or that may "break" the program (but not harm the computer). Learning to cope with the limits of computation gets easier and less traumatic after you have done it a few times. We understand, however, that many instructors may not appreciate a computer's failure that they cannot explain, and so we have assembled an *Instructor's Survival Guide*, available upon request through the Web.

It is important that students become familiar with the material in the text before they come to lab. A lecture helps, but reading and working through the materials is essential. Even though it may not be that hard to work through the problems in lab by having instructors and fellow students prompt you with the appropriate commands to enter, there is little to be gained from that. Our goal is to make this introductory experience very much a "lab" that develops an attitude of

experimentation and discovery and that nurtures rewarding feelings when a project finally computes correctly.

Many of the problems we assign require numerical solution and therefore are not covered in elementary texts. These include nonlinear oscillators, the motion of projectiles with drag, and the rotation of cubes about an arbitrary axis. Notwithstanding our prediction that other elementary courses and texts will eventually be more multidisciplinary, some readers may feel that we are requiring them to understand phenomena at a level higher than in their other courses. Our hope is that the readers will recognize that they are pioneers, will be stimulated by their new-found powers, and will help modernize their other courses.

On the technical side, we have developed the materials with Maple 6–9.5, Mathematica 4.2, and the Java Development Kit (JDK or Java 2 Standard Edition, J2SE) [SunJ]. Even though studio and workbench environments are available, we prefer the pedagogical value in using a shell to issue separate commands for compilation and execution, and then having to deal with the source and compiled files directly. In addition, many students are able to load JDK onto their home computers and work there as well.

We have found that *WinEdt* [WinEdt] and *TextPad* work well to edit and run source code on a Windows platform, and that *Xemacs* [Gnu] with Java tools is excellent for Unix/Linux machines. In addition, *jEdit*, the Open Source programmer's editor [jEdit], is an excellent tool that, because it itself is written in Java, runs on most any platform. Visualization is very important in computational science. It is built into Maple and Mathematica and is excellent. Part of the power of Java is that it has strong graphical capabilities built right into the language, although calling them up is somewhat involved. Consequently, we have adopted the free, open source package *PtPlot* [PtPlot] as our standard approach to plotting with Java, and *Dislin* for Fortran 90. However, *gnuplot* [Gnuplot] and *Grace* [Grace] are also recommended as stand-alone applications. For 3-D graphics, a more specialized application, we give instructions on using *gnuplot* and refer the interested reader to *OpenDx* [DX] and *VisAd* [Visad]. These too are free, powerful, and available for Unix/linux and Windows computers.

1.4 EXERCISES TO COME BACK TO

Consider the following list of problems to which a computational approach could be applied. Indicate with an "M," "J," or "E" whether the best approach would be the use of Maple, Java, or either.

1. calculate the escape velocity from Jupiter
2. write a spreadsheet (accounting) program from scratch
3. solve problems from a calculus textbook

4. prove an algebraic identity
5. determine the time required for a sky diver to reach terminal velocity
6. write a compiler for a programming language

PART 1

MAPLE OR MATHEMATICA BY DOING
(see Text or CD)

PART 2

FORTRAN BY DOING

Chapter Nine

Getting Started with Fortran

9.1 ANOTHER WAY TO TALK TO A COMPUTER

As we have seen in our work with Maple, a computer is an incredibly powerful device that can be incredibly helpful too when it does what you intend. Maple is a fairly easy way to tell the computer what to do, and that is why we started with it; you enter a command and Maple responds. If Maple cannot understand your command, or if its response is not what you like, then you just keep changing the command until all is well. In this line-by-line mode with a response after each command line, Maple is acting as an *interpreter*. Having the immediate response of an interpreter is useful, and when it is combined with a friendly graphical user interface (GUI), as in Maple, we have a powerful environment for scientific work.

To provide you with a broader experience in scientific computing than is possible with Maple, we now look at a different approach to computing. Rather than entering commands one at a time and getting an immediate response to each, we shall instead create a file that contains all the commands we want the computer to perform, and then we will give this entire *program* file to the computer for processing in one fell swoop. Handling all your commands at once, however, requires the help of a *compiler*, that is, a special system-level program which converts a user-level program file (written in a specific syntax understood by the compiler) into another equivalent binary file that the computer can process and execute.

Overview of a compiler

To solve a numerical problem, we create a distinct, unambiguous set of instructions using a pre-defined syntax (or grammar) specifically designed for the computer language we want to use (Fortran in this case). We enter and store these instructions in a file which is commonly referred to as the *source file* or *source code*.

A special system program called the *Fortran compiler* translates our source code and creates an equivalent binary set of instructions designed for only the computer to read and understand. This step is referred to as the *compile* step. This binary file is called the *object file* or the *object module*.

This object file is then linked with other required system-level library binary files to produce the final *executable* file (also called the *load module*). This resulting file can now be loaded into the computer's memory and executed, whereby each instruc-

tion that we originally designed to solve our numerical problem is actually performed by the computer.

Once this compiling process is completed and the executable file has been generated, the program instructions can be performed many times without re-entering these instructions or re-compiling the program. It is likely that we have different sets of input data which require the use of our program's instructions. Creating a file once containing the needed instructions to solve a numerical problem, and then using that file multiple times thereafter, is a clear advantage.

However, if changes are required to improve the program or add new features and calculations, then we can edit the source file appropriately (just the parts that need improvement), recompile this newer source file to produce a new executable file, then execute (or *run*) our program using various input data as needed.

To understand the value of a compiled language, imagine telling a story to friends at party in which you were restricted to speaking only one sentence at a time, and could not continue your story until someone else responded to each of your sentences. While you might be able to get through your story, you could probably weave a better tale if you did not have to stop all the time. In particular, without interruptions you might be able to weave some very interesting and satisfying logical connections between the different parts of your story. Also, the story telling would progress much more rapidly without all the interruptions.

Likewise it is with a compiled approach to computing. While it may take more skill and time to weave a long program, you can make it do some very powerful things. If you are good, you can make it do exactly what you want, in exactly the way you want it done. And if the program gets used more than once (with different data), you do not have to repeat the process.

Just as you probably have found it rather rare to be able to enter many Maple commands without making at least one error, it will also be rather rare that you can write a long program perfectly on the first try. However, there are many rewards to be had by trying, and once you get it right it can be made to really zip along. But not to worry, we will start you with programs to modify rather than making you start from scratch.

We have decided to introduce you to Fortran as your first compiled language (although we have a version of essentially identical material using Java in the paper book, which we encourage you to read as well). We have picked Fortran for a number of reasons [F90_CSEP, F90_UL, ChapF 04]:

1. Fortran is widely used in the scientific computing community, especially for large projects. The name itself stands for "Formula Translation" and it was designed specifically for the solution of complex, numerically challenging problems.

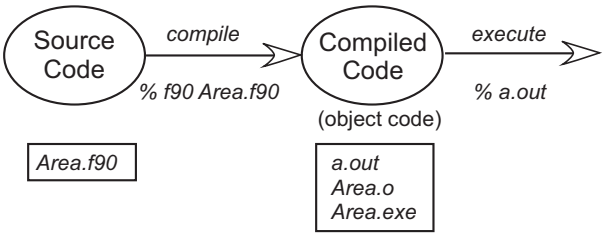


Figure 9.1 Steps in compiling and executing.

2. Fortran code tends to be simple for humans and compilers to understand. You may not be able to do everything with it that you can with C and Java, but this also means less trouble.
3. Many computing packages are available for Fortran.
4. Fortran is probably the fastest, and most robust computing languages for scientific computing, and especially for matrix computation.
5. Fortran90 is a version of the Fortran compiler which has been modernized to contain features, such as object-oriented programming, coercion (numeric polymorphism), and dynamic memory management, that were heretofore missing. Yet old Fortran programs still compile and run properly using the Fortran90 compiler.
6. Fortran90 contains primitive data types for matrix and complex objects which permit the user to perform matrix manipulations and computations without having to deal with the individual components. It thus becomes easy to scale up a program for parallel processing.
7. Fortran90 contains data-parallel capabilities which lead to easier and less error-prone parallelization of code.

9.2 FORTRAN PROGRAM PIECES

Whether simple or complex, developing Fortran programs requires the two basic elements indicated in Fig. 9.1. The first is a *source file* that you have written using Fortran commands and has a filename ending with `.f90`. The second is the binary, executable file that the computer generates, or "compiles," using the Fortran90 compiler. It may have the name ending with `.o`, or `.exe` (Windows), or be called `a.out` (Unix).

For instance, on the CD you will find a simple program `Area.f90` which calculates the area of a circle, and with which we will experiment. As briefly explained earlier, there are a number of processes involved in order to convert (translate) `Area.f90` to something closer to what a computer can understand and run. Here are some facts and terms to help further clarify the needed process:

- You enter a command to *compile* the source program. You must be in an

operating-system's *shell* (a command-line interpreter) to enter commands. You know that you are "in" a shell by the prompt, which may contain a `>` or a `%`, or something containing the name of the directory/folder in which you are located, all depending upon the operating system on your computer.

- The Fortran90 compiler is brought into action with the `f90` command from the command line:
  Under Unix:

```
> cd Courses/Week5                    Change dir. to where source resides
> f90 Area.f90                        Run Fortran compiler on Area.f90
```

  Under Windows (Start/Run/command):

```
> cd Courses\Week5                    Change dir. to where source resides
> f90 Area.f90                        Run Fortran compiler on Area.f90
```

  When we illustrate commands like above, the first character (or field of characters) is the shell's prompt, the second field, in **`monospaced bold type`**, is the command that you enter, and the third field (justified to the right) contains notes of explanation which should not be entered.
- For these commands to work, you must issue the `f90` command from within the directory in which the file `Area.f90` is stored[1]. If the compilation was successful (you get no error messages), the Fortran compiler will have written another file in the same directory, named either `Area.exe` on Windows, `a.out` on Unix, or possibly `Area.o`, where the `.o` indicates "object file." As mentioned before, this compiled program generated by the F90 compiler is what is called the *load module* or *executable file*. This file contains the binary commands which can be loaded into the computer and then executed.
- A compiled program is actually produced in a number of stages. First, the source code is compiled into a binary file which lacks some of the things needed to be complete (such as math library routines). The second stage is called *linking*, where one or more object files are combined with needed libraries to produce the final executable. This executable will run on your computer, or on one with similar hardware, without having to invoke the compiler again. In contrast, the compiled code generated by Java can be executed on almost any computer, but the Java run-time package must be installed in order to do so, and it runs more slowly.
- The last step of the process in Fig. 9.1 is *executing* or *running* your program. This is done by entering the name of the resulting executable file (depending upon your operating system) at your operating system's command line prompt. Here is an example from Unix:

```
> a.out                               Execute the file called a.out
```

[1] If you are using a programming or studio environment package such as *Code Warrior*, some of this will be done automatically for you, or maybe you will have to push a button. Nevertheless, it's a good idea to know what files you have created.

On some Unix machines, the system may not be set up to look for executables in your current directory, in which case you will need to change to the directory (`cd directory_name`) containing your executable, or you can simply provide its full pathname instead.

9.3 ENTERING AND RUNNING YOUR FIRST PROGRAM

As a general rule, you do not learn how to compute just by reading about it. To understand a program you must get it running and giving correct answers. So before we try to explain what is inside a Fortran program, let us get one running.

In Lst. 9.1 is the Fortran source code for the program `Area.f90` that calculates the area of a circle. The line numbers on the left are there to help us explain the program to you; they are not part of the program and should not be entered. The Fortran statements in the middle are the instructions the computer needs to process. The text after the ! characters is a *comment* for explanation purposes that does not have to be entered, but should be for explanation and future reference. When the compiler encounters a "!", it ignores everything on that line following the !.

Listing 9.1 Area.f90

```
1  ! Area.f90: Calculates the area of a circle, sample program
2  ! ------------------------------------------------------------
3  Program Circle_area                          ! Begin main program
4    Implicit None                              ! Declare all variables
5    Real*8 :: radius, circum, area             ! Declare Reals
6    Real*8 :: PI = 3.14159265358979323846      ! Declare, assign Real
7    Integer :: model_n = 1                     ! Declare, assign Ints
8    print *, 'Enter a radius:'                 ! Talk to user
9    read *, radius                             ! Read into radius
10   circum = 2.0 * PI * radius                 ! Calc circumference
11   area = radius * radius * PI                ! Calc area
12   print *, 'Program number =', model_n       ! Print program number
13   print *, 'Radius =', radius                ! Print radius
14   print *, 'Circumference =', circum         ! Print circumference
15   print *, 'Area =', area                    ! Print area
16 End Program Circle_area                      ! End main program code
```

• Use a *text editor*[2] to enter `Area.f90` into a file named `Area.f90` in the directory/folder set aside for your work. By "enter" we mean actually type the program from scratch rather than *cut and paste* it into a file. You may view this as mindless clerical work, but reading and looking at what you are entering helps make this foreign language seem not quite so alien.

[2]An editor is a program like *Notepad* on a Windows PC, or *nedit* or *Xemacs* on Unix. Word processing programs like *Word* or *Word Perfect* are possible, but you must remember to save the file as *Text* to avoid control characters that will confuse the compiler.

• After you have entered the entire program, `save` it into a file named `Area.f90`. Make sure that this file is in the proper directory for your personal work.

• Open a *shell* from which you may enter commands, change to the directory where you have saved `Area.f90`, and *compile* it:

```
> f90 Area.f90                                   Get help if this doesn't work
```

• We remind you: the `>` here is the computer's operating system prompt to you, the command you enter is in **`bold monospaced font`**, and the last field of text is our note of guidance or explanation (not to be typed by you).

• In order to stop Fortran from yelling at you when you compile the program, you may have to correct some silly typographical errors made when you entered the program. To correct an error, go to your editor (it's a good idea to keep it open in a separate window while you compile in the shell's window), and change what's there so that it agrees with what we have in the preceding `Area.f90` program. Then save `Area.f90` in the same directory where you are doing the compiling.

• If compilation is successful, you will not be told much (you have to do something wrong to get noticed). However you can check that the object code, `Area.exe` or `a.out`, was created by listing all the files, including creation times:

```
> ls -l                                          Long list, Unix/Linux command prompt
> dir                                            List, Windows command prompt
```

• Make sure that you have saved and closed your `Area.f90` file. (Saving writes a copy to the hard disk for storage, and closing removes it from the grasp of the editor.) The executable file (`Area.exe` or `a.out`) is a binary-type file and is not meant for human comprehension; it is not recommended that you try to view it. What you would see would not be illuminating and may even mess up your editor.

• If you do not like the name `a.out` for your executable, you may use the `-o` option to specify your preference:

```
> f90 Area.f90 -o Area.o                         Name your object file Area.o
```

• To see how Fortran responds to not doing things correctly, be adventurous and try below some unconventional variations on the theme:

```
> f90 Area                                       A forgotten .f90 after the filename
> f90 AREA.F90                                   Capitalized the filename; may be OK in DOS
> f90 Area.f90 -o                                No object filename given
```

• Check again that you have created the executable version of your Fortran program; that is, ensure there is a file `a.out` or `Area.exe` or whatever you may have named it:

```
> ls -l                                          List files in Unix/Linux
> dir                                            List files in DOS
```

• As we see from Fig. 9.1, once we have the compiled code in the file `Area.exe` or `a.out`, we get the computer to execute it by simply typing in its file name:

```
> Area                                           Execute compiled Area.exe (Windows)
> a.out                                          Execute compiled a.out (Unix)
```

• Note that Windows does not require the `.exe` extension to be entered, although it's OK if you do. Now that you have a program that is running, check that it is actually doing what you intended by entering various values for the radius. In particular, try entering a value of 10 for the radius at the prompt written by your program:

```
Enter a radius:
10
```

As you know, the program should tell you that the circumference of your circle is equivalent to 20 π, and that the area is equivalent to 100 π.

9.4 LOOKING INSIDE AREA.F90

We now look at `Area.f90` to get some idea of how a program goes about its business. Though a simple program, it is not trivial in that it contains a number of important parts. Admittedly it is hard to understand at first all that is going on; be patient, after seeing them a few times it begins to make more sense.

`! Comments` - At the top of `Area.f90` is an exclamation mark, some information about the program, and a line of dashes. All the stuff after `!` on a given line is a comment placed there as information for the user, and not processed by the Fortran. This means you can write whatever you want after exclamation marks without affecting computation. We encourage the use of comments and blank lines to make code clearer to you or another programmer using or improving your code. Nevertheless, we are frugal with comments and blank lines in our examples in order to spare printed space.

Program Circle_area - This line declares the name of the program to be `Circle_area`, and begins the program. The name of the program must be different from the names of any variables used in the program, so be sure to specify a name which is descriptive and unique. If we had simply used `area` as the name for the program, an error would be generated by the Fortran compiler because `area` is declared on line 5 as a variable. If the program were named `Area` or `AREA`, the compiler would still complain because Fortran 90 treats all code as if it were one case; it does not distinguish between uppercase and lowercase letters. In fact, the statements in file `Area.f90` could be typed entirely in uppercase and would execute no differently

from the version in lowercase. Actually, the only difference would be that the quoted text printed by the computer would be displayed on the screen in all capital letters since this text is taken literally.

The `Program` line is where the execution of the program begins. The body of the main program is made up of all the lines of code between the `Program` statement (line 3) and the `End Program` statement (line 16). The code lines between `Program` and `End Program` are executed in order, one at a time.

Implicit None - The `Implicit None` statement on line 4 indicates that there are no implicit variables, and so all variables must be declared before they can be used. This is not a required command but it is highly recommended as a good way to catch errors (like misspellings). Without `Implicit None`, variables with names starting with `i, j, k, l, m, n` are automatically declared as integers (whole numbers), and all other variables are automatically declared as reals (numbers with fractional components called *floating-point* numbers). All variables must be declared immediately following the `Implicit None` statement. This means you can always look up to this area in your code to find a list of all variables used in the program.

Real*8 :: radius, circum, area - Before you use a variable in Fortran, you should *declare* its name and data type. Our use of `Implicit None` requires you to do this. The first declaration line in this program lists the variables `radius, circum` and `area` to be of type `Real*8`, that is, double precision. The next line declares the variable `PI` to be of type `Real*8`, and simultaneously assigns a specific value to it. The last declaration line states the variable `model_n` to be an `Integer`. (We will describe data types in Chapter 10).

Notice that in each of these lines there is first a data type, `Real*8` or `Integer`, followed by the double colon `::` separator, and then a series of one or more variables which will be stored as the stated data type. This is the basic Fortran syntax for variable declaration statements. Multiple variables can be placed on a single line, although it is probably best to add additional declaration lines rather than have one line so long that its end is hidden beyond the edge of the screen.

print *, 'Enter a radius:' - Line 8 uses the `print` command to write a message to the screen. When a number follows the `print` statement, it indicates the label on a special `format` statement used to customize the format in which output lines are printed. When an asterisk `*` is used instead of a format label number (as in this case), it indicates that no format statement will be used, but instead we will let the compiler decide the format of the lines to be output. Since format statements contain significant complexity, we will save that for later and simply use this easier `*` method. In this statement, the comma is a separator. The part inside the single quotes is a *string* of characters which is printed out to a new line on the screen.

Anything that is inside matching pairs of single or double quotes is understood by the compiler to be a character *string*. You can think of a string as a variable which contains standard English characters and letters. Fortran makes no effort to assign symbolic or numerical meaning to strings, but instead stores and outputs them as pure text. In this case, we are using the `print` command in order to inform the person running the program that he/she should enter a number. The number itself is retrieved by the `read` command on the following line.

read *, radius - This `read` command captures text entered from the keyboard and assigns it to the variable `radius`. When the program runs, a text input cursor is displayed on the screen and the program waits for the user to enter some text and press Return. Since `radius` is a `Real*8` (double-precision) variable, the characters you enter will be converted to a number before being stored in the variable. If the user of the program is uncooperative and enters some alpha text instead of a number, the program will generate an error and terminate (commonly called a *crash* in computer lingo).

circum = 2.0 * PI * radius - Lines 10 and 11 are *assignment* statements. They compute the numerical value of the expression on the right side of the = sign, and then assign that value to the variable on the left side. Here, we use the word "assign" to mean store a numerical value in the memory location reserved for that variable. This is the same as the assignment operator `:=` in Maple.

Arithmetical operations in Fortran are denoted by the standard +, -, * and / symbols. However, the symbol ^, often used for exponentiation, is not used in Fortran; instead, the symbol ** is used for exponentiation. In this example, we squared the variable `radius` by simply entering `radius * radius`, although we could have used `radius**2` just as easily.

print *, 'Program number =', model_n - This writes a string and a variable to a single line on the screen. After the asterisk, the variables to be printed should be separated by commas.

End Program Circle_area - The last line in `Area.f90` is the matching `End Program` statement for the `Program` statement on line 3. The `Program` and `End Program` statements always form a pair.

9.5 KEY WORDS

assignment statement	compiled code	object	comments
compile	execution	linking	shell
executable	real	program	print
source file/code	data type	text editor	read
string	main	running	

9.6 SUPPLEMENTARY EXERCISES

1. Modify `Area.f90` into a new program `Volume.f90` which calculates the volume of a liquid in a spherical tank of radius R, when the liquid is a height H above the bottom of the tank. We have already studied this problem with Maple in Chapter 8, *Searching, Programming; Dipsticks*, where we derived that the volume $V = \pi H^2(R - H/3)$. Make sure to test your program for $H = 0$, R, and $2R$, that is, for an empty, half-full, and full tank.
2. Explain in just a few words:
   a. what the Fortran compiler does
   b. what the Fortran linker does
   c. how Fortran differs from an operating system
   d. what a shell is

Chapter Ten

Data Types, Limits, Procedures; Rocket Golf

10.1 PROBLEM AND THEORY (SAME AS CHAPTER 3)

Michele, our golf fanatic, is still in pursuit of the world's record for the fastest golf ball. Recall, she hits her golf balls from a rocket moving to the right with a velocity $v = c/2$ relative to observer Ben on earth. Michele hits her drive with a speed of $U = c/\sqrt{3}$ at an angle $\theta = 30^o$ with respect to the moving rocket. She observes her drive to remain in the air for a (hang) time $T' = 2.6 \times 10^7$ seconds.

1. How would Ben, watching Michele's drive from the earth, describe the golf ball in terms of its speed, angle ϕ and hang time T?
2. How would the answer to 1. change if Michele hit her ball to the left, that is, in a direction opposite to the rocket's velocity?
3. If Michele hit the ball with a speed $U = c$, how would the answers change?

The formulas we need from special relativity relate the description of the same moving object as it appears to observers in different frames. If Michele in O' sees her golf ball having velocity components:

$$U' = (U'_x, U'_y), \tag{10.1}$$

then Ben in O, who sees Michele moving to the right with velocity v, will see her golf ball move with velocity components:

$$U = (U_x, U_y). \tag{10.2}$$

The two sets of components are related by the following:

$$U_x = \frac{U'_x + v}{1 + vU'_x/c^2}, \qquad U_y = \frac{U'_y}{\gamma(1 + vU'_x/c^2)}, \qquad \gamma = \frac{1}{\sqrt{1 - v^2/c^2}}. \tag{10.3}$$

10.2 FORTRAN'S PRIMITIVE DATA TYPES

Before we begin to calculate with Fortran, it is in your best interest to explore the limitations of its numerical capabilities. The limitations arise from the schemes

Name	Type	Bits	Bytes	Range & Precision
`LOGICAL`	logical	1	-	`true` or `false`
`CHARACTER`	string	16	2	'\u0000' $\leftrightarrow$ '\uFFFF' (ISO Unicode)
`INTEGER`	integer	32	4	$-2,147,483,648 \leftrightarrow +2,147,483,648$
`REAL`	floating-point	32	4	$\pm 1.401298 \times 10^{-45} \leftrightarrow \pm 3.402923 \times 10^{+38}$
`DOUBLE PRECISION` or `REAL*8`	floating-point	64	8	$\pm 4.94065645841246544 \times 10^{-324} \leftrightarrow \pm 1.7976931348623157 \times 10^{+308}$

Table 10.1 Fortran's basic data types and their sizes in bytes (B)

computers use to store information. As you have seen in `Area.f90`'s statements containing `Integer` and `Real*8`, Fortran is specific about the types of variables or data with which it deals. Variables are stored as specific data types. The data types and the amount of memory they occupy are described in Table 10.1. These basic types are rather standard for most computer languages. Variables of these types are stored in the computer's memory in small blocks of memory locations called *words*. The length of a word depends upon the memory structure of the computer you are using. For most computers, one word is represented as a group of 4 bytes (or 32 bits), as will be discussed in the next section. One of the chores you have in programming is deciding how much precision you need your variables to have. So, to be precise about *precision*, we need to have a word about *words*.

10.3 INTEGERS

The most elementary unit of memory is, effectively, a little magnet that is either charged or not (imagine a compass needle pointing up or down). If we associate "down" with the numeric digit 0 and "up" with the digit 1, then we have a physical device that symbolically stores 0's and 1's. It should then be no great surprise to hear that all numbers on the computer are ultimately represented in *binary* form, that is, in terms of the binary digits (abbreviated *bits*) 0 and 1.

Just like the digits in the *decimal* system indicate the number of times that each 10^0, 10^1, 10^2, $\ldots$, is contained in a numeric value, so it is with the binary system, except that the base is 2 and not 10. To demonstrate, in Table 10.2 we see the three bits abc representing the eight numbers from 0 to 7. Likewise, N bits are used to represent integers (whole numbers) up to 2^N. In practice within the computer's numeric storage scheme, the first bit is used to represent the sign of the integer; therefore, we effectively lose one bit. This means that it is possible to represent integers only up to 2^{N-1} with N bits.

Long strings of binary zeros and ones are fine for computers, but are awkward for people. For example, decimal 75 is 001011 in binary. Consequently, binary strings are represented instead as *octal* or *hexadecimal* numbers, which are compact. The problem with our beloved *decimal* number system is that it is not one-to-one compatible with the binary number system, since decimal is based on

Binary abc	$= a \times 2^2$	$+b \times 2^1$	$+c \times 2^0$	= Decimal
000	0	0	0	0
001	0	0	1	1
010	0	1	0	2
011	0	1	1	3
100	1	0	0	4
101	1	0	1	5
110	1	1	0	6
111	1	1	1	7

Table 10.2 The use of the three binary digits (bits) a, b, and c, to represent the eight decimal integers from 0 to $2^3 - 1 = 7$.

the power of 10 and not the power of 2. Octal (2^3) or hexadecimal (2^4) are both easily and effectively translated directly to binary since they share 2 as their base. Therefore, computer memory schemes for storing numerical values are based on either octal or hexadecimal representations of binary, depending upon the computer design. In practice, the most common memory scheme is based on *hexadecimal* (2^4), using 4 bits to represent one hexadecimal digit. Then, only during the output phase, values are converted to decimal for display to humans.

The point of all this discussion about what goes on in the "guts" of memory is to provide a brief taste of some of the complexities that drive computer design. Computer texts delve into these details for chapters. The main point for you to extract is that in one way or another, you must generally understand how many bits will be required to store the values of each of your variables. The number of bits is called *word length* and is often expressed in *bytes*, where a byte is a "mouthful of bits":

$$1\,\text{byte} \equiv 1\,\text{B} \stackrel{\text{def}}{=} 8\,\text{bits}. \tag{10.4}$$

Conventionally, storage size is measured in bytes or kilobytes (K). For this reason, while only one bit is adequate to store a *binary* digit 0 or 1, two or more *bytes* are required to store a decimal system value, or a letter, or a special character. In practice, 8 bits or 1 byte are more than adequate to store an integer value, in which case the integers lie in the range 1–2^7 or 1–128. The more usual practice of computer design, however, is to utilize 4 bytes (4 B = 32 bits = 1 word) to store integers. This allows for larger integers which means that the maximum positive integer is $2^{31} \simeq 2 \times 10^9$. In spite of what may seem a very large range for values, it really is not so large when compared with the range of values encountered in the physical world. To illustrate, the ratio of the size of the universe to the size of a proton is 10^{40}. Though integers have limited range, they are stored exactly on the computer if they fall within this range; otherwise, some precision (in the form of significant digits) is lost.

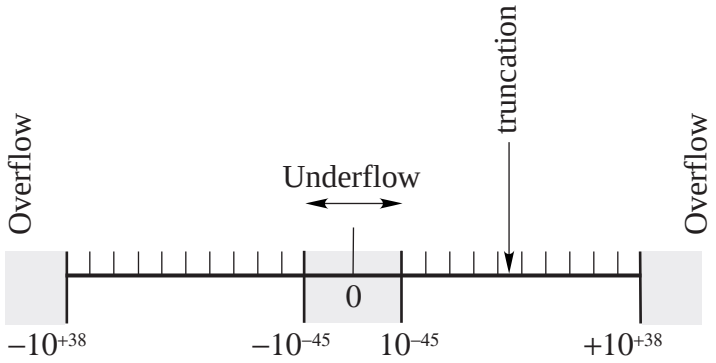


Figure 10.1 A schematic representation of the limits of single-precision floating-point numbers, and the consequences of exceeding those limits.

10.4 FLOATING-POINT NUMBERS

Scientific work primarily uses *floating-point* numbers, which are called `Reals` in Fortran. In contrast to `Integers` that are whole numbers, floating-point numbers have fractional components. The number x is stored as a sign, a mantissa, and an exponent:

$$x_{\text{real}} = (-1)^s \times \text{mantissa} \times 2^{\text{exp}} \; . \tag{10.5}$$

Here, the mantissa contains the *significant digits* of the number, s is the sign bit, and the exponent permits very large, as well as very small, numbers[1]. A single-precision 32-bit word may allocate 8 bits of computer memory for the exponent in (10.5), which leaves 23 bits for the mantissa and 1 for the sign. This 8-bit "exponent" has the range $[-127, 128]$. In practical terms, this means that single-precision (4-byte numbers) has 6-7 significant digits of precision (1 part in 2^{23}) and magnitudes typically in the range

$$2^{-149} \simeq 10^{-45} \leq \text{single-precision} \leq 2^{128} \simeq 10^{38} \; . \tag{10.6}$$

These ranges are represented schematically in Fig. 10.1. If you write a program declaring *double-precision*, then 64-bit words (8-bytes) will be used in place of the 32-bit (4-byte) words. With 11 bits used for the exponent and 52 for the mantissa, double-precision numbers have about 16 decimal places of precision and typically have magnitudes in the range

$$10^{-324} \leq \text{double-precision} \leq 10^{308} \; . \tag{10.7}$$

[1]In practice, a number called the *bias* is subtracted from the exponent `exp` so that the stored exponent is always positive.

10.5 MACHINE PRECISION

One consequence of computers using the floating-point representation to store numbers is that these numbers are stored with the limited precision demonstrated above. Examine again the schematic representation of this in Fig. 10.1, where you are to imagine floating-point numbers being stored along the vertical hash marks. If you call for a number that lies between the hash mark, the computer will move you to the closest number. The corresponding loss of precision is called *truncation error*.

The exact precision obtained in a calculation depends on the details within the program. For a single operation, single-precision usually yields 6-7 decimal places for a 32-bit word, and double-precision 15-16 places. To understand how *machine precision* affects calculations, what would you guess as the result of the simple addition of two single-precision numbers: $7 \; + \; 2.0 \times 10^{-8}$?

a) 7.00000002 b) 9×10^{-8} c) 7 d) 7.02

The correct answer is c). Because there is no more room left to store the significant digits in 2.0×10^{-8}, they are lost or truncated from the answer and the addition yields 7.

This loss of precision is categorized by defining the *machine precision* ϵ_m as the maximum positive number that may be added to the number stored as 1 without changing the number stored as 1 on the computer:

$$1_c + \epsilon_m = 1_c \; . \tag{10.8}$$

Here, the subscript c is a reminder that this value is the number stored in the computer's memory. Likewise, x_c, the computer's representation of an arbitrary number x, and the actual number x, are related approximately by

$$x_c \simeq x(1+\epsilon), \qquad |\epsilon| \leq \epsilon_m \; . \tag{10.9}$$

So remember, $\epsilon_m \simeq 10^{-7}$ for single-precision 32-bit words and $\epsilon_m \simeq 10^{-16}$ for double-precision 64-bit words.

10.6 UNDER- AND OVERFLOWS

If a single-precision number x is larger than 2^{128}, an *overflow* occurs. If x is smaller than 2^{-149}, an *underflow* occurs. We visualize this in Fig. 10.1 by observing that any number that tries to get closer to 0 than 10^{-45} in magnitude falls in the gray region at the center, and leads to an underflow condition. Likewise, any number whose magnitude exceeds 10^{+38} falls in the gray region at the ends of the line, and leads to overflow. The number x_c that the computer stores may result in what is called `NAN` (not a number), or it may be a non-computable `infinity`, or it

may be zero. Because underflow is a loss of information, a scientific programmer must be sensitive to its occurrence. Because the only difference between the representations of positive and negative numbers on the computer is the one sign bit, the same considerations hold for negative numbers.

In our experience, serious scientific calculations almost always require double-precision (8B or 64-bits). And if you need double precision in one part of your calculation, you probably need it all over, including double-precision library routines. If you want to be a scientist or an engineer, learn to say "no" to single-precision numeric representations if there is any question about significant accuracy.

Numerical Constants

In Fortran, a numerical constant is represented by a fixed value such as `1.0, -2.5, -54, 203.`, or `956.47`. When a numerical constant is used without a decimal point in a program, it is assumed by Fortran to be an `integer`; if it includes a decimal point, it is assumed to be a `real` number (even if it is simply `2.`, which is not followed by a `0` or a fractional part). It is important for you to understand this because calculations are done differently for integers and reals. If it is required for a constant to be *double-precision*, you must append `_8` (or `d0`) to the constant, as in `1.5_8` or `1.5d0` for a double-precision `1.5` value. When either the `_8` or `d0` is not present, the Fortran compiler will assume *single-precision*. You may actually want to append `_4` (or `e0`) after the number to clearly demonstrate and document that the value is stored as *single*-precision.

Within the calculations of two *floating-point* numbers, like `1.0/x` (assume `x` to be `real*8` for this discussion), most Fortran compilers will adjust the precision of the results to that of the single-precision `1.0`. For mixed precision with *integers* and reals, like `1/x`, Fortran will generally do the calculation in *real* arithmetic. For best results, do not count on rules or assumptions about mixed precision, but rather be consistent within formulas so that you are certain of the precision level. For instance, when double-precision is needed, be explicit and use something like `1.0_8/x`. So, although it is perfectly fine to use constants in Fortran programs, you should be specific about their *type* so that you are in control of how the calculations will be executed. For simplicity, some of our sample programs are not specific regarding the precision of constants and thereby let the compiler's assumptions take effect.

10.7 EXPERIMENT: DETERMINE YOUR MACHINE'S PRECISION

Lst. 10.1 contains the small, but significant, program `Limits.f90` for you to "test drive" during your "break in" period with Fortran. By repeatedly comparing $1+\epsilon_m$

to 1 as ϵ_m is made smaller, `Limits.f90` demonstrates the machine precision by showing when ϵ_m becomes too small to impact the calculation's precision. Notice that in the program, ϵ_m is called `epsilon_m` and it is compared to a variable called `one` defined as having the initial value of 1.

Listing 10.1 Limits.f90

```
1 !Limits.f90:  Determines machine precision
2 ! ------------------------------
3 Program Limits
4   Implicit None
5   Integer :: i, n
6   Real*8 :: epsilon_m, one
7   n=60            ! Establish the number of iterations
8   ! Set initial values:
9   epsilon_m = 1.0
10  one = 1.0
11  ! Within a DO-LOOP, calculate each step and print.
12  ! This loop will execute 60 times in a row as i is
13  !   incremented from 1 to n (since n = 60):
14  do  i = 1, n, 1                 ! Begin the do-loop
15    epsilon_m = epsilon_m / 2.0   ! Reduce epsilon_m
16    one = 1.0 + epsilon_m         ! Re-calculate one
17    print *, i, one, epsilon_m    ! Print values so far
18  end do                          ! End loop when i>n
19 End Program Limits
```

1. The code within the lines `14` to `18` form what is called a Fortran *do-loop*. This is merely code that is repeated over and over (*looped*) until a certain condition finally ends it. There are several syntax versions of *do-loops*, and they will be explained in more detail in Chapter 12. Briefly for now, the variable `i` in this example is continually incremented by `1` from the initial value of `1` until `i` finally exceeds the last value, `n`. At this point, the loop terminates and the statements after `end do` are executed.
2. Enter `Limits.f90` by hand into a file of the same name, compile it, and run it to determine the machine precision ϵ_m for `double-precision` on your computer. Once you understand how this program works, you may want to modify it so that it starts closer to the final answer, or so that it runs longer in order to get to the final answer. As is true for many of the programs we give you, they are meant as models for you to modify, extend, and experiment.
3. Modify `Limits.f90` to determine the machine precision ϵ_m for single-precision numbers. To modify it correctly, you need to change the phrase `Real*8` to `Real or Real*4` in the declaration statements.
4. **Underflow**: Modify this program to determine the smallest positive single-precision numbers that Fortran handles. *Hint:* Continuous division of `epsilon_m` by 2 will determine the smallest number within a factor of 2. So, modify the line `one = 1.0 + epsilon_m` to a division and increase the value of `n` so the

program repeats for more iterations.

5. **Overflow**: Modify this program to determine the largest positive single-precision numbers that Fortran handles. *Hint:* Rather than divide by 2, multiply by 2.
6. Determine the underflow and overflow limits (smallest and largest numbers) for double-precision.
7. Modify this program to determine the largest and most *negative* integers. *Hint:* Change `epsilon_m` to an integer (`Integer`) and keep subtracting 2 to determine the most negative integer; add 2 to determine the most *positive* integer.
8. Determine Fortran's value for $3(1/3)$ using double-precision values for 1 and 3, and then integer values for 1 and 3. Explain your result for the integer version.

10.8 NAMING CONVENTIONS

All computer languages have syntax rules as to how to name variables. The rules of the Fortran language are actually very simple:

1. No more than 31 characters.
2. A letter must be the first character, any remaining characters must be either a letter, a number, or the underscore.
3. Names are not case-sensitive.
4. Fortran does not have specified reserved words, that is, words that are restricted from use as variable names.

Although these are the only syntax rules for naming variables in Fortran, we suggest some conventions to help prevent errors and make your programs clearer. Deviate from these conventions at your own risk!

- Write regular variables in lowercase letters. Even though a variable may start with a lower case letter, it may contain capital letters within, such as `hiMass` and `loMass`, to aid understanding by programmers. Recall that variable names are not case sensitive, so `iLoveYou` and `iLOVEYou` reference the same thing.
- We have seen `PI` spelled with all capital letters. If the value of variable does not change as the program runs, then it is a *constant* and is often denoted by all capital letters.
- Although there are no formal *reserved words* in Fortran, there are many *keywords*. Keywords are commands and references like `PRINT, WRITE, FORMAT, REAL, INTEGER, COMPLEX, IF, ELSE, DO, PROGRAM, CALL, USE, FUNCTION, SUBROUTINE, MODULE, END`, and many more. As per the rules, you can use these as variable names, but experience suggests that this is a *bad* idea.
- Names should be meaningful and indicate the variables they represent. For

program readability and future modification, time spent initially naming variables appropriately is time saved during the life of the code.

10.9 MATHEMATICAL (INTRINSIC) FUNCTIONS

Our physics description of projectile motion used the trigonometric functions $\cos\theta$ and $\sin\theta$. While Fortran itself has no knowledge of trigonometry (in contrast to Maple or Mathematica), it does have a whole bunch of mathematical functions built into it. As opposed to user-defined functions, the math functions are *intrinsic* and can be used without any preface, loading or defining:

Listing 10.2 Math.f90

```
1 ! Math.f90: demo some Fortran math functions
2 ! ------------------------------------------
3 Program Math_test             ! Begin main program
4   Real*8 :: x = 1.0, y, z    ! Declare variables x, y, z
5   y = sin(x)                  ! Call the sine function
6   z = exp(x) + 1.0            ! Call the exponential function
7   print *, x, y, z            ! Print x, y, z
8 End Program Math_test         ! End main program
```

Here is a table of all of Fortran's Math functions:

Math	Fortran	Math	Fortran	Math	Fortran
$\sin\theta$	`sin(x)`	$\cos\theta$	`cos(x)`	$\tan\theta$	`tan(x)`
$\sin^{-1}\theta$	`asin(x)`	$\cos^{-1}\theta$	`acos(x)`	$\tan^{-1}\theta$	`atan(x)`
$\tan^{-1}(y/x)$	`atan2(y, x)`	e^x	`exp(x)`	$\ln x$	`log(x)`
$\lvert x\rvert$	`abs(x)`	$\sqrt{x}$	`sqrt(x)`	$\log_{10}(x)$	`log10(x)`
remainder	`mod(x, y)`	$\max(x,y)$	`max(x,y)`	$\min(x,y)$	`min(x,y)`
next int	`ceiling(x)`	previous int	`floor(x)`		
nearest int	`nint(x)`	nearest `Real` int	`anint(x)`		

In most cases these functions take various argument types, such as `Real*4` or `Real*8`, and return the same variable type as their value. The functions `abs`, `cos`, `exp`, `log`, `sin`, and `sqrt` also work with complex numbers (such as `Complex*4` or `Complex*8`):

```
a = sin(x**2)          This gives a = sin(x^2)
b = log(abs(x))        This gives b = ln|x|, i.e, natural log
```

While the built-in mathematics functions can be counted on to give accurate answers, they do not know everything about mathematics, and will give error messages or non-numeric answers if pushed too far:

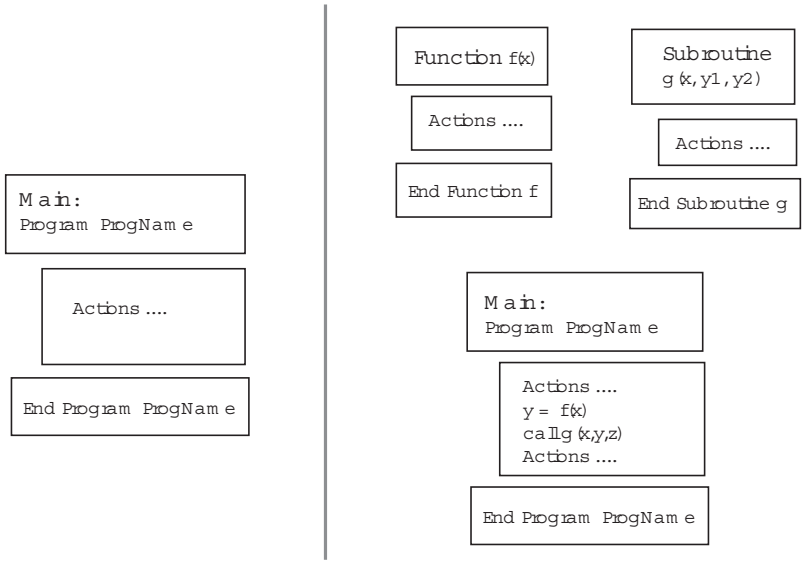


Figure 10.2 *Left:* The structure of a Fortran program with no procedures. *Right:* The structure of a Fortran program with a main program, a subroutine, and a function. The main program and the two procedures are contained in the same source file, for instance, `ProgName.f90`.

Exercise: Take `Math.f90` and modify it to evaluate:

$$\sqrt{-1}, \qquad \cos^{-1}(2), \qquad \log(0)\ . \tag{10.10}$$

10.10 USER-DEFINED PROCEDURES: FUNCTIONS AND SUBROUTINES

In some ways it makes sense to do all the computing in the main program (the code within the `Program` and `End Program` block). Since the compiler looks to this main program as the place to begin executing your code, you may as well put your work there. For small programs this is fine since you can probably just look at the code inside the `Program` code block to figure out everything that is happening; however, programs get longer and more complicated as they are called on to do more work. It is much easier to write and understand long programs when they are divided into smaller parts, each having its own specific task. Also remember that the presence of documentation, comments embedded in the program code, grows more important as the length of the code increases.

So far we have only examined Fortran programs that do all of their work in the main program. We have placed all needed variables and code between the `Program` and corresponding `End Program` statements. This idea is shown on the

left of Fig. 10.2. Just as Maple and Mathematica permit the user to define their own functions, so does Fortran. We will demonstrate and study two Fortran constructs that allow this capability: the `Function` and the `Subroutine`. Both functions and subroutines, jointly called *procedures*, act as mini-programs. They can perform calculations, print out results, and even invoke other subroutines or functions. As indicated on the right side of Fig. 10.2, in the simplest setup, this function `f` and subroutine `g` are placed right in the same source file with the main program; they can actually occur before or after the main program. In other programming models that we will discuss later, related procedures are grouped together into *modules*, or they can be created as separate programs called *external procedures*.

The concept that each basic task (or set of related tasks) can be created in a separate code unit is called *modular programming*. In good modular design, the main program is the control unit, or administrator of the entire program. Like a good boss, it calls upon others to get the work done without doing any of the work itself. In this approach, you get a good idea of what an entire program does just by reading through its main program steps, without worrying about the details of the actual calculations contained in each of the separate units or *procedures*.

The major difference between functions and subroutines is the number of values that get returned to the program calling them. A `function` has only one value returned, and it is the value of the function. A `subroutine` may have a number of values returned, and they are returned via the argument list of the subroutine. We now look at each of these types of procedures separately.

Functions

Lst. 10.3 gives an example of a small program that contains a user-defined function $f(x, y) = 1 + \sin(x\,y)$:

Listing 10.3 Function.f90

```
! Function.f90: Program calls a simple function
! ----------------------------------
Real*8 Function f(x,y)
  Implicit None
  Real*8 :: x, y
  f = 1.0 + sin(x*y)
End Function f
!
Program Main
  Implicit None
  Real*8 :: Xin=0.25, Yin=2., c, f  ! declarations (also f)
  c = f(Xin, Yin)
  write(*,*) 'f(Xin, Yin) = ',c
End Program Main
```

In this example, the function is defined in the top half of the source file and the main program is on the bottom. Although the order does not matter within the rules of Fortran syntax, it is common to have the main program last (after all procedure units are defined and listed). Notice that our function example has two arguments in the main program (`Xin` and `Yin`) that are *passed* to the function procedure and represented locally by the variables `x` and `y` that are used to compute the function value `f`. Any number of arguments are permitted to be passed to the function procedure. These values are used in this function to compute the value of `f`, and then that value is returned to the calling program as the value of `f` in the main program. This is basically how functions work, and we will provide more examples to examine.

Exercise: Compile and execute `Function.f90` and record the results. ♠

It is important to note how `f` is *called* (invoked) with the arguments `Xin` and `Yin` in main, yet defined with the arguments `x` and `y` internally in `function f(x,y)`. Calling a function with differently named arguments outside the function is permitted because the variable names within the function `f` are defined locally for that procedure (dummy variables in the math sense). However, it is required that the argument *types* agree in the main program and in the function since they reference the same corresponding memory locations. When the program is executed, the values stored in the memory locations represented by these variables are used. In addition, notice how line 11 in main declares `f` as `Real*8`, as though the function were a local variable, while line 3 declares the function to be `Real*8 Function f(x,y)`; these parts must also agree.

Subroutines

In addition to functions, Fortran procedures include *subroutines*. Lst. 10.4 gives an example of a small program that contains a user-defined subroutine.

Listing 10.4 Subroutine.f90

```
1  ! Subroutine.f90: Demonstrates the call for a simple subroutine
2  ! ------------------------------------------------------
3  Subroutine g(x, y, ans1, ans2)
4    Implicit None
5    Real(8) :: x, y, ans1, ans2 ! Declare variables
6    ans1 = sin(x*y) + 1.        ! Use sine intrinsic func.
7    ans2 = ans1**2
8  End Subroutine g
9  !
10 Program Main_program          ! Demos the CALL
11   Implicit None
12   Real*8 :: Xin=0.25, Yin=2.0, Gout1, Gout2
13   call g(Xin, Yin, Gout1, Gout2)  ! Call the subr g
```

```
14   write (*, *) 'The answers are: ', Gout1, Gout2
15 End Program Main_program
```

Subroutines can do everything that functions do, and more. For example, `Subroutine.f90` takes the same arguments as `Function.f90` as input, but then *returns* two answers back to the main program. The results from a subroutine are not returned as values represented by the subroutine's name (like with functions), but rather are assigned as the values of the corresponding arguments (variables). In our example, the values of the variables in `Main_program` (`Xin`, `Yin`, `Gout1`, and `Gout2`) are shared with the variables in `Subroutine g` (`x`, `y`, `ans1`, and `ans2`), respectively, that is, is the same in both. Another important difference from functions is how subroutines are invoked. Unlike functions that are used like variables that take arguments, subroutines are *called*, as you see on line 13.

Exercise: Compile and execute `Subroutine.f90` and check that the first answer is the same as that obtained with `Function.f90`. Check that `ans2` is the square of the `ans1`. ♠

If you use a procedure (function or subroutine) with an argument other than the type for which it was defined, you should get an error message or the wrong answer. Try it: replace the main program in `Function.f90` with this one which attempts to evaluate `f(i)` for an `Integer` value `i`, and compile and run it:

```
1 Program bad_program      ! Calls f with the wrong type
2   Implicit None
3   Real*8 :: Yin=2., f    ! Declares all variables
4   Integer :: i           ! Wrong argument TYPE
5   do i = 1, 10
6     print *, 'f(', i, Yin ') =', f(i,Yin)   ! i is wrong
7   end do
8 End Program bad_program
```

The possible outcomes include:

1. *The Fortran compiler gives an error.* This is good because evaluating `f` with an `Integer` rather than `Real*8` argument is the wrong match with the function definition.
2. *The program gives the correct answer.* This indicates that you have a non-standard compiler, which means your codes will not be portable. In this case, you may want to be more careful with the matching of argument types, or start using *modules* or *interfaces* (discussed in Chapter 18) as these do the desirable type checking for you.
3. *The program crashes or gives incorrect results.* Crashing is OK since it lets you know there is an error. A wrong answer without an error indication is

always a problem (since you may not *know* it is the wrong answer). Again, you may want to be more careful with the matching of argument types.

If you really do need to evaluate the function with an integer argument, a correct approach would be to create a temporary variable of the correct type (`real*8`), and then use it in the function call in this fashion:

```
Program good_program        ! Calls f with right type
  Implicit None
  Real*8 :: Yin=2.0, f
  Integer :: i
  Real*8 :: u               ! Temporary variable
  do i = 1, 10
    u = i                   ! Assign value to temporary
    print *, 'f(', u, Yin') =', f(u,Yin)   ! u is right
  end do
End Program good_program
```

Enter these changes into `Function.f90` and compile and execute it. Check that this runs and gives the correct answers. The use of temporary variables is a general approach that is useful in many situations.

Finally, here is an important point about *constants*. Note that we have carefully avoided evaluating the function `f` above using an explicit *constant* value for the first argument. For instance, we could have demonstrated `f(5,Yin)`, using the constant `5` as the value of the first argument, `u`. Recall that when a constant is represented without a decimal point, it is assumed to be an `integer`; therefore, using `5` as the first argument should lead to an error as in the previous demonstration of using the wrong type. It also may not be acceptable to enter a *real* constant `f(5.,Yin)`, since this generally implies `Real*4`; and `Real*8` may actually be required as the argument type. So, constants may still be used as arguments, but you should be explicit about their type, for instance, `5.0_8` or `5.0_4`. The lesson here is caution regarding *type* and *precision*.

10.11 SOLUTION: VIEWING ROCKET GOLF

We will now use Fortran90 to solve the same Rocket Golf problem as we did with Maple in Chap. 3. With the application of modular programming, we will now write separate procedures, as we did with Maple's user-defined functions, to compute γ, U_x, and U_y, and plot a graph of γ *versus* v using *Dislin*[2]. Our solution program thus contains a main program, three computational functions, and one plotting subroutine. (Note: If you have trouble with the plot or your system does not have `Dislin` already installed, simply remove the subroutine `plotGamma` and

[2]We describe the use of *Dislin* in Chap. 11, *Visualization with Fortran*. We recommend that you read that chapter soon, however, you may work through the present sections just by copying the plotting commands.

its `call` statement to do the exercises in this section). To determine how round-off error affects the results, we describe and run the program that solves the problem in double-precision, and then ask you to modify it so that it solves the same problem in single-precision (we give the results of both, but only the source file for the double-precision version).

Listing 10.5 Golf.f90

```
! Golf.f90: The GOLF program with DISLIN plot
! ------------------------------------
Program golf_double
  Implicit None
  Real*8 :: Up, Tp, v, theta, T, uxval, uyval, u, phi
  Real*8, Parameter :: PI = 3.14159265358979323846_8
  Real*8 :: gamma, Ux, Uy
  ! Set initial variables for Michelle:
  Up = 1.0_8 / Sqrt(3.0_8)
  Tp = 2.6e7_8
  v = 0.5_8
  theta = 30.0_8 * PI / 180.0_8
  ! Call functions to get variables for Ben:
  T = gamma(Up) * Tp
  uxval = Ux(Up, theta, v)
  uyval = Uy(Up, theta, v)
  u = Sqrt(uxval ** 2 + uyval ** 2)
  phi = Atan2(uyval, uxval)
  ! Display results:
  print *, 'T =', T
  print *, 'ux =', uxval
  print *, 'uy =', uyval
  print *, 'u =', u
  print *, 'phi =', phi
  ! Produce the Plot of gamma(v) vs. v:
  call plotGamma
End Program golf_double

! Function gamma(v) - time dilation function
Real*8 Function gamma(v)
  Implicit None
  Real*8 :: v
  gamma = 1.0_8 / Sqrt(1.0_8 - v ** 2)
End Function gamma

! Function Ux returns x velocity
Real*8 Function Ux(Up, theta, v)
  Implicit None
  Real*8 :: Up, theta, v, Upx
  Upx = Up * Cos(theta)
  Ux = (Upx + v) / (1.0_8 + v * Upx)
End Function Ux

! Function Uy returns y velocity
```

```
45 Real*8 Function Uy(Up, theta, v)
46   Implicit None
47   Real*8 :: Up, theta, v, Upx, Upy
48   Real*8 :: gamma
49   Upx = Up * Cos(theta)
50   Upy = Up * Sin(theta)
51   Uy = Upy / (gamma(v) * (1.0_8 + v * Upx))
52 End Function Uy
53
54 ! Subroutine to perform the plot
55 Subroutine plotGamma
56   Use dislin   ! Required for DISLIN routines
57   Implicit None
58   Integer, Parameter :: n=1000
59   Integer :: i
60   Real*8, Dimension(n) :: xv, yg
61   Real*8 :: v, gamma
62   ! Create X and Y arrays to plot:
63   v = 0.0
64   do i = 1, n
65     xv(i) = v
66     yg(i) = gamma(v)
67     v = v + .001_8
68   end do
69   ! Call DISLIN plotting routines:
70   call metafl('XWIN')              ! define display format
71   call disini                      ! initialize Dislin
72   call titlin('gamma(v) vs. v',1)  ! define line 1 title
73   call name('v', 'X')              ! X axis label
74   call name('gamma(v)', 'Y')       ! Y axis label
75   call graf(-0.05_8,1.05_8,0._8,.1_8,-1._8,26._8,0._8,5._8)
76   call grid (1,1)                  ! axis marks and grids
77   call title                       ! insert title
78   call color('RED')                ! color of curve
79   call curve(xv,yg,n)              ! draw the line
80   call disfin                      ! finish Dislin
81 End Subroutine plotGamma
```

Lst. 10.5 shows the program `Golf.f90`. It creates initial data for the angle and velocity of Michele's golf ball, and computes the velocities and angles as seen by Ben. If you set the speed of light c equal to a nearly-infinite number, the relativistic effects vanish. Observe that in this example we have placed the main program first in the file, followed by the functions `gamma`, `Ux`, and `Uy`, and then the subroutine `plotGamma`. Some programmers prefer the main program to be listed first as it shows from the start how the entire program works. This is *top-down programming*. Other programmers prefer defining all the procedures (functions and subroutines) first before they are called. That is *bottom-up programming*. The Fortran compiler does not care if you call a procedure before it is defined as long as it is defined somewhere within the source file (or in other modules that are

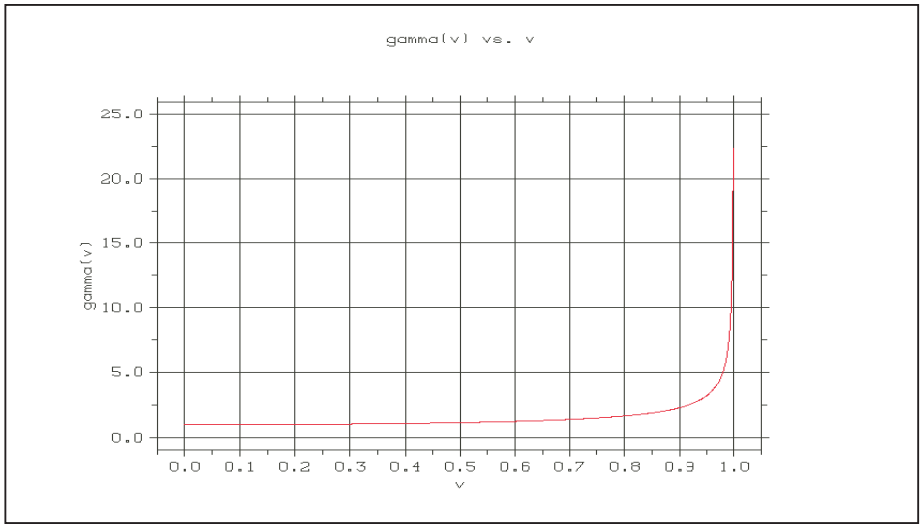


Figure 10.3 A plot of the function $\gamma(v)$ output from the program `Golf.f90`. Compare to similar plot done with Maple in Chap. 3.

referenced by the `Include` statement).

Find where the function `Uy` calls the function `gamma`. Having one function call another is acceptable modular programming practice. Also note that we have kept the main program neat by placing all the plotting commands in the subroutine `plotGamma`. This keeps the technical details from interfering with our ability to see the logical flow of the program, and permits us to modify the plotting without messing up the rest of the program. The plot obtained is shown in Fig. 10.3, and looks much like the Maple plot in Chap. 3.

To create a single-precision version of the program, you will have to convert `Real*8`'s into `Real*4`'s and change the `_8`'s attached to the numerical constants:

```
10  v = 0.5_4
11  theta = 30.0_4 * PI/180.0_4
```

Running this code on a PC with Compaq Fortran, produces the following output for the two cases:

/* Golf: Double-Precision */	/* Golf: Single-Precision */
T = 31843366.6561813	T = 3.1843368E+07
ux = 0.800000000000000	ux = 0.8000000
uy = 0.200000000000000	uy = 0.2000000
u = 0.824621125123532	u = 0.8246211
phi = 0.244978663126864	phi = 0.2449787

This output is the Fortran solution to the first part of the problem, where Michele hits the golf ball forward. Realize that the exact output of these values (format and

significant digits) may differ based on machine type, operating system, compiler, etc. Generally, we will see that Fortran is smart enough to realize that its single-precision calculations have 6-7 places of precision, and so prints out just 8 significant figures (an extra one or two places so you are able to see the imprecision). In contrast, Fortran prints out 16 or 17 significant figures for double-precision, where we expect 15-16 places of precision.

In the second part of the problem, Michele hits the golf ball backwards (at $\theta = 150°$). To find a solution here, we have to modify the program. In the double-precision version, we should change the line `theta = 30.0_8 * PI / 180.0_8` so that it reads

```
theta = 150.0_8 * PI / 180.0_8
```

After making this modification, the results we obtained when running the program on a PC are the following:

/* Golf: Double-Precision */	/* Golf: Single-Precision */
T = 31843366.6561813	T = 3.1843368E+07 s
ux = -1.480297366166875E-016	ux = -7.9472862E-08
uy = 0.333333333333333	uy = 0.3333333
u = 0.33333333333333333	u = 0.3333333
phi = 1.57079632679490	phi = 1.570797

Notice where the differences in precision occur between single and double in these examples. Also notice the differences between these results here as run on a PC and the results you have produced on your computer system.

When we solve the third part of the problem and look at the x component of velocity that Ben sees for the backward hit, we find that the double and single-precision calculations give:

```
ux = -1.4802973661668753E-016      ux = -7.9472862E-08
```

Assessment

It is honest to say that the single- and double-precision results differ by a factor of 100 million! Yet this is not good computational science. It is better to say that within the precision of the calculation, both calculations agree that the x-velocity is zero, but that it is very hard for floating-point computations to give a result that is exactly *zero*.

Another way of interpreting these numbers is to say that the double-precision computation appears to have error in the 16th decimal place while the single-precision computation has an error in the 8th place. This is the respective, expected levels of precision, and so both results are small numbers consistent with

zero. Since the velocities predicted by the programs are much smaller than the velocities used as input, this is a case in which subtractions within the program have cancelled off all of the significant figures in our input numbers, and the output numbers are of (very) questionable significance.

10.12 YOUR PROBLEM: MODIFY GOLF.F90

1. Save and print out a copy of `Golf.f90` from the CD.
2. Draw a box around and label each of the procedures in the program.
3. Label where each procedure is *called*.
4. Remove or *comment out* all the Dislin references and calls. This will eliminate any problems with these routines in case Dislin is not installed on your system. The exercises in this section do not require the plotting components.
5. Copy `Golf.f90` into a new file `GolfReal.f90`. Convert the copy into a single-precision program.
6. Compile and then execute both programs and compare the results for the output (the velocity seen by Ben). We do not expect a large difference for such a simple problem, yet it should be noticeable. The difference might get to be much larger if the calculation were repeated millions of times, as realistic calculations often are.
7. To keep track of your results, make a table of the form:

	U_x^M	U_y^M	U^M	θ^M	U_x^B	U_y^B	U^B	ϕ^B
single								
double								
$\vdots$								

   where the superscripts M and B are used for Michele and Ben.
8. Modify both programs so that $\theta = 150°$. This angle corresponds to Michele's ball hit to the left at an angle of $30°$ above the negative x' axis. Enter your results into your table.
9. Compile and execute both programs. Compute the relative error of the double-precision value for `ux` with respect to the single-precision value for `ux`:

$$u^{rel} \stackrel{\text{def}}{=} \frac{u^D - u^S}{u^S}. \tag{10.11}$$

   Also compute the relative error of the single-precision value for `ux` with respect to the double value for `ux`. Compare the two relative errors. Is relative error a useful metric for comparing numbers such as these?
10. Modify the double-precision program so that $\theta = 0°$.
11. Compile and execute the program. Record results in your table. According to Ben and equation (10.3), what is the x velocity of Michele's golf ball? What would the x velocity be if nonrelativistic physics were used? (This corresponds to using these same equations with the speed of light $c = \infty$, so

$\gamma = 1$.)

12. Modify the double-precision program so that $\theta = 180°$, and interpret the results.
13. Determine what Ben sees as the x velocity of the golf ball when the golf ball is hit $30°$ below the left horizon, that is, at $\theta = 210°$. Compare this to the result from classical physics.

10.13 KEY WORDS

array	binary numbers	bits
bytes	underflow	local variables
machine precision	memory words	word length
modular programs	naming conventions	overflow
data type	significant digits	double-precision

10.14 FURTHER EXERCISES

1. Explain in just a few words what is meant by:
   a. an integer
   b. a floating-point number
   c. double-precision
   d. truncation error
   e. round-off error
   f. a string
   g. machine precision and underflow
   h. number of significant digits and overflow
2. Approximately what are the overflow and underflow limits for double-precision computations in Fortran?
3. Approximately what are the number of significant digits in single and double-precision in Fortran?
4. Imagine that you have just landed that dream job of yours in the local hamburger joint. Your first assignment is to write a program that figures out what change to make if a person buys one item that costs less than a dollar, and pays for it with a dollar bill.
   a. Compute what change in quarters, dimes, and pennies you would give, using the minimum number of coins. *Hint:* Work entirely in integer arithmetic, and define the integers: `price, change, quarters, dimes`, and `pennies`. To name an instance, 27 cents would be represented by the integer 27, and `change = 100 - price, quarters = change/25`, *etc*.
   b. Modify the program so that it uses floating-point variables and comment on the different results.
5. Write the following numbers in scientific notation so that they reflect the given number of significant digits:
   a. 25.3 to four significant figures.

b. 0.00005 to two significant figures.
c. 1.351 to two significant figures.
d. 84000 to three significant figures.

6. Suppose that the floating-point number system on your computer has two-digit mantissas and exponents ranging from -2 to 1. Indicate whether the following expressions each would result in Overflow, Underflow, Roundoff error, or an Exact answer.
   a) 20. + 20. b) 50. * 50. c) 20. + 0.01
   d) 20. * 0.01 e) 0.01 + 0.01 f) 0.01 * 0.01
7. You have encountered a number of examples in which a program was constructed to contain several functions or subroutines rather than do all of the computations within the main function. In reflecting on these examples, does dividing a program into procedures make it
   a. longer or shorter?
   b. harder or easier to understand?
   c. more or less difficult to debug?
   d. more or less likely to reuse components?
8. Consider a Fortran program in which the main program uses a function `Integer f(x,y)` (where x is `Real*8` and y is `Integer`) that is defined just once. Indicate whether each of the following is true or false:
   a. the main program must *call* `f` with variables having the names `x` and `y`
   b. the function f may change the value of $f(x, y)$
   c. if the main program calls f with two integer arguments, f will convert the first one to double-precision.
   d. the function f returns a double-precision value
   e. the variable type returned by f depends upon the arguments given to it.
9. Indicate the values that Fortran would produce for the following expressions:
   a. 4/2 =
   b. 4/3 =
   c. 3/4 =
   d. 3./4 =
   e. 4/3. =
10. Explain what the values of `a` and `b` are at the end of the following bit of code.

```
Integer ::  i = 3, j = 2
Real*8 ::  a, b, c = 3
a = 1 + i/j + j/i
b = 1 + c/j + j/c
```

Chapter Eleven

Visualization with Fortran

11.1 AN OVERVIEW OF DISLIN

We have already seen in Maple how visualizations are helpful in understanding results and in debugging computations, as well as adding some fun to your work. There are two basic ways to do visualization in Fortran. The traditional, and still effective, approach is to run your computation and save the output to a file (easy in Fortran), and then use a separate program such as *Gnuplot* [Gnuplot] or *Grace* (*AceGr*) [Grace] to plot the data as a separate process. When the simulations are time consuming ("expensive"), or the data sets large, this is often the preferred approach.

The second approach to data visualization is to do plotting from within your program using Fortran-callable routines. The Fortran language itself does not contain universal graphical constructs, as exist in Java. We recommend and demonstrate *Dislin* [Dislin]. It is powerful, free for many users, appropriate for science, and available for many Fortran compilers. While we find it somewhat harder to install and use than PtPlot in Java, it is the best we can find for Fortran. If you have trouble with it, remember that Gnuplot and AceGr will still serve you well.

The *DISLIN* data-plotting library was written by Helmut Michels at the Max Planck Institute in Lindau, Germany. Its name is an abbreviation for *Device-Independent Software LINdau* since it is system-independent to a high degree, which allows Fortran programs using Dislin to be ported from one operating system to another without changes. Dislin consists of subroutines and functions for graphically displaying data in the form of curves, bar graphs, pie charts, 3-D color plots, surfaces, contours, and maps. You should reference the Dislin home page at `www.dislin.de` for more information. If your system does not have Dislin installed, then you cannot run our example programs. (§11.2 describes how to install Dislin).

2-D Graphs within Fortran using Dislin

Listing 11.1 EasyDislinPlot.f90

```
! EasyDislinPlot.f90: Simple plot using Dislin
!-----------------------------
Real Function f(x)              ! could substitute any task
  Real, intent(in) :: x
  f = cos(x)
End Function f
!
Program EasyDislinPlot
  Use dislin
  Implicit None
  Integer, parameter :: n=500
  Real, dimension(n) :: xValues, yValues
  Real :: f, factor, xstep
  Integer :: i
  ! Create some data for the X and Y arrays to plot
  factor = 10./n
  do i=1,n,1
     xValues(i) = (i-1) * factor
     yValues(i) = f(xValues(i))
  end do
  ! Dislin plotting routines
  call metafl('XWIN')           ! define intended display
  call disini                   ! dislin initialize
  call pagera
  call complx
  call axspos(450,1800)
  call axslen(2200,1200)
  call name('x-axis','X')
  call name('y-axis','Y')
  call labdig(-1,'X')
  call ticks(10,'XY')
  call titlin('DISLIN graphics demonstration with CURVE',1)
  call titlin('f(x) = cos(x)',3)
  xstep = xValues(n)/10.        ! increment for labels
  call graf(0.,xValues(n),xValues(1),xstep, -1.,1.,-1.,.2)
  call title
  call color('RED')
  call curve(xValues,yValues,n) ! connect the data points
  call color('FORE')
  call dash
  call xaxgit
  call disfin                   ! dislin finish
End Program EasyDislinPlot
```

The program `EasyDislinPlot.f90` in Lst. 11.1 plots $\cos(x)$ *versus* x. We recommend that you use this code as a "black box", that is, without trying to understand any more of it than you have to! We do not expect you to understand every Dislin

command without extensive review of documentation. However, here are some brief explanations that should help you understand much about this program:

• The function `f` contains a simple function which is to be plotted. You could easily place your own function in its place for experimentation. We suggest you try a few modifications.

• On line 9 is the command `use dislin`. This permits your program to import the plotting package. It is a requirement in order to use any of the Dislin routines.

• Lines 16 through 20 initialize the variables. Note that the variables `xValues` and `yValues` are arrays (discussed later in Chapter 17) with multiple elements to hold the x and y values to be plotted. The `do` - `end do` pair of statements bracket the popular Fortran *do-loop* covered in detail in Chapter 12.

• Lines 22 through 42 are the Dislin subroutine calls. Line 22 establishes the intended display format before the other Dislin routines are called into action; line 23 and line 42 act as a pair to *initialize* and then *finish* the Dislin routines.

• Lines 24 through 27 set up the plotting environment for an A4 page (standard metric size of 2969 by 2099 pixels - picture elements, or dots). The `axspos` command places axes on this page with the arguments of the function corresponding to the lower left corner of an axis system and `axslen` sets the length of the axes. If the `call`s in this segment of code are omitted, you will be left with the Dislin default settings, which are quite acceptable.

• `name`: labels the axes. The first argument is the string name for the axis, the second is the internal string name for the axis.

• `titlin`: defines the title string. Then `title` builds and inserts the title appropriately.

• `graf`: plots a 2D-axis system with the stated ranges and tick marks.

• `color`: sets the color of the plotted points and the curves connecting them.

• `curve`: connects data points with lines (or can plot them with symbols).

• `dash`: changes the line plotting style from solid to dashed. The `xaxgit` command plots only the $y = 0$ line.

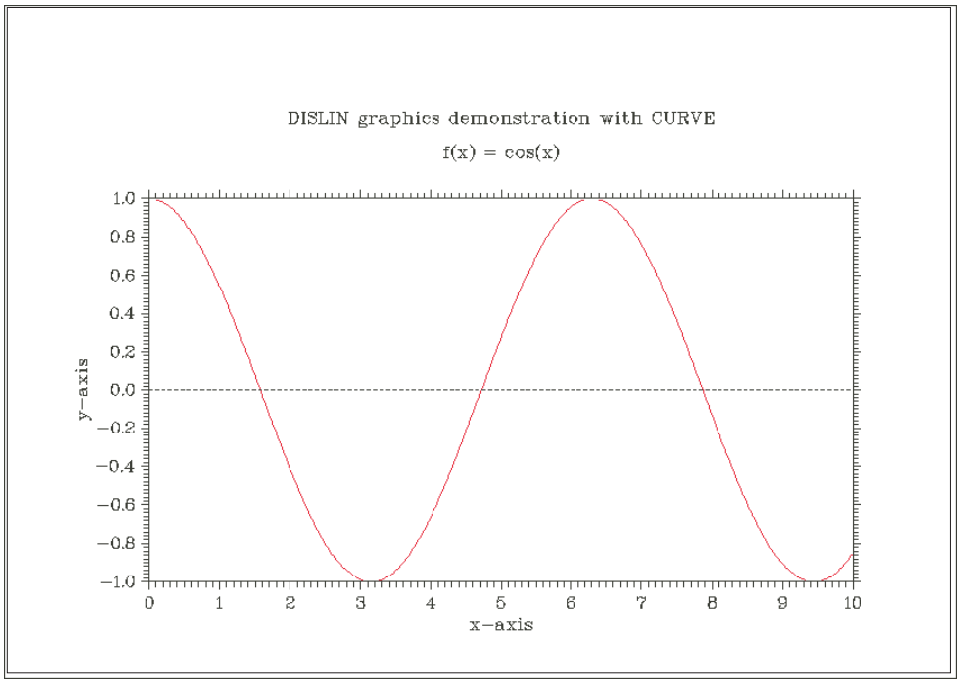


Figure 11.1 The Dislin output from the program EasyDislinPlot.f90.

Running the Sample Program

Run the sample program `EasyDislinPlot.f90` to see the plot for yourself. On a PC, once Dislin is installed, simply use the `f90link` command that is supplied with the Dislin package. With its `-a` option, it is customized to handle the compile, link, and execution of your program (including all needed Dislin libraries). When using this command, do not enter the `.f90` extension. Here is the command with the option you will need:

```
> f90link -a EasyDislinPlot          Compile, link, and run using single-precision
```

If the double-precision Dislin libraries are required for a program, then you would simply add the `-r8` option like this example:

```
> f90link -a -r8 EasyDislinPlot      Use double-precision libraries
```

Dislin allows for a variety of graphic output capabilities. In our program, we used `XWIN` to indicate output to a small window on the screen. If we had preferred to create a file containing our plot, then we could have used `BMP` to create a *bit-map* file. By default, it would be automatically named `DISLIN.BMP`; however, by

adding the `setfil` routine, we can establish the filename of our choice. Here is an example:

```
call metafl('BMP')                          define bit-map as the output format
call setfil('my_plotfile.bmp')              name the bit-map file
```

Once created, this file could be printed or saved for future reference. When creating a file to contain a graph image for printing, it is important to note the foreground/background contrast. In Dislin, the default image contains a black background and white foreground which looks great on a screen, but is difficult to print. One might prefer these colors to be reversed. This can easily be done with the following call placed right after the calls to `metafl` and `setfil`:

```
call scrmod('REVERS')                       reverse foreground/background
```

These calls that define the output display structure must all be given *before* initializing Dislin with `call disini`.

The table below lists all the possible output options. Although some may require other routines for full support, this list will provide a good overview of the Dislin output capabilities:

Device	Output Type
CONS	VGA screen (full window)
XWIN	VGA screen (small window)
GKSL	A GKSLIN metafile will be created
CGM	A CGM metafile will be created
PS	A PostScript file will created
EPS	An EPS file will created
PDF	A PDF file will created
WMF	A Windows metafile will be created
HPGL	A HPGL file will be created
SVG	A SVG file will be created
JAVA	A Java applet file will be created
GIF	A GIF file will be created
TIFF	A TIFF file will be created
PNG	A PNG file will be created
PPM	A PPM file will be created
BMP	A BMP file will be created
IMAG	An Image file will be created

Look at the `EasyDislinPlot.f90` program to verify what kind of output was in-

dicated. See if you can make the needed changes to instruct the plot to go to a file instead.

Dislin Matrix Plots

Dislin has the capability to display the results of a matrix. In the file `MatrixPlot.f90`, we demonstrate this fairly straightforward approach.

Listing 11.2 MatrixPlot.f90

```
! MatrixPlot.f90: Example to demonstrate matrix plotting
! ------------------------------------------------------
Subroutine plot(xs,xf,ys,yf,zs,zf, plot_title, A,n,m)
  ! parameters:
  ! xs = starting x-axis value
  ! xf = final x-axis value
  ! ys = starting y-axis value
  ! yf = final y-axis value
  ! zs = starting z-axis value
  ! zf = final z-axis value
  ! plot_title = title character string
  ! A  = matrix to be plotted with dimensions n & m
  Use dislin
  Implicit None
  Character (len=*), intent(in) :: plot_title
  Real, intent(in) :: xs,xf, ys,yf, zs,zf
  Integer, intent(in) :: n, m
  Real, dimension(n,m), intent(in) :: A
  call metafl('XWIN')          ! Set display
  call disini                  ! Initialize Dislin
  call titlin(plot_title,4)    ! Title
  call name('X-axis','X')      !
  call name('Y-axis','Y')      ! Define axis labels
  call name('HEIGHT','Z')      !
  call graf3d(xs,xf,xs,(xf-xs)/10.,ys,yf,ys,(yf-ys)/10., &
              zs,zf,zs,(zf-zs)/10.)   ! 3-D grid system
  call box3d                   ! Provide a 3-D box border
  call title                   ! Apply the title
  call color('RED')            ! Assign object color
  call surmat(A,n,m,1,1)       ! Surface matrix plot
  call disfin                  ! Finalize Dislin
End Subroutine plot
!
Program MatrixPlot
  Implicit None
  Integer, parameter :: rows=30, columns=20
  Integer :: i, j
  Real :: z_height, top
  Real, dimension(rows,columns) :: matrix
  ! create some test matrix data
```

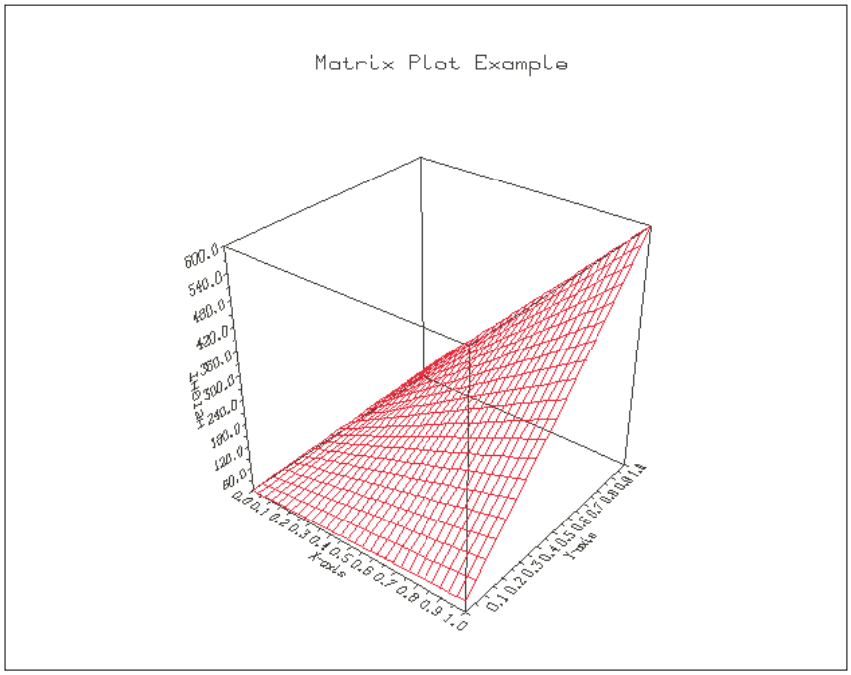


Figure 11.2 The Dislin output from the program MatrixPlot.f90.

```
41   do i=1,rows
42     do j=1,columns
43       z_height = i*j
44       matrix(i,j) = z_height
45     end do
46   end do
47   top = rows*columns
48   call plot(0.,1.,0.,1.,0.,top,  &
49       'Matrix Plot Example', matrix,rows,columns)
50 End Program MatrixPlot
```

This program houses a subroutine `plot` that handles the Dislin requirements to initialize and deploy the plotting of the data matrix. The main program `MatrixPlot` simply creates the matrix data and then hands it off to the subroutine.

Dislin Surface Plots

The Dislin plotting package can also make some beautiful 3-D surface plots, which is just what is needed for the capacitor problem of Chapter 20. Here is an example of this problem solved in Fortran using Dislin, `Laplace.f90`, shown in Lst. 11.3.

Listing 11.3 Laplace.f90

```
! Laplace.f90: Laplacian electrostatics on square grid, Plot
! ------------------------------------------
Subroutine plot(xs,xf,ys,yf,zs,zf, plot_title, A,xdim,ydim)
  ! parameters:
  ! xs = starting x-axis value
  ! xf = final x-axis value
  ! ys = starting y-axis value
  ! yf = final y-axis value
  ! zs = starting z-axis value
  ! zf = final z-axis value
  ! plot_title = title character string
  ! A  = Laplace matrix of dimension xdim & ydim
  Use dislin
  Implicit None
  Character (len=*), intent(in) :: plot_title
  Real, intent(in) :: xs,xf, ys,yf, zs,zf
  Integer, intent(in) :: xdim, ydim
  Real, dimension(xdim,ydim), intent(in) :: A
  call metafl('XWIN')
  call disini
  call name('X-axis','X')
  call name('Y-axis','Y')
  call name('Z-axis','Z')
  call titlin(plot_title,4)
  call graf3d(xs,xf,xs,(xf-xs)/10.,ys,yf,ys,(yf-ys)/10., &
              zs,zf,zs,(zf-zs)/10.)
  call box3d
  call title
  call color('GREEN')
  call surmat(A,xdim,ydim,1,1)
  call disfin
End Subroutine plot
!
Subroutine solve(V, tmax, imax)    ! Create the data
  Implicit None
  Integer, intent(in) :: imax, tmax
  Real, intent(out), dimension(imax,imax) :: V
  Integer :: i, j, k, right, left, top, bottom
  right = imax/3       !right edge of the capacitor
  left = 2*imax/3      !left edge of the capacitor
  top = 2*imax/3       !upper capacitor
  bottom = imax/3      !lower capacitor
  !Initialize the matrix V to zero
  V=0.
  !Boundary conditions
  V(right:left,bottom) = -1.0
  V(right:left,top) = 1.0
  V(1,1:imax) = 0.0
  V(imax,1:imax) = 0.0
  V(1:imax,1) = 0.0
  V(1:imax,imax) = 0.0
```

```
52   !Jacobi relaxation using "nested" Do-Loops
53   do k=1,tmax,1           ! Loops tmax times
54     do i=2,imax-1,1       ! Controls row subscript
55       do j=2,imax-1,1     ! Controls column subscript
56         V(i,j) = 0.25*(V(i+1,j)+V(i-1,j)+V(i,j+1)+V(i,j-1))
57         !Continue applying interal boundary conditions
58         if(j==bottom .AND. i>=right .AND. i<=left) then
59           V(i,j) = -1.0
60         else if(j==top .AND. i>=right .AND. i<=left) then
61           V(i,j) = 1.0
62         end if
63       end do
64     end do
65   end do
66 End Subroutine solve
67 !
68 Program Laplace
69   Implicit None
70   Integer, parameter:: imax=101
71   Integer :: tmax=1001
72   Real, dimension(imax,imax) :: A
73   call solve(A, tmax, imax)
74   call plot(0.,1.,0.,1.,-1.,1.,    &
75        'Realistic Capacitor', A, imax, imax)
76 End Program Laplace
```

Notice that the matrix data, `V`, is generated in a subroutine called `solve`. Then in the subroutine `plot` which uses Dislin, we have isolated the plotting routine requirements just like in the previous example. The main program, `Laplace`, drives the show. Check out the nice surface plot in Fig. 11.3

11.2 SETTING UP DISLIN *

You can download Dislin from `http://www.dislin.de`. The installation depends on your computer's operating system, and there may be a license and cost associated with the download. It is possible to test the appropriate version free before making commitments [Dislin]. For Redhat linux, use the `RPM` files for installation. For Windows systems, here are instructions that may help:

1. Download the appropriate file into a work folder that you have created for Dislin. You will find a `readme` file that should be quite helpful.
2. Run the program `setup.exe` by double-clicking on it. As it suggests, allow it to load the files into a directory called `C:\dislin` which it will automatically create.
3. Reconfigure your Windows environment so that the needed files are recognized upon execution. First, you will need to create a new *environment vari-*

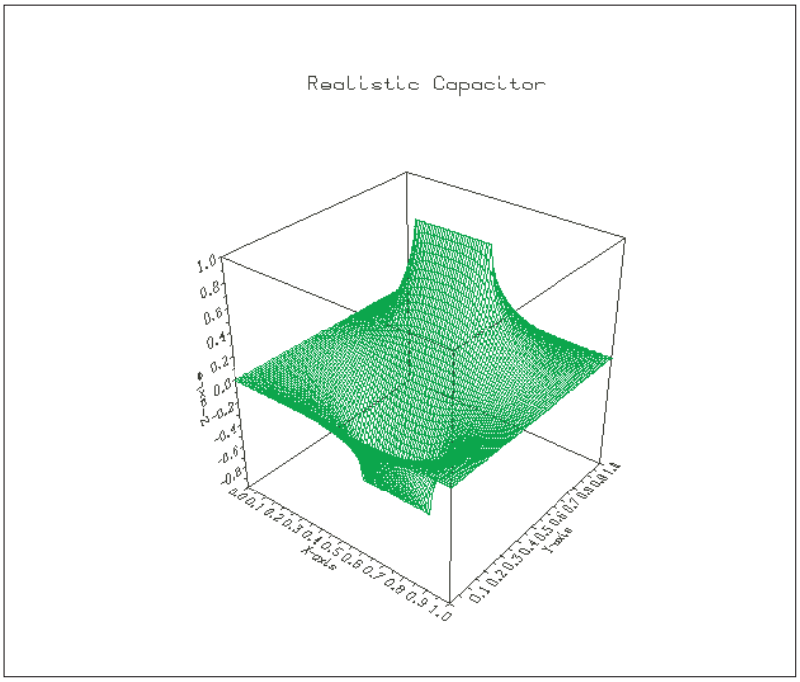


Figure 11.3 The Dislin output from the program Laplace.f90.

able named `dislin` with `C:\dislin` as its value. Next, you will need to add `C:\dislin\win` to your current `PATH` environment variable (the subdirectory `\win` contains the Dislin library routines). Both of these actions may be done from the `My Computer` icon on your desktop:

a. Right-click on `My Computer`, then click on `Properties`, then `Advanced`, then `Environment Variables`. Do *NOT* change `System Variables`.
b. To create a new Dislin environment variable, click on `New`, then enter `dislin` as the variable name and `C:\dislin` as the variable value. Then click `OK`.
c. To add the correct Dislin search path to your `PATH` environment variable, click on `PATH` inside the box that lists the variables. Now click `Edit` (not `New`). Within the `variable value` box, click your pointer directly behind the last search path item, then add a semi-colon and the additional search path (with no spaces). This addition should look like `;C:\dislin\win`, then click `OK`.
d. Finally, click `OK` in the `Environment Variables` window. Click `Apply`, then `OK` in the `System Properties` box.

4. Now, change directories to `C:\dislin` and compile `dislin.f90` in that directory. It is best to do this from within a *DOS window*. Once you have initiated a DOS window:

```
> cd /d C:\dislin                    Change to the Dislin directory
> f90 -c dislin.f90                  Compile to ensure compatibility
```

Note that the file `dislin.mod` contains interfaces for all Dislin routines. It must be referenced from all procedures that utilize Dislin routines via the command `Use dislin` at the top of each procedure requiring it.

Compilation and execution of Fortran programs that use Dislin should best be performed from a DOS window when running on a PC. As mentioned earlier, the Dislin package supplies a convenient *linking* command called `f90link` located in the `\win` subdirectory. Remember that DOS can always find it now because you have added its search path to your `PATH` environment variable. This command will compile, link, and execute (including all needed libraries) in one fell swoop (by using its `-a` option). By default, `f90link` uses the single-precision Dislin libraries. For double-precision, add the `-r8` option to reference the double-precision Dislin libraries. Here are examples again for your reference:

```
> f90link -a prog                    Compile, link, run prog.f90 using single
> f90link -a -r8 prog                Compile, link, run prog.f90 using double
```

For other operating systems you may have to download the compressed source code[1] and compile it for their system.

11.3 GNUPLOT BASICS

Gnuplot is a versatile 2-D and 3-D graphing package that makes Cartesian, polar, surface, and contour plots. It is open software, available for free on the Web, and produces many output formats.

You begin gnuplot with a file of $(x\ y)$ data points. In Fortran so far, we have only demonstrated how to write data and text to the screen. Chapter 13 presents more details about various *input/output (I/O)* capabilities. Briefly, here is a short code segment that may help for now:

```
1  open(UNIT=1,FILE='graph.dat') ! Open file 1, call it graph.dat
2  do i=1,imax,1
3    write(UNIT=1,*) x(i), y(i)  ! Write to graph.dat
4  end do
5  close(UNIT=1)                 ! Close file 1
```

Once you have generated a file of your $(x\ y)$ data, say, `graph.dat`, then you issue the **`gnuplot`** command from a shell or from the Start menu. You then

[1] If you have never done this before, you may want to do it with a friend. In Windows, you can use WinZip to unzip files. In Unix, you can try double-clicking the file in the file manager, or decompress it from the terminal with the `gunzip` or `tar -xvf` commands.

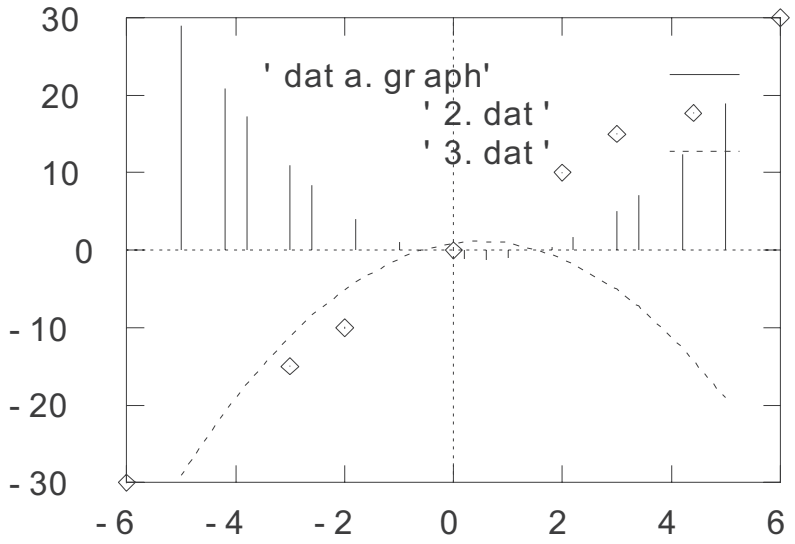


Figure 11.4 A gnuplot graph for three data sets with impulses and lines.

get a new window with the gnuplot prompt `gnuplot>`.You construct your graph by entering gnuplot commands at the gnuplot prompt or by using the pull-down menus in the gnuplot window:

```
> gnuplot                                   // Start gnuplot program
Terminal type set to 'x11'                  Type of terminal for Unix
gnuplot>                                    The gnuplot prompt
gnuplot>  plot "graph.dat"                  Plot data file graph.dat
```

In Figure 11.4 we plot a number of graphs on the same plot using several data files and the command:

```
gnuplot>  plot 'graph.dat' with impulses, '2.dat', '3.dat' with lines
```

The general form of the 2-D plotting command is:

```
plot{ranges} function{title}{style} {, function ...}
```

with points	Default. Plot symbol at each point
with lines	Plot lines connecting the points
with linespoint	Plot lines and symbols
with impulses	Plot vertical lines from x axis to points
with dots	Plot small dot at each point (scatter plots)

It is necessary to place all file or directory names in single or double quotes and to separate file names with a comma. Explicit values for the x and y ranges are set

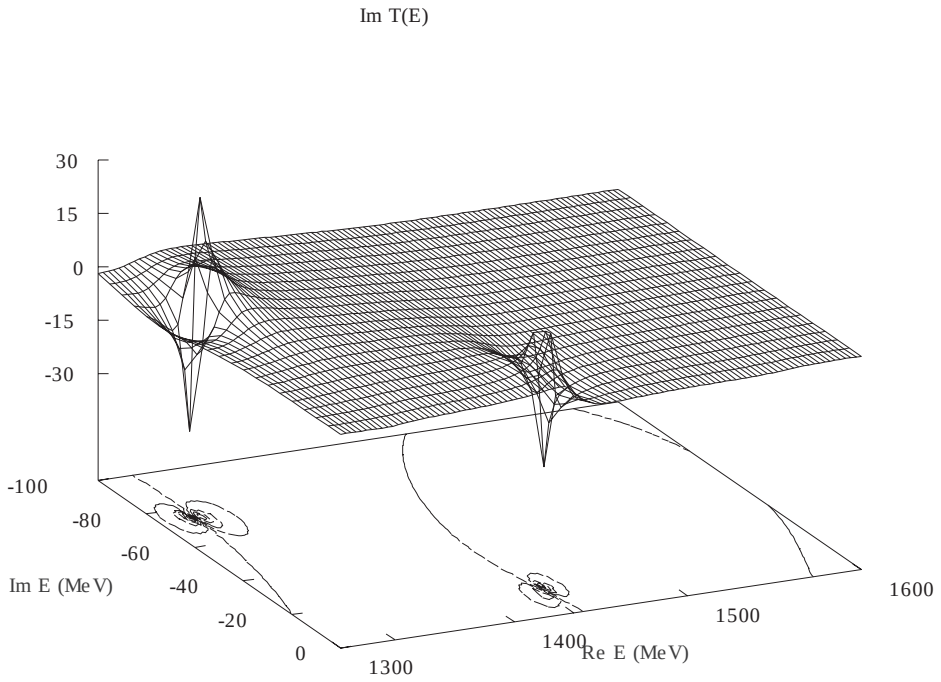


Figure 11.5 Gnuplot's surface plot of a scattering amplitude ImT as a function of complex energy E.

with the plot options:

```
gnuplot> plot [xmin:xmax] [ymin:ymax] "file"                    Generic
gnuplot> plot [-10:10] [-5:30] "graph.dat"                      Explicit
```

Printing Plots

Gnuplot supports a number of printers including PostScript ones. The basic method of printing plots is:

1. Set "terminal type" for your printer.
2. Send the plot output to a file.
3. Replot figure for new output device.
4. Quit Gnuplot (or get out of Gnuplot window).
5. Print file.

You could also import the file into a word processor to have it appear in a document or into a drawing program to fix it up just right. To see what type of printers and other output devices are supported by Gnuplot, enter the `set terminal` command without any options into a gnu window for a listing of available terminal types. Here is an example of creating a PostScript figure and printing it:

```
gnuplot> set terminal postscript          Choose local printer type
Terminal type set to 'postscript'         Gnuplot response
gnuplot> set term postscript eps          Another option
gnuplot> set output "plt.ps"              Send figure to file
gnuplot> replot                           Plot again so file is sent
gnuplot> quit                             Or get out of gnu window
% lp plt.ps                              Unix print command
```

Gnuplot Surface (3-D) Plots

The surface (3-D) plot command is `splot`, and it is used in the same manner as `plot`—with the obvious extension from (x, y) to (x, y, z). As discussed in Chapter 4, a surface, such as that in Figure 11.5, is specified by giving just the z values for successive rows, with different rows separated by blank lines:

```
-4.043764                                 Data for 3-D plot
...
-11.000000
```

Because there are no explicit x and y values given, Gnuplot labels the x and y axes with the row and column number. At present Gnuplot does not have the capability to rotate 3-D plots interactively. Instead, you adjust your plot with the command:

```
gnuplot>  set view rotx, rotz, scale, scalez
```

where $0 \leq$ `rotx` $\leq 180^o$ and $0 \leq$ `rotz` $\leq 360^o$ are angles in degrees, and the scale factors control the size. Any changes made to a plot are made when you redraw the plot using the `replot` command.

To see how this all works, here we give a sample Gnuplot session that we will use in Chapter 20 to plot a 3-D surface from numerical data. Listing 11.3 contains the actual code used to output data in the form for a Gnuplot surface plot.

```
> gnuplot                                 Start gnuplot system from a shell
```

Then, a special Gnuplot shell starts up with the prompt `gnuplot>`, and you are ready to give *gnuplot* your subcommands.[2]

```
gnuplot> set hidden3d                     Hide surface whose view is blocked
gnuplot> set nohidden3d                   Show surface though hidden from view
gnuplot> splot 'Laplace.dat' with lines   Surface plot of Laplace.dat with lines
gnuplot> set view 65,45                   Set x and y rotation viewing angles
gnuplot> replot                           See effect of your change
gnuplot> set contour                      Project contours onto x-y plane
```

[2]Under Windows, there is a graphical interface that is friendlier than the gnuplot subcommands. The subcommand approach we indicate here is reliable and universal.

```
gnuplot> set cntrparam levels 10                          10 contour levels
gnuplot> set terminal PostScript                          Output in PostScript format for printing
gnuplot> set output "Laplace.ps"                          Plot output to be sent to file Laplace.ps
gnuplot> splot 'Laplace.dat' w l                          Plot again, output to file
gnuplot> set terminal x11                                 To see output on screen again
gnuplot> set title 'Potential V(x,y) vs x,y'              Title graph
gnuplot> set xlabel 'x Position'                          Label x axis
gnuplot> set ylabel 'y Position'                          Label y axis
gnuplot> set zlabel 'V(x,y)'; replot                      Label z axis and replot
gnuplot> help                                             Tell me more
gnuplot> set nosurface                                    Do not draw surface, leave contours
gnuplot> set view 0, 0, 1                                 Look down directly onto base
gnuplot> replot                                           Draw plot again; may want to write to file
gnuplot> quit                                             Get out of gnuplot
```

For this sample session, the default output for your graph is your terminal screen. To print a paper copy of your graph, we recommend first saving it to a file as a *PostScript* document (suffix `.ps`), and then printing out that file to a PostScript printer. You create the PostScript file by changing the terminal type to `Postscript`, setting the name of the file, and then issuing the subcommand `splot` again. This plots the result out to a file. If you want to see plots on your screen again, you need to set the terminal type to `x11` again (for Unix's *X Windows System*), and then plot it again.

Chapter Twelve

Flow Control via Logic; Projectiles

In this chapter, we begin discussing some of the control structures that make programming so interesting and that make your programs so intelligent. We will have you write your own program to simulate projectile motion, a standard problem of elementary physics. In Chap. 15, *Differential Equations with Maple and Fortran; Projectile Motion with Drag*, we solve this same problem including air resistance. We start with a review of the kinematic equations that may be scanned for those familiar with it, and then go on to describe program design and control structures.

12.1 PROBLEM: FRICTIONLESS PROJECTILE MOTION

Figure 12.1 shows the trajectory of a projectile fired from a cannon at time $t = 0$, with initial velocity V_0, at an angle θ relative to the horizon. The projectile remains in the air for a total hang time of T, and hits the ground at a horizontal distance or range $x = R$ away from the origin.

Problem: write a program to simulate the motion of this projectile. Have it perform the following steps:

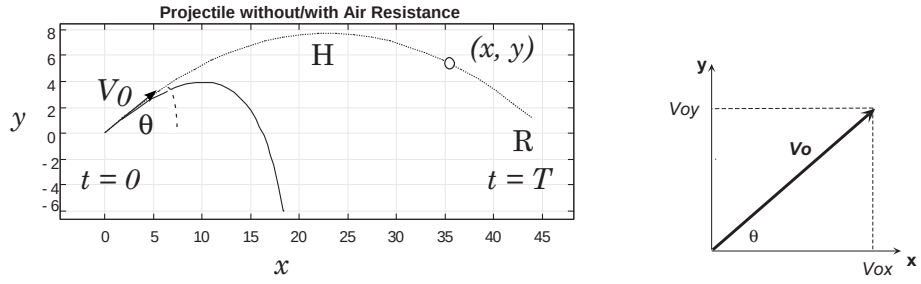


Figure 12.1 *Left:* The trajectory of a projectile fired with initial velocity V_0 in the θ direction. The nonparabolic curve includes air resistance. *Right:* The components of the initial velocity V_0 projected onto the x and y axes.

1. store the initial values V_0, g, and θ as constants,
2. compute the "hang time" T from the equations of kinematics,
3. compute the range R and height H from the equations of kinematics,
4. compute each position (x, y) of the projectile starting at $-T/2$ and running in 100 uniform steps to $3T/2$.

12.2 THEORY: KINEMATICS

If we ignore air resistance, a projectile has only the force of gravity acting on it. This force causes the projectile to fall towards the center of the earth with a constant acceleration $g = 9.8m/s^2$. The resulting solutions to the equations of motion are

$$x(t) = V_{0x}t, \quad y(t) = V_{0y}t - \tfrac{1}{2}gt^2. \tag{12.1}$$

Here V_{0x} and V_{0y} are the horizontal and vertical components of the initial velocity, and, as we see from Fig. 12.1,

$$V_{0x} = V_0 \cos\theta, \quad V_{0y} = V_0 \sin\theta. \tag{12.2}$$

Likewise, the x and y components of the velocity as functions of time are:

$$v_x(t) = V_{0x}, \quad v_y(t) = V_{0y} - gt \tag{12.3}$$

Although the equations of motion yield x and y as functions of time, they do not tell us the shape of the trajectory. For these simple equations it is easy to derive that the shape is a parabola. We will, instead, determine the shape by plotting up the solution, a simpler approach as the equations become more realistic.

We determine the hang time T by observing that at the time of firing $t = 0$, and landing $t = T$, the projectile has a height $y = 0$. If we set $y = 0$ in (12.1), we obtain an equation for the time at which the height is zero:

$$y(t) = V_{0y}t - \tfrac{1}{2}gt^2 = 0, \quad \Rightarrow \quad V_{0y}t = \tfrac{1}{2}gt^2 \tag{12.4}$$

$$\Rightarrow \quad T = \frac{2V_{0y}}{g}, \tag{12.5}$$

where we ignore the $t = 0$ solution. Once we know T we find the range R as the horizontal distance $x(T)$:

$$R = x(T) = V_{0x}\frac{2V_{0y}}{g} = \frac{2V_0^2 \sin\theta\cos\theta}{g}. \tag{12.6}$$

The projectile reaches its maximum height H at the midpoint of the trajectory at time $t = T/2 = V_{0y}/g$:

$$H = y(\frac{T}{2}) = V_{0y}\frac{V_{0y}}{g} - \tfrac{1}{2}g\left(\frac{V_{0y}}{g}\right)^2 \tag{12.7}$$

$$\Rightarrow \quad H = \frac{V_0^2 \sin^2\theta}{2g}. \tag{12.8}$$

12.3 COMPUTER SCIENCE: DESIGNING STRUCTURED PROGRAMS

Now that you are into the program construction business, it is a good idea to understand not only the grammar of each instruction within the selected programming language, but also the general structure of a well-built code. Books have been written on program design, but then again, it is a good idea not to believe everything you read! Yet, as seems to be true in much of life, it is helpful to follow the rules until you know better. Once you have truly mastered a subject, you may then make new rules. We view programming as a written art that blends elements of science, mathematics, and computer science into a set of instructions which permits a computer to accomplish a scientific goal. Good programs should:

- Give the correct answers.
- Be clear and easy to read, with the action of each part of the code easy to identify and analyze.
- Document themselves for the sake of programmers (yourself included) who must improve or expand the code later.
- Be easy to use.
- Be structured so that it is easy to modify and robust enough to keep giving the right answers!
- Be passed on to others to use and to develop further.

One way to make your programs clearer is to *structure* them in an organized fashion. You may have already noticed that sometimes we show our coding examples with indentation, skipped lines, and strategically placed dividers. This is done to provide visual clues as to the functioning of the different program components. These organized components are the “structures” in structured programming. Although the Fortran compiler ignores these visual clues, the human readers are aided in understanding and modifying the program by having it arranged in a manner that not only looks good, but also makes the different parts of the program manifest to the eye.

The organization of code into clearly organized task components, is known as *structured programming*. This method, presented with the visual clues to help humans easily identify program components, is *highly* recommended! It is a valuable approach when dealing with complex tasks involving multiple control structures, especially when some structures are nested within others. The basic idea is simple: blocks of code performing specified tasks should be organized together, and then indented and physically isolated to set them apart. Even though in our examples (in order to conserve printed space) we do not skip as many lines or add as many comments as we would like, we recommend that you take these extra steps in your code design as programs grow in complexity.

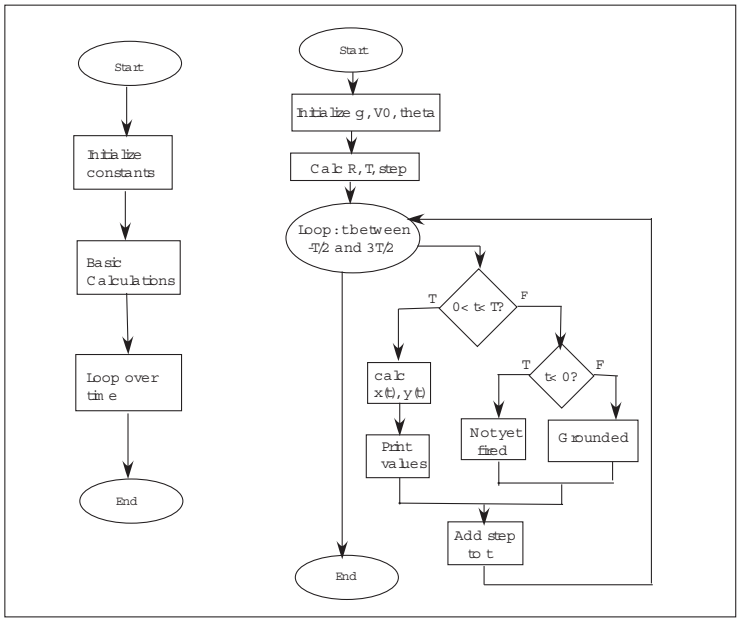


Figure 12.2 A flow chart illustrating a possible program to compute projectile motion. On the left, we show the very high-level basic components of the program, while on the right, we show some of the details of the logic flow structures.

Flow Charts and Pseudocode

In Fig. 12.2 we present a *flowchart* that illustrates a possible program to compute projectile motion. A flowchart is not meant to be a detailed description of a program with all calculations and exact variable names, but rather a graphical aide to help visualize its logical flow. As such, it is independent of which computer language you may choose and is useful for developing and understanding the basic structure of an overall task. We recommend that you draw some type of flowchart or construct some type of *pseudocode* every time you write a program. Pseudocode is like a text version of a flowchart. Here is an example:

1. Store g, V_0, and θ
2. Calculate R and T
3. Calculate uniform $step$
4. Loop 100 times from $-T/2$ to $3T/2$ by $step$
   a. If $t < 0$, print "not yet fired"
   b. If $t > T$, print "grounded"
   c. If projectile is in the air, calculate and print $x(t)$ and $y(t)$
   d. increment t by $step$
5. End program

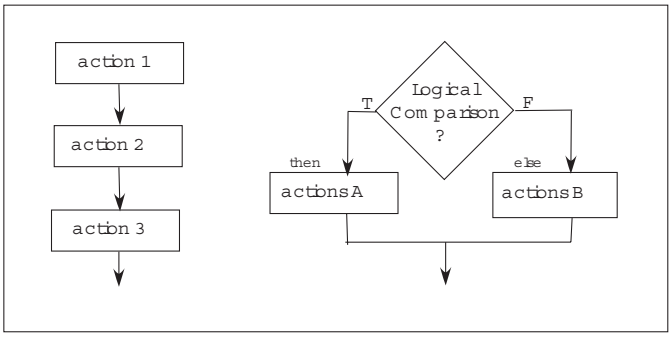


Figure 12.3 *Left:* Sequential or linear programming. *Right:* The if-then-else structure, one of several structures used to write nonlinear program segments based on logical decisions.

One way of viewing program design is in terms of data flow. Simplistically, your program starts with some initial data (read or created), decides what to do with it, does it, and then outputs the results. Usually boxes in flowcharts like Fig. 12.3 show straight-forward actions, while elongated or angled boxes show decisions to be made and the related actions that are needed based on those decisions. The direction of logical flow of the program is shown by the arrows, with flow moving down, except when arrows redirect it.

For our projectile problem, we need to calculate the position for 100 different times. The easiest way to do this is to have the same block of code repeated 100 times, with only the value of the time, t, changing. Such a structure is called a *loop*. Like a clock, our example will run for a total time

$$T_{\text{total}} = T_f - T_i = \frac{3}{2}T - \frac{-T}{2} = 2T, \tag{12.9}$$

in steps (ticks) of

$$\Delta t = \frac{2T}{100} = \frac{T}{50}. \tag{12.10}$$

Whence, the time increases by Δt for each repetition of the loop:

$$t = t + \Delta t, \tag{12.11}$$

with the program using its intelligence to decide what to compute and print depending upon the particular value of t.

12.4 FLOW CONTROL VIA LOGIC

One of the most satisfying aspects of programming is getting the computer to do just what you want. The control of the computer can be direct, as when you have the computer evaluate functions in a "calculator" mode like we did with Maple, or it can be more subtle, as when you try to make the computer act intelligently and

"think" logically about what it should do next. Making a program *think logically* is done by including some symbolic logic in your program using *logical* or *Boolean* variables. A Boolean variable is a Fortran data type that is either *true* or *false*. Fortran represents these two values as `.TRUE.` and `.FALSE.`, where the periods are syntactically part of the name. A Boolean expression is a combination of variables or expressions containing Boolean operators that when evaluated has a value that is either *true* or *false*. When Boolean expressions are used with *control structures* in your program, you are able to affect the order in which statements are executed.

Conditional and Relational Operators

Conditional operators act between two variables or expressions to form a new logical expression that have `.TRUE.` or `.FALSE.` values. There are two types of conditional operators, *relational operators* and *logical operators*. *Relational operators* start with variables like `Real::x` and `Real::y`, and create a logical relationship between them, for example, `x > y`, that is either `.TRUE.` or `.FALSE.`, depending on the numerical values of `x` and `y` . This logical relationship is stored in a *logical variable* that has `.TRUE.` or `.FALSE.` values. *Logical operators* combine logical variables to create more complex expressions, for example, `(x>3) .AND. (y>4)`.

To be more specific, here are all relational operators:

Operator (Old Form)		Example	Return **.TRUE.** if
`>`	`(.GT.)`	`x > y`	x is greater than y
`>=`	`(.GE.)`	`x >= y`	x is greater than or equal to y
`<`	`(.LT.)`	`x < y`	x is less than y
`<=`	`(.LE.)`	`x <= y`	x is less than or equal to y
`==`	`(.EQ.)`	`x == y`	x and y are same *object*
`/=`	`(.NE.)`	`x /= y`	x and y are not equal

In each of the cases in the table, the result is either `.TRUE.` or `.FALSE.`. Thus, even though `x > y` may be constructed from two `Real`'s, Fortran knows that it is a logical variable because it has the relational operator `>` connecting them. Since these relational operators are never used in assignment statements, there can be no confusion. Indeed, this is the reason for introducing the double equal sign `==` as a relational operator; `x = y` is an assignment statement, while `x == y` is a logical variable.

Logical operators combine Boolean variables to create more complex Boolean expressions. For example:

Operator	Example	Name: Return `.TRUE.` if
`.AND.`	`x .AND. y`	**Logical and:** `x` and `y` both true, conditionally evaluates `y`
`.OR.`	`x .OR. y`	**Logical or:** either `x` or `y` true, conditionally evaluates `y`
`.NOT.`	`.NOT.x`	**Not:** `x` is false
`.EQV.`	`x .EQV. y`	**Equivalence:** `x` and `y` the same
`.NEQV.`	`x .NEQV. y`	**Nonequivalence:** `x` and `y` not the same

The basic logic behind logical operators is that if we have a compound statement constructed from two simpler statements, then the truth table

True	False
true and true	*true and false*
true or true	*false and false*
true or false	*false or false*
not false	*not true*

determines the truth of the compound statement. The logical *and* is represented by `.AND.`, while `.OR.` represents the logical *or*. As a case in point, if `x` is *true*, then `x .OR. y` is *true* regardless of *y*. Believe it or not, what makes the use of logical expressions so powerful is that it is possible to combined the different logical expressions to describe the basis for *all* logical decision.

Exercise: If a is your age in years, w your weight in pounds, and h your height in inches, construct Boolean expressions that will be *true* only when the following statements are true:

1. you are old enough to obtain a driver's license but too young to retire, (`(a > 16) & (a < 65)`),
2. you are not a teenager,
3. you are either younger than 20 and less than 150 pounds, or you are older than 40 and more than 6 feet,
4. you are neither old enough to vote, tall enough to hit your head on a five-foot door frame, nor of the right weight to box in the 132–140 pound division. ♠

Control Structures

The combination of Boolean expressions and *control structures* permits you to construct programs that can handle all possible logical situations. Table 12.1 contains a more complete list of control structures, of which we will use only a subset. The type of logic that is possible with these logical expressions is shown in the flowcharts of Figures 12.3-12.4. If you look at both the command in the table, and the action in the figure together, you should get a good idea how the command works.

The left of Fig. 12.3 shows *linear* or *sequential* programming in which the statements are executed in the order in which they are encountered, that is, from

Name	Example	Comment
if-then	**If** `( (x < 3) .AND. (y == 12) ) z = y * x`	Single line if

multi-line if-then	**If** `( (x >= 1) .AND. (x <= 9) )` **Then**
	`y = 1`
	EndIf
		If statement
		Multi-line code block
		End of the If statement

if-then-else	**If** `(x <= 0.)  Then`
	`y = y ** 2`
	Else
	`y = 3 * y`
	EndIf
		If statement
		Do this if true
		Only one `Else` (“catchall”)
		Do this if false
		End of the If statement

if-then-else-if	**If** `(score >= 90)` **Then**
	`grade = "A"`
	Else If `(score >= 80)` **Then**
	`grade = "B"`
	Else If `(score >= 70)` **Then**
	`grade = "C"`
	EndIf
		Multi-way if
		
		Any number of `Else If`’s OK.
		
		Inaccessible if other `Else If`’s true.
		
		End of the If statement

do	**Do** `count = 1, 100, 1`
	`<statements>`
	End Do
		Initial value, final value, increment
		Multi-line code block
		End of the Do structure

do-while	**Do While** `(x + sin(x) < 0.8)`
	`x = x + 0.01`
	End Do
		Repeat as long as true
		Block of code that is repeated
		End of the Do structure

select case	**Select Case (month)**
	Case `(1)`
	`s = "Jan"`
	Case `(6, 7, 8)`
	`s = "Summer"`
	Case `(10:12)`
	`s = "Winter"`
	End Select
		Jump to the appropriate case
		
		Do this if month equals 1
		
		Do this if month equals 6, 7 or 8
		
		Any month between 10 and 12, inclusive

Table 12.1 Logical flow control structures in Fortran.

top to bottom. Control structures introduce the possibility that the execution of a program may “split” into different paths depending on the values of certain variables, or may repeat certain sections a number of times. If the program does “split,” then the program is no longer sequential.

The *if* or *if-then-else* structure, illustrated on the right of Fig. 12.3, is one of the most common control structures in programming. It permits your program to make decisions leading to multiple outcomes, with the decision based on changing situations. For example: *if* it is raining before 10 AM, *then* I will take my umbrella. This structure can be enhanced with the *else* option: *if* it is sunny, *then* I will take my sunglasses, *else* I will take my umbrella. The angle-edged box contains a logical expression constructed with the *if* statement. If the logical expression is *true*, then `actions A` are executed, *else* `actions B` are executed.

In the Fortran *if-then-else* structure illustrated in Fig. 12.3, there are two sets of possible actions; one set is executed if the condition is true and the other set of actions is executed if the condition is false. The syntax requires that *actions A*

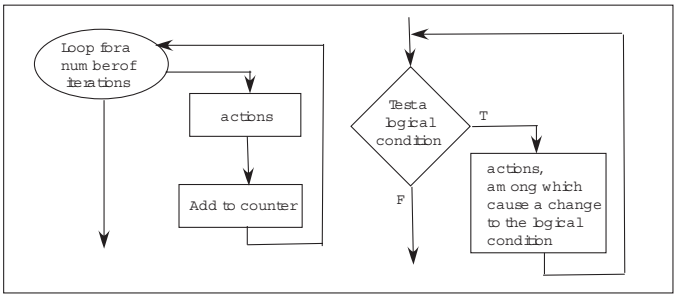


Figure 12.4 *Left:* The Fortran Do-loop iteration. *Right:* The Fortran Do-while loop uses a general test, in contrast to the Do-loop's iteration count.

consist of a block of statements contained within the paired `if` and `end if` statements. (When there is only one line of actions needed, however, the `then` and `end if` statements are not required; the single action statement simply follows the `if` logical condition on the same line). If *actions B* are included as an alternate path of actions, they must be preceded with the word `Else`, which occurs on a separate line of its own right prior to the list of *actions B*.

Fig. 12.4 illustrates the Fortran *do*-loop, probably the most popular looping control structure. The *do*-loop is used to *iterate*, that is, repeat a section of code for as long as some condition is true. This is useful for things like summing a series or repeating a calculation until the error gets smaller than some fixed amount. Because a counter is used to keep track of the number of iterations, it is also called a *counting loop*. The statements that are repeatedly executed are bracketed by the initial `do` statement and the closing `end do` statement.

In the first line of the *do* structure:

```
do var = 1, 100, 2                                    a simple example for discussion
```

after the keyword `do`, there is a variable (here, it is named `var`). It is used to count the loop iterations; it can be referred to as the *counter* or the *index* variable. It is followed by three fields separated by commas; these may be constants or variables or arithmetic expressions, and they may be positive or negative. The first field, `1` in this case, gives the initial value for the index variable. The second field, `100`, gives the final value for the index. The third field is the increment (or decrement, if it is negative) for the index variable. For each iteration through the statements within the loop, this value is added to the index variable. If the third field is missing, it is assumed to be the positive value `1`. The program will continue to repetitively execute all the statements between `do` and `end do` until the index variable has passed the final value. (Note that it is possible to start at `100`, decrementing down to `1` using a change value of `-1`).

```
do var = 100, 1, -1
```

The iteration of a do-loop ends when the index (or counter) has passed the final value. However, if the final condition is already met before any actions are taken, then the statements inside the `do`-loop will never be executed. In either case, if iteration has ended, or never executed, the program moves on to the first executable statement after the `end do` statement.

These few ways to incorporate logic into programs are sufficient for the problem at hand. Nevertheless, there are other commands that use a somewhat different approach in controlling the flow of your program, and we shall now discuss some. Consider again the right of Fig. 12.3 illustrating the *if-then-else* structure. We see in Table 12.1 that the `else` part of this control structure can have an *if-then* structure following it to form an *if-then-else-if* structure. What's more, there can be any number of `else if` statements included to handle any number of special cases. For the example in the table, we may add more `else if` statements to assign grades of *A+*, *A-*, *B+*, *etc.*.

The *Do-While* construct, shown on the right of Fig. 12.4, is similar to the *do*-loop, and can serve a similar function. It repeats the statements between `do while` and `end do` until the stated logical condition is `.False.`. The problem is that without a fixed number of iterations, the loop may never end if the programmer has made an error (*infinite loop*). It is literally up to the programmer to decide upon a good test for a general condition, and to be certain that the general condition will eventually be false so that the `do while` loop ends. However, it's perfectly legal to start with the logical condition `.FALSE.`, in which case the loop is never executed.

It is common to have the counter index variable (`var` in our example, but any variable name is acceptable) in a `do` loop be an integer, yet this is not required. A real variable may be used, and it may be given a *real* initial value, termination value, and increment value. You may also use either an integer or real variable in a logical expression within a `do while` loop to determine if it is time to end the loop. Be that as it may, be warned that round-off error makes it rather meaningless to demand true equality between two floating-point numbers, or between a real and an integer. For example, the following loop may never complete:

`Real*8 ::  x = 0.`
`do while (x /= 100.)`
`  x = x + 1.`
`end do`
	Comparison may never be satisfied
	Increment

To make sure that the loop will eventually stop, it is better to use the greater-than and less-than operators `<=`, `<`, `>=`, `>` when comparing values as part of the logical expression for a `do while` loop. The `==` and `/=` operators should be avoided, unless there is a good reason to use them, such as when one wishes to determine if a sequence has converged *exactly* to a given value.

Listing 12.1 Select.f90

```
1 ! Select.f90: Demonstrates the use of Select Case control
    structure
2 !-------------------------------------------------------------
3 Program select_example
4   Implicit None
5   Integer :: wake, month
6   do month = 12, 0, -1                          ! Loop for 13 months
7     print *, 'Month is ', month                 ! Print current month
8     select case (month)                         ! Choose case via month
9       case (0)                                  ! Here for month 0
10        print *,' Last year'
11      case (1)                                  ! Here for month 1
12        wake = 9
13        print *,' January,  wake =', wake
14      case (2)                                  ! Here for month 2
15        wake = 7
16        print *,' February, wake =', wake
17      case (4)                                  ! Here for month 4
18        wake = 6
19        print *,' April,    wake =', wake
20      case default                              ! Here other cases
21        wake = 8
22    end select                                  ! End Select
23  end do                                        ! End Do
24 End Program select_example
```

Select Case

The *Select Case* structure is a neat way to execute different blocks of code depending upon the value of an integer or string variable. While this also can be accomplished with a series of `if-then` statements, as we said, *Select Case* is neater. As an instance, consider the program `Select.f90` in Lst. 12.1 that we use to *program* our clock radio.

Compile and execute `Select.f90` to see what the output looks like. Note that the *do*-loop on line 6 counts *down* on the value of `month` from 12 to 0. Once within the loop, there is a printout on line 7 that outputs the value of `month`. Next within the do-loop is the `select case` on line 8 controlled by the value of `month`. If `month =` 0, 1, 2, 4, execution will jump to the lines starting with `case` followed by the corresponding value of `month`. If the value of `month` is not one of these cases, then execution jumps to the `case default` on line 20. When execution jumps to one of the listed cases, the value of the wakeup time variable `wake` is set and printed. Once execution is completed within a `case` set of actions, the program jumps to the next statement after the `end select` statement.

Select Case Exercise:
1. Add in wakeup times and printouts for three more months.
2. See what happens if you leave off the `case default` component on lines 20-21.
3. See what happens if the order of the `Case` statements is changed. ♠

12.5 IMPLEMENTATION: PROJECTILE.F90

1. Before you write your program, write down the equations to be solved. Hand this in with your assignment, and refer to it while working on the program.
2. Create a flow chart, or some pseudocode, that shows the logic of your program. Think of the logic flow as the path that converts the initial data to the printed results. Hand this in, too.
3. Using the program `limits.f90` from Chapter 10 as an *initial program framework*, open it with your editor and then save it as `projectile.f90`, a new file in the appropriate directory/folder for this week.
4. Modify `projectile.f90` to compute and print T, H and R in a line before the *do*-loop. (In other words, strip out the "limit" tasks and insert the "projectile" tasks).
5. Modify the *do*-loop so that it loops over i = 1 to 100.
6. Declare a variable t_2, the time variable that changes as the projectile flies through the air. Within the *do*-loop, calculate t_2 by the formula $t_2 = -T/2 + 2 * T * (i/100.)$.
7. Inside the *do*-loop, use *if* statements to decide if the projectile has not yet been fired, if it's in the air, or if it has already hit the ground.
8. Have your program print "not yet fired" or "grounded" as appropriate. Otherwise, print explicit values for $x(t_2)$ and $y(t_2)$.
9. Have your program make a plot of the particle trajectory. Your instructor may have to help you determine the available plotting packages for your system.

12.6 SOLUTION: PROJECTILE TRAJECTORIES

1. Run your program for a variety of initial conditions, that is, for your choice of different values for V_0 and θ. Try some for which you know the expected answer (like $\theta = 0$ and $\theta = \pi/2$).
2. Check that the range R increases as the initial velocity V_0 increases in magnitude, and as the elevation θ increases from 0 to $\pi/4$.
3. Check that the range $R = 0$, but that the hang time $T \neq 0$, for $\theta = 90^o$ (this corresponds to shooting the cannon straight up in the air, not the most clever thing to do).
4. Check that the y values never exceed the maximum height H.

12.7 KEY WORDS

air resistance	Boolean variable	select case	conditional op
control structure	counting loop	decrement op	flowchart
if-then-else	increment op	flow control	do while
iteration	linear program	logical operators	operator
pseudocode	relational op	repetition structure	structured program

12.8 SUPPLEMENTARY EXERCISES

1. Take the program `Limits.f90` from Chapter 10 that uses a *do*-loop to determine machine precision, and make a copy of it.
   a. Compile and run `Limits.f90` as a check that it is still running, and to get some output for comparison.
   b. Modify the *do*-loop so that the counter is a double precision variable. You should get all the same results.
   c. Change the *do*-loop to a *do while* loop that accomplishes the same task. (*Hint:* `while (1.0 + epsilon_m /= 1.0)` will repeat the loop until there is no difference between the stored values of `1.0 + epsilon_m` and `1.0`.)
   d. Include an *if-then* construct in the *loop* so that the program prints only one line of output, and that is for the first time that `1.0 + epsilon_m = 1.0`.
   e. Include an *if-then-else* construct in the *loop* so that the program prints the message "not there yet" if `1.0 + epsilon_m /= 1.0`, and then the usual `one` and `epsilon_m` the first time that `1.0 + epsilon_m = 1.0`.
2. If a is your age in years, w is your weight in pounds, and h is your height in inches, construct Boolean expressions that will be *true* only when the following conditions are met:
   a. you are old enough to obtain a driver's license and you do not weigh 1000 pounds.
   b. you are not a teenager.
   c. you are either younger than 20 and less than 150 pounds, or older than 40 and greater than 6 feet.
   d. you are neither old enough to vote, tall enough to hit your head on a five-foot door frame, nor the right weight to box in the 132–140 pound division.
3. Let: A be your age, Y the number of years you have been in college, and D the number of dollars you have in the bank. Construct Boolean expressions that will be *true* when the following conditions are met:
   a. you are a millionaire but you are not a senior.
   b. you are either too young to vote or you are not a freshman.
   c. you are either younger than 20 and broke, or older than 90 and have more than $100,000.
   d. you are 16 years old and your number of years in college is greater than the number of dollars you have in the bank.

4. **Iteration** refers to the repetition of the same lines of code until a condition is satisfied. To illustrate, in Lst. 12.2 is a code that computes $1/1-x$ for $x^2 < 1$ via its infinite series expansion

$$\frac{1}{1-x} = 1 + x + x^2 + x^3 + x^4 + \cdots . \tag{12.12}$$

The summation of terms is repeated 1000 times or until the new terms have an insignificant effect on the sum ($x^n < 10^{-7}$ sum).

Listing 12.2 Iterate.f90

```
! Iterate.f90: Series expansion of 1 / (1 - x)
!-------------------------------------
Program Iterate
  Implicit None
  Integer, parameter :: N = 1000          ! Integer constant
  Real*8, parameter :: eps = 1.0e-5       ! Real constant
  Real*8 :: sum = 0.0, x, term = 1.0
  Integer :: i = 0
  print *, 'Enter a value between 0 and 1:'
  read *, x
  do while (i <= N .AND. term > eps)      ! Repeat N times,
    sum = sum + term                      ! or till term<eps
    term = term * x
    i = i + 1
    print *, 'i = ', i, '  sum =', sum, 'term =', term
  end do
End Program Iterate
```

a. Enter, compile, and execute `Iterate.f90`. Check that it runs with no errors.
b. Try several *positive* values of $x < 1$. Check that convergence is obtained in each case, but with a differing number of terms needed.
c. See how close you are able to get to $x \simeq 1$ before the algorithm fails, that is, 1000 summations are made.
d. The series (12.12) is valid for negative x values, but the code has assumed that $x > 0$. Modify the program so that it will work for negative x, and check your results for $x = -0.5, -0.1, -0.9$.

Chapter Thirteen

Fortran Input and Output

The Fortran programmer is blessed with a variety of powerful and high-level I/O statements and formats to use. "High-level" here means that you do not have to worry about all the details, while "format" means that Fortran will produce a form for your output so that it looks just the way you want, including alignment. Here in Lst. 13.1 is a sample code that contains a number of I/O statements:

Listing 13.1 FortranIO.f90

```
1  ! FortranIO.f90: Input/Output example
2  ! ---------------------
3  Program demo_IO
4    Implicit None
5    Real :: PI, r, power, area
6    Character (LEN=50) :: name
7    PI = 3.14159265358979
8    write(*, *) 'Enter your name: '     ! Output string
9    read(*, *) name                     ! Input string
10   name = trim(name)
11   write(*, *) 'Ciao, ', name          ! Output string+variable
12   write(*, *) 'Now, enter radius & power: '
13   read(*,*) r, power                  ! Input 2 variables
14   area = PI*(r**power)                ! Compute area
15   write(*, 100) name, r, area, power ! Write to screen
16   write(*, 200) name, r, area, power ! Write to screen
17   open(1, FILE='data.dat',STATUS='REPLACE')  ! Open file
18   write(1, 100) name, r, area         ! Write to file + format
19   close(1, STATUS='KEEP')             ! Close file
20   write(*, *) 'See data.dat for the answer in the file.'
21   stop
22   100 Format(/,'Example 1:',/,'Hello,',T10, A8,/, 5X,  &
23    'The area of a circle with radius ', F12.6,      &
24    2X, 'is  ',2E12.3//)
25   200 Format('Example 2:  ', a8, 1X, 3G12.3, //)
26 End Program demo_IO
```

A typical session of running `f90` produces the screen output:

```
 Enter your name:
Loren
 Ciao, Loren
 Now, enter radius & power:
10    2

Example 1: Hello,   Loren
     The area of a circle with radius    10.000000  is     0.314E+03   0.200E+01

Example 2:  Loren        10.0        314.        2.00

 See data.dat for the answer in the file.
```

13.1 BASIC I/O

The simplest input and output in Fortran is writing to the computer screen and reading from the keyboard. This is the default, and is thus called *standard I/O*. For example, on lines 8 and 11 of `FortranIO.f90` we have

```
 8.  write(*, *) 'Enter your name:  '
11.  write(*, *) 'Ciao, ', name
```

The `write` command is used for output. It takes two arguments in parenthesis, the first indicating *where* to write (what device), and the second indicating what *format* to use when writing. Both these commands above do the simplest thing, which is to use an asterisk * in place of an explicit argument. The first * indicates that no `UNIT` is being specified on which to write, so Fortran will use the *default* output device (the computer screen). The second * indicates that no `format` statement is being specified (*free format*), and so Fortran will decide what format to use based on the types and values of the variables being output.

Usually a comma-separated list of variables follows the parenthesis in I/O commands. On line 8, the quotes indicate an explicit *string* to be output, rather than the name of a variable. On line 11, the first element in the list is again an explicit string, while the second element is `name`, a `character` variable.

Input can be just as simple. For example:

```
13.  read(*,*) r, power
```

Here, two variables previously declared as `real` are read in, with the variable names separated by a comma. Notice that since we have used the default * again, we do not specify things like where the decimal point occurs in the input or how many spaces are used to separate the values on input. In fact, the two values to be input can be on one line separated by spaces, or commas, or tabs, or even on separate lines. The default input device is the computer keyboard, and Fortran figures out the format on its own from the data it finds.

13.2 FORMATTED OUTPUT

When you want to specify the exact format of input or output, then the second * in the `read` and `write` statements above should be replaced by a numbered label. This label is included as part of the `format` statement in your program, making it possible for any I/O statements in the program to reference and use it. The internal part of the Format statement defines your desired I/O layout:

```
15 write(*, 100) name, r, area, power
16 write(*, 200) name, r, area, power
22 100 Format(/,'Example 1:',/,'Hello,',T10, A8,/,5X, &
23      'The area of a circle with radius ', F12.6, &
24      2X, 'is ',2E12.3//)
25 200 Format('Example 2:  ',a8, 1X, 3G12.3,//)
```

Here the write statements on lines 15 and 16 output identical variables but with different `format` statements (as indicated by the labels `100` and `200`). The format statements and their labels are found on lines 22 and 25, but may be placed anywhere in the program since they are just used for reference (not executable action commands). To keep `format` statements out of the way of the program's logical flow, they are often placed at the end of the code, right before the `End Program` statement. We see that format statement `100` produces verbose output, while `200` contains minimal output, but with variables in a slightly different output format.

Some of the basic input and output types in Fortran are:

Type	Meaning	Type	Meaning
`A`	`CHARACTER` (**A**lphanumeric) data	`E`	`REAL` (**E**xponential) data
`T`	Tab to a column	`F`	`REAL` (**F**ixed decimal point) data
`G`	**G**eneral format	`I`	**I**`NTEGER` data
`X`	Blank space	`L`	**L**ogical

Here `A` denotes alphanumeric (strings), `E` denotes a real number in the exponential form, `F` produces a fixed decimal point, `G` is a general format that switches between `F` to `E` depending upon the size of the number, `X` represents a blank space, `L` is for logical (`.TRUE.`, `.FALSE.`) variables, and `T` is used to tab to some column.

These format types take modifiers to control the total number of digits (width) used for the I/O, how many digits appear after the decimal point, and whether some field is repeated:

Example	Meaning
A8	8 character alphnumeric
E12.3	exponential of width 12 digits, with 3 after decimal
F12.6	fixed point of width 12 with 6 after decimal
G12.3	general of width 12 digits, with 3 after decimal
I4	Integer of width 4 digits
2X	2 blank spaces
3G12.3	3 general fields of width 12 with 3 digits after the decimal

So now if we go back and look at `100 format` on line 22, we see a bunch of format instructions separated by commas. The slash / tells the computer to skip a line, `'Example 1:'` and `'Hello,'` are strings that get output exactly as entered. Accordingly, `T10` causes the line to tab to column 10 and `A8` prints out an alphanumeric variable (your name) using 8 places. The `/` causes the output to skip to another new line, which is then indented by 5 spaces as indicated by the `5X`. The long string enclosed in quotes is printed on that new line. After that, `F12.6` outputs a real value in a field 12 places wide, with 6 digits after the decimal point. Then there is another string followed by two real values printed as `2E12.3`, that is, as 12-character wide fields with 3 digits after the decimal points, and with an exponent. Notice how the formatting of these same variables gets changed when `200 Format` is used!

13.3 FILE I/O

Fortran has many options for file I/O that give you the flexibility needed for various tasks. However, since file I/O deals with the hardware and operating system of your local computer, it is important to check out various versions of these commands to see which ones work best on your local system.

File I/O can be either free-form or formatted (the second argument to the `read` and `write` commands). In order to write to a file or read from a file, you must specify a `UNIT` number associated with the file (this goes in place of the first * in the I/O commands parentheses). In `FortranIO.f90`, we associated `UNIT` 1 with the filename `data.dat`, in the same directory as the executing program, via

```
17 open(1, FILE='data.dat', STATUS='REPLACE')
```

where `REPLACE` indicates that it is OK to replace an older version of the file. This statement may appear anywhere in your program, although we prefer to put `OPEN` statements at the top of the program, so they do not get in the way of the logic and so it is easy to see which `UNIT`s and filenames are being used. The actual writing to the file is done with the `write` command:

```
18 write(1, 100) name, r, area
```

which uses the same `Format` statement that we used for screen output. You may

construct any format to meet your needs, but you do not want to spend a lot of time engineering a beautiful output for a one-time run. Since `format` statements are referenced by their numerical labels, several I/O statements can use the same format.

Although not strictly needed in many cases, it is always good housekeeping to close any files that you may have opened:

```
19 close(1, STATUS='KEEP')
```

This releases resources for other tasks, as well as protects the file you have written in case of crashes.

Chapter Fourteen

Numerical Integration and Nested Loops; Energy Usage

In this chapter we explore how computers evaluate integrals numerically. We look at the same energy problem solved with Maple or Mathematica in Chap. 6, where numerical integration also had to be used. This chapter provides a fundamental technique for computational science and a better understanding of calculus. You also gain experience with nested loops and plotting.

14.1 PROBLEM (SAME AS CHAPTER 6): POWER AND ENERGY

In Chapter 6 we modeled the electrical power use $P(t)$ over time by some simple functions. For example,

$$\text{Model 3:} \quad P_3(t) = \left(4 + \frac{t}{365} + \tfrac{1}{2}\sin\frac{\pi t}{91}\right)\left(2 + e^{-\sin 2\pi t}\right), \tag{14.1}$$

where the power is in GW (10^9 watts) and the time t is in days (the top curve in Figure 14.1). Your **problem** is to determine the total energy used over 1000 days (approximately three years) by evaluating the integral

$$E(100) = \int_0^{100} P_3(t)dt, \tag{14.2}$$

for Model 3, the one that cannot be done analytically.

The highly oscillatory patterns in Figure 14.1 make (14.2) a challenging integral to evaluate (as demonstrated by the quite different answers obtain by casual application of Maple and Mathematica). Consequently, we want you to compare the results from the two different integration rules (trapezoid and Simpson), and to see if the value for the integral converges as we improve the approximations. There are fancier integration rules, but we leave those to the references [Press 94, CP 05].

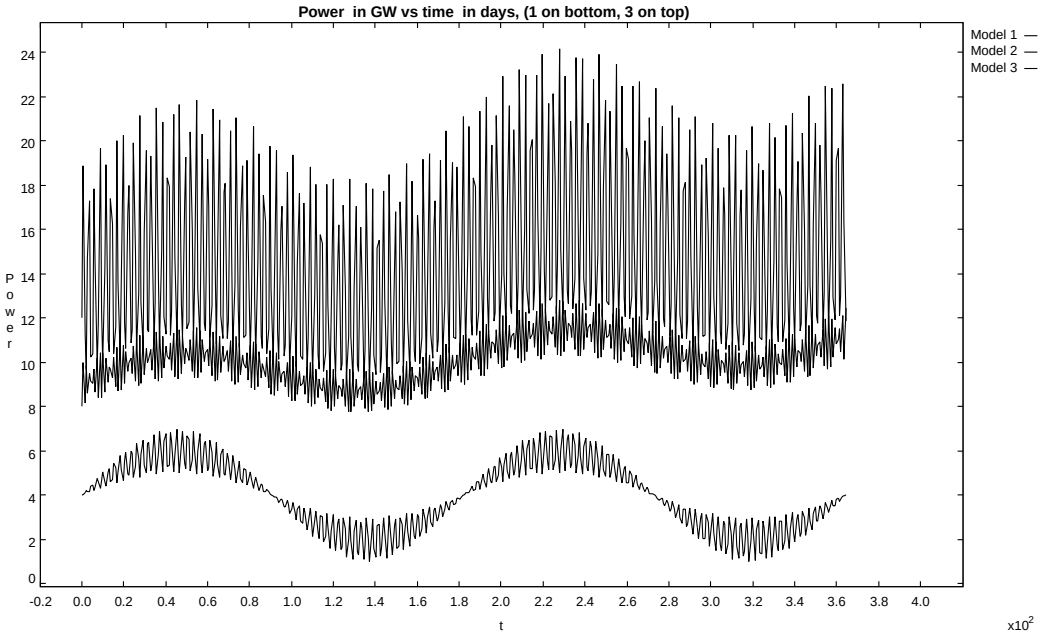


Figure 14.1 The three models of power consumption. Time t is in 100 days and power is in GigaWatts.

14.2 ALGORITHMS: TRAPEZOID AND SIMPSON'S RULES

Consider Figure 14.2 showing an arbitrary function $f(x)$ in the interval $a \leq x \leq b$. The definite integral of this function,

$$I = \int_a^b f(x)dx, \tag{14.3}$$

is equal to the area under the curve. A traditional way to measure this area is to plot the integrand on a piece of graph paper and add up the number of little boxes lying below the curve. Seeing that boxes are "quadrilaterals," numerical integration is also called *numerical quadrature*, even when it gets beyond the explicit box-counting stage.

Numerical-integration techniques usually break the total interval up into small columns, as also shown in Figure 14.2, and then use different approximations to determine the area of each column. When the areas of all the columns are added together, we obtain an approximation for the integral as a weighted sum over the integrand $f(x)$ evaluated at points within the integration region:

$$I = \int_a^b f(x)dx \;\; \simeq \;\; \sum_{i=1}^{N} f(x_i)w_i. \tag{14.4}$$

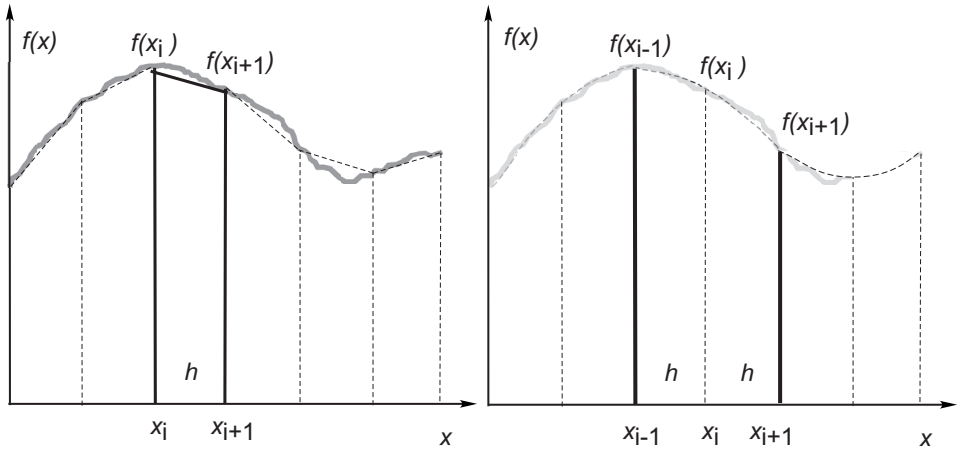


Figure 14.2 *Left:* Straight-line sections used for the trapezoid rule. An individual trapezoid with area $\frac{h}{2}[f(x_i) + f(x_{i+1})]$ is highlighted. *Right:* Parabolas used in Simpson's rule (a single parabola is fit to each pair of consecutive intervals.

Here the x_is are *integration points* and the w_is are *integration weights*. This is a handy way to program up an integration algorithm, since we need only change the specific values for w_i (and maybe x_i) for different rules.

As the number of integration points N is made progressively larger (narrower columns used), the sum in (14.4) should become a progressively better approximation to the integral. However, if too many points are used, then round-off error tends to accumulate and the approximation stops converging to the exact answer. Consequently, the best integration rules are those which obtain the desired level of precision with the least number of points.

Trapezoid Rule

The trapezoid rule is the simplest integration algorithm. As shown on the left of Figure 14.2, we divide the area under $f(x)$ into a series of columns each of width h, and form trapezoids by approximating the integrand $f(x)$ by a straight line connecting the endpoints of each interval. In this case the integral is approximately equal to the sum of the areas of the columns.[1] The approximation gets progressively better as the widths of the trapezoids are made progressively smaller, and the integrand is better approximated by a straight line within the interval.

[1] An alternative way of viewing the trapezoid rule is as forming a series of rectangles with horizontal tops, the height of each rectangle being $(f_i + f_{i+1})/2$. If we use f_i as the height, then we have a version of the trapezoid rule known as *Euler's rule*.

We first deduce an equation that gives the x value for each integration point. If we evaluate the function at N points, then we have $N-1$ columns (it takes two points to define one column) covering the range b–a. It follows then that the width of each interval in Figure 14.2 is

$$h = \frac{b-a}{N-1}, \tag{14.5}$$

and the discrete x values. They are thus enumerated with the subscript i:

$$x_i = a + (i-1)h, \quad i = 1,\ N. \tag{14.6}$$

To calculate the area of each trapezoid in Figure 14.2, we look at the isolated column i. The area of this one trapezoid is its width times its average height:

$$\int_{x_i}^{x_i+h} f(x)dx \ \simeq\ \tfrac{1}{2}h[f(x_i) + f(x_{i+1})], \tag{14.7}$$

which is the same area one gets if each column is considered as a rectangle with the top passing through the midpoint of the slanted top. The total area in the integration region from a to b is the sum of the areas of all the columns:

$$\int_a^b f(x)dx \simeq \frac{h}{2}f(x_1) + hf(x_2) + \ldots hf(x_{N-1}) + \frac{h}{2}f(x_N). \tag{14.8}$$

Observe that because each internal point gets counted twice, it gets weighted by h, whereas the endpoints get counted just once and so are weighted by only $h/2$. In terms of the notation of our standard integration rule (14.4), the points and weights for the trapezoid rule are:

$$x_i = a + (i-1)h, \quad w_i = \left[\frac{h}{2}, h, \ldots, h, \frac{h}{2}\right], \quad i = 1,\ N. \tag{14.9}$$

Listing 14.1 Trap.f90

```
!Trap.f90: Trapezoid integration
!————————————
Program trap
Implicit None
  Integer :: N, i
  Real :: A, B, sum, h, w, t
  write(*,*) "Enter initial and final t values:"
  read(*,*) A, B
  write(*,*) "Enter number of integration points:"
  read(*,*) N
  h = (B - A)/(N - 1)
  sum = 0.
  do i=1, N, 1
     t = A + (i-1)*h
     if (i==1 .OR. i==N) then
        w = h/2.
     else
```

```
18            w = h
19         end if
20         sum = sum + w*t**2
21      end do
22      write(*,*) 'The sum is ', sum
23  End Program trap
```

14.3 IMPLEMENTATION: TRAP.F90

In Lst. 14.1 we present a simple Fortran program `Trap.f90` that uses the trapezoid rule to evaluate the integral of $\int_A^B t^2 dt$. The function t^2 to be integrated is written directly into the program on line 20. Notice that the calculation of the sum over intervals is accomplished within a `do`-loop. See §10.10 if you prefer the integrand t^2 to be a separate function.

1. Compile and run `Trap.f90`.
2. Modify it to determine the energy usage $\int_0^{100} P_3(t)dt$ for Model 3.
3. Use a plotting program to produce a plot of Model 3's $P(t)$ for 100 days, in steps of 10 days.
4. Make a crude graphical estimate ($\sim$ 25% accuracy) of the area under your $P_3(t)$ *vs.* t curve. Use 10-day column widths, and count the number of boxes in each column (count fractionally filled boxes as full if they are more than half full), or calculate the area of each trapezoid. Make your boxes big so that you do not have too much work to do, and make sure you have the right units (energy=GW-days) for your choice of boxes.
5. Compare the answer for the integral from your program using 10 day widths with that found from your graph. Do they agree within measurement error?

Implementation with Nested Loops

Our approximation to the area under a curve as the sum of columns with straight or rounded tops should get more accurate as we increase the number of columns `N-1`. However, if we make `N` too large, then round-off error accumulates to the point where increasing `N` further leads to a less accurate answer. To get a feel for the level of accuracy of our numerical integration technique, we want to increase `N` and determine in what decimal place the answer changes. If the answer changes in, say, the fourth decimal place, then we would expect the answer to be good to at least three places.

Modify a copy of `Trap.f90` so that it prints out the value of the integral and the value for eight values in the range $25 \leq N \leq 200$. Rather than running the

program eight times (increasing the value of N for each run), make the computer do this for you; modify the program so that it loops over eight values of N:

```
1  do N = 25, 200, 25              !Outer loop increases N
2      ...
3      do i = 1, N, 1              !Inner loop sums over columns
4           ...
5      end do
6  end do
```

The procedure outlined in this code fragment uses *nested loops*, that is, one *do*-loop contained within another *do*-loop. Study how for each value of the total number of points N, everything within the outer loop gets repeated, and this *includes* the inner-loop's i-summation over the areas of each column. Consequently, it is important to check that the program re-initializes the value of variables used in the inner loop, in particular, h and sum. If you look at line 11 of Trap.f90, you will notice that the column width h varies with the number of integration points used N. Because of that, we need to begin the new outer do-loop before line 11. Next, notice that the summation over column areas is ended by the end do on line 21. As a consequence, we need to end the outer loop incrementing N after line 21.

Improved Method: Simpson's Rule

Simpson's rule for integration will also give us an algorithm of the form (14.4), but with different weights than the trapezoid rule. The integration range is again divided into equal-width columns, with the points given by (14.6). Now, as we show on the right of Figure 14.2, we approximate $f(x)$ at the top of every two columns as a parabola:

$$f(x) = \alpha x^2 + \beta x + \gamma. \tag{14.10}$$

This is a better than using a straight line. With each parabola containing three unknown parameters, α, β, and γ, we need three values of $f(x)$ to determine them. We do that by having one parabola pass through the tops of two adjacent columns. Whereas the determination of these constants requires some algebra, the idea is simple. What is important is that because parabolas are being fit to successive pairs of columns, *Simpson's rule requires an odd number of integration points* N so that there will be an even number of columns.

*Simpson's Rule Weights**

After approximating the integrand $f(x)$ by a parabola in (14.10), it is easy to calculate the area of two columns:

$$\int_{x_{i-1}}^{x_{i+1}} (\alpha x^2 + \beta x + \gamma) dx = \frac{\alpha}{3}(x_{i+1}^3 - x_{i-1}^3) + \frac{\beta}{2}(x_{i+1}^2 - x_{i-1}^2) + \gamma(x_{i+1} - x_{i-1}). \tag{14.11}$$

However, to make an algorithm out of this we need to express the parameters α, β, and γ in terms of the values of the integrand within the intervals. To do that we evaluate the integrand $f(x)$ at the three integration points ($x_{i-1} = 0$, $x_i = h$, $x_{i+1} = 2h$) and solve for the parameters in terms of them:

$$\alpha = \frac{1}{h^2}\left(\frac{f(x_{i+1}) + f(x_{i-1})}{2} - f(x_i)\right), \tag{14.12}$$

$$\beta = \frac{1}{h}\left(2f(x_i) - \frac{f(x_{i+1}) + 3f(x_{i-1})}{2}\right), \tag{14.13}$$

$$\gamma = f(x_{i-1}). \tag{14.14}$$

This leads to the elementary Simpson rule for the area of two columns:

$$\int_{x_{i-1}}^{x_{i+1}} f(x)dx \simeq \frac{hf(x_{i-1})}{3} + \frac{4hf(x_i)}{3} + \frac{hf(x_{i+1})}{3}. \tag{14.15}$$

When we apply Simpson's rule to the entire column, the endpoints of each two-column section get counted twice, since they belong to two different pairs. Yet the first and last endpoints only get counted once:

$$\int_a^b f(x)dx \simeq \frac{h}{3}f(x_1) + \frac{4h}{3}f(x_2) + \frac{2h}{3}f(x_3) + \frac{4h}{3}f(x_4) + \cdots + \frac{4h}{3}f(x_{N-1}) + \frac{h}{3}f(x_N). \tag{14.16}$$

In terms of the notation of our standard integration rule (14.4), Simpson's rule is:

$$x_i = a + (i-1)h, \quad w_i = \left\{\frac{h}{3}, \frac{4h}{3}, \frac{2h}{3}, \frac{4h}{3}, \ldots \frac{4h}{3}, \frac{h}{3}\right\}, \tag{14.17}$$

where the sequence repeatedly alternates between $4h/3$ and $2h/3$, except for the endpoints. *Remember*, N must be odd!

14.4 ASSESSMENT: WHICH RULE IS BETTER?

1. Modify your trapezoid-rule program so it prints out running values for the sum, the integrand, the weight, and the loop counter i in equation (14.4)'s notation. By "running values" we mean the value of a variable as it actually changes during the calculation. Looking at these values and checking that they change in the way expected is one of the best ways to ensure that your program is working correctly.

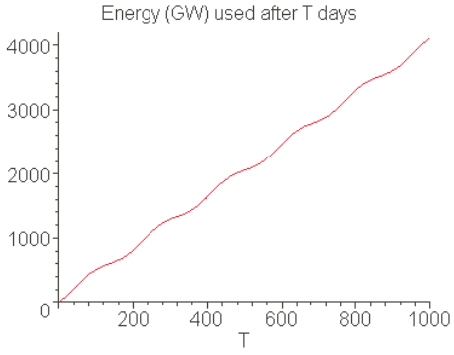


Figure 14.3 Energy consumption as a function of time for model 1 computed with Maple.

2. Modify a copy of your program so it evaluates the integral using Simpson's rule. Make the program smart enough to use only an odd number of points for Simpson's rule. *Hint:* You test if an integer `j` is even by testing if `2 * (j/2) = j` or if `j % 2 = 0`. The first test takes advantage of the fact that integer arithmetic rounds *down*, while the second test uses a Java method to determine if the remainder is 0 after division by 2.
3. The numerical approximation for integration should improve as the number of points `N` increases. Make a table or plot showing the level of accuracy obtained for the integral for various numbers of points:

N-1	I (trapezoid)	I (Simpson)
8	-	-
⋮		
512	-	-

4. Make a plot of the energy consumption *versus* time, like that in Figure 14.3, only here for Model 3. How does that compare to the Maple results?

14.5 KEY WORDS AND CONCEPTS

integration	integration points	integration weights	modular programs
nested loops	quadrature	Simpson's rule	trapezoid rule

1. Explain succinctly the difference between the trapezoid and Simpson rules.
2. Is a parabola or a straight line a better approximation for a function?

14.6 SUPPLEMENTARY EXERCISES

1. You are given nonrepeated weights of 1, 2, 4, 8, 16, and 32 kg. Your boss claims that you should be able to use these six weights to weigh any rice sack weighing an integer number of kilograms between 1 and 63. For instance,

$63 = 1 + 2 + 4 + 8 + 16 + 32$. Write a program to check if the claim is true. The program should write the sack weight and the weights used.

2. As a more challenging version of the above problem, imagine that you are given a sack of rice of undermined integer weight less than 64 kg. Write a program that determines the sack's weight and the individual weights you used to determine it.
3. The Chinese scholar Sun Tsu in the year 400 BCE found solutions to the problem:

   > find an integer such that if divided by 3 the residue is 2, if divided by 5 the residue is 3, and if divided by 7 the residue is 2.

   Write a program that finds the numbers less that 500 that satisfy these three conditions.

(These three problems come via courtesy of M. Páez.)

Chapter Fifteen

Differential Equations with Maple and Fortran; Projectile Motion with Drag*

In this chapter we solve for projectile motion with drag by solving simultaneous, second-order ordinary differential equations (ODEs) using both Fortran and Maple. This is a fairly realistic problem that is often missing from undergraduate education, but whose solution looks familiar to students. The ODE solver used in our Fortran program is simple, useful for other problems as well, and provides some insight into numerical differentiation. The Maple solution, however, gets to be surprisingly involved at times due to the equations being both second-order and coupled, and having a solution that cannot be plotted simply. A comparison of the two approaches is quite educational. We have marked this chapter as optional, in part because there are no new materials that are used elsewhere, and in part because some students may not yet be familiar with differential equations. However the subject matter is important for many fields, and it does provide a good balance to the solution of a partial differential equation studied in Chapter 20.

15.1 PROBLEM: PROJECTILE MOTION WITH DRAG

We want to determine if the inclusion of air resistance in projectile motion leads to trajectories that are much steeper at their ends than their beginnings (golf balls that appear to drop out of the sky). We saw in Chapter 12, "Flow Control via Logic," that frictionless projectile trajectories are parabolas, symmetric about their midpoints, and so do not describe balls dropping out of the sky.

In Figure 12.1 we showed trajectories of a projectile fired from a cannon without friction (solid curve) as well as one with air resistance or drag included (dashed curve). The projectile is fired at time $t = 0$ with velocity V_0 at an angle θ relative to the horizon. It remains in the air for a total or hang time $t = T$ and hits the ground a horizontal distance or range $x = R$ away from the origin, with all these quantities apparently changing if air resistance is included. The problem solution requires us to write a program that predicts or *simulates* these different trajectories for a given set of initial conditions.

15.2 MODEL: VELOCITY-DEPENDENT DRAG

The basic physics behind this problem is Newton's second law of motion, which relates force, mass, and acceleration:

$$\mathbf{F} = m\mathbf{a} = m\frac{d^2\mathbf{x(t)}}{dt^2}, \tag{15.1}$$

where the bold symbols indicate vector quantities. This is a second-order ordinary differential equation. A differential equation because it contains a derivative; an ordinary derivative because there is only one independent variable and so no partial derivatives (no ∂), the time t; and second-order because it is a second derivative. Because the equation of motion involves vectors, there are separate differential equations for the x and y components of each vector:

$$\frac{d^2x}{dt^2} = \frac{1}{m}F_x^f, \qquad \frac{d^2y}{dt^2} = \frac{1}{m}(F_y^f - mg), \tag{15.2}$$

where we have divided through by the mass, switched the derivative to the LHS (a standard form), and substituted components for the frictional force $\mathbf{F}^f$ and the gravitational force.

The force of friction $\mathbf{F}_f$ is not a basic force of nature with a definite form for all situations. Indeed, it is just an approximate description of the physics of viscous flow, with no one expression being accurate for all velocities. We know it always opposes motion, which means it is in a direction opposite to that of the velocity. The simplest model, and the one often studied in texts [M&T 88], assumes that the force of air resistance is proportional to some power n of the projectile's speed:

$$\mathbf{F}^f = -k\,m\,|v|^n\,\frac{\mathbf{v}}{|v_x|}. \tag{15.3}$$

The $\mathbf{v}/|v|$ term ensures that the frictional force changes direction, to keep opposing motion, when the velocity changes sign. If $n = 1$, or any odd power, then we may have the frictional force proportional to $-v^n$ and get the correct sign. However, if n is even, then we have to use these absolute values to ensure the correct sign. Though a frictional force proportional to a power of the velocity is a more accurate description than a constant force, it is still a simplification. Indeed, physical measurements indicate the power n appears to change with velocity, and so the most accurate model would be a numerical one that uses the empirical velocity dependence $n(v)$.

With a simple power-law dependence, the equations of motion take the

form:[1]

$$\frac{d^2x}{dt^2} = -k\,v_x^n\,\frac{v_x}{|v_x|}, \qquad \frac{d^2y}{dt^2} = -g - k\,v_y^n\,\frac{v_y}{|v_y|}. \tag{15.4}$$

We shall consider three values for n, each of which represents a different model for the air resistance: (1) $n = 1$ for low velocities; (2) $n = \frac{3}{2}$, for medium velocities; and (3) $n = 2$ for high velocities. For comparison, we will compare these solutions with air resistance to the analytic results for the frictionless case:

$$x(t) = v_{0,x}t, \qquad y(t) = v_{0,y}t - \frac{1}{2}gt^2, \tag{15.5}$$

$$v_x(t) = v_{x,0}, \qquad v_y(t) = v_{y,0} - gt. \tag{15.6}$$

15.3 ALGORITHM: NUMERICAL DIFFERENTIATION

In Chapter 5, "Solving Equations, Differentiation," we used Maple to explore the accuracy of numerical differentiation. We now apply those techniques in the Fortran solution of differential equations. Most calculus courses start off with the definition of the derivative as:

$$\frac{df(x)}{dx} \stackrel{\text{def}}{=} \lim_{h\to 0}\frac{f(x+h) - f(x)}{h}. \tag{15.7}$$

It is a challenge to apply this definition on a computer, because its finite word length causes the numerator to fluctuate between 0 and round-off error as the denominator approaches zero. A simple and effective solution is to make h small but not zero:

$$\frac{df(x)}{dx} \simeq \frac{f(x+h) - f(x)}{h}. \tag{15.8}$$

Equation (15.8), known as the *forward-difference* rule, is not the most accurate approximation [CP 05], but it will suffice.

15.4 MATH: SOLVING DIFFERENTIAL EQUATIONS

When air resistance is present, the force on the projectile depends upon the value of the velocity, which means that the acceleration is not constant. This makes an analytic solution to the differential equation difficult, but it is not a challenge to a numerical solution. The trick is to figure out an approximation that takes the solution at the initial time 0 and advances it a small amount Δt. Because Δt is small, it is not hard to find an approximate solution. One then takes the solution so found and uses it as the new "initial conditions" to provide the solution at time $2\Delta t$. The process is continued until the solution at the desired time is found. With $[x(t=0),\ y(t=0)]$ and $[v_x(t=0), v_y(t=0)]$ as the initial conditions, we have

[1]For more realistic models used for friction, these equations may become coupled, e.g., because $(V^2)_x \neq (V_x)^2$. For simplicity, we ignore the coupling of the equations, although the technique for the numerical solution remains unchanged.

all the information we need to start the solution and will keep it going as long as we like.

The simplest algorithm for solving differential equations is a variation of *Euler's rule*. It is based on the fact that for a sufficiently short time interval Δt, we may assume that the acceleration is constant and use the equations for constant acceleration from elementary physics to advance the velocity and position for a short time from t to $t + \Delta t$:

$$x(t+\Delta t) \simeq x(t) + v_x \Delta t + \frac{1}{2} a_x (\Delta t)^2, \quad v_x(t+\Delta t) \simeq v_x(t) + a_x(t)\Delta t. \tag{15.9}$$

Here $a_x(t)$ is the variable acceleration at time t. Likewise, there are similar equations for y and v_y. To make the algorithm simpler, we assume that Δt is so small that the terms quadratic in Δt are negligible with respect to the terms linear in Δt:

$$x(t+\Delta t) \;\simeq\; x(t) + v_x \Delta t, \qquad\qquad v_x(t+\Delta t) \;\simeq\; v_x(t) + a_x(t)\Delta t. \tag{15.10}$$

This means that as long as we keep the time step Δt small, we may ignore the acceleration term in (15.9). The acceleration still affects the solution via (15.10), but it does so by changing the velocity, which, in turn, changes the position. For this simple method to work, we must choose small values of Δt. As a rule of thumb, if the physical system under study has a characteristic time or period of T, then we would start with $\Delta t \simeq T/100$ and then see the effect of decreasing it. A higher-order technique might require fewer steps but use more complicated formulas.

As our example, we take the simple $n = 1$ law for air resistance:

$$\frac{d^2x}{dt^2} = -kv_x, \qquad\qquad \frac{d^2y}{dt^2} = -g - kv_y. \tag{15.11}$$

The terms on the right-hand sides of these equations are the accelerations we need for the algorithm. The algorithm starts with the given initial position (the origin) and the velocities $v_{x,0}$, $v_{y,0}$, and then steps through many time steps. We label the time steps with the index $i = 1 \ldots N$. For each time step, the position and velocity will be updated according to the algorithm:

$$x^{i+1} \;=\; x^i + v_x^i \, \Delta t, \qquad\qquad y^{i+1} \;= y_x^i + v_y^i \, \Delta t, \tag{15.12}$$

$$v_x^{i+1} \;=\; v_x^i - k \, v_x \, \Delta t, \qquad\qquad v_y^{i+1} \;=\; v_y^i - k \, v_y \, \Delta t - \, g \, \Delta t. \tag{15.13}$$

The acceleration due to gravity enters only for the y component of velocity, since gravity acts only in the y direction. And as we have said, the forces act by directly changing the velocity, but change the position only indirectly through the changes in velocity.

Implementation: ProjectileAir.f90

The implementation of the algorithm is given in the code `ProjectileAir.f90` shown in Listing 15.1. The program `ProjectileAir.f90` contains:

1. a *main* program that sets initial parameters and directs the computation; and
2. a `buildPlot` subroutine that solves the equations of motion analytically and numerically and plots each result using the `Dislin` package.

Examine how the `Dislin` package is imported with the `Use dislin` command on line 4. On lines 54 and 55 we set the values for the constants, and then initialize the computation by calling `buildPlot` on line 56. The calculation does the frictionless and with-friction computations simultaneously, with the analytic calculation on lines 21 and 22, and the numeric calculations on lines 27-28. The Dislin plotting routines are called on lines 32-49, and should give a plot looking like the trajectory in Figure 12.1.

15.5 ASSESSMENT: BALLS FALLING OUT OF THE SKY?

1. Compile and run `ProjectileAir.f90` and check that you get a plot similar to that in Figure 12.1. If your program is not plotting, and your system has Dislin loaded, then you may need to review some of the materials on using Dislin in Chapter 11, "Visualization with Fortran."
2. In general, it is not be possible to compare analytic and numerical results to realistic problems, because analytic expressions do not exist. However, we do know the analytic expressions (15.5) for the frictionless case, and we may turn friction off in our numerical algorithm and then compare the two. This is not a guarantee that we have handled friction correctly, yet it is a guarantee that we have something wrong if the comparison fails. Add some lines of code to `ProjectileAir.f90` that calls the `buildPlot` procedure a second time, this time with the friction coefficient $k = 0$.
3. We have deliberately given you a program in which the total number of time steps `N` is rather *small*, and, inversely, the time step `dt` is rather *large*. Consequently, the results from the given program will be of low precision. Repeat the calculation with ever-increasing values for `N`, until you cannot tell the difference on the plot between the numeric and the analytic results for frictionless flight.
4. Once the results look good to you, determine the level of precision of the calculation by comparing the actual numbers for the analytic and the numerical calculations. If necessary, increase `N` until the relative difference is less than 1%. Use this value of `N` for future computations.
5. Add lines to the program so that it prints out and plots up the components of the velocity as a function of time.
6. Study your plots of the velocity components *versus* time and draw a line on

Listing 15.1 ProjectileAir.f90

```
!ProjectileAir.f90: Projectile Program using Dislin
! ------------------------------------
Subroutine buildPlot(Npts, x0, y0, v0, theta, k)
  Use dislin
  Implicit None
  Integer, intent(in) :: Npts
  Real*8, intent(in) :: x0, y0, v0, theta, k
  Real*8 :: dt, g, vx, vy, xmax, ymax, t
  Real*8, dimension(Npts) :: aXValues, aYValues, nXValues,
      nYValues
  Integer :: i
  t = 0.
  g = 9.81
  dt = 2*v0*sin(theta)/(Npts*g)
  vx = v0*cos(theta)
  vy = v0*sin(theta)
  nXValues(1) = x0
  nYValues(1) = y0
  do i=1,Npts,1
    t = (i-1)*dt
    ! Analytic
    aXValues(i) = x0+v0*cos(theta)*t
    aYValues(i) = y0+v0*sin(theta)*t-g*t*t/2.
    if(i>=2) then
      vx = vx-k*vx*dt
      vy = vy-g*dt-k*vy*dt
      ! Numberic
      nXValues(i) = nXValues(i-1) + vx*dt
      nYValues(i) = nYValues(i-1) + vy*dt
    end if
  end do
  ! Dislin plotting routines
  call metafl('XWIN')
  call disini
  call name('x-axis','X')
  call name('y-axis','Y')
  call labdig(-1,'X')
  call ticks(10,'XY')
  call titlin(' Analytic (green) vs Numerical',1)
  xmax = 2.*v0**2*cos(theta)*sin(theta)/g
  ymax = (v0*sin(theta))**2/(2.*g)
  call graf(x0,x0+xmax,x0,xmax/10.,y0,y0+ymax,y0,ymax/10.)
  call title()
  call color('RED')
  call curve(aXValues,aYValues,Npts)
  call color('GREEN')
  call curve(nXValues,nYValues,Npts)
  call color('FORE')
  call dash
  call disfin
End Subroutine buildPlot
!
Program projectileAir
Implicit None
! Set-up all needed parameters
Integer :: Npts = 200
Real*8 :: x0=0., y0=0.,v0=20., theta=3.14159265358979/4., k=0.25
call buildPlot(Npts, x0, y0, v0, theta, k)
End Program projectileAir
```

them that appears to be the terminal velocity for each.

7. Print out several plots showing the effect of air resistance for different initial angles θ. For each, determine the value of R, T, and H, and compare the three with the corresponding values for frictionless flight. All three should decrease.
8. We know that for frictionless flight, the maximum range R occurs for $\theta = 45^o$, and that there is the same range for trajectories with θ or $90^o - \theta$ (for example, for 30^o and 60^o). Investigate how both these statements change when drag is included. In other words, if you want to hit your golf ball with a maximum range, should you hit it at an angle that is greater than or less than 45^o?
9. Investigate and describe how the range R varies with initial velocity v_0 for a fixed value of θ. Explain the results you obtain (does it go further, for example?).
10. Up until this point, you have looked at results for the model of friction (15.3) with $n = 1$. This corresponds to a low-velocity model. Modify `ProjectileAir.f90` to investigate $n = 2$, (high-velocity model) and $n = 3/2$ (medium-velocity model). To make a realistic comparison, adjust the value of k for the latter two cases such that the initial force of friction, kV_0^n, is the same for all three cases.

Improved Algorithm: Verlet*

In developing the Euler integration algorithm (15.10), we ignored the direct affect of the acceleration on the position, as is present in (15.9). We now want to modify the algorithm so that it includes the acceleration term directly in the solution for $x(t + \Delta t)$. An approach that is nearly as simple as Euler yet appears to work very well is called the *Verlet algorithm*. It uses second-order terms for x_n, but only first-order ones for v_n:

$$x_{n+1} = x_n + v_n \Delta t + \frac{1}{2} a_n (\Delta t)^2, \qquad v_{n+1} = v_n + \frac{1}{2}(a_{n+1} + a_n)\Delta t. \quad (15.14)$$

We see that while the algorithm for the velocity is only first-order in Δt, it makes up for this somewhat by using the average acceleration over the time interval and not just its value at the beginning of the interval. The only problem with this method is that it is not fully self-contained; to determine v_{n+1} you first have to use the Euler method to obtain a_{n+1}.

Exercise: Implement the Verlet algorithm and compare the number of steps needed to obtain the three-place precision with both the Euler and Verlet methods. (The better algorithm should use fewer steps.) ♠

15.6 MAPLE: DIFFERENTIAL-EQUATION TOOLS

Maple is competent at finding solutions to most any differential equation that you give it. It many cases it can find an analytic solution with no help from you, while in other cases you may have to give it some hints based on your understanding of the mathematics of your equation. If all else fails, or if all you want is a plot of the solution, ask Maple to solve the differential equation numerically.

In this section we solve the projectile-with-drag problem in Maple. Because we have already done this with Fortran, it affords you the opportunity to compare the two approaches to the same problem. You should decide for yourself; we find the Fortran solution easier and faster.

Warning: Although the commands we have given here appear to work properly on various versions of Maple, we have noted that the *order* in which Maple returns the solution changes with different versions. To cite an instance, $[y(t), x(t)]$ in one case and $[x(t), y(t)]$ in another. Consequently, you may have to reverse the order of some of the arguments in the plotting commands given below.

We start by repeating the differential equations for projectile motion in a uniform gravitational field with no friction, and their solution:

$$\frac{d^2}{dt^2}\,x(t) = 0, \qquad \frac{d^2}{dt^2}\,\mathrm{y}(t) = -g, \tag{15.15}$$

$$x(t) = v_{o,\,x}\,t, \qquad y(t) = -g\,t^2 + v_{o,\,y}\,t. \tag{15.16}$$

where we have assumed $x(0) = y(0) = 0$, and used the symbols $v_{o,\,x}$ and $v_{o,\,y}$ to denote the x and y components of the initial velocity v_0. We start by solving these equations with Maple and seeing if we get these same solutions. To start simply, first we solve for just the y motion, and then for the simultaneous x and y motions. Then we add in air resistance. This careful approach to a final solution is standard in scientific problem solving.

To keep from repeatedly entering complicated expressions, we give our equations names and enter the names rather than long expressions. We write our differential equations using the `diff` command, and later show how to do it with the `D` operator (see Chapter 5 for our discussion of derivatives). Yet even if `diff` is used for the equations, `D` *must* be used for the initial conditions.

There are various ways to enter differential equations, and we demonstrate several of them (all of which give the same answer of course). We use the variables `diff_eq` and `diff_eq2` for two different representations of the equation for the y component of a projectile's motion, and solve each with the `dsolve` command:

```
> restart;      with(DEtools):                    # Initialize
> Vy := diff(y(t), t);                            # Vy = expression for velocity
```

$$Vy := \frac{d}{dt}\,\mathrm{y}(t)$$

```
> diff_eq := diff(Vy, t) = - g;       diff_eq := diff(diff(y(t),t),t) = - g;
> diff_eq := diff(y(t), t, t) = - g;      diff_eq := diff(y(t), t$2) = - g;
> diff_eq := diff(y(t), t$2) + g;                    # Multiple forms, same result
```

$$diff\_eq := \frac{d^2}{dt^2}\,\mathrm{y}(t) = -g$$

Two simpler ways of entering the same differential equation is with the `D` operator:

```
> diff_eq2 := (D2)(y)(t) = -g;      diff_eq2 := D(D(y))(t) = -g;        # With D
```

$$diff\_eq2 := (\mathrm{D}^{(2)})(y)(t) = -g$$

We solve this equation with the `dsolve`. Even though the command accepts a number of arguments, some of which direct Maple along specified paths for solutions, we leave it all up to Maple:

```
> dsolve(diff_eq);                                   # Solve ODE with dsolve
```

$$\mathrm{y}(t) = -\frac{g\,t^2}{2} + \_C1\;t + \_C2$$

We see that we do indeed get the general solution with integration constants $\_C1$ and $\_C2$ to be determined by the initial conditions (the leading underscores indicate that the computer has chosen these names).

We incorporate the initial conditions into the solution by setting $\_C1 = Vy0$ and $\_C2 = 0$, or by giving `dsolve` the initial conditions along with the equations and letting it do all the work for us:

```
> init_con := y(0) = 0,       D(y)(0) = Vyo ;                # Initial conditions
> dsolve( {diff_eq2, init_con}, y(t) );
```

$$init\_con := \mathrm{y}(0) = 0, \qquad \mathrm{D}(y)(0) = Vyo$$
$$\mathrm{y}(t) = -g\,t^2/2 + Vyo\,t$$

This looks great and is the right answer, yet we have had to be careful. First, even if the differential equation is entered with `diff`, the initial conditions *must* be entered with the D. Second, because we have a second-order ODE, there are two initial conditions (displacement and velocity), and we group them as the set `init_con`, with items separated by a comma, but order irrelevant.

As a check of the solution, we take the derivative of $y(t)$ to obtain an expression for the velocity, and then take the derivative of the velocity to see if we get the acceleration back. To do this, we need to keep in mind that Maple returns an expression in the form of an equation for $y(t)$ as the solution. Even though the expression looks like a function, it is not. Even though it is possible to transform $y(t)$ into a function, the procedure is not straightforward. (One needs to first define a procedure that contains the solution, and then create a function that calls the

procedure). We take a simpler approach in which we define the expression `soltn` to contain the solution, and then manipulate `soltn`:

```
> soltn := dsolve( {diff_eq2, init_con}, y(t) );     # Solve ODE with initial conditions
> diff( soltn, t );      diff( soltn, t, t );        # 1st, 2nd derivatives
```

$$soltn := \mathrm{y}(t) = -\frac{1}{2}\,g\,t^2 + Vyo\,t$$

$$\frac{d}{dt}\,\mathrm{y}(t) = -g\,t + Vyo \qquad\qquad \frac{d^2}{dt^2}\,\mathrm{y}(t) = -g$$

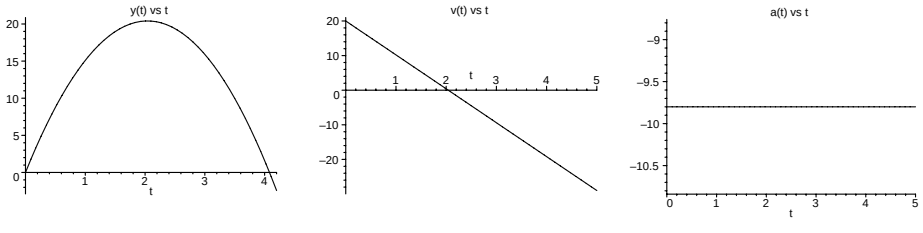
Extract Right-Hand Side: rhs

In order to plot the solution, all parameters must be assigned numerical values. In addition, since the expression for the solution is an equation with $y(t)$ on the LHS, we must use Maple's `rhs()` function to extract the RHS. We extract the velocity and acceleration from the solutions by taking time derivatives:

```
> g := 9.8;       Vyo := 20;
```

$$g := 9.8 \qquad\qquad Vyo := 20$$

```
> plot( rhs(soltn), t = 0..4.2, title = 'y(t) vs t' );              # y(t), left plot
> plot( rhs(diff(soltn, t)), t = 0..5, title = 'v(t) vs t' );  # Vy(t), right plot
> plot( rhs(diff(soltn, t, t) ), t = 0..5, title = 'a(t) vs t' ); # a, right plot
```



Check over how the velocity starts off positive and then gets more and more negative, and that the acceleration is constant at $g = 9.8$ in the negative y direction.

Second-Order ODE with Plot

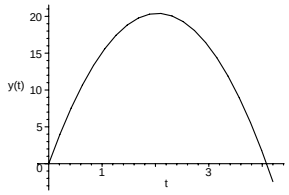
Maple has the `DEplot` command that both solves a differential equation and plots up the solutions. As before, while our solution was analytic and contained initial conditions as parameters, to plot a graph we need to have a completely numeric answer, and so we must assign values to all parameters. The value of the parameter

g is assigned first, and the values of the initial conditions are placed within the `DEplot` command:

```
> restart:    with(DEtools):                                  # Initialize, load DEplot
> diff_eq := diff( y(t), t, t ) = - g ;     g:= 9.8;          # Enter ODE
```

$$diff\_eq := \frac{d^2}{dt^2}\,\mathrm{y}(t) = -g \qquad g := 9.8$$

```
> DEplot( diff_eq, y(t), t = -0..4.2, [[y(0)=0,D(y)(0)=20 ]] );    # [init conds]
```



Observe how we include the initial conditions as a *list* for the third argument.

System of ODEs

When the systems in an environment depend on each other, we usually have simultaneous differential equations to solve. As a case in point, in our projectile problem we have simultaneous equations for the x and y motions (we solved only for the y motion so far). The same commands and procedures are used for a *set* of equations as for a single equation. The additional step is to define the variable that gave the equation to solve to now be a set of equations ("set" because the order does not matter):

```
> diff_eqs := ( D@@2 ) (y)(t) = -g,       (D@@2) (x)(t) = 0;
```

$$diff\_eqs := (\mathrm{D}^{(2)})(y)(t) = -g, \qquad (\mathrm{D}^{(2)})(x)(t) = 0$$

Here we use `D` to define the equations, although `diff` would do. The solution is obtained by specifying the set of equations `{diff_eqs}` as the first argument to `dsolve`, and the set `{y(t), x(t)}`, the solution we desire, as the second argument:

```
> dsolve( {diff_eqs}, {y(t), x(t)} );          # Maple will generate integration constants
```

$$\{\mathrm{y}(t) = -\frac{g\,t^2}{2} + \_C3\,t + \_C4 \qquad \mathrm{x}(t) = \_C1\,t + \_C2\}$$

We see that Maple does find the expected forms of the solution (15.16), with the constants still to be determined. As before, we include the initial conditions into the set of equations we give as the first argument to `dsolve`, this time with four conditions:

```
> init_cons := y(0)=0,   D(y)(0)=Vyo,      x(0)=0,      D(x)(0)=Vxo ;
```

$$init\_cons := \mathrm{y}(0) = 0, \qquad \mathrm{D}(y)(0) = Vyo, \mathrm{x}(0) = 0, \qquad \mathrm{D}(x)(0) = Vxo$$

```
> dsolve( {diff_eqs, init_cons}, {y(t), x(t)} );          # Includes initial condition
```

$$\{\mathrm{y}(t) = -\frac{1}{2}\,g\,t^2 + Vyo\,t, \mathrm{x}(t) = Vxo\,t\}$$

Indeed, we get the familiar solutions as a set within braces. If we want numeric values for x and y, we must assign numeric values to the parameters.

15.7 MAPLE SOLUTION: DRAG $\propto$ VELOCITY

We now return to our projectile-with-drag problem, applying some of the Maple tools we have just learned. If the frictional force is proportional to the first power of the velocity, the equations to solve are:

$$\frac{d^2x}{dt^2} = -k\,v_x, \qquad \frac{d^2y}{dt^2} = -g - k\,v_y. \tag{15.17}$$

Because the x and y motions are independent, there is no mixing of the x motion into the y equations, and vice versa, and we could actually solve each equation separately. However, simultaneous equations generally are coupled, and we will solve them as if they were. We enter the equations into `dsolve` as a *set* with elements separated by commas. We use a set because order does not matter, and later we will place brackets around the elements. We define a variable `diff_eqs` as the set of equations, and then solve the set. Because the equations involve the second derivatives, there will be a `diff` with respect to `t$2` and `t`:

```
> diff_eqs := diff( x(t), t$2 ) = -k * diff( x(t), t),
      diff( y(t), t$2) = -g -k * diff( y(t), t );
```

$$diff\_eqs := \frac{d^2}{dt^2}\,\mathrm{x}(t) = -k\,(\frac{d}{dt}\,\mathrm{x}(t)), \qquad \frac{d^2}{dt^2}\,\mathrm{y}(t) = -g - k\,(\frac{d}{dt}\,\mathrm{y}(t))$$

Observe that we must indicate the t dependence of x and y as $x(t)$ and $y(t)$, because otherwise Maple would treat them as constants whose derivatives vanish.

Now we see if Maple is smart enough to find an analytic solution. We enter the set of equations as the first argument to `dsolve` with braces around `diff_eqs` to indicate that it is a set:

```
> dsolve({diff_eqs});                                    # Solve differential equation
```

$$\{\mathrm{x}(t) = \_C3 + \_C4\,e^{(-k\,t)}, \qquad \mathrm{y}(t) = -\frac{\_C1\,e^{(-k\,t)} + g\,t - \_C2\,k}{k}\}$$

We see that Maple returns a set of solutions with four constants. Good, we are on our way. Rather than solving for the constants in terms of the initial conditions, we define a set of initial conditions, and include them as a second argument to `dsolve`.

We specify initial velocities as the initial values for the first derivatives, and use the `D()` operator (`diff` does not work for initial conditions):

```
> InitConds := x(0) = 0,      D(x)(0) = Vox,      y(0) = 0,      D(y)(0) = Voy;
> soltn := dsolve( {diff_eqs,InitConds} );
```

$$InitConds := \mathrm{x}(0) = 0, \quad \mathrm{D}(x)(0) = Vox, \quad \mathrm{y}(0) = 0, \quad \mathrm{D}(y)(0) = Voy$$

$$soltn := \left\{ \mathrm{x}(t) = \frac{Vox}{k} - \frac{Vox\, e^{(-k\,t)}}{k}, \quad \mathrm{y}(t) = -\frac{\frac{(g+Voy\,k)\, e^{(-k\,t)}}{k} + g\,t - \frac{g+Voy\,k}{k}}{k} \right\}$$

This appears to be the analytic solution we want, and it is worth investigating its properties. Yet before we do that, we need to see if the velocities exhibit reasonable behaviors, that is, if they attain a terminal speed and then stop increasing. We do that by taking the derivative of our solution:

```
> diff(soltn,t);                                  # Take derivative of y(t) to get velocity
> diff(y(t),t) = -(-(g+Voy*k)*exp(-k*t)+g)/k};
```

$$\{\frac{d}{dt}\,\mathrm{x}(t) = Vox\, e^{(-k\,t)}, \qquad \frac{d}{dt}\,\mathrm{y}(t) = -\frac{-(g + Voy\,k)\, e^{(-k\,t)} + g}{k}\}$$

Indeed, we see that the x component of velocity decreases exponentially as a function of time from its initial value and approaches zero for long times. This is a consequence of the frictional force always opposing the velocity in the x direction, but with no other force present to increase the velocity in the x direction. In contrast, the y velocity approaches a terminal value -g/k for long times. This is a consequence of the force of gravity $F = -mg$ exactly balancing the frictional force $F = mkv$ when $v = -g/k$:

```
> Vox := 22*cos(Pi/4);     Voy := 22*sin(Pi/4);     := 9.8;     k := 1.0;
```

$$Vox := 11\sqrt{2} \qquad Voy := 11\sqrt{2} \qquad g := 9.8 \qquad k := 1.0$$

Once the parameters are assigned, we check the solution:

```
> soltn;
```

$$\{\mathrm{x}(t) = 11.0000\sqrt{2} - 11.0000\sqrt{2}\, e^{(-1.0\,t)},$$
$$\mathrm{y}(t) = -1.00000\,(9.8 + 11.0\sqrt{2})\, e^{(-1.0\,t)} - 9.80000\,t + 9.80000 + 11.0000\sqrt{2}\}$$

We see our solution in its numeric form, as a set with a comma separating the equation for $x(t)$ from that of $y(t)$. Though they make the equations hard to read, the zeros after the decimal point indicate the Maple's level of precision for this floating-point calculation (it became floating point when we made g and k floating-point numbers).

15.8 EXTRACT OPERANDS

To plot up the results we need to extract the RHSs of both equations. For a simple expression this was done with the `rhs` command. Now that `soltn` is a set of equations, we need to indicate which equation it is in the set that we want the RHS of. For this purpose Maple has the `op` command to extract *operands* (pieces) from an expression. Consequently, `rhs(x)` is equivalent to `op(2, x)`. First we test that we know how to extract each equation, and then we take RHSs:

```
> op(1,soltn);          op(2,soltn);          # Extract operand 1, 2 from soltn
```

$$\mathrm{x}(t) = 11.000\sqrt{2} - 11.0000\sqrt{2}\,e^{(-1.0\,t)}$$
$$\mathrm{y}(t) = -1.00000\,(9.8 + 11.0\sqrt{2})\,e^{(-1.0\,t)} - 9.800000\,t + 9.80000{+}11.0000\sqrt{2}$$

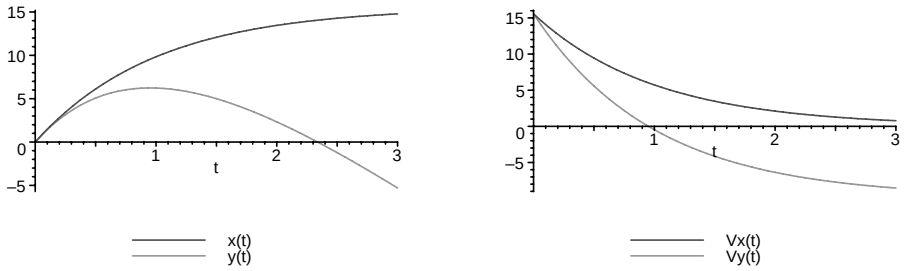
Now that we have a way to separate out each equation, we take the RHSs of each:

```
> rhs(op(1,soltn));       rhs(op(2,soltn));       # Extract RHSs of x(t), y(t) solution
```

$$11.00000000\sqrt{2} - 11.00000000\sqrt{2}\,e^{(-1.0\,t)}$$
$$-1.00000\,(9.8 + 11.0\sqrt{2})\,e^{(-1.0\,t)} - 9.80000\,t + 9.80000{+}11.0000\sqrt{2}$$

At last we have what we need to make our plots. We plot the position and velocity (time derivative) by entering lists (in square brackets) as the first argument to the `plot` command (a list because order matters for the plot):

```
> plot( [rhs(op(1,soltn)), rhs(op(2,soltn))], t = 0..3, legend = [`x(t)`, `y(t)`] );
> plot( [diff(rhs(op(1, soltn)), t), diff(rhs(op(2, soltn)), t )],
        t = 0..3, legend = [`Vx(t)`, `Vy(t)`]);
```

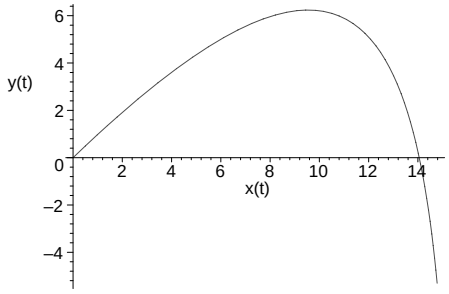


These plots show an initial linear increase of $x(t)$ with time t, as expected for constant velocity. However, as time increases, the action of the frictional force in the x direction is to keep decreasing V_x until, for large time, V_x vanishes (except that the ground gets in the way). Then the x motion ceases and x remains constant.

In turn, $y(t)$ shows a parabolic dependence on time initially, as expected for uniform acceleration, but only a linear increase for later times. The linear increase of y with time indicates that v_y has attained its terminal value as the drag force exactly balances that of gravity. As expected from the analytic expressions, and our discussion in the previous paragraph, we see that V_x approaches zero and V_y approaches a constant value for large times (or would if the ground did not get in the way).

The usual plot of a trajectory is a plot of $y(t)$ as the ordinate *versus* $x(t)$ as the abscissa. As you will recall from our discussion of visualization in Chapter 4, this type of plot is known as a *parametric plot*. A parametric plot is constructed by moving the time limits into the list that is used as the first argument (the list contains $\{x(t), y(t)\}$ and the range of t):

```
> plot([rhs(op(1, soltn)), rhs(op(2, soltn)), t = 0..3 ], labels=[`x(t)`, `y(t)`]);
```



We see a trajectory that is much distorted from the symmetric parabola that occurs for frictionless flight, and looks similar to that computed with Fortran and shown in Figure 12.1. In fact, this looks very much like the type of trajectory seen for a golf ball or a baseball in which the ball rises at first and then appears to "drop out of the sky" at the end of its trajectory.

Solution for R and T

We now want to determine if we can solve for the range R and hang time T for the general projectile-with-friction problem. We start by again placing our solution in symbolic form:

```
> diff_eqs := diff(x(t), t$) =                                    # DE for x, y motion
     -k * diff(x(t), t),     diff(y(t), t$) = -g -k * diff(y(t), t);
```

$$diff\_eqs := \frac{d^2}{dt^2}\,\mathrm{x}(t) = -k\,(\frac{d}{dt}\,\mathrm{x}(t)) \qquad \frac{d^2}{dt^2}\,\mathrm{y}(t) = -g - k\,(\frac{d}{dt}\,\mathrm{y}(t))$$

```
> InitConds := x(0) = 0, D(x)(0) = Vox, y(0) = 0, D(y)(0) = Voy;
```

$$InitConds := \mathrm{x}(0) = 0,\ \mathrm{D}(x)(0) = Vox,\ \mathrm{y}(0) = 0,\ \mathrm{D}(y)(0) = Voy$$

```
> soltn := dsolve( { diff_eqs, InitConds } );
```

$$soltn := \left\{ \mathrm{x}(t) = \frac{Vox}{k} - \frac{Vox\, e^{(-k\,t)}}{k},\ \mathrm{y}(t) = -\frac{\frac{(g + Voy\,k)\, e^{(-k\,t)}}{k} + g\,t - \frac{g + Voy\,k}{k}}{k} \right\}$$

The range R corresponds to the value of x at which the height $y = 0$ (in addition to the initial firing). We start our solution for the hang time T at which $y(T) = 0$ by extracting $x(t)$ and $y(t)$:

```
> X := rhs(op(1, soltn));        Y := rhs(op(2, soltn));     # Extract RHS x(t), y(t)
```

$$X := \frac{Vox}{k} - \frac{Vox\, e^{(-k\,t)}}{k} \qquad Y := -\frac{\frac{(g + Voy\,k)\, e^{(-k\,t)}}{k} + g\,t - \frac{g + Voy\,k}{k}}{k}$$

Now we use `solve` to find the time T at which $Y(T) = 0$:

```
> solve(Y, t);
```

$$0$$

This just tells us the initial time, which we know, but is not what we want. Instead, let us be more specific and ask for the time at which the height y just becomes negative, namely, when it just hits the ground:

```
> solve(Y < 0, t);
```

We see that Maple returns nothing. It apparently has given up. If we look at the expression for $y(t)$, we see that it contains the time t in both an exponential and a linear term. Apparently, setting $Y = 0$ leads to a transcendental equation that has no analytic solution. The only solution, then, is obtained numerically, and we shall do that with Java (although Maple will also work).

15.9 DRAG $\propto V^2$ (EXERCISE)

Modify the analysis performed for a drag force proportional to the first power of the velocity so that it is appropriate for a force proportional to the square of the velocity. Maple should be able to solve this analytically as well, and you should be able to follow all of the steps we have for Model 1. *Note:* since v^2 does not change sign when v does, you will need to add a $v/|v|$ factor in the frictional force for the y motion (it is not needed for the x motion because the x component of velocity is always positive).

15.10 DRAG $\propto V^{3/2}$

We now want to solve the projectile-with-friction problem for a drag force proportional to the 3/2 power of the velocity. The equations we need to solve are:

$$\frac{d^2\,x}{dt^2} = -k\,v_x^{3/2}, \qquad \frac{d^2\,y}{dt^2} = -g - k\,v_y^{3/2}\frac{v_y}{|v_y|}. \tag{15.18}$$

The acute reader might notice that there is a problem with this last equation; namely, if v_y is negative, then the square root operation, which is part of raising v_y to the 3/2 power, will return an imaginary number. This is clearly not what we want. We solve that problem by taking the absolute value of v_y before raising it to a power:

$$\frac{d^2y}{dt^2} = -g - k\left|\frac{dy}{dt}\right|^{3/2}\frac{v_y}{|v_y|}. \tag{15.19}$$

This expression is simpler to read and easier to enter into Maple. (An alternative approach would be to use the `sign` function, which extracts the sign of its argument, to keep track of the sign of the velocity and adjust the frictional force appropriately.)

We set about solving this model just as we did the problem with drag proportional to the first power. First we define the set of differential equations and give it the name `diff_eqs`:

```
> restart; with(DEtools):                                   # Clean the slate
> diff_eqs := diff(x(t),t$2) = -k * diff(x(t), t)^(3/2),    # DE
diff(y(t),t$2) = -g - (diff(y(t),t)/abs(diff(y(t),t))) *k*abs(diff(y(t), t))^(3/2);
```

$$diff\_eqs := \frac{d^2}{dt^2}\,\mathrm{x}(t) = -k\,(\frac{d}{dt}\,\mathrm{x}(t))^{(3/2)},\quad \frac{d^2}{dt^2}\,\mathrm{y}(t) = -g - (\frac{d}{dt}\,\mathrm{y}(t))\sqrt{\left|\frac{d}{dt}\,\mathrm{y}(t)\right|}\,k$$

This looks fine, and so we go on to look for a general solution by first entering the initial conditions:

```
> InitConds := x(0) = 0, D(x)(0) = Vox, y(0) = 0, D(y)(0) = Voy;
```

$$InitConds := \mathrm{x}(0) = 0, \qquad \mathrm{D}(x)(0) = Vox,\ \mathrm{y}(0) = 0, \mathrm{D}(y)(0) = Voy$$

Warning: You may have to stop the following command by hand. We now ask Maple to solve the differential equations labeled `diff_eqs` with the initial conditions labeled `InitConds`:

```
> soltn := dsolve( [diff_eqs, InitConds], [x(t), y(t)] );
```

$$soltn := \cdots \texttt{\_RootOf} \cdots$$

We see from the time it takes Maple to search for a solution, from the volume of output that Maple produces, and from the `_RootOf` command being returned, that Maple is having trouble finding a simple solution to our problem. In any case, an

analytic solution with this level of complexity is not illuminating, and probably not even good for computation (the numerous subtractions and evaluations of the multivalued $\tan^{-1}$ and $\ln$ functions is error-prone).

Now we try a *numerical* approach. We start it by assigning values to the initial conditions and parameters:

```
> Vox := 22.*cos(Pi/4);      Voy := 22.*sin(Pi/4);      g := 9.8;      k := 1/5.;
```

$$Vox := 15.556349186 \qquad Voy := 15.556349186 \qquad g := 9.8 \qquad k := 0.2000$$

We get a numeric solution using the same procedures as before, only now by including the `type=numeric` argument to `dsolve`:

```
> soltn := dsolve( [diff_eqs, InitConds], [x(t), y(t)], type=numeric );
```

$$soltn := \textbf{proc}(x\_rkf45) \ldots \textbf{end proc}$$

Regardless of this not appearing to be a clear signal that a solution is in hand, this is Maple's way of telling us that it has written a *procedure* named `soltn` that will produce a numerical solution to the equations. Recall, a procedure in Maple is like a function that requires more than one line to define. It is like a *method* in Java or a subroutine in Fortran. In the present case, the argument to the procedure tells us that the procedure takes an argument and then returns the solution. Since Maple assumes we are solving for $y(x)$, the argument is indicated generically as x. In our case, the second argument to the `solve` command was the list `[x(t), y(t)]`, and so we are solving for x and y as functions of the time t. Therefore the argument to the procedure is the time t. (The subscript `rkf45` to `x` is meant for aficionados; it indicates that a fourth order Runga-Kutta numerical method is used, and that it uses adaptive step size to obtain fifth-order precision.)

We now have a procedure `soltn(t)` that numerically solves the differential equations for a single input value of time t and returns `[x(t), y(t)]` in some form. To see how this works, let us look at the solution returned for time 0 (which should just be the initial conditions we entered):

```
> soltn(0);
```

$$[t = 0.,\ \mathrm{x}(t) = 0.,\ \frac{d}{dt}\,\mathrm{x}(t) = 15.556349186,\ \mathrm{y}(t) = 0.,\ \frac{d}{dt}\,\mathrm{y}(t) = 15.556349186]$$

OK, the solution appears to be working properly and returning `[t,x(t),Vx(t),y(t),Vy(t)]`. So we try some nonzero times:

```
> soltn(0.1);      soltn(0.5);
```

$$[t = 0.1,\ \mathrm{x}(t) = 1.496606466606704,\quad \frac{d}{dt}\,\mathrm{x}(t) = 14.3981782622785151,$$
$$\mathrm{y}(t) = 1.44943382860698944,\quad \frac{d}{dt}\,\mathrm{y}(t) = 13.47199594152410]$$
$$[t = 0.5,\ \mathrm{x}(t) = 6.49693016854386762,\quad \frac{d}{dt}\mathrm{x}(t) = 10.8534715090049546,$$

$$\mathrm{y}(t) = 5.458067069137789,\ \ \frac{d}{dt}\,\mathrm{y}(t) = 6.980924610035919]$$

We use Maple to pick out the individual pieces of the solution:

```
> Soltn := soltn(0.5);                                    # Store list of solutions in Soltn
```

$$Soltn := [t = 0.5,\ \mathrm{x}(t) = 6.49693016854386762,\ \frac{d}{dt}\,\mathrm{x}(t) = 10.8534715090049546,$$
$$\mathrm{y}(t) = 5.45806706913778950,\ \frac{d}{dt}\,\mathrm{y}(t) = 6.98092461003591946]$$

```
> op(1, Soltn);  op(2, Soltn); op(3, Soltn); op(4, Soltn); op(5, Soltn);
```

$$t = 0.5 \qquad \mathrm{x}(t) = 6.49693016854386762 \qquad \frac{d}{dt}\,\mathrm{x}(t) = 10.8534715090049546$$
$$\mathrm{y}(t) = 5.45806706913778950 \qquad \frac{d}{dt}\,\mathrm{y}(t) = 6.98092461003591946$$

In order to plot the solutions, we pick off just the RHS of these expressions:

```
> rhs(op(2, Soltn));      rhs(op(4, Soltn));      # Extract RHS of x(t), y(t) in Soltn
```

$$6.49693016854386762 \qquad 5.45806706913778950$$

Defining Functions from Procedures

It seems like we should now be able to plot up the solution. However, Maple's `plot` command accepts expressions and functions as input but not procedures. (We discussed Maple programming and procedures in Chapter 8.) This means we cannot have `plot` call our procedure and plot it. It is not much work to define a function from a procedure and then to plot up the function; essentially, you just define an arrow function as a call to your procedure. Here we do that for the x and y positions and velocities:

```
> X := (t) -> rhs(op(2, soltn(t)));       Y := (t) -> rhs(op(4, soltn(t)));
> Vx := (t) -> rhs(op(3, soltn(t)));    Vy := (t) -> rhs(op(5, soltn(t)));
```

$$X := t \to \mathrm{rhs}(\mathrm{op}(2,\ \mathrm{soltn}(t))) \qquad Y := t \to \mathrm{rhs}(\mathrm{op}(4,\ \mathrm{soltn}(t)))$$
$$Vx := t \to \mathrm{rhs}(\mathrm{op}(3,\ \mathrm{soltn}(t))) \qquad Vy := t \to \mathrm{rhs}(\mathrm{op}(5,\ \mathrm{soltn}(t)))$$

```
> plot([X,Y], 0..3, title = "x(t) & y(t), 3/2-power friction");        # Plot 1
> plot([Vx,Vy], 0..3, title = "Vx(t) & Vy(t), 3/2-power friction");    # Plot 2
> plot([X,Y,0..3], title = "y(t) vs x(t) for 3/2-power friction");     # Plot 3
```

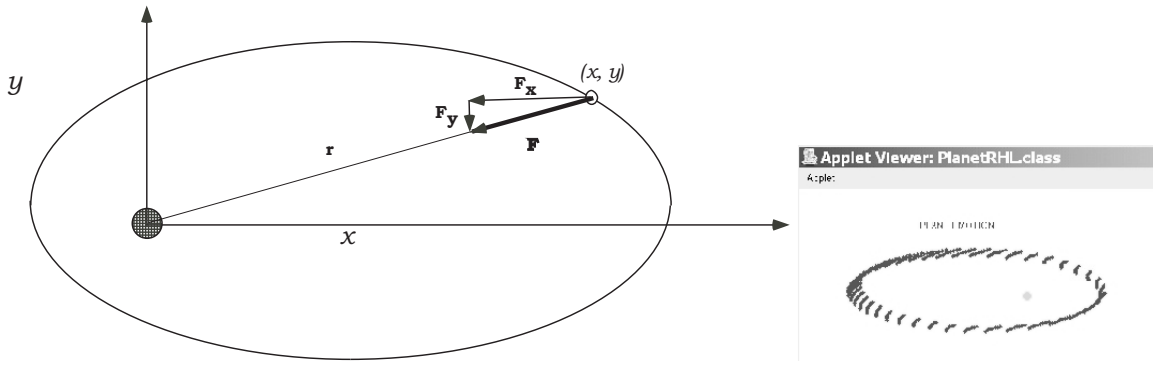
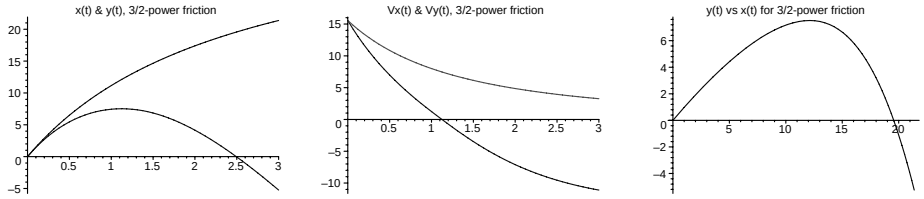


Figure 15.1 *Left:* The gravitational force on a planet a distance r from the sun. The x and y components of the force are indicated. *Right:* Output from the applet `PlanetRHL` showing the precession of a planet's orbit when the gravitational force $\propto 1/r^4$.



We see similar results to Model 1, only now with an even more drastic "drop out of the sky effect" at the end of the trajectory.

15.11 EXPLORATION: PLANETARY MOTION*

Newton's explanation of the motion of the planets in terms of a universal law of gravitation is one of the great achievements of science. He was able to prove that the planets traveled along elliptical paths with the sun at one vertex, and with periods that agree with observation. All Newton needed to postulate is that the force between a planet of mass m and the sun of mass M is given by

$$F = -\frac{GmM}{r^2}, \tag{15.20}$$

where r is the distance between the planet of mass m and sun of mass M, and G is a universal constant. The minus sign indicates that the force is always attractive and lies along the line connecting the planet and sun, as indicated on the left of Figure 15.1. The hard part for Newton was solving the resulting differential equations, since he had to invent calculus to do it. Whereas the analytic solution is complicated, the numerical solution is not.

Even for planets, the basic equation of motion is

$$\mathbf{F} = m\mathbf{a} = m\frac{d^2\mathbf{x}}{dt^2}, \tag{15.21}$$

with the force now given by (15.20). If we look at Figure 15.1 we see that

$$F_x = F\cos\theta = F\frac{x}{r}, \qquad F_y = F\sin\theta = F\frac{y}{r}, \qquad r = \sqrt{x^2+y^2}, \tag{15.22}$$

with F given by (15.20). If we write the equation of motion in component form and substitute (15.22) for the force components, we obtain the differential equations we need to solve:

$$\frac{d^2x}{dt^2} = -GM\frac{x}{r^3}, \qquad \frac{d^2y}{dt^2} = -GM\frac{y}{r^3}. \tag{15.23}$$

Implementation: Planet.java*

On the Java section of the CD you will find the applet `Planet`. Although this is Java based, you can run the applet with just a browser. We suggest that you run the class file to see how the solution behaves. To keep the calculation simple without changing the physics, assume that the units we use are such that $GM = 1$, and that the initial conditions are [Feyn 63]:

$$x(0) = 0.5, \qquad y(0) = 0, \qquad v_x(0) = 0.0, \qquad v_y(0) = 1.63. \tag{15.24}$$

1. Modify `projectileAir.java` so that it solves (15.23) as functions of time.
2. Make the number of time steps large enough so that the planet's orbit repeats on top of itself.
3. You may need to make the time step small enough so that the orbit closes upon itself (it should) and just repeats. This should be a nice ellipse.
4. Experiment with initial conditions until you obtain the ones that produce a circular orbit (a circle is a special case of an ellipse).
5. Once you have good precision, see the effect of progressively increasing the initial velocity until the orbit opens up and becomes an hyperbola.
6. Use the same initial conditions as produced the ellipse. This time investigate the effect of the power in (15.20) being $1/r^4$ rather than $1/r^2$. You should find that the orbital ellipse now rotates (precesses), as in Figure 15.1.

15.12 KEY WORDS

equations of motion	Euler's method	forward difference
numerical ODE solution	numerical differentiation	ODE

15.13 SUPPLEMENTARY EXERCISES

1. Use Maple, Fortran, or both to find the solution of the differential equation
$$\frac{dn(t)}{dt} = -\lambda n(t), \tag{15.25}$$
with the initial condition $n(0) = 100$. Plot up the answer for $\lambda = 0.3$.
2. In Chapter 16 you will encounter an RLC electrical circuit and the differential equation describing it. We consider here the same circuit without a capacitor and described by the differential equation
$$V(t) = RI + L\frac{dI}{dt}. \tag{15.26}$$
Solve this equation for the current in the circuit for the same values of R and L as in the problem, and for a constant $V(t)$ (same magnitude as in problem). Plot your solution.
3. Consider exponential growth starting with an initial population $N(0) = 2$.
   a. Solve the equation describing exponential growth
   $$\frac{dN(t)}{dt} = \lambda N(t). \tag{15.27}$$
   Assign the integration constants to correspond to $n(0) = 100$, and $\lambda = 0.3$, and plot up the solution.
   b. Solve for exponential growth with a modulated growth parameter
   $$\frac{dy(t)}{dt} = \lambda\left[\frac{10000 - N(t)}{10000}\right] \times N(t). \tag{15.28}$$
   This is the differential-equation version of the logistics map studied in Chapter 19, "Discrete Math, Arrays as Bins." The term in square brackets is insignificant for small $N(t)$.
4. Solve for and plot up the solution to the differential equation
$$\frac{dy(x)}{dx} = \sin(xy), \tag{15.29}$$
with initial condition $y(0) = 1$.
   a. Verify that Maple cannot find an analytic solution to this equation.
   b. Have Maple find a numerical solution to this equation.
5. Solve for and plot up the solution to the differential equation
$$\frac{d^2y(x)}{dx^2} = -y(x)^3, \tag{15.30}$$
subject to the initial conditions $y(0) = 1$, $y'(0) = 0$. *Hint:* If Maple cannot find an analytic solution, you may need to try a numerical one.

Chapter Sixteen

Object-Oriented Programming, Abstract Data; Complex Currents*

Note to the instructor: As an alternative to the full version of this chapter, you may skip the study of the RLC circuit and just focus on the mathematics of complex numbers and their representation in terms of objects. And even with that, you may defer the use of *nonstatic* (object-oriented) methods to a later time.

16.1 PROBLEM: RESONANCE IN RLC CIRCUIT

We are given the circuit shown on the left of Figure 16.1 containing a resistor of resistance R, an inductor of inductance L, and a capacitor of capacitance C. All three elements are connected in series to an alternating voltage source

$$V(t) = V_0 \cos \omega t. \tag{16.1}$$

Problem: Determine the magnitude and time dependence of the current in this RLC circuit as a function of the frequency of the external voltage. We will solve the RLC circuit problem for you within this chapter. *Your problem* is to repeat the calculation for a circuit in which there are two RLC circuits in parallel, as shown on the right of Figure 16.1. You may assume a single value for inductance and capacitance, and three values for resistance:

$$L = 1000 \text{ H}, \qquad C = \frac{1}{1000} \text{ F}, \qquad R = \frac{1000}{1.5}, \ \frac{1000}{2.1}, \ \frac{1000}{5.2} \ \Omega. \tag{16.2}$$

Consider frequencies of applied voltage in the range $0 < \omega < 2/\sqrt{LC} = 2/s$.

16.2 MATH: COMPLEX NUMBERS

Try to remember your first exposure to square roots in elementary school. Though it was straightforward to understand that $5^2 = 25$, it was a challenge to understand that $\sqrt{25} = \pm 5$, and more of a challenge to understand what was the true value of $\sqrt{24}$. For many of us, it was downright impossible to understand the true value of $\sqrt{-1}$. Mathematicians, being a rather clever and proud bunch, have handled this

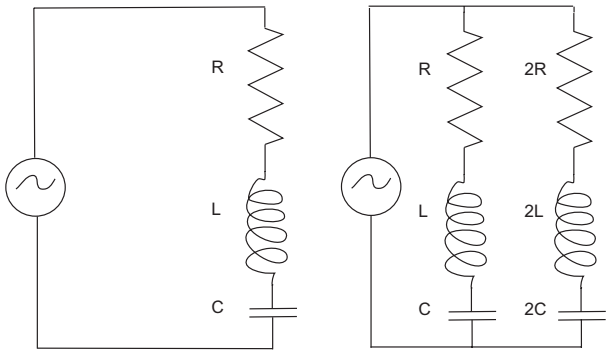


Figure 16.1 *Left:* An RLC circuit connected to an alternating voltage source. *Right:* Two RLC circuits connected in parallel to an alternating voltage. Observe that one of the parallel circuits has double the values of R, L, and C as does the other.

affront to their abilities by inventing the number i as the answer,[1]

$$i \stackrel{\text{def}}{=} \sqrt{-1}, \qquad i^2 = -1, \tag{16.3}$$

where the "def" over the equal sign indicates a definition. In this way mathematicians have a way of going ahead with their calculations, even if they do not know what i means. Whereas defining away one's ignorance may appear to be a swindle, mathematicians are an honest bunch at heart and so tell the world that they have really just made up the answer by calling i the *imaginary* number. "Imaginary" is a good name for i, since there is no way to measure this number in the real world, but, then again, there is no law forbidding us from imagining such a number.

In common mathematical usage, i is called *the* imaginary number, while any multiple of i is called *an* imaginary number. So, for example, $2i$, i, and $100i$ are all imaginary numbers. Once we have extended our minds to imagine an imaginary number, it is not much of a stretch to imagine adding a real number to an imaginary number to form something more complex. To name an instance, $1 + i$, $1 + 2i$, and $100 + 76i$. These numbers with both real and imaginary parts are called *complex numbers*.[2] The term "complex" indicates that these numbers have a number of parts, and *not* that they are hard to understand!

Complex numbers are very useful in mathematics and science since they let us double our work output with only the slightest increase in input. This is accomplished by employing the familiar operations of algebra and calculus on complex numbers, and then separating off the real and imaginary parts of the answer at the end. By way of example, even if z is a complex number, z^3/z still equal z^2, without our having to express the individual numbers in terms of their real and imaginary parts.

[1] The invention is credited to Girolama Cardana (1501–1576).
[2] The term, as well as much in applied mathematics, is credited to Carl Friedrich Gauss (1777–1855).

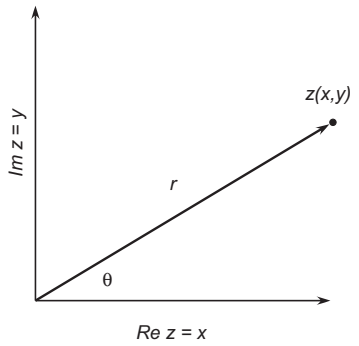


Figure 16.2 Representation of a complex number as a vector in space.

To do algebra with complex numbers, we define the symbol z to represent a number with both real and imaginary parts:

$$z = x + iy. \tag{16.4}$$

In turn, the complex nature of z is indicated by giving its real and imaginary parts:

$$\text{Re } z = x, \qquad \text{Im } z = y. \tag{16.5}$$

In a strict sense, y is the magnitude of the imaginary part of z, and iy is the imaginary part. Yet y is usually called "the imaginary part."

Because complex numbers have independent real and imaginary parts, a useful way to visualize them is to imagine a coordinate system in which the ordinate points off into imaginary space and the abscissa points off into real space. As seen in Figure 16.2, we then visualize $z = x + iy$ as a point with y projection along the imaginary axis and x projection along the real axis. This is analogous to a *vector* in a 2-D space, except that part of this vector lies in an imaginary space. The analogy between complex numbers and 2-D vectors is taken one step further by applying the *polar coordinate* representation of a vector to complex numbers. On account of this, the point z in Figure 16.2 may be located not only by giving its Cartesian coordinates x and y, but also by specifying it polar coordinates r, the length of the vector from the origin to point z, and θ, the angle that the vector makes with the abscissa. The complex number is the same in either case, and, indeed, the two representations are related via simple trigonometry:

$$\begin{array}{rclcrcl} r & = & \sqrt{x^2 + y^2}, & & \theta & = & \tan^{-1}(y/x), \\ x & = & r\cos\theta, & & y & = & r\sin\theta. \end{array} \tag{16.6}$$

Complex Arithmetic Review

The essence of the computing aspect of our problem is the programming of the rules of arithmetic for complex numbers. This is an interesting chore because while Java contains all the rules for real numbers, you must educate Java as to the rules for complex numbers. Indeed, since complex numbers are not *primitive data types* like *doubles* and *floats*, we will construct complex numbers as *objects*.[3]

We start with two complex numbers, which we distinguish with subscripts:

$$z_1 = x_1 + i\,y_1, \tag{16.7}$$
$$z_2 = x_2 + i\,y_2. \tag{16.8}$$

The rules of arithmetic follow by applying algebra to the real and imaginary parts:

Addition:
$$z_1 + z_2 = (x_1 + x_2) + i(y_1 + y_2), \tag{16.9}$$

Subtraction:
$$z_1 - z_2 = (x_1 - x_2) + i(y_1 - y_2), \tag{16.10}$$

Multiplication:
$$\begin{aligned} z_1 \times z_2 &= (x_1 + iy_1) \times (x_2 + iy_2), \\ &= (x_1x_2 - y_1y_2) + i(x_1y_2 + x_2y_1), \end{aligned} \tag{16.11}$$

Division:
$$\begin{aligned} \tfrac{z_1}{z_2} &= \frac{x_1 + iy_1}{x_2 + iy_2} \times \frac{x_2 - iy_2}{x_2 - iy_2}, \\ &= \frac{(x_1x_2 + y_1y_2) + i(y_1x_2 - x_1y_2)}{x_2^2 + y_2^2}. \end{aligned} \tag{16.12}$$

In deducing the rule for complex division we employed the fact that a complex number z multiplied by its *complex conjugate*

$$z^* = x - iy, \tag{16.13}$$

always yields a real number:

$$z \times z^* = (x + iy)(x - iy) = x^2 + y^2. \tag{16.14}$$

We observe that this result is the same as the square of the length r defined in (16.6). The length r is commonly referred to as the *modulus* or *magnitude* of the complex number z, and is expressed by using the absolute value symbol: $r = |z|$. The product of the complex number z and its complex conjugate z^* is then

$$zz^* = |z|^2 = r^2. \tag{16.15}$$

Exercise: Consider the complex numbers

$$b = 1 + 2i, \qquad c = 4 + i. \tag{16.16}$$

What are the values of $b + c$, $b - c$, $b \times c$, $|c|$, and b/c? ♠

[3]Because complex numbers are used so often in science and engineering, there is a move afoot to have some future versions of Java incorporate complex numbers as a primitive data type in order to speed up execution and make them easier to use.

We end our little review with an examination of functions of complex numbers. The new idea here is an ingenious theorem, derived by Euler, that if you raise e, the base of the natural logarithm system, to a purely imaginary power, then you get a complex number with a sine and cosine as its real and imaginary parts:

$$e^{i\theta} = \cos\theta + i\sin\theta \qquad \text{(Euler's theorem)}. \tag{16.17}$$

If the angle θ is real and in radians, then $\cos\theta$ and $\sin\theta$ are identified with the projection of $\exp i\theta$ along the real and imaginary axes, respectively. So if we look at both Figure 16.2 and Eqs. (16.4) and (16.6), we see that it is also possible to express a complex number z in terms of its polar representation:

$$z \equiv x + iy = re^{i\theta} = r\cos\theta + i\,r\sin\theta. \tag{16.18}$$

An important application of Euler's theorem is to define what it means to raise a number to a complex power z, for example,

$$e^z = e^{x+iy} = e^x e^{iy} = e^x(\cos y + i\sin y). \tag{16.19}$$

We shall find (16.19) useful in our exercises.

16.3 THEORY: RESISTANCE BECOMES IMPEDANCE

The basic rules of circuit theory are called Kirchoff's laws [R & M 93], and we now apply one of them to the circuit on the left of Figure 16.1. We work our way around the circuit, setting the external voltage $V(t)$ equal to the sum of the voltage drops across the resistor, the inductor, and the capacitor. If $I(t)$ is the current in the circuit, we end up with the basic differential equation of circuit theory

$$\frac{dV(t)}{dt} = R\frac{dI}{dt} + L\frac{d^2I}{dt^2} + \frac{I}{C}, \tag{16.20}$$

where we have taken an extra derivative to eliminate an integral. The solution to our problem follows by solving (16.20) when the voltage has the form $V(t) = V_0\cos\omega t$. An elegant way to do that is to recognize that

$$V_0\cos\omega t = \text{Re}\, V_0 e^{-i\omega t} = \text{Re}\ (V_0\cos\omega t - iV_0\sin\omega t)\,. \tag{16.21}$$

Because (16.20) is a linear equation (only first power of I occurs), the law of linear superposition holds. This means that if we imagine the circuit being driven by a complex voltage source, whose real part is the physical voltage, then the resulting current $I(t)$ will also be complex, with its real part the physical current. Thus we assume that the current has the form

$$I(t) = I_0 e^{-i\omega t}. \tag{16.22}$$

If we substitute this and the complex $V(t)$ into (16.20), we obtain

$$V_0 e^{-i\omega t} = ZI_0 e^{-i\omega t}. \tag{16.23}$$

Here Z is called the *impedance* and is the complex expression

$$Z = R + i\left(\frac{1}{\omega C} - \omega L\right). \tag{16.24}$$

Equation (16.23) is the generalization of Ohm's law, $V = IR$, to alternating current circuits.[4] We see that the real part of the impedance is just the resistance R, while the imaginary part (called the *reactance*) is $1/\omega C - \omega L$. As we shall see, the imaginary part of the impedance determines the phase of the current.

Solution for Complex Current

If we now solve (16.23) for the complex current, we obtain

$$I(t) = \frac{1}{Z}V_0 e^{-i\omega t} = \frac{V_0 e^{-i\omega t}}{R + i\left(1/\omega C - \omega L\right)}. \tag{16.25}$$

Eq. (16.25) is more illuminating if the complex impedance is expressed in polar form:

$$Z = |Z|e^{i\theta}, \tag{16.26}$$

$$|Z| = \sqrt{R^2 + (1/\omega C - \omega L)^2}, \quad \theta = \tan^{-1}\left(\frac{1/\omega C - \omega L}{R}\right). \tag{16.27}$$

We see now that the complex current is

$$I(t) = \frac{V_0}{|Z|}e^{-i(\omega t + \theta)}, \tag{16.28}$$

which means that the physical current is its real part:

$$I_{\text{physical}}(t) = \text{Re}I(t) = \frac{V_0}{|Z|}\cos(\omega t + \theta). \tag{16.29}$$

Equation (16.29) states that the amplitude of the current in the circuit is given by the amplitude of the voltage divided by the magnitude of the complex impedance, and that the phase of the current, relative to that of the voltage, is given by θ. If the phase $\theta > 0$, then the current *leads* the voltage in time, that is, the current reaches its maximum before the voltage reaches its maximum. If θ is negative, then the current *lags* the voltage.

Finally, what is to be done if we have two such RLC circuits in parallel, as shown on the right of Figure 16.1. The analysis is the same as that done with ordinary resistors. If two impedances are in series, then they have the same current passing through them. If two impedances are in parallel, then they have the same

[4] Some elementary texts may refer to $|Z|$ as the impedance and then use the concept of *phasors* to describe the effect of the impedance on the phase. We view the use of complex impedance as more direct and elegant.

Figure 16.3 An abstract drawing, or what?

voltage across them. If two impedances are connected in series, then the voltages add, and this leads to:

$$Z = Z_1 + Z_2 \qquad \text{(series connection)}. \tag{16.30}$$

If the impedances are connected in parallel, then the currents add, and this leads to

$$\frac{1}{Z} = \frac{1}{Z_1} + \frac{1}{Z_2} \qquad \text{(parallel connection)}. \tag{16.31}$$

Hence in the parallel case, the impedances add in inverse, with all the steps of the calculation being performed with complex arithmetic.

16.4 CS: ABSTRACT DATA TYPES, OBJECTS

What do you see when you look at the *abstract* object in Figure 16.3? Some readers may see a face in profile, others may see some parts of human anatomy, and others the total absence of artistic ability. This figure is abstract in the sense that it does not try to present a true or realistic picture of the object, but rather uses a symbol to suggest more than meets the eye.

Abstract or formal concepts pervade mathematics and science because they make it easier to describe nature. For example, we often use the symbol $\mathbf{v}(t)$ to denote the velocity of an object as a function of time. Despite velocity being a familiar concept, it is actually abstract in the sense that we cannot see it. What we see is the position of the object as a function of time, and then we infer from that the velocity by determining how rapidly that position is changing.

In computer science (CS), we create an abstract object by using a symbol to describe a collection of items. We have already seen that data, or variables in Java or Fortran may be integers, floating-point numbers, Booleans, or strings. These types of variables are built into the languages and therefore are called *primitive data types.* In addition, computer languages let the user define *abstract data types* of their own by combining primitive data types into more complicated structures called *objects*. These objects are abstract in the sense that they are named with a single symbol, yet they represent a number of parts.

For instance, one might create an object with parts that contain a student's school record (ID, grades, etc.), as well as other parts that contain methods to calculate the grade point average, etc. Of course, one would need many such objects because there are many students. To distinguish between the general structure of this student-record object and specific record objects for individual students, the general object is called a *class*, while the object for specific cases is called an *instance* of the class, or just an *object*.

In this chapter, our *objects* will be complex numbers, while in other chapters they may be plots, vectors, or matrices. The classes that we form will be combinations of abstract data types and associated methods for modifying those data. The entire class may also be thought of as objects. In a formal sense, computer science requires abstract data types to possess the three properties [Zach 96]:

Typename: procedure to construct the new data type from elementary pieces.

Set values: mechanism for assigning values to the defined data type.

Set operations: rules that permit operations on the new data type (you would not have gone to all the trouble of declaring a new data type unless you were interested in doing something with it).

In terms of these properties, when we declare a variable to be complex we satisfy property (1). When we declare $\mathrm{Re}z = x$ and $\mathrm{Im}z = y$ as *doubles*, we satisfy (2). When we define the rules of arithmetic and trigonometry for complex numbers (as reviewed in § 16.2), we satisfy (3).

Before we examine how these properties are applied in our programs, let us review the structure we have been using in our Java programs. When we start off our programs with a declaration statement such as `double x`. This tells the Java compiler the kind of variable `x` is, so that Java will store it properly in memory and use proper operations on it. The general rule is that *every variable we use in a program must have its data type declared.* For primitive (built-in) data types, we declare them to be `double, float, int, char, long, short,` or `boolean`.

If our program employs some user-defined, abstract data types, then they too must be declared. This declaration must occur even if we do not define the meaning of the data type until later in the program (the compiler checks on that). Consequently, when our program refers to a number z as complex, the compiler must be told at some point that there is *both* a real part x and an imaginary part y that makes up a complex number.

16.5 OBJECTS IN FORTRAN

One of the powerful and modern aspects of Fortran is its *object-oriented* approach to programming. Since the new concepts of object-oriented programming (OOP) can be a rather hard to grasp as purely theoretical concepts, we will gain some experience with objects and in the process learn some of the concepts of OOP. After all, a child learns to speak a language in a similar way!

For present purposes, it is easiest to think of an *object* in Fortran as a non-primitive data type with a number of specified components. Examples will include complex numbers and *arrays*. Indeed, when we set up a class file in Fortran containing our definition of a new, multi-component data type, and include various methods we have written to assign values to and operate on the new data type, we have created an object. Actually the class file containing a collection of data (variable) definitions and the associated methods to operate on the data that we have been calling a *class*, is a version of an *object*.

Data by Reference

Before we start working with objects in a program, it is necessary to understand that even though we may assign a variable name like `x` to an object, because objects have multiple components, you do not assign one explicit *value* to the object. Indeed, since you are the one who defines just how big and complicated your objects will be, it will also be your job to assign values to all of the different parts of the objects you create.

When Fortran deals with objects it does so by what computer scientists call by *reference*. More simply, when you refer toan object, Fortran understands that to mean that you are referring to the *location in memory* where your object is stored and not to the explicit values of the object's parts. If you remember that objects are referenced by their locations in memory rather than their actual values, the procedures we are about to describe will make more sense.

Sample Data Type

We suspect that all the words in the preceding section mean little without a real example to follow. So let's now examine an example of an object in Fortran, after which you should be able to go back and make more sense of the preceding words. The "object" that we will create with Fortran will be a complex number (about which, after sections § 16.2-16.2, you should be an expert). Despite the fact that Fortran has its own intrinsic complex data type, we create a custom data type for

its instructive attributes.

Below we present the Fortran class file `ComplexType.f90` that produces complex-number objects:

Listing 16.1 ComplexType.f90

```
! ComplexType.f90: Using complex procedures and type definitions
! ------------------------------------------------------------
Module complex_module
  !Define complex type
  Type public_complex
     Real :: re, im
  End Type public_complex
  !Add the functions for complex mathematics
  Contains
  ! ------- Add two complex numbers
  Function add(a,b)
    Implicit None
    Type (public_complex), intent(in) :: a,b
    Type (public_complex) :: add
    add%re = a%re + b%re
    add%im = a%im + b%im
  End Function add
  ! ------- Subtract b from a complex numbers
  Function subtract(a,b)
    Implicit None
    Type (public_complex), intent(in) :: a,b
    Type (public_complex) :: subtract
    subtract%re = a%re - b%re
    subtract%im = a%im - b%im
  End Function subtract
  ! ------- Multiply two complex numbers
  Function multiply(a,b)
    Implicit None
    Type (public_complex), intent(in) :: a,b
    Type (public_complex) :: multiply
    multiply%re = a%re * b%re - a%im * b%im
    multiply%im = a%re * b%im + a%im * b%re
  End Function multiply
  ! ------- Divide two complex numbers
  Function divide(a,b)
    Implicit none
    Type (public_complex), intent(in) :: a,b
    Type (public_complex) :: divide
    divide%re = (a%re * b%re + a%im * b%im)/(b%re**2 + b%im**2)
    divide%im = (a%im * b%re - a%re * b%im)/(b%re**2  + b%im**2)
  End Function divide
End Module complex_module
!
Program Main_program
  Use complex_module
  Implicit None
```

```
47    Type(public_complex) :: a,b,c
48    Character ch
49    a%re=1.
50    a%im=2.
51    b%re=3.
52    b%im=4.
53    if(a%im>=0.0) then
54       ch = '+'
55    else if(a%im<0.0) then
56       ch = '-'
57    end if
58    write(*,*) 'a = ', a%re, ch, abs(a%im), 'i'
59    if(b%im>=0.0) then
60       ch = '+'
61    else if(b%im<0.0) then
62       ch = '-'
63    end if
64    write(*,*) 'b = ', b%re, ch, abs(b%im), 'i'
65    c = add(a,b)
66    if(c%im>=0.0) then
67       ch = '+'
68    else if(c%im<0.0) then
69       ch = '-'
70    end if
71    write(*,*) 'a + b = ', c%re, ch, abs(c%im),'i'
72    c = subtract(a,b)
73    if(c%im>=0.0) then
74       ch = '+'
75    else if(c%im<0.0) then
76       ch = '-'
77    end if
78    write(*,*) 'a - b = ', c%re, ch, abs(c%im),'i'
79    c = multiply(a,b)
80    if(c%im>=0.0) then
81       ch = '+'
82    else if(c%im<0.0) then
83       ch = '-'
84    end if
85    write(*,*) 'a * b = ', c%re, ch, abs(c%im),'i'
86    if(b%re==0. .AND. b%im==0.) then
87       write(*,*) 'Error: division by zero ',b%re,'+',b%im,'i'
88    else
89       c = divide(a,b)
90       if(c%im>=0.0) then
91          ch = '+'
92       else if(c%im<0.0) then
93          ch = '-'
94       end if
95       write(*,*) 'a / b = ', c%re, ch, abs(c%im),'i'
96    end if
97 End Program main_program
```

Exercise:

1. The program `ComplexType.f90` provides all Abelian algebraic operations on complex numbers, with the complex numbers represented as objects. Enter the `ComplexType.f90` program file by hand, trying to understand it as best you can in the process.
2. Compile and execute this program, and check that the output agrees with the results you obtained in the exercises in §16.2.

The first thing to notice about `ComplexType.f90` is that the data type is declared with the statements

```
5.  Type public_complex
6.    Real ::  re, im
7.  End Type public_complex
```

The main program is declared on line 44 with

```
44.  Program Main_program
```

These are the same techniques we have employed before (it's good when some things stay the same in life). However, on line 6 we see that the variables `re` and `im` are declared for the public data type with the statement

```
6.  Real ::  re, im
```

These variables are part of a dynamicdata type, an object. These variables are dynamic in thesense that they will be different for each object (instance of the class) created. That is, if we define `z1` and `z2` to be `Complex` objects, then the variables `re` and `im` will be different for `z1` and `z2`.

We extract the component parts of our complex object by a projection operation. [For those readers who know some vector analysis, you can think of this as similar to extracting the components of a vector in space by taking dot products with the unit vectors to project the vector onto different axes.] The projection operation is denoted by a %:

`z1%re`	real part of object `z1`
`z1%im`	imaginary part of object `z1`
`z2%re`	real part of object `z2`
`z2%im`	imaginary part of object `z1`

Object Constructors

To construct an object from a module in Fortran, one must simply state the use of the module before any other declarations, such as in `ComplexType.f90`.

```
45.  Use complex_module
```

Some explanation is clearly in order! Modules are covered in more details in Chapter 18. Briefly now, a *module* acts like a nested class in Fortran. A module can be used by other programs once it has been compiled. All of the procedures in a module are static. A procedure or variable can be declared private, protected, or public. The declaration of user rights to the module's constituents is important only if you are developing programs with security in mind. Once a module is used inside of a procedure or program, all of the module's procedures and data types become available for use therein. For example, look at the program `ComplexType.f90` for a demonstration of multiple functions that may be used by other programs via the module `complex_module`.

Exercise Add two functions to `complex_module` that allow a user to create a default complex number and a custom complex number from two real initial values.

Declaring and Creating Objects

Now let's take stock of what we have up until this point. On line 6, our program has defined the variables `re` and `im` that each object of this program will have. For this reason these variables that project the component parts are often referred to as *instance variables*. We also have a complex number algebra module that can perform algebraic operations on two complex numbers simultaneously.

In the `main_program`, we can see how to create objects using the data type `public_complex`. On line 47, in the usual place for declaring variables, we have the statement

```
47.  Type(public_complex) ::  a, b, c
```

The compiler knows from the `type` command that `public_complex` is not one of its primitive (built-in) data types. In the present case this program contains the module named `complex_module` with the data type `public_complex`. Hence, the compiler does not have to look very far to know what you mean by a public_complex data type[5]. Accordingly, when the statement `Type(public_complex) :: a,b,c` declares the variables `a`, `b`, and `c` to be `public_complex` data objects, we know that they are manifestly objects since instanced data types are *not* static.

Recall that declaring a variable type, such as `real` or `integer`, does not assign a value to the variable, but, instead, tells the compiler to add the name of the variable to the list of variables it can be expected to encounter. Likewise, the declaration statement `type(public_complex) :: a,b,c` lets the compiler know what custom data type these variables are.

To actually *create the objects*, and not just register their names, we have to place numerical values in the memory locations that have been reserved for the objects. Since an object has multiple parts, we cannot give all its parts initial values with a simple assignment statement like `a = 0;`, so some other method is needed.A new instance of a complex data type is created as above, then values are assigned by accessing the internal instance variables of the complex data type variable.

```
49.  a%re = 1.          Sets the internal real part by reference
50.  a%im = 2.          Sets the internal imaginary part by reference
```

Instances

Since the name of the parent object and the names of objects that are created are the same, it sometimes is useful to use yet another word to distinguish one from the other. Accordingly, the phrase *instance of a class* is used to refer to the created objects (in our example, `a`, `b`, and `c`), that is, the created objects are each a single reference to the parent object. This designation distinguishes them from the definition of the abstract data type.

16.6 FORTRAN SOLUTION: COMPLEX CURRENTS

1. Extend the program `ComplexType.f90` by adding new functions to subtract, take the modulus, take the complex conjugate, and determine the phase of

[5]Other Fortran files could also use our complex methods, but we have enough to worry about right now.

complex numbers.

2. Test your methods by checking that the following identities hold for a variety of complex numbers:

$$\begin{array}{lllllll} z+z & = & 2z, & z+z* & = & 2\,\mathrm{Re}z & \\ z-z & = & 0, & z-z* & = & 2\,\mathrm{Im}z & \\ zz* & = & |z|^2, & zz* & = & r^2 & \text{(which is real)} \end{array} \tag{16.32}$$

   Hint: Compare your output to some cases of pure real, pure imaginary, and simple complex numbers that you are able to evaluate by hand.

3. Equations (16.29) and (16.27) are the solution for the current in a single RLC circuit. It tells us that the magnitude of the current is given by

$$|I| = \left|\frac{V_0}{Z}\right|, \tag{16.33}$$

   and that the phase of the current (relative to the $\cos\omega t$ time dependence) is

$$\theta_I = \tan^{-1}\left(\frac{1/\omega C - \omega L}{R}\right). \tag{16.34}$$

   Modify the given complex arithmetic program so that it performs the complex arithmetic required by (16.33). *Hint:* You do not have to solve this from scratch! Instead, use the techniques you have already programmed for determining the magnitude to determine $|I|$.

4. Compute and then make a plot of the magnitude and the phase of the current in the circuit as a function of frequency ω of the external voltage source. For our problem, a good range is $0 \leq \omega \leq 2$.

5. **Assessment:** You should notice a resonance peak in the magnitude at the same frequency for which the phase vanishes. The smaller the resistance R, the sharper should the circuit pass through resonance. These types of circuits were used in the early days of radio to tune to a specific frequency. The sharper the peak, the better the quality of reception.

6. The second part of the problem dealing with the two circuits in parallel is very similar to the first part. You need to change only the value of the impedance Z used. To do that, explicitly perform the complex arithmetic implied by (16.31), deduce a new value for the impedance, and then repeat the calculation of the current.

16.7 MAPLE SOLUTION: COMPLEX CURRENTS

Recall from way back in Chapter 3 that Maple knows all about complex numbers and complex arithmetic. Indeed, you have probably already seen an occasional I pop up as the solution to some equations. The point here is that Maple reserves the symbol I as the $\sqrt{-1}$, and this lets us use its I at our pleasure:

```
> restart:          `sqrt (-1)` = sqrt (-1);          # What's returned?
```

$$sqrt\ (\ -1\ ) = I$$

```
> expand((a + I*b)^2);                                    # See if Maple uses I as sqrt(-1)
```

$$a^2 + 2\,I\,a\,b - b^2$$

In §16.3 we applied circuit theory and complex analysis to the RLC circuit and found that if the exciting voltage is the real part of

$$V(t) = V_0\, e^{(-I\,\omega\,t)}, \tag{16.35}$$

then the current in the circuit is

$$I(t) = \frac{V_0 \cos(\omega\,t - \theta)}{|Z|}. \tag{16.36}$$

Here Z is the complex impedance,

```
> Z  := R + I * (1/(omega*C) - omega*L);
```

$$Z := R + (\frac{1}{\omega\,C} - \omega\,L)\,I.$$

We will start our Maple investigation by trying to determine magnitude $|Z|$ and phase θ of Z using Maple's capabilities for complex analysis. Whereas needing to know the magnitude and phase of Z is just another way of saying that we need to know Z in polar notation, we have Maple calculate these for us:

```
> Re(Z);                    Im(Z);                    # ReZ, ImZ
```

$$\mathrm{Re}(R + (\frac{1}{\omega\,C} - \omega\,L)\,I) \qquad \mathrm{Im}(R + (\frac{1}{\omega\,C} - \omega\,L)\,I)$$

This is just a fancy way of giving us back the input. The problem is that Maple does not know that ω, R, L, and C are real, and so it cannot do the complex arithmetic. To tell Maple that the constants are real, we tell it what to assume:

```
> assume (R, real); assume(L, real); assume(C, real); assume (omega, real);
> Re(Z);      Im(Z);                                      # Now check again
```

$$R^{\sim} \qquad \frac{1}{\omega^{\sim}\,C^{\sim}} - \omega^{\sim}\,L^{\sim}$$

It has taken some work, but now we are ready for some complex arithmetic. Look at the polar form:

```
> polar(Z);
```

$$\mathrm{polar}(\sqrt{R^{\sim 2} + (\frac{1}{\omega^{\sim}\,C^{\sim}} - \omega^{\sim}\,L^{\sim})^2},\quad \mathrm{argument}(R^{\sim} + (\frac{1}{\omega^{\sim}\,C^{\sim}} - \omega^{\sim}\,L^{\sim})\,I))$$

This is giving us the correct magnitude, but Maple appears unable to compute the phase. This seems to be a Maple failure. However, we get it to work by combining the `map` and `evalc` commands:

```
> Polar := map( evalc, polar(Z) );                    # Polar is variable name
```

$$Polar := \mathrm{polar}(\sqrt{R\tilde{}^2 + (\frac{1}{\omega\tilde{}\, C\tilde{}} - \omega\tilde{}\, L\tilde{})^2},\quad \arctan(\frac{1}{\omega\tilde{}\, C\tilde{}} - \omega\tilde{}\, L\tilde{},\ R\tilde{}))$$

Here `map` applies the procedure `evalc` to each element of `polar(Z)` and returns the polar form $(|Z|, \theta)$. This simplifies the elements into the form needed. [It should not be this hard!] We get separate expressions out for the real and imaginary parts by using the command `op(numb,expression)` to extract the `numb`-th operand from `expression`:

```
> op(1,Polar);     op(2,Polar);                       # Extract operands
```

$$\sqrt{R\tilde{}^2 + (\frac{1}{\omega\tilde{}\, C\tilde{}} - \omega\tilde{}\, L\tilde{})^2} \qquad \arctan(\frac{1}{\omega\tilde{}\, C\tilde{}} - \omega\tilde{}\, L\tilde{},\ R\tilde{})$$

The expressions we have just derived are the standard ones for the magnitude and the phase of the impedance Z. However, since the current is proportional to $1/|Z|$, we will plot $1/|Z|$. We could compute $1/Z$ and work with that as well.

Maple's Surface Plots of Complex Impedance

We want to examine the current that occurs in the RLC circuit as a function of the driving frequency. We have already discussed in Chapter 4 some of Maple's commands for plotting complex functions. Irrespective of them being illuminating, for the problem given here, we first use `plot3d` to make the familiar $z(x, y)$ surface plot of the magnitude and phase of the current as functions of *both* the frequency of the external voltage ω and of the resistance R:

```
> restart:          Zinv := 1/(R + I * (1/(omega*C) - omega*L));  # Compute 1/Z
> L := 1000;        C :=1/1000;                                   # Assign numerical values
```

$$Zinv := \frac{1}{R + (\frac{1}{\omega C} - \omega L)\, I} \qquad L := 1000 \qquad C := \frac{1}{1000}$$

```
> assume (R, real);        assume (omega, real);      # Tell Maple the constants are real
> Polar := map( evalc, polar(Zinv) );                 # Convert 1/Z to polar form
> polar(1/((R^2+(1000/omega-1000*omega)^2)^(1/2)),
> arctan(-1000/omega + 1000*omega, R));
```

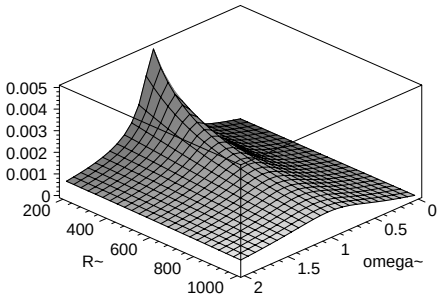
$$Polar := \mathrm{polar}\left(\frac{1}{\sqrt{R\tilde{}^2 + (\frac{1000}{\omega\tilde{}} - 1000\,\omega\tilde{})^2}},\ \arctan(-\frac{1000}{\omega\tilde{}} + 1000\,\omega\tilde{},\ R\tilde{})\right)$$

```
> mag := op(1, Polar);       op(2, Polar);            # Extract magnitude, phase
```

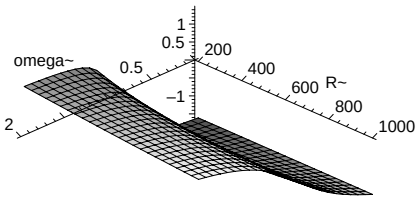
$$mag := \frac{1}{\sqrt{R\tilde{}^2 + (\frac{1000}{\omega\tilde{}} - 1000\,\omega\tilde{})^2}} \qquad \arctan(-\frac{1000}{\omega\tilde{}} + 1000\,\omega\tilde{},\ R\tilde{})$$

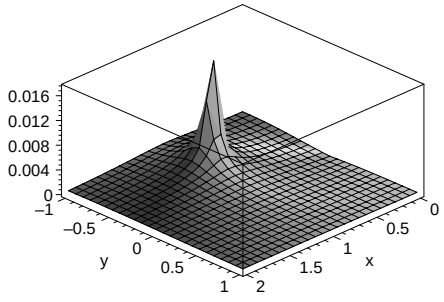
Check out how the magnitude has a maximum when the external frequency $\omega = 1/\sqrt{LC}$. This is the *resonance* frequency. For our choice of constants this corresponds to $\omega = 1$. We now make some plots to see if this is true.

```
> op(1, Polar);       with(plots):
> plot3d(op(1, Polar), omega = 0..2, R = 200..1000, axes = BOXED); # 1/Z vs ω
```




```
> plot3d(op(2,Polar), omega=0..2, R=200..1000, axes = NORMAL);        # θ vs ω
```





Sure enough, the plot of $1/|Z|$ shows that the magnitude of the current has a maximum at $\omega = 1$. (It helps to grab and rotate these plots to see them better.) We also see that as the resistance R in the circuit is made smaller, the maximum current becomes progressively larger. The second plot of the phase shows that below resonance, $\omega < 1$, the current lags the voltage, while above resonance the current leads the voltage. Another way to visualize complex functions is with the command `complexplot3d`. It makes a 3-D visualization of a complex function of a complex argument. Here we do it by treating the frequency $\omega = x + iy$ as a complex number:

```
> with(plots):          R := 1000;
> complexplot3d( 1/(R+I*(1/(w*C)-w*L) ), w = 0-I..2+I, axes = BOXED);    # 1/Z
```

We see in the right plot above that there is a sharp peak at $x = \mathrm{Re}(\omega) = 1$,

as expected. The color change indicates where $\mathrm{Im}(1/Z)$ changes sign. If we look closely at this graph we will also see that there is a maximum for a negative imaginary value of ω. This is related to how long the resonance stays excited, a statement we do not try to prove here.

16.8 KEY WORDS

abstract data types	reactance	frequency	imaginary number
complex conjugate	constructor	current	default constructor
complex arithmetic	class	impedance	instance of object
instance variable	magnitude	modulus	nonstatic methods
nonstatic variable	object	voltage	polar representation
complex number	reference call	resonance	user-defined data types
object creation			

16.9 FORTRAN AND MAPLE EXERCISES

1. Since Fortran actually has a primitive data type `complex*4 or complex*8`, rewrite `ComplexType.f90` in Lst. 16.1 as a more simplified Fortran program using the correct primitive data types. (In other words, complex *objects* do not have to be built). Call your modified code `Complex.f90`; compare results with `ComplexType.f90`.
2. Complex mathematics is much easier if we have the pure imaginary number $i = \sqrt{-1}$. Define a new complex variable `i` with a value equal to i. (In Maple, this is just the built-in variable I.)
3. Now that you have a specific variable for i, write some new methods that compute the following functions of complex numbers (objects):
   a. $\exp(z) = e^{x+iy} = e^x e^{iy} = e^x(\cos y + i \sin y)$
   b. $\sin(z) = (e^{iz} - e^{-iz})/2i$
4. Verify that your methods work by trying some simple cases. (In Maple, compare to the built-in functions that also handle complex numbers.) Examples of simple cases might be the use of complex numbers like $z_1, z_2 = 0, 1, i, 2, 2i, \ldots$, where you can easily figure out the answers.
5. Let $z = 3 + 3\sqrt(3)i$.
   a. Determine $|z|$ and check that you get 6.
   b. Determine the phase θ of z and check that you get $\theta = \pi/3$.
6. Find $\mathrm{Re}(2-3i)^2/(2+3i)$, $\mathrm{Im}(1/z^2)$, $|(1+z)/(1-z)|$, and $(2-3i)/(2+3i)$.
7. Consider the complex number $z = x + iy$. Use gnuplot to make a surface plot of $\mathrm{Im}\cos(z)$ and $\mathrm{Re}\cos(z)$.
8. Consider the complex function $f(z) = z^3/(1 + z^4)$.
   a. Make a plot of $f(x)$ *versus* x, that is, assume z is real and vary it along the real axis.
   b. Make surface plots of $\mathrm{Re}z$ and $\mathrm{Im}z$. Restrict the range of x and y values to lie close to where the "action" is.

Chapter Seventeen

Arrays I: Vectors, Matrices; Rigid-Body Rotations

Vectors and matrices play key roles in scientific computing. Indeed, much of so-called *high-performance computing* involves computations with very large arrays. These arrays may contain experimental data needing processing, or they may contain the results of a simulation in which the discrete nature of breaking space up into small units leads naturally to the use of arrays. And since real-world problems are often complicated and require high precision (small units), arrays with thousands or millions of elements are not unusual. Even though this means that the matrices used in realistic problems will be big and will require efficient algorithms, often the programs using matrices are simple and direct.

In this chapter, we deal with techniques needed to create and manipulate vectors and matrices in Fortran. We will solve the same problem, *Rotations of Rigid Bodies*, as we did in Chapter 7, *Matrices and Vectors; Rotation* with Maple. If needed, you should review that chapter for some of the physics and mathematics background (even elementary stuff like the definition of a vector), and then return here for the Fortran approach.

It is interesting to observe how the Fortran and Maple approaches to this problem tend to complement each other. Maple is able to do the calculations symbolically, whereas Fortran is not. However, Maple is slow in performing numeric calculations involving large matrices, while compiled languages tend to be much faster. In addition, Maple excels in its capacity for interactive 3-D graphics, while the interactivity is harder to do in Fortran or even Mathematica. We have seen how Dislin and Gnuplot can produce surface plots, but only the latest version of Gnuplot can interact with the surfaces.

17.1 PROBLEM: RIGID-BODY ROTATIONS

Your problem is similar to the one in Chapter 7. Here we will deal with a plate and a square whose moments of inertia are given, and we will focus on the matrix multiplication that converts the angular velocity $\boldsymbol{\omega}$ into the angular momentum $\mathbf{L}$.

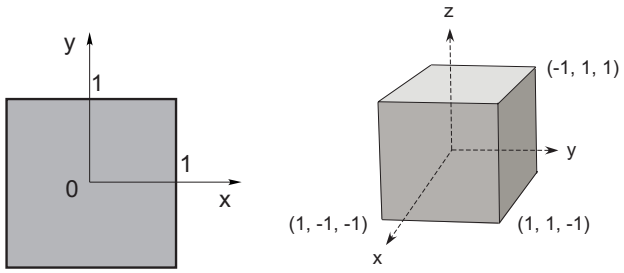


Figure 17.1 *Left:* A plate sitting in the $x - y$ plane with a coordinate system at its center. *Right:* A cube sitting in the center of a three-dimensional coordinate system.

Problem 1: Consider the two-dimensional rectangular metal plate shown on the left of Figure 17.1. It has sides $a = 2$ and $b = 1$, mass $m = 12$, and inertia tensor for axes through the center:

$$\{I\} = \begin{pmatrix} mb^2/12 & 0 \\ 0 & ma^2/12 \end{pmatrix} = \begin{pmatrix} 1 & 0 \\ 0 & 4 \end{pmatrix}. \tag{17.1}$$

The plate is rotated so that its angular velocity vector $\boldsymbol{\omega}$ always remains in the $x-y$ plane (which does not mean the plate remains in the $x - y$ plane). Specifically, it is rotated with the three angular velocities:

$$\boldsymbol{\omega} = (1, 0), \qquad \boldsymbol{\omega} = (0, 1), \qquad \boldsymbol{\omega} = (1, 1). \tag{17.2}$$

Write a Fortran program that computes the angular momentum vector $\mathbf{L}$ via the matrix multiplication $\mathbf{L} = \{I\}\boldsymbol{\omega}$. Plot $\boldsymbol{\omega}$ and $\mathbf{L}$ for each case, and compare the results to those found with Maple.

Problem 2: Consider now the rotation of the cube on the right of Figure 17.1. The cube has side $b = 1$, mass $m = 1$, and, for axes on the corner, an inertia tensor [M&T 88]:

$$\{I\} = mb^2 \begin{pmatrix} +2/3 & -1/4 & -1/4 \\ -1/4 & +2/3 & -1/4 \\ -1/4 & -1/4 & +2/3 \end{pmatrix} = \begin{pmatrix} +2/3 & -1/4 & -1/4 \\ -1/4 & +2/3 & -1/4 \\ -1/4 & -1/4 & +2/3 \end{pmatrix}. \tag{17.3}$$

The cube is rotated along axes passing through two sides and the diagonal, explicitly, with the three angular velocities:

$$\boldsymbol{\omega} = (1, 0, 0), \qquad \boldsymbol{\omega} = (0, 1, 0), \qquad \boldsymbol{\omega} = (1, 1, 1). \tag{17.4}$$

Write a Fortran program that computes the angular momentum vector $\mathbf{L}$ via the requisite matrix multiplication. Plot $\boldsymbol{\omega}$ and $\mathbf{L}$ for each case, and compare the results to those found with Maple.

17.2 THEORY: ANGULAR-MOMENTUM DYNAMICS

Recall from Chapter 7 that the angular-momentum vector $\mathbf{L}$ is given by the product

$$\mathbf{L} = \{I\}\,\boldsymbol{\omega}. \tag{17.5}$$

Here $\boldsymbol{\omega}$ is the angular-velocity vector and $\{I\}$ is the inertia tensor represented by the matrix $[I_{ij}]$:

$$\{I\} = [I_{ij}] = \begin{pmatrix} I_{xx} & I_{xy} & I_{xz} \\ I_{yx} & I_{yy} & I_{yz} \\ I_{zx} & I_{zy} & I_{zz} \end{pmatrix}. \tag{17.6}$$

Despite the use of x, y, and z to label the three axes being standard in elementary science, the mathematics and the computing gets easier if we label each direction with a number rather than a letter. Consequently, we now change to a notation in which A_0 indicates the component of the vector $\mathbf{A}$ in the x or "0" direction, A_1 indicates the component in the y or "1" direction, and A_2 indicates the component in the z or "2" direction:

$$\begin{pmatrix} A_x \\ A_y \\ A_z \end{pmatrix} \Leftrightarrow \begin{pmatrix} A_1 \\ A_2 \\ A_3 \end{pmatrix}, \quad \begin{pmatrix} I_{xx} & I_{xy} & I_{xz} \\ I_{yx} & I_{yy} & I_{yz} \\ I_{zx} & I_{zy} & I_{zz} \end{pmatrix} \Leftrightarrow \begin{pmatrix} I_{11} & I_{12} & I_{13} \\ I_{21} & I_{22} & I_{23} \\ I_{31} & I_{32} & I_{33} \end{pmatrix}. \tag{17.7}$$

Here we use $1, 2, 3$ for the indices, as is standard math, yet Java and C would use $0, 1, 2$.

With this new notation, the relation stating that the angular momentum equals the product of the moment of inertia tensor times the angular velocity vector is represented by the *matrix multiplication*

$$\begin{pmatrix} L_1 \\ L_2 \\ L_3 \end{pmatrix} = \begin{pmatrix} I_{11} & I_{12} & I_{13} \\ I_{21} & I_{22} & I_{23} \\ I_{31} & I_{32} & I_{33} \end{pmatrix} \begin{pmatrix} \omega_1 \\ \omega_2 \\ \omega_3 \end{pmatrix}. \tag{17.8}$$

In terms of the individual components, (17.8) is an elegant way of representing the three simultaneous equations,

$$L_1 = I_{11}\omega_1 + I_{12}\omega_2 + I_{13}\omega_3, \tag{17.9}$$
$$L_2 = I_{21}\omega_1 + I_{22}\omega_2 + I_{23}\omega_3, \tag{17.10}$$
$$L_3 = I_{31}\omega_1 + I_{32}\omega_2 + I_{33}\omega_3. \tag{17.11}$$

These three equations may also be written in a convenient form for computations:

$$L_k = \sum_{j=1}^{3} I_{kj}\omega_j, \quad k = 1, 2, 3. \tag{17.12}$$

17.3 COMPUTER SCIENCE AND MATH: ARRAYS VS. VECTORS AND MATRICES

We saw in Chapter 16 dealing with complex-number objects, how the mathematics and the programming becomes simpler if we deal with a variable's single name, even though it has multiple parts. In computer science, we store an entire table of numbers in a single variable when that variable is declared as an ***array***. You then deal with the entire array as one symbol, or deal with individual elements in the array via a subscript or index notation. As a case in point, we store the final exam scores of all students in the array `Final(i)`, with the different values of the index `i` referring to different students. In this case, `Final(10)= 99` would be the score of student number `10`, and `Final(100) = 13` would be the (sorrowful) score of student number `100`.

There is no requirement that arrays have only one index. To prove the point, we could use the 2-D array `HW(i,j)` to store all homework scores for all students, with `i` representing the assignment number and `j` representing the student number. It follows then that `HW(2,3)=99` would be second homework score for student number `3`, while `HW(10,500)=12` would be the tenth homework score for student `500`. As you see here, the two indices (subscripts) needed to reference the elements in a 2-D array are separated by a comma, and enclosed in a single set of parenthesis. Likewise, a 3-D array would have three indices separated by commas. The number of dimension, and the length in each dimension is up to the programmer.

You may think of a computer program's array as a concrete representation of the mathematical abstraction of matrices and vectors. Arrays are collections of values of a certain data type, data types that themselves may be abstract or primitive. The values contained in an array are called *elements*, and the shape of an array (200×300) is referred to as its *dimensions*.

In mathematics and science we deal with objects called *vectors* and *matrices*. Taking into account that only one index is needed to distinguish the components of a vector, they are stored on a computer in a 1-D *array*. Because matrices have rows and columns, their elements are stored in arrays with two indices. As an example, the matrix I_{xy}, representing the inertia tensor, could have its components stored in the array `I(x,y)`. To repeat, the mathematical objects are vectors and matrices, the computer storage is in single-indexed and double-indexed *arrays*.

As is often the case when science and computer science are combined, confusion results from using different words to describe the same idea. Even worse, we often use the same words to describe different ideas. Of necessity we now must point out that our use of the words "two dimensional" and "three dimensional" is from physics where the number of dimensions of space equals the number of components (indices) that a vector may have, and where the number of values that

any one index spans is also equal to the number of dimensions. Yet on a computer, a "three-dimensional array" refers to a data type with three indices, such as `A(i,j,k)`, and there is no limit on the range spanned by any one of the indices `i`, `j`, and `k`. In the latter case, the maximum value of some index is referred to as the "size" or "length" of the array, not its "dimension". To name an example, the two-dimensional physics vectors used to describe motion in a plane, are stored in one-dimensional arrays of length 2.

In most programming languages arrays are stored in memory as a continuous 1-D string of numbers. For example, an $n \times m$ array would be stored as one long string, with some algorithm used to determine how the location in the string relates to the row and column indices. In Fortran, the mapping of an array to its 1-D form is done in *column major order*, that is, first all the elements from column 1 of the array are stored, then all the elements from column 2 are stored, and so forth, until all columns are stored.

17.4 ARRAY DECLARATION AND INSTANTIATION

The elements in an array are accessed by stating the array's name followed by a comma-separated list of the indices enclosed in parenthesis, for example, `A(1)` or `B(i,j)`. As is true for other primitive data types, arrays too must be declared and assigned values (*instantiation*) before their use. First, you state an array's data type followed by two colons, and then the array's name. For example, these two definitions produce the same fixed size array:

```
1  Real*8, dimension(8) :: A
2  Real*8 :: A(8)
```

This same scheme is one of the ways used to assign values to array elements:

```
1  Real*8 :: A(8)
2  Integer :: i
3  A(2) = 1.
4    do i=1, 8, 1
5      A(i)=i
6    end do
```

When using a 2-D array to store a matrix, the first index refers to the column of the matrix and the second to the row. Take the rotational inertia matrix as an example:

```
1  Real*8 :: I(3,3)      ! Declare array type
2  Integer :: i,j
3  do i=1,3,1
4    do j=1,3,1
5      I(i,j) = i-j      ! Initialize matrix
6    end do
7  end do
```

An all-too-common error when dealing with arrays is going out of bounds, that is, telling the computer to go to a location in memory which does not contain your array. This is a forbidden because you can do a lot of harm there, even if it is not your intention. Going out of bounds usually occurs when an array index becomes greater than the dimension of an array, or negative.

A powerful array operation in Fortran is the ability to access entire rows, columns, or even the whole array at once, and then to perform subsequent operations on them. You may even access sections of an array. This not only saves you time, but also leads to a program that runs faster. As an example, here we multiply two $n \times n$ matrices A and B to form C

$$C_j = \sum_{i=1}^{n} B_{i,j} A_j$$

```
Integer :: i,j
Integer, parameter :: n=10
Real*8, dimension(n,n) :: A, B, C
A = 1.
B = 0.1_8
C = 0.
do i=1,n,1
  do j=1,n,1
    C(1:n,i) = C(1:n,i) + B(i,j)*A(1:n,j)
  end do
end do
```

The output should be C = 1. Here `C(1:n,i)` accesses the entire i^{th} column (elements between `1` and `n`) in one sweep. To access only a range of elements in the column or row, you only need to describe the range by `i:j`. For instance, `C(i,3:5)` or `D(2:10, 4:15, j)`.

Exercise: The program `OneDArray.f90` sets up two vectors $\mathbf{A}$ and $\mathbf{B}$, each of length 3, and then forms the cross product $\mathbf{C} = \mathbf{A} \times \mathbf{B}$:

Listing 17.1 OneDArray.f90

```
! OneDArray.f90: Arrays as vectors for cross product
! ---------------------------------------
Program main_program
  Real :: a(3), b(3), c(3)
  Integer :: i
  a(1:3) = 1./sqrt(3.)
  b(1) = 2./sqrt(6.)
  b(2:3) = -1./sqrt(6.)
  c(1) = a(1)*b(3)-b(2)*a(3)
  c(2) = a(3)*b(1)-a(2)*b(3)
  c(3) = a(1)*b(2)-b(1)*a(2)
```

```
12   write(*,*) (c(i),i=1,3)
13 End Program main_program
```

1. Compile and execute this program.
2. Convert the three 1D vectors `A, B` and `C` to 2D vectors. You can do this with statements of the form

   ```
   real ::  A(1,3)
   do i=1,3,1 A(1,i) = Math.sqrt(i)
   ```

3. Compile and execute the 2D version of the program and see how much longer it now takes. (The extra time arises from having to move through larger blocks of memory. We found a 0.3% effect, not big, but real.)
4. Modify `OneDArray.f90` to have it print values for `a(0)` and `a(3)`. Fortran should not let you access `a(0)` because it is out of bounds. ♠

Array Sizes

The number of elements contained within an array is referred to as the *length* or *size* of the array. The length of the array can be determined by the `size` command:

```
real ::  A(105)                                  Declaration and creation of A
write(*,*) "size(A,DIM=1) =", size(A,DIM=1)      ⇒ size(A,DIM=1) = 105
```

The `DIM` keyword here designates which index of the array we want the size of. If you want to know the length of an array with several indices, use the `size` command, with specification as to which dimension (`DIM`) interests you:

```
real ::  B(20,10)                                20 columns, each with 10 rows
write(*,*) "size(A,DIM=1) =", size(A,DIM=1)      ⇒ size(A,DIM=1) = 20
write(*,*) "size(A,DIM=2) =", size(A,DIM=2)      ⇒ size(A,DIM=2) = 10
```

Fixed-Sized Arrays

A *fixed-size array* is one in which the dimensions are defined during its declaration. We have been using them in our examples so far. To declare a fixed-sized array, you may use either the `dimension` modifier (especially useful when defining multiple arrays of the same dimension), or you may declare each array individually:

```
1   Real*8, dimension(8,8) :: D, E, F
2   Integer :: A(7,8), B(9,10), C(11,12)
```

Assumed-Shape Arrays*

An *assumed-shape array* is one whose dimensions are not specified when the array is declared, but rather, is determined by the size of a fixed-size array that is passed to it. This is called *dynamic memory allocation* and is useful for creating procedures that have arrays passed as arguments. Assumed-shape arrays are declared by stating their type, then optional modifiers, then the usual two colons, and finally the array name. The difference between this declaration and the usual one is that the dimensions are declared with colons as *place holders* rather than explicit numbers. For example, here we multiply two arbitrary `Real*8` arrays, which works as long as the number of columns on the left array equals the number of rows of the right:

```
  Subroutine multiply (A,B,C)
  Implicit none
  Real*8, dimension(:,:), intent(in) :: A, B
  Real*8, dimension(size(A,DIM=1),size(B,DIM=2)) :: C
  Integer :: i,j,k,h,m,n
  k = size(A,DIM=1)
  h = size(A,DIM=2)
  m = size(B,DIM=1)
  n = size(B,DIM=2)
  if (h/=m) then
    write(*,*) 'Error: Dimensions do not match.'
    stop
   end if
   do i=1,n,1
      do j=1,h,1
         C(1:k,i) = C(1:k,i) + B(j,i)*A(1:k,j)
      end do
   end do
  End Subroutine multiply
```

Allocatable Arrays*

Listing 17.2 Timer.f90

```
! Timer.f90: Times the matrix multiply for expanding arrays
! ------------------------------------------------------------
Module linalg
  Contains
  Subroutine multiply (A,B,C)
    Implicit None
    Real*8, dimension(:,:), intent(in) :: A, B
    Real*8, dimension(size(A,DIM=1),size(B,DIM=2)) :: C
    Integer :: i,j,k,h,m,n
    k=size(A,DIM=1)
```

```
11      h=size(A,DIM=2)
12      m=size(B,DIM=1)
13      n=size(B,DIM=2)
14      if(h/=m) then
15        write(*,*) 'Error: Dimensions do not match.'
16        stop    ! Aborts program in the case of this error
17      end if
18      do i=1,n,1
19        do j=1,h,1
20           C(1:k,i) = C(1:k,i) + B(j,i)*A(1:k,j)
21        end do
22      end do
23    End Subroutine multiply
24  End Module linalg
25  !
26  Program timer
27    Use linalg
28    Implicit None
29    Real*8, allocatable, dimension(:,:) :: A, B
30    Real*8 :: t
31    Integer :: Ti(8), Tf(8), i, inc=20
32    Integer, parameter :: n=500
33    Real*8 :: C(n,n)
34    Character (len = 12) :: clock(3)
35    do i=2,n-1,inc
36      allocate(A(i-1,i),B(i,i+1))
37        A = 1._8
38        B = 0.2_8
39        call date_and_time(clock(1),clock(2),clock(3),Ti)
40        call multiply(A, B, C)
41        call date_and_time(clock(1),clock(2),clock(3),Tf)
42      deallocate(A,B)
43      t = 60.0*(Tf(6)-Ti(6))+(Tf(7)-Ti(7))+(Tf(8)-Ti(8))/1000.0
44      write(*,FMT=100) i,t
45    end do
46    100 Format(I5, ' ', F12.6)
47  End Program timer
```

An *allocatable array* is a dynamic array that can be resized as needed. To create and destroy a dynamic array, you issue subroutine calls to `allocate` and `deallocate` memory for the array. For example, in Lst. 17.2, we find the compute times for matrix multiplication for various array sizes. As the allocatable size of the arrays gets larger, the matrix multiplication takes more time. Give it a try and watch it slow down as the matrices grow in size.

Notice in the `multiply` procedure above how the intrinsic function `size(array,DIM)` is used. This is a good way to handle cases where the dimensions are determined by the size/shape of the arrays passed as input arguments. The `size` function determines the needed parameters, which can then be assigned

to integer variables for use in the procedure.

17.5 ARRAYS AS ARGUMENTS TO PROCEDURES*

Listing 17.3 ChangeArray.f90

```
! ChangeArray.f90:  Change array with function
! ------------------------------------------
Program ChangeArray
  Implicit None
  Interface
     Function f(x)
       Real, dimension(:), intent(in) :: x
       Real :: f(size(x))
     End Function
  End Interface
  Real :: x(3)
  Integer :: i
  x(1)=1.0
  x(2)=2.0
  x(3)=3.0
  write (*,*) 'Input:'
  write(*,*) (x(i),i=1,3)
  x = f(x)
  write (*,*) 'Output:'
  write(*,*) (x(i),i=1,3)
End Program ChangeArray

Function f(x)
  Implicit None
  Real, dimension(:), intent(in) :: x
  Real :: f(size(x))
  f = x/3.0
End Function f
```

We have already indicated that in Fortran, functions and subroutines (*procedures*) are called *by reference*, that is, with reference to memory locations rather than by having actual values sent to them. This is in contrast to Java and C, where procedures are called *by value*. This idea gets to be more interesting when an array is used as an argument to a procedure, since an array is an object with multiple parts. Indeed, one of the reasons Fortran is so fast when dealing with arrays is that only the single address of the first element gets passed to the procedure, and not, possibly, *massive* amounts of data.

While a procedure cannot change the memory address passed when there is an array argument, it is free to the change the actual values of the array elements stored in memory. Run `ChangeArray.f90` to see how this works. Note that all three values of `x` are changed by the function `f(x)`, and that the changed values of `x` are seen by the `main` program. Also note that the `write` statements here

demonstrate the use of *implied do-loops* contained within the write statements themselves, thus, permitting several values of an array to be written on a single line.

Exercise: What we have said about functions dealing with single-indexed arrays hold equally true for multi-indexed arrays. Modify a copy of `ChangeArray.f90` so that `x(3)` is now the two-indexed array `x(3, 2)`, and see if you can make changes to `x(i,1)` for `i=1 ...3`. ♠

Computation: Changing Array Arguments in Functions

Usually we write a function to return a value for the function name, but not necessarily to change the value of any of the function's arguments. As discussed above, when arrays are used in the argument list to a function, the situation may have unexpected surprises (since a change in an *argument* array within a function will impact that array's values in the calling routine as well). This is also true for subroutines, but the task at hand with array arguments in these types of procedures tend to be more blatant and less likely to surprise us. Here's a more extensive example of using arrays within the argument list for a function:

Listing 17.4 ArrayMod.f90

```
1  ! ArrayMod.f90: Array Modification
2  ! ----------------------
3  Program array_mod
4    Implicit None
5    Interface
6      Integer Function modify(x,y,z)
7      Real, intent(inout) :: x, y(:), z(:,:)
8      End Function modify
9    End Interface
10   Integer :: i, j                !Instance arrays
11   Real :: a, b(3), c(3,3)
12   a = 1.6180339887               !Initialize arrays
13   do i=1,3,1
14     b(i) = a*i*i
15       do j=1,3,1
16         c(i,j) = a*sin(1.0*i)*cos(1.0*j)
17       end do
18   end do
19   write(*,*) 'Scalar-before a = ', a
20   write(*,*) 'Vector-before b = [', (b(i),i=1,3,1), ']'
21   do i=1,3,1
22      if(i==2) then
23        write(*,*) 'Before c = [', (c(i,j),j=1,3,1), ']'
24      else
25        write(*,*) '           [', (c(i,j),j=1,3,1), ']'
26      end if
```

```
27    end do
28    call modify(a,b,c)
29    write(*,*) 'Scalar-after a = ', a
30    write(*,*) 'Vector-after b = [', (b(i),i=1,3,1), ']'
31    do i=1,3,1
32       if(i==2) then
33         write(*,*) 'After c = [', (c(i,j),j=1,3,1), ']'
34       else
35         write(*,*) '          [', (c(i,j),j=1,3,1), ']'
36       end if
37    end do
38  End Program array_mod
39  !
40  Integer Function modify(x,y,z)
41    Implicit None
42    Integer :: i, j
43    Real, intent(inout) :: x, y(:), z(:,:)
44    x = 3.14159265358979
45    do i=1,3,1
46       y(i) = x*(i**3)
47       do j=1,3,1
48          z(i,j) = x*sin(2.0*i)*cos(3.0*j)
49       end do
50    end do
51    modify = 1
52  End Function modify
```

1. Compile and execute `ArrayMod.f90`.
2. Add numbers to the output lines corresponding to the line number in the code that produced the output.
3. Modify the code so that the procedure reverses the original order of the elements in the vector and array, and test it.
4. Modify the code so that the matrix `c` in the main program is now a 4×4 array, and test it.

An unusual consequence of arrays being accessed by reference to their locations in memory, rather than the values of their individual elements, arises when two arrays are set equal to each other without explicit reference to individual elements. So here we give you a riddle. The code `ArraysEqual.f90` is simple and declares the two vectors `a` and `b`, and then in line 13 sets part of vector `b` equal to part of `a`. The riddle is to first guess what will be the output, and then to see if your guess agrees with the results of running the code.

Listing 17.5 ArraysEqual.f90

```
1  ! ArraysEqual.f90: Shows methods modifying arrays
2  ! ------------------------------
3  Program arrays_equal
```

```
4   Implicit None
5   Integer :: i          !Instance
6   Real :: a(3), b(3)
7   do i=1,3,1            !Initialize
8      a(i) = 1.0*i*i
9      b(i) = sqrt(1.0*i)
10  end do
11  write(*,*) 'vector a before = [', (a(i),i=1,3,1), ']'
12  write(*,*) 'vector b before = [', (b(i),i=1,3,1), ']'
13  b(1:2) = a(2:3)
14  a(1:3) = 1.0
15  write(*,*) 'vector a after = [', (a(i),i=1,3,1), ']'
16  write(*,*) 'vector b after = [', (b(i),i=1,3,1), ']'
17 End Program arrays_equal
```

17.6 APPLICATION TO ROTATIONS

Let's apply these rules and conventions to our rotation problem. We set up our moment of inertia tensor M_{jk} on the computer by declaring that the variable `M` is to be an array with two indices, and that the angular momentum and angular velocity are arrays with one index:

```
real ::  M(3,3);          Declare M as 2-D array
real ::  L(3), w(3);      Declare L and w as 1-D arrays
```

So far we have told the compiler that `M`, `L` and `w` are to be arrays of real numbers, and have reserved the right number of locations in memory for them. But we have not actually placed any values for these variables in memory. The most direct way to assign values to do it for each individual component. For example, if our angular velocity vector ω were pointing along the x axis with magnitude 10, we would assign it the three components:

```
w(1) = 10.        The x (1) component
w(2) = 0.         Explicit y (2) component
w(3) = 0.         Explicit z (3) components
w(2:3) = 0.       Implied y (2) and z (3) components
```

Declaring values for the moment of inertia tensor `M` with its two indices is more interesting. We use two loops to assign values to each component:

```
k=0
do i=1,3,1
   do j=1,3,1
      k=k+1
      M(i,j) = k
   end do
end do
```

Indeed, you can even build up an array with each row having a different length, or with the elements of an array being sub-arrays. In this way, arrays can be used to build what is called *structured data sets*.

17.7 KEY WORDS

angular momentum	angular velocity	arrays	arrays as arguments
declaration	creation	dimension	eigenvalue problem*
inertia tensor	initializer	linear equations*	matrix library
matrix multiplication	rigid bodies	rotation	

Explain in just a few words what is meant by

1. a subscript and an array index being different
2. an abstract data type
3. a complex number
4. an array
5. the dimension of an array
6. the length of an array
7. a matrix
8. a local variable
9. a vector

17.8 SUPPLEMENTARY EXERCISES

1. Do problems 1 and 2 from the beginning of this chapter!
2. Consider the two-dimensional rectangular metal plate shown in Fig. 17.1. For this symmetric orientation, and for rotations in the $x-y$ plane, the moment of inertia tensor is

$$I = \begin{pmatrix} 1 & 0 \\ 0 & 4 \end{pmatrix}. \tag{17.13}$$

   Write a Fortran program that does matrix multiplication to compute the angular momentum vector according to (17.5) and (17.12). Write a procedure to do the matrix multiplication and use the length field of the array so that the procedure does not have to be rewritten for the following exercise. For each calculation, take your answer and draw the angular momentum $\mathbf{L}$ and the angular velocity vector ω on a piece of graph paper, taking note if the two are parallel. Consider the cases:
   a. $\omega = (1, 0)$
   b. $\omega = (0, 1)$
   c. $\omega = (1, 1)$
3. Consider now a three-dimensional cube of mass $m = 1$ and side $l = 1$, like Fig. 17.1. Its moment of inertia tensor is [M&T 88]

$$I = \begin{pmatrix} +2/3 & -1/4 & -1/4 \\ -1/4 & +2/3 & -1/4 \\ -1/4 & -1/4 & +2/3 \end{pmatrix}. \tag{17.14}$$

For each calculation, take your answer and draw the angular momentum **L** and the angular velocity vector ω on a piece of graph paper, taking note if the two are parallel. *Hint:* Make a perspective drawing of three perpendicular axes first, and then mark off components on each axis (or use Maple).] Consider the cases:

a. $\omega = (1, 0, 0)$.
b. $\omega = (0, 1, 0)$.
c. $\omega = (1, 1, 1)$.

4. The determinant of a 3×3 matrix is defined to be

$$\det \begin{pmatrix} M_{11} & M_{12} & M_{13} \\ M_{21} & M_{22} & M_{23} \\ M_{31} & M_{32} & M_{33} \end{pmatrix} = M_{11}M_{22}M_{33} + M_{12}M_{23}M_{31} + M_{13}M_{21}M_{32} - M_{31}M_{22}M_{13} - M_{32}M_{23}M_{11} - M_{33}M_{21}M_{12}.$$

Listing 17.6 Determinant.f90

```
! Determinant.f90: Finds determinant of 3x3 matrix
! ------------------------------------
Program determinant
  Real :: a(3,3), det
  Integer :: i, j
  call random_number(a)
  a = a*5.
  det = a(1,1)*(a(2,2)*a(3,3) - a(3,2)*a(2,3)) &
       + a(1,2)*(a(3,1)*a(2,3) - a(2,1)*a(3,3))  &
       + a(1,3)*(a(2,1)*a(3,2) - a(3,1)*a(2,2))
  do i=1, 3, 1
     if(i==2) then
        write(*,*) 'A = [',(a(i,j),j=1,3,1),']'
     else
        write(*,*) '     [',(a(i,j),j=1,3,1),']'
     end if
  end do
  write(*,*) '|A| = ', det
End Program determinant
```

The program `Determinant.f90` calculates the determinant of a 3 by 3 matrix. Try this program out on a variety of $3\times$ matrices:

a. A diagonal matrix, whose determinant should be the product of the diagonal elements.
b. A matrix with two identical rows (determinant should vanish).
c. A matrix with two identical columns (determinant should vanish).
d. A matrix in which the third row is the sum of the first two (vanish again).

5. Here is the program `ArrayTest.f90` that finds the maximum and minimum of a two-dimensional array of arbitrary length:

Listing 17.7 ArrayTest.f90

```
! ArrayTest.f90: Find min, max of 2D array
! ----------------------------
Subroutine findMinMax(x,min,max)
  Implicit None
  Real, dimension(:,:), intent(in) :: x
  Real :: min, max
  Integer :: i, j, imax, jmax
  imax = size(x,DIM=1);
  jmax = size(x,DIM=2);
  min = x(1,1)
  max = x(1,1)
  do i=1,imax,1
     do j=1,jmax,1
        if(x(i,j)<=min) min = x(i,j)
        if(x(i,j)>=max) max = x(i,j)
     end do
  end do
End Subroutine findMinMax
!
Program main
  Implicit None
  Interface
    Subroutine findMinMax(x,min,max)
      Real, dimension(:,:), intent(in) :: x
      Real :: min, max
      Integer :: i, j, imax, jmax
    End Subroutine findMinMax
  End Interface
  Real :: min, max, PI
  Real, dimension (10,10) :: a
  Integer :: i, j
  PI = 3.141592658979
  !Initialize the array
  do i=1,10,1
     do j=1,10,1
        a(i,j) = exp(-(i*i+j*j)/25.0)*sin((i-1)  &
                 *PI/7)*cos((j-1)*PI/5)
     end do
  end do
  call findMinMax(a,min,max)
  write(*,*) 'Minimum = ', min, 'Maximum = ', max
End Program main
```

   a. Enter `ArrayTest` and get it running.
   b. Write a program `ArraySort.f90` that uses `ArrayTest` to rearrange all the elements of a 2D array from minimum to maximum values of the array elements. Here are some suggestions:

• Use as much as you can of `ArrayTest` just to save yourself some typing.
• The simplest, though not the most efficient, way to sort a list of numbers is to start with the first two elements, `a[1]` and `a[2]`, interchange them if `a[1]` is larger than `a[2]`, and then go on to the next adjacent pair, `a[2]` and `a[3]`, and perform the same operation. Continue until you get to the end of the array, `a[n-1]` and `a[n]`. While one pass through the array will not be sufficient, $n - 1$ passes will.
• Design a procedure that implements the sort procedure just described. While the program will turn out to be quite simple, it may take some thinking to get the logic thought through. Accordingly, *first* sketch out a flowchart or some pseudocode containing the key logical questions and consequences. Hand in your design with the rest of your assignment.
• Write a procedure `xChange(a, i)` that sorts a 2D array. To be safe, you may want to read the discussion on changing an array element in the *call* to a procedure in § 17.5.

Chapter Eighteen

Advanced Function Methods, Objects*

In §16.5 we defined objects as non-primitive data types with multiple components. We have already used such objects to represent complex numbers and arrays, and have investigated some of the interesting characteristics of employing objects as array arguments. In this chapter we examine some more advanced use of procedures and objects.

18.1 PROCEDURE MODULES AND INTERFACES

The simple function and subroutine examples we have presented so far are all that a beginning user should need for their work. However, Fortran does permit some more sophisticated techniques for dealing with procedures, and we shall discuss some of that now in the context of *object-orientated programming (OOP)*. While incorporating these newer features in your programs does require some extra lines of code, OOP techniques lead to less-error prone programs, aid in debugging, and lead to more efficient compilation and optimization (speed) [F90_UL, ChapF 04]. In addition to saving you time in the long run, we employ some of these techniques in our examples, and so it is worthwhile for you to understand them somewhat.

Most of the procedures we have used as examples have been *external*. This is the older, Fortran77 approach in which each procedure is treated like a separate program unit that are developed and compiled independently from the main program and the other procedures, and eventually linked together for execution.

In Fortran90, you may also group together user-defined data types and the procedures that manipulate these types into *modules*. These modules are equivalent to what is called *objects* or classes in the Java and C++ programming languages. Since an object is, in some sense, defined by the methods used to manipulate it, it makes sense to couple together new data types and the methods used to manipulate them.

The real world often maintains some of the traditions of the past, and so sometimes you may need to include some already-written, tried-and-true, older

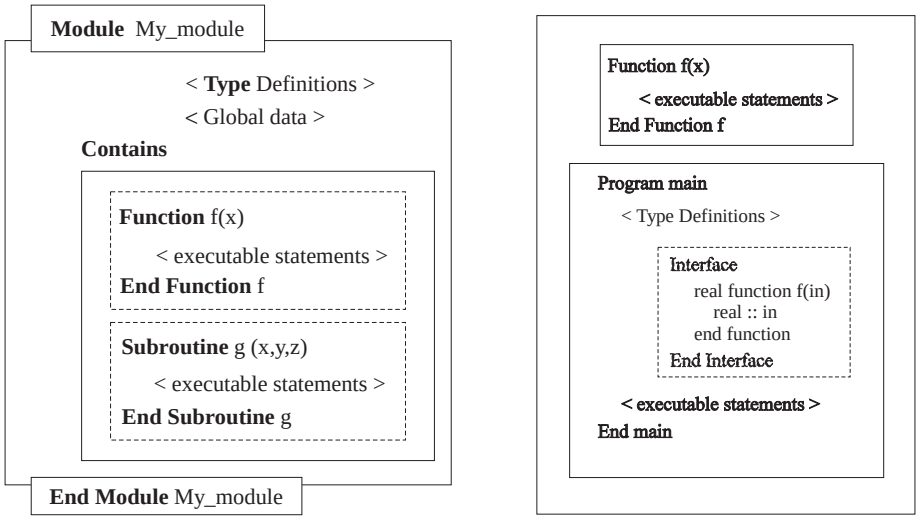


Figure 18.1 *Left:* The structure of a Fortran program containing a procedure module. *Right:* The structure of a Fortran program containing a procedure interface.

Fortran *external* procedures in your modern programs. This is probably a better idea than writing over from scratch complex programs and procedures just to be modern. Fortunately, some of Fortran90's advanced features and compiler optimizations are possible when using older external procedures, if an *interface block* is used to incorporate them into your program environment.

To understand the difference between a procedure module and an interface block, consider Fig. 18.1. The module on the left, and in Lst. 18.1, is a collection of subroutines and functions bound together within the `Module` and `End Module` commands. The procedures have their usual forms and follow the `Contains` command.

The new aspect of a `module` is that at its top we have the opportunity to declare the type definitions for all the arguments used by the procedures in the module, as well as `global data` that will be available to all procedures within the module. In addition, all other programs may utilize procedures from within this module if they simply issue the `Use My_module` command (indicating the inclusion of this module into their program environment). Once the Fortran module file is compiled, it is stored as a file with a `.mod` suffix. It is then available for other programs to `use`. A simple example of a module is `ModuleExample.f90` in Lst. 18.1.

Exercise: Compile `ModuleExample.f90` and before executing note that a file `my_module.mod` has been created. Now run `ModuleExample`, note the results, and try running it again after removing the `my_module.mod` file. ♠

Listing 18.1 ModuleExample.f90

```
! ModuleExample.f90: A module demo with a function and subroutine
! ------------------------------------------------------
Module my_module
  Contains
    Real*8 Function f(x, y)
      Implicit None
      Real*8 :: x, y
      f = sin(x*y) + 1.0
    End Function f
    !
    Subroutine f_sub(x,y,c)
      Implicit None
      Real*8 :: x, y, c
      c = sin(x*y) + 1.0
    End Subroutine f_sub
End Module
! --- The main program that uses the module
Program main
  Use my_module
  Implicit None
  Real*8 :: u=2.1415, v=1., c
  c = f(u,v)
  write(*,*) 'function''s answer   = ', c
  call f_sub(u,v,c)
  write(*,*) 'subroutine''s answer = ', c
End Program main
```

In contrast to a module, the `interface` block illustrated on the right of Fig. 18.1, and in Lst. 18.2, is only a block *within the calling program* that describes the types and arguments of procedures to be *called*. The block is contained between the `Interface` and `End Interface` commands within the main program's header definitions. This block is not a separate entity that needs to be called, and it does not *encapsulate* the executable statements of the functions or subroutines (the two cannot be mixed in an interface). The interface block merely indicates to the compiler which external procedures (and their types and arguments) are to be included in this program's environment. Including those procedures in this way allows for more complete error-checking.

So, even though a *module* is a preferable technique over an `interface`, there are some circumstances when you might use an `interface` block to incorporate an existing, older procedure while still utilizing some of Fortran90's advanced features, as well as making it easier for the compiler to find errors or problems. A simple example of an interface is `FuncInterface.f90` in Lst. 18.2.

Exercise: Compile `FuncInterface.f90`, run it, and compare the results to the previous example, `ModuleExample.f90`. ♠

Listing 18.2 FuncInterface.f90

```
! FuncInterface.f90: Calls a function utilizing an interface
!-----------------------------------------------
Real*8 Function f(x,y)
  Implicit None
  Real*8 :: x, y
  f = sin(x*y) + 1.0
End Function f
!
Program main
  Implicit None
  Interface
    Real Function f(in1, in2)
      real*8 :: in1, in2
    End Function f
  End Interface
  Real*8 :: in1=2.1415, in2=1., c
  c = f(in1, in2)
  write(*,*) 'function''s answer = ',c
End Program main
```

18.2 CHANGING THE ARGUMENTS OF A FUNCTION

Listing 18.3 Change.f90

```
! Change.f90:  function arguement change demo
!-----------------------------------------------
Program change
  Implicit None
  Integer :: a=1, b=2, z, f
  write(*,*) 'In the main program, a =', a, ', b =', b
  z = f(a,b)
  write(*,*) 'In the main program, f(a,b) =', z, ', then...'
  write(*,*) 'In the main program, a =', a,  ', b =', b
End Program change
!
Integer Function f(x,y)
  Integer, intent(inout) :: x, y
  x = 5
  write(*,*) 'In the function, x =', x, ', y =', y
  f = x + y
End Function f
```

This is a warning of something that is more obvious with subroutines than with functions. When you call a function, the memory location of the argument is *passed* to the function from the calling program. In other words, the main program sends only a *memory reference* or memory location to the function, not the actual value of the variable itself (the same as with arguments in subroutine calls).

Accordingly, it is quite acceptable to call a function with the variable `x` as the argument to the function, and then use a different variable like `y` for the corresponding variable within the function itself. As a general rule, *a function may change the value of its arguments, and those changes will be reflected in the corresponding variables of the invoking routine.* In other words, the *changed* value of the variable *will* be returned to the calling program or procedure. Since we only concentrate on the single value of the function name itself, we often do not think about this *side effect*.

The short program `Change.f90` in Lst. 18.3 gives surprising results regarding the values of function arguments. To see them:

1. Compile and run `Change.f90`. You should get the results:

```
In the main program, a = 1, b = 2
In the function, x = 5, y = 2
In the main program, f(a, b) = 7, then ...
In the main program, a = 5, b = 2
```

   Look at the code and locate the four `write` statements. Compare them to the printed results. Note that we start off with `a = 1` and `b = 2` in the main program. Then the function $f(a, b)$ is evaluated, where a and b are assigned to the variables x and y inside f. Inside `f(x, y)`, the code sets `x = 5`, prints out the values (`x = 5, y = 2`), and returns a value of `7` for `f`. When we get back to the main program, we print out `f(a, b) = 7`, and find that the assignment statement inside the function has caused the value of `a` in the main program to change!

   What is happening here is that the variables in each function are *local* to that particular function and not "seen" by other parts of the program. The variables passed to the function `f` do not have their numerical values sent, but rather just their location in memory. Accordingly, the values for variables `a` and `b` in the main program are stored in the same memory locations as the values for variables `x` and `y` in the function `f`. If `a` is changed in `f`, then the corresponding variable `x` will reflect that change in the main program. Since memory locations, not values, are transferred between a calling program and a procedure, it is said that Fortran procedure calls *call by reference* rather than *call by value*.
2. Copy `Change.f90` to `Change2.f90`. Modify the function declaration so that the arguments are reversed:

```
12.  Integer Function f(y, x)                    Reversed arguments
```

   Compile and execute `Change2.f90`. This program should convince you that the variable names inside a function are arbitrary and may be changed according to your preference, as long as the argument types still match. The

function's arguments are, after all, called "dummy" variables to indicate that they are only place holders.

3. As discussed before, you must always write your programs such that the arguments in the call to a function and in the function's declaration match exactly in type. So, for example, if a function is declared with the second argument as a `Real*8`, then the calls to that function must have the second argument as a `Real*8`, *or else!*. To see "or else what", change the dummy arguments on line 12 to `Real*8`. Recompile `Change.f90` and see the results.
4. Not only must the arguments of a function being called match the types of the function arguments declared by the function, but the value that the function returns must also match the type declared for the function in the calling program. For example, make a working copy of `Change.f90`, and replace the function declaration in on line 12 with one that declares the function value returned to be a `Real*8`. Recompile `Change.f90` and see the results. You should get an error message with a reference to "incompatible types" or "function result types differ." These responses are signals of inconsistent data types.

18.3 FUNCTIONS CALLING FUNCTIONS

So far, we have used the main program to call other functions. Actually, functions can call other functions as well. Lst. 18.4 is an example that includes functions calling other functions. Note that we are using double-precision variables and also the tangent function from the Fortran Math library.

Exercise: Compile and run `Function3.f90`. Compare the output you obtain with the `print` lines in the code and explain why the results are as they are.

Listing 18.4 Function3.f90

```
1  ! Function3.f90: Example of functions calling functions
2  !-----------------------------------------------------
3  Module func_module
4    Contains
5      Real*8 Function f(x, y)
6        Implicit None
7        Real*8 :: x, y, w
8        w = g(x)              ! call to function g
9        print *, 'In f(x, y), x =', x, 'g(x) =', w
10       f = w * w + y * y     ! calculate value for f
11     End Function f
12     Real*8 Function g(x)    ! Module function g
13       Implicit None
14       Real*8 :: x
15       g = tan(x) ** 3       ! use tan function for g
16     End Function g
17 End Module func_module
18 !
```

```
Program main_program
  Use func_module           ! Incorporate the module
  Implicit None
  Real*8 :: x = 1.0, y = 3.0, z
  print *, 'In main, x =', x, 'y =', y
  z = f(x, y)               ! call to function f
  print *, 'In main, f = ', z, 'x =', x, 'y =', y
End Program main_program
```

18.4 FUNCTION OVERLOADING

Now that we have warned you how important it is to have the data types of a function's arguments be the same in the calling program as it is in the function declaration, we show you the exceptions to this rule. With *function overloading* you define a number of different functions having the same name but arguments that somehow differ. Explicitly, each function must have either 1) a different selection of argument types, 2) a different number of arguments, or 3) a different return type. You then can use the same function name for differing data types (with the help of the `interface` feature), and, incredibly enough, the compiler will choose the proper one based on the unique combination of arguments and return type. While this may seem too forgiving, functions that differ only in return type are not acceptable.

Listing 18.5 Overload.f90

```
! Overload.f90: overloading via interface
! ---------------------------------------
Integer Function f_int(i)
  Implicit None
  Integer :: i
  f_int = i*i
End Function f_int
Real*8 Function f_real(x)
  Implicit None
  Real*8 :: x
  f_real = x*x
End Function f_real
Complex*16 Function f_complex(z)
  Implicit None
  Complex*16 :: z
  f_complex = z*conjg(z)
End Function f_complex
!
Program main
  Interface f
    Integer Function f_int(i)
      Integer :: i
    End Function f_int
    Real*8 Function f_real(x)
```

```
25        Real*8 :: x
26      End Function f_real
27      Complex*16 Function f_complex(z)
28        complex*16 :: z
29      End Function f_complex
30    End Interface f
31    Integer :: a=2
32    Real*8 :: b=2.
33    Complex*16 :: c=(1.,1.)
34    write(*,*) 'Integer: f(a) = ', f(a)
35    write(*,*) 'Real: f(b) = ',    f(b)
36    write(*,*) 'Complex: f(c) = ', f(c)
37  End Program main
```

The program `Overload.f90` in Lst. 18.5 is an example of overloading, using an `interface`.

1. How many functions `f` do you see in this code?
2. Compile and execute `Overload.f90` to see function overloading in action. Check that you get three lines of output and compare that output to the code to ensure that three different functions with the same name were called.
3. Extend `Overload.f90` so that it also calls a function `f(x,y,z)` where `x, y,` and `z` are `Real*8`, as well as the function `f(i,j,k)`, where `i, j,` and `k` are integers.

Chapter Nineteen

Discrete Math, Arrays as Bins; Nonlinear Bug Dynamics*

This chapter gives an example of how computations are used to model the population dynamics of biological systems. It employs some fairly simple discrete mathematics that leads to some unusual nonlinear behaviors to explore. One-dimensional arrays are used to bin store the results of the simulations, with the arrays then written to a file and visualized with Gnuplot. This chapter introduces no new Fortran tools, but instead reviews a number of previously introduced techniques. So even though there is no advanced material in it, we mark this chapter as optional since it may be skipped without interrupting the logical flow.

19.1 PROBLEM: VARIABILITY OF BUG POPULATIONS

As is true with much in nature, insect populations do not appear to follow any simple patterns. At times they appear stable, at other times they vary periodically, and at still other times they appear chaotic, only to settle down to something simple again.[1]

Problem: Deduce and explore a simple model of the population as a function of time that seems capable of producing complicated behaviors.

19.2 THEORY: SELF-LIMITING GROWTH, DISCRETE MAPS

Imagine a bunch of insects reproducing, generation after generation, in your garden. We start with N_0 of them, in the next generation we have to live with N_1 bugs, and after i generations there are N_n of them to bug us. We want a model for how N_n varies with the discrete generation number n.

We look to the simple model of radioactive decay and growth for guidance.

[1] We actually use *chaos* as a technical term meaning complicated behavior in which there does not appear to be any order, but in which there is some simple mathematical description [CP 05].

There we know that an exponential time dependence,

$$N(t) = N(0)e^{\pm\lambda t}, \tag{19.1}$$

follows from the simple discrete law [CP 05]:

$$\Delta N_n = \pm\lambda N_n \Delta t. \tag{19.2}$$

Here λ is a rate constant, N_n is the number of particles present in generation n, and Δt is the time for one generation. The model states that the change in the number of particles in one generation is proportional to the number of particles present and to the length of that generation.

We know that a discrete equation such as (19.2) produces growth or decay for positive or negative values of λ respectively, but not both. If bugs just bred, then an exponential-growth model might make sense; yet bugs cannot live on love alone. As their numbers grow the bugs start running out of food and this leads to their numbers growing only to some maximum value N_*. We modify the simple growth model by introducing a growth rate λ that decreases as the population number reaches some maximum:

$$\lambda = \lambda'(N_* - N_n), \qquad \Rightarrow \qquad \Delta N_n = \lambda' \Delta t (N_* - N_n) N_n. \tag{19.3}$$

We expect that when N_n is small compared to the maximum N_*, the bugs will grow exponentially, but as N_n becomes comparable in magnitude to N_*, the growth rate should decrease and possibly become negative if it exceeds N_*.

To make the mathematics of this model clearer, we change to dimensionless variables

$$x_n = \frac{\lambda' \Delta t}{1 + \lambda' \Delta t N_*} N_n \simeq \frac{N_n}{N_*}, \qquad \mu = 1 + \lambda' \Delta t N_*. \tag{19.4}$$

In terms of these, our model assumes the simple form

$$x_{n+1} = \mu x_n (1 - x_n). \tag{19.5}$$

Here x_n is essentially the fraction of the maximum population attained in generation n, and μ is a constant we call the *survival rate*. The range of variation of these dimensionless parameters are limited to:

$$0 \le x_n \le 1, \qquad 0 \le \mu \le 4. \tag{19.6}$$

Notwithstanding the simplicity of (19.5) with its one variable x and one parameter μ, it leads to surprisingly complicated behaviors. This is a consequence of its being a *nonlinear equation* in which the variable x occurs to the second power. It is called the *logistics map* [Rash 90], and your **problem** is now reduced to exploring the properties of (19.5).

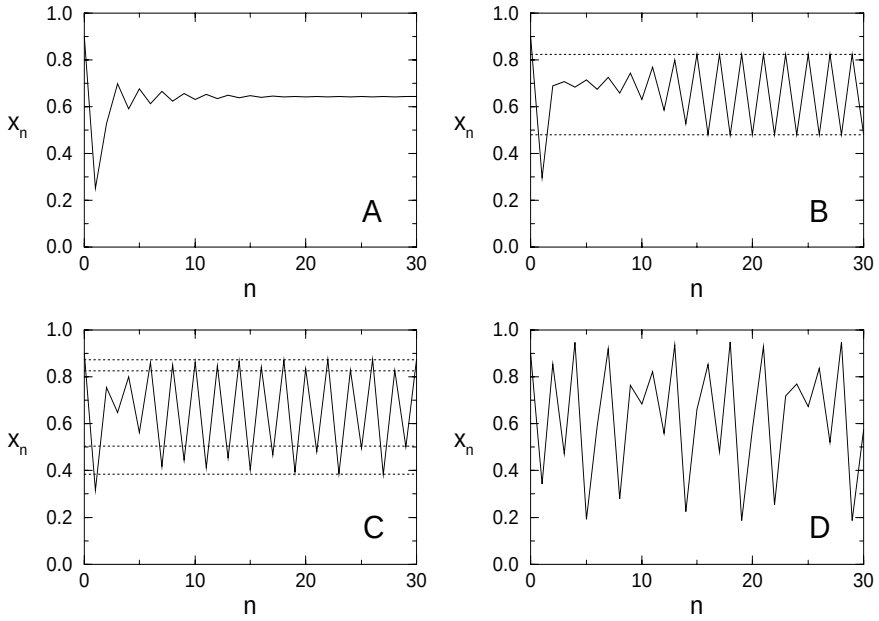


Figure 19.1 The insect population x_n versus generation number n for various survival rates: (A) $\mu = 2.8$, a period-one cycle; (B) $\mu = 3.3$, a period-two cycle; (C) $\mu = 3.5$, a period-four cycle; (D) $\mu = 3.8$, a chaotic regime.

19.3 ASSESSMENT: PROPERTIES OF NONLINEAR MAPS

Rather than do some fancy mathematical analysis [Rash 90] to determine properties of the logistics map (19.5), study it directly on the computer. Write a simple program that uses the logistics map to produce a sequence of population values x_n as a function of the generation number n. Start with a *seed* population $x_0 = 0.75$ and plot up x_n *versus* n for $\mu =$ (0.4, 2.8, 3.3, 3.5, 3.8). The last four simulations should yield results similar to those in Figure 19.1.

Explore other nearby values of μ and identify on printed copies of your graphs the following characteristics:

Transients: Irregular behaviors before reaching a steady state.

Extinction: If the survival rate is too low, the population dies off.

Stable states: Single-population stable states for $\mu < 3$.

Two and three cycles: At two-cycle fixed points, the population jumps back and forth between two semistable x_* values. At three-cycle fixed points the population

jumps between three x_* values. These *attractors* occur only for $\mu > 3$.

Intermittency: Try to find solutions for $3.8264 < \mu < 3.8304$ in which the system appears stable for a while but then jumps all around, only to become stable again.

Chaos: The system's behavior in the chaotic region is critically dependent on the exact value of μ and x_0. Systems may start out much the same but end up quite different. Compare the long-term behaviors of starting with the two essentially identical seeds:

$$x_0 = 0.75, \qquad x_0' = 0.75(1+\epsilon), \tag{19.7}$$

where $\epsilon \simeq 2 \times 10^{-14}$ for a double-precision calculation. Repeat the experiment with $x_0 = 0.75$, but with what should essentially be two identical survival parameters:

$$\mu = 4.0, \qquad \mu' = 4.0(1-\epsilon). \tag{19.8}$$

19.4 EXPLORATION: BIFURCATION DIAGRAM, BUGS.F90*

Listing 19.1 Bugs.f90

```
! Bugs.f90: Bifurcation diagram for logistic map.
! bug population - bifurcation map of m*y*(1-y)
! ------------------------------------------------
Program Bugs
  Implicit none
  ! Declarations (range and resolution for m)
  Real*8 :: m_min, m_max, m, step, y
  Integer :: x
  m_min = 1.0
  m_max = 4.0
  step  = 0.01
  open(6, File='bugs.dat', Status='REPLACE')
  ! Loop for m values, arbitrary starting value for y
  do m=m_min, (m_max-step), step
     y = 0.5
     !Wait until transients die out
     do x=0, 200
        y = m*y * (1 - y)
     end do
     ! Record 200 points
     do x=201, 401
        y=m*y * (1 - y)
        write (6,50) m,y
     end do
  end do
  close(6, Status='KEEP')
  ! Output stored in file: bugs.dat
  50 Format (f5.3,f10.6)
End Program Bugs
```

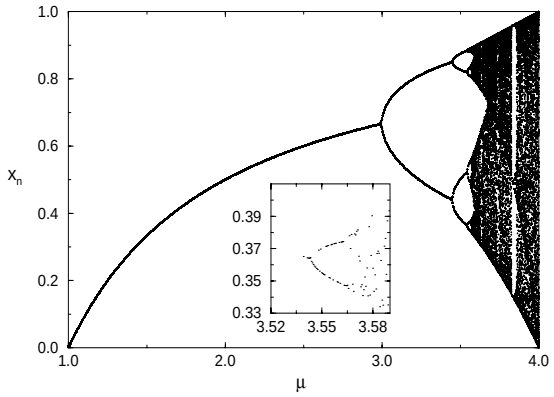


Figure 19.2 The bifurcation plot, attractor populations versus survival rate, for the logistics map.

Listing 19.1 gives our basic program for producing the data for a bifurcation diagram. Computing and watching the population change with each generation gives a good idea of the basic phenomena. However, it is hard to discern the simple and beautiful behavior that lies within these complicated behaviors. One way to gain such understanding is to concentrate on the semistable population values x_* that the system jumps among over time. These are called *attractors* to indicate that the system is somehow attracted to these populations.

A plot of these attractors as a function of the survival parameter μ is an elegant way to summarize the results of extensive computer simulations. One such *bifurcation diagram* is given in Figure 19.2. It shows the output from the program `BugSorted.java` given here. For each value of μ, hundreds of iterations were made to make sure that all transients died out. The value of x that the system settled down to is recorded as x_*. The doublet of points (μ, x_*) that occur after the transients have died out were then written to a file. To be sure that all attractors have been reached, the calculation is repeated for various values for the initial populations x_0.

Implementation: Bifurcation Diagram

Create your own version of Figure 19.2 using Gnuplot. To get an idea of how many points may be needed, let us assume that your screen resolution is $\sim$100 dots per inch and that your laser printer's resolution is $\sim$300 dots per inch. This means that you are plotting approximately $3000 \times 3000 \simeq 10$ million elements. *Beware:* this will take a long time to print and require more memory than some printers have available.

Binning

1. Break up the range $1 \leq \mu \leq 4$ into 1,000 steps and loop through them. These are the "bins" into which we will place the x_* values.

2. In order not to miss any structures in your bifurcation diagram, loop through a range of initial x_0 values as well.

3. Wait at least 200 generations for the transients to die out, and then print out the next several hundred values of (μ, x_*) to a file.

4. Print out your x_* values to no more than three to four decimal places. You will not be able to resolve more places than this on your plot, and this will keep your output files smaller by permitting you to remove duplicates[2]. Another approach is to remove two successive x_n values that differ only beyond the third place. We do that in the sample code `Bugs.f90` by taking advantage of the fact that x_n is in the range

$$0 \leq x_n \leq 1. \tag{19.9}$$

For this reason, before we write out to a file, we multiply our x_n's by 1,000, cast them as integers, and compare with the previous x_n. If they are not equal, then we know the x_n value differs in the first three decimal places from the previous x_n value.

5. Plot up your file of x_* versus μ. Use small symbols for the points and do not connect them.

6. Enlarge sections of your plot and notice that a similar bifurcation diagram tends to be contained within portions of the original (this is called *self-similarity*).

7. Look over the series of bifurcations occurring at

$$\mu_k \simeq 3,\ 3.449,\ 3.544,\ 3.5644,\ 3.5688,\ 3.569692,\ 3.56989, \ldots . \tag{19.10}$$

The end of this series is a region of chaotic behavior.

8. Inspect the way this and other sequences begin and then end in chaos. The changes sometimes occur quickly over a very short range of μ, and so you may have to make plots over a very small range of μ values to see the structures.

9. Close examination of Figure 19.2 shows regions where a very large number of populations suddenly change to very few populations with a slight increase in μ. Whereas these may appear to be artifacts of the video display, this is a real effect and these regions are called *windows*. Check that at $\mu = 3.828427$, chaos turns into a three-cycle population.

[2] For example, with the Unix command *sort -u*.

Table 19.1 Maps for generating x_n sequences and bifurcation plots.

Name	$f(x)$
Logistics	$\mu x(1-x)$
Tent	$\mu(1-2\,\lvert x-1/2\rvert)$
Ecology	$xe^{\mu(1-x)}$
Quartic	$\mu[1-(2x-1)^4]$

19.5 EXPLORATION: OTHER DISCRETE MAPS*

Only nonlinear systems exhibit unusual behavior like chaos. Yet systems are nonlinear in any number of ways. Table 19.1 has maps that you may use to generate x_n sequences and bifurcation plots.

Chapter Twenty

2-D Arrays: File I/O, PDE's; Realistic Capacitor

The main purpose of this chapter is to get more experience with arrays, and especially with the large 2-D ones used to solve realistic problems. As is often the case with realistic problems, no analytic solution exists and we must employ a variety of techniques to solve our problem. These include breaking space into a discrete lattice and applying a simple algorithm to solve a partial differential equation (PDE), file I/O, and 3-D visualization of numerical data using gnuplot. The simplicity of the solution follows from the use of large arrays that permit us to manipulate thousands or millions of variables with just a few lines of Fortran code.

20.1 PROBLEM: FIELD OF REALISTIC CAPACITOR

Figure 20.1, adapted from a first-year physics text, shows an idealized parallel-plate capacitor on the left. The equal spacing and single direction of the electric-field line (arrows) indicate that the field is uniform. On the right is a photograph of the electric field between the plates of a realistic parallel-plate capacitor, visualized by small pieces of thread suspended in oil. Observe how the field fringes out near the ends and extends beyond the ends of the capacitor. Textbooks usually analyze the ideal capacitor because only it can be treated analytically.

Problem: Determine and visualize the electric field within and surrounding a realistic capacitor, and compare to Figure 20.1.

20.2 THEORY AND MODEL: ELECTROSTATICS AND PDES

Capacitors are devices with the *capacity* to store electric charge. They usually consists of two conductors of similar shape separated by a small gap, with equal—but opposite–charge on each conductor. Because the electric field, which would otherwise extend over all of space, is condensed to the region between the plates, capacitors are also called *condensers*.

In spite of real electrical devices being three-dimensional, we model the capacitor, as shown on the left of Figure 20.2, as two wires connected to a battery

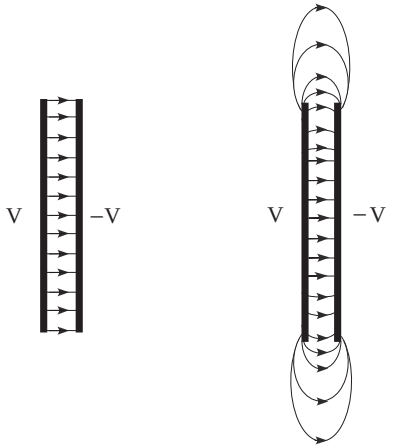


Figure 20.1 *Left*: Electric-field lines betweens the plates of an ideal parallel-plate capacitor. The equal spacing and single direction indicate a uniform field. *Right:* A representation of the field between the plates of a realistic capacitor. The field tends to "fringe" and extend beyond the ends of the plates.

that keeps the top plate at 100 volts and the bottom one at -100 volts. To permit a solution to this capacitor, we need to contain it in a limited space, which we do by placing it inside a charge-free box made from wires that are "grounded" (kept at 0 volts). After we solve this problem, we may make it more realistic still by enlarging the box so that it has no effect on the field near the capacitor. This is little work for us, but much more work for the computer.

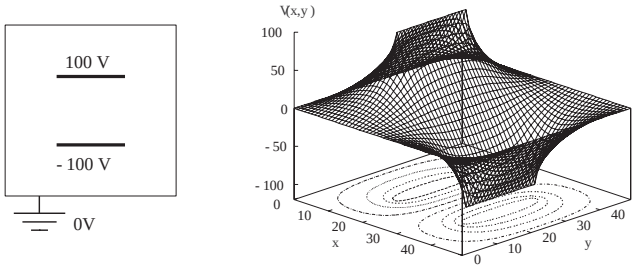


Figure 20.2 *Left:* A parallel-plate capacitor within a grounded box. A realistic capacitor would have the plates closer together in order to condense the field. *Right:* A visualization of the electric potential for this geometry. The contours projected onto the xy plane give the equipotential surfaces.

Laplace's Partial Differential Equation*

Rather than solve for the electric field $\mathbf{E}(x, y)$, we solve for the electric potential $V(x, y)$. This is easier because V is a scalar while $\mathbf{E}$ is a vector, and because the equations for V are simpler. Insofar as the field $\mathbf{E}$ equals the derivative (gradient) of V, the two solutions are equivalent. (Electric-field lines are perpendicular to equipotential surfaces and run from higher to lower potential values.)

Elementary physics texts show how to determine electric-field configurations using Gauss's law. We skip the mathematical manipulations needed to prove it and just state that the solution using Gauss's law is equivalent to finding a solution of the partial differential equation (PDE),

$$\frac{\partial^2 V(x,y)}{\partial x^2} + \frac{\partial^2 V(x,y)}{\partial y^2} = 0. \tag{20.1}$$

This is called Laplace's equation, in honor of the genius who first did those mathematical manipulations. The rounded Ds in (20.1) indicate that the derivatives are partial not *ordinary*. The reason this is a *partial* DE is that the potential function $V(x, y)$ is a function of *both* x and y. We cannot solve the x behavior without simultaneously solving for the y behavior. The real world is full of such equations.

20.3 ALGORITHM: FINITE DIFFERENCES

The simplest method for solving our PDE is illustrated in Figure 20.3 and is known as a *finite-difference technique*. There we see on the left a grid of x and y values, each uniformly spaced by a step size $\Delta x = \Delta y = \Delta$. Rather than trying to find a solution for the continuous range of all possible values of x and y, we try to find the solution only for the finite number of values that lie on the nodes of the grid. Yet even for the small 100×100 grid, this requires solving for 10,000 unknowns. Though not a trivial problem, the use of arrays makes it simple.

Approximating derivatives numerically is quite easy. In fact, in Chapter 15 we show how to do it in order to find a solution of an ordinary differential equation. In the present case, the second partial derivatives are approximated simply as [CP 05]:

$$\frac{\partial^2 V(x,y)}{\partial x^2} \simeq \frac{V(x+\Delta, y) + V(x-\Delta, y) - 2V(x,y)}{\Delta^2}, \tag{20.2}$$

$$\frac{\partial^2 V(x,y)}{\partial y^2} \simeq \frac{V(x, y+\Delta) + V(x, y-\Delta) - 2V(x,y)}{\Delta^2}, \tag{20.3}$$

where Δ is the spacing of the grid in Figure 20.3. After we substitute this expression, and the analogous one for the second y derivative, into Laplace's equation

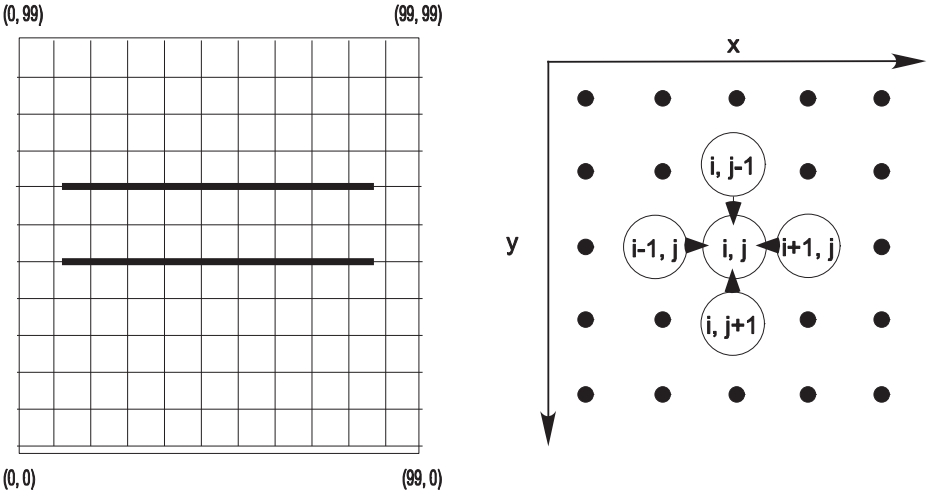


Figure 20.3 *Left:* The capacitor's field is computed only for those (x, y) values on the grid. The voltage of the plates and containing box are kept constant. *Right:* The algorithm for Laplace's equation in which the potential at the point $(x, y) = (i, j)\Delta$ equals the average of the potential values at the four nearest-neighbor points.

(20.1), we obtain a finite-difference approximation to the Laplace equation:

$$V(x,y) \simeq \tfrac{1}{4}\left[V(x+\Delta,y) + V(x-\Delta,y) + V(x,y+\Delta) + V(x,y-\Delta)\right],$$

$$\texttt{V[i][j]} = \tfrac{1}{4}\texttt{(V[i+1][j] + V[i-1][j] + V[i][j+1] + V[i][j-1])} \ . \quad (20.4)$$

Equation (20.4) is the specific implementation of the equation we need to solve in terms of the 2-D Java array elements `V[i][j]` in which we store the potential values for $x = i\Delta$ and $y = j\Delta$. This equation is visualized on the right of Figure 20.3, which shows that the potential at point (i, j) is the average of the potential values at the four nearest neighbors. If we wrote down this equation for each of the 10,000 points on our grid, we would have 10,000 simultaneous equations for 10,000 simultaneous unknown values of $V_{i,j}$.

Even though it is possible to solve these 10,000 simultaneous equations in a direct fashion, we will follow a simple, indirect approach know as a *relaxation* technique. Equation (20.4) states that a solution for the potential must be the average of the potential values at the four nearest neighbors. With that in mind, we try an iterative approach to finding the solution:

1. Start with an initial guess $V(x, y) = 0$ every place except on the plates.
2. To obtain an improved guess, apply (20.4) to all points in space, except the plates and box.
3. Keep applying (20.4) to obtain an even more approved guess until the improvement becomes smaller than some preset level of tolerance.

There is no guarantee that this relaxation method converges for arbitrary choices

of Δ and geometries. Yet if we do come up with a solution that ends up satisfying (20.4), then we have our solution and it's nobody's business how we got it!

20.4 IMPLEMENTATION: LAPLACEFILE.F90

Listing 20.1 LaplaceFile.f90

```
! LaplaceFile.f90: Laplacian electrostatics on square grid, File
! ------------------------------------------------------------
Subroutine solve(tmax, imax)
  Implicit None
  Integer, intent(in) :: imax, tmax
  Real, dimension(imax,imax) :: V
  Integer :: i, j, k
  !Initialize the matrix V to zero
  V=0.
  !Boundary conditions
  V(imax/5:4*imax/5,imax/3) = 100.0
  V(imax/5:4*imax/5,2*imax/3) = -100.0
  V(1,1:imax) = 0.0
  V(imax,1:imax) = 0.0
  V(1:imax,1) = 0.0
  V(1:imax,imax) = 0.0
  !Jacobi relaxation
  do k=1,tmax,1
    do i=2,imax-1,1
      do j=2,imax-1,1
        V(i,j) = 0.25*(V(i+1,j)+V(i-1,j)+V(i,j+1)+V(i,j-1))
        !Continue applying interal boundary conditions
        if(j==imax/3 .AND. i>=imax/5 .AND. i<=4*imax/5) then
          V(i,j) = 100.0
        else if(j==2*imax/3 .AND. i>=imax/5 .AND. i<=4*imax/5)
            then
          V(i,j) = -100.0
        end if
      end do
    end do
  end do
  !Write the matrix to a file
  open(UNIT=1,FILE='LaplaceData.dat',STATUS='REPLACE')
  do i=1,imax,1
    write(1,40) (V(i,j), j=1,imax,1)
    write(1,50)
  end do
  close(UNIT=1,STATUS='KEEP')
  40 format(100F13.6)
  50 format(' ')
End Subroutine solve
!
Program LaplaceFile
  Implicit None
  Integer :: tmax, imax
```

```
45   imax=101
46   tmax=1001
47   call solve(tmax,imax)
48 End Program LaplaceFile
```

We have prepared two implementations of our solution for a realistic capacitor. The program `LaplaceFile.f90`, shown above in Lst. 20.1, solves the problem and writes the solution to a file for separate visualization. The program `Laplace.f90`, given in Lst. 11.3 in Chapter11, solves the problem and then uses Dislin to produce a 3-D visualization.

The pseudocode for our solution involves an outermost loop for iterations, and inner loops over `imax` x and y values:

```
1    set potential = 0 on surrounding around box
2    set potential = 100, -100 on upper, plates
3    repeat 1000 times
4        for imax x space steps
5        for imax y space steps
6            V = average of 4 nearest neighbors
7            keep potential on box and plates fixed
8    write V matrix to a file
```

This algorithm is well suited to the use of arrays because the indices vary, but not the equations. This means, as we see in Listing 20.1, that it does not take too many lines of code to implement. We store the potential $V(x, y)$ in an array `V(i,j)`, with the first index spanning all possible x values and the second index corresponding to all possible y values (measured in steps of Δ). The array is declared as `V(imax,imax)`, and we set $imax = 101$ steps for the sample program. The voltages of the capacitor plates are kept fixed, while the boundary conditions for the box are imposed on lines 11-16.

The only tricky parts in `LaplaceFile.f90` are ensuring that we do not let the algorithm change the value of the externally-applied voltages, and that we do specify any indices that lie outside of the array. Once there are no errors, we just let the program repeat itself `tmax=1001` times and relax into what we hope will be a stable value for the potential.

LaplaceFile.f90 Orientation and Visualization

Before trying to modify a complicated or subtle program, it is good to make sure that it runs, and to orient yourself as to how it goes about its business. Likewise, it is good to do some visualizations of the results so that you get a feel for what

its output should be. For that purpose, make a copy `LaplaceFile.f90` to your personal directory and read through it to get a general idea of what it does.

• Make a paper listing of `LaplaceFile.f90` and draw boxes on it illuminating the following structures in the code:

1. the loop that initializes the potential to 0,
2. the loop that keeps the box's potential at 0,
3. the loops that sets the voltage on the capacitor plates,
4. the loops that sweep over all x and y values,
5. the loop that implements the algorithm (20.4),
6. the statements that output data to a file,

• Before you make any modifications, compile and execute `LaplaceFile.f90`. Make sure the data was indeed written to the file `LaplaceData.dat`. Open the file `LaplaceData.dat` with an editor. It should contain the solution $V(x, y)$ in a format that does not bother telling you the x and y values. There are 100 values of V followed by a blank line. These are the values of `V(1:100,0)`. The second 100 numbers are `V(1:100,1)`, and so forth. *Note:* The plotting program `gnuplot` automatically uses the row and column indices as the values for x and y values that it plots, which means that you do not have to enter them.

• Use `gnuplot` (or whatever you have available, such as Dislin) to construct a 3-D surface plots of the potential versus both x and y. Experiment with the options to create the best visualization. We show you how to do this with Gnuplot in §11.3, with the explicit command given in § 11.3. Your plot should look like Figure 20.2. The height of the surface represents the value of the potential $V(x, y)$ for the x and y values along the horizontal plane. Take stock of how the potential of the bounding box is always zero, and that of the plates are 100 and -100 volts. On the horizontal plane we have placed contour lines that correspond to curves along which the potential is constant. These contours are also called *equipotential surfaces*. In general, the field is seen to be compressed between the plate and to vary more slowly outside the plates than within them.

• This program opens a file called `UNIT=1`, writes to it, and then closes the file. The `REPLACE` option on the `open` statement indicates that the old version of the file will be written over, while the `KEEP` option on the `close` command ensures that the file written to disk remains after the program stops executing.

20.5 EXPLORATION

• The program `LaplaceFile.f90` assumes that the potential must have relaxed to its final value after 1000 iterations. Decide if this is a good assumption:

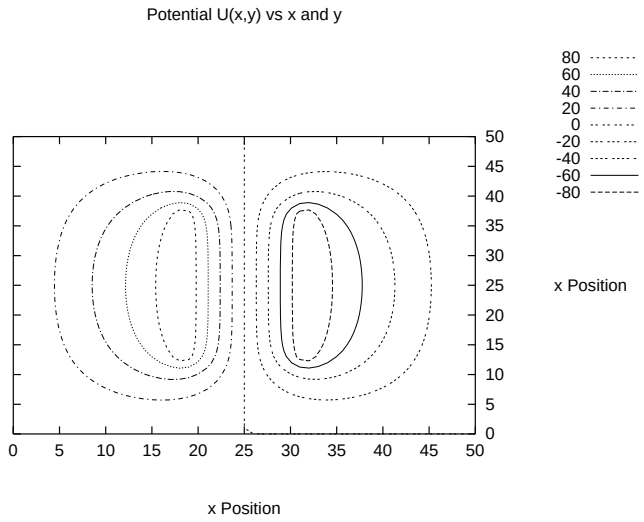


Figure 20.4 Contour plot of equipotential surfaces. The electric field lines are perpendicular to these contours and point toward lower potential.

a. Modify a copy of `LaplaceFile.f90` so that it outputs to a file `Laplace2.dat`.
b. Rather than try to discern small changes in a highly-compressed surface plot, use a measure of precision with a numerical value. If you look at the surface plot of the potential in Figure 20.2, you will see that a line along the diagonal samples the whole range of the potential. On this account, define a variable

$$\texttt{trace} = \sum_i \left|\texttt{V(i,i)}\right| \tag{20.5}$$

and print out the value of `trace` for each iteration. If `trace` changes in the sixth decimal place, you probably have five good places of precision.
c. Now modify the program so that it always obtains at least three places of precision. Define `tol = 0.001` as the desired relative precision and modify `LaplaceFile.f90` so that it stops iterating when the relative change from one iteration to the next is less than `tol`:

$$\left|\frac{\texttt{trace - traceOld}}{\texttt{traceOld}}\right| < \texttt{tol}. \tag{20.6}$$

You can use a `do-while` loop for this purpose.

• Now that you know the code is accurate, make it simulate a more realistic capacitor in which the plate separation is 1/10th of the plate length. You should find the field more condensed between the plates.

• Compare the results of your simulation to the sketch on the right of Figure 20.1. Whereas we are computing the electric potential while the sketch is of the electric field, some processing is necessary. We make use of the fact that the electric field lines point from regions of higher to lower potential, and that the electric field lines are perpendicular to the equipotential surfaces (the contours in our plots). Though you may try to draw lines perpendicular to the contours projected onto the base of the 3-D plot, it would be easier to look down directly onto the base. Make a 2-D graph, like Figure 20.4, of just the contour lines. The needed `gnuplot` commands are
```
gnu> set nosurface
gnu> set view 0,0,1
```

• Draw in the electric field lines by hand, starting from the plate at -100 volts, always keeping the field lines perpendicular to the equipotential surfaces and going downhill.

20.6 EXPLORATION: 3-D CAPACITOR*

Our solution for the capacitor is general and realistic. However, our model is only two dimensional (the "plates" are lines not squares). Extend `LaplaceFile.f90` so that the plates have finite dimensions in both the x and y directions. You will now have to solve for $V(x, y, z)$, and make three, 3-D surfaces of $V(x, y, z)$ *versus* $x - y$, $x - z$, and $y - z$.

20.7 KEY WORDS

boundary conditions	equipotential surfaces	electric field	electrostatics
electric potential	equipotential lines	capacitor	gnuplot
hidden lines	finite difference method	initial conditions	postscript
partial derivatives	reference pass	contour lines	

Chapter Twenty One

Web Computing (see Java Version)

PART 3

LaTeX SURVIVAL GUIDE

Chapter Twenty Two

LaTeX for Text: Scientific Document Processing, Markup Language

22.1 WHY LaTeX?

In the 1980's Donald Knuth created a revolutionary computer program called TeX ("tech") for producing book-quality technical documents [Knuth 86] that rivaled the documents produced by a professional human typesetter. In 1985 Leslie Lamport combined a number of TeX commands into a set of higher-level macros known as LaTeX [Lap 85] that has fewer details to worry about than TeX. At present LaTeX is widely used for scientific journals, books, and reports, and for good reason. LaTeX automatically generates tables of contents, lists of figures, cross-references, equations, section and figure numbers, indexes, and bibliographies. Nevertheless, it is the mathematics that LaTeX does better than any other electronic system. This chapter describes enough of the basic elements of LaTeX for you to produce your own beautiful-looking scientific documents [Wilk 95].

To produce a LaTeXdocument, all you have to do is prepare a source file with some LaTeX commands embedded at the correct places, and then compile the source. Because the source file is an ASCII file, it is possible to write and read it on any computer system with no special programs. Of course one needs a copy of the LaTeX program to typeset this ASCII file, but the program is free and readily available.

The basic steps for using LaTeX are shown in Figure 22.1. It is seen to be quite similar to compiling and running a code, and so makes a useful last part for this book. By convention, the LaTeX source file has a `.tex` extender and the compiled version of the source has a `.dvi` extender. We have called our source file `Paper.tex` and the compiler has produced the *device-independent* file `Paper.dvi` from it. A `.dvi` file is independent of any specific computer system or hardware, and is viewed on a computer screen with a program known as a *viewer*, or converted to a PostScript (`.ps`) or Portable Document Format (`.pdf`) file to be sent to a printer or posted on the Web.

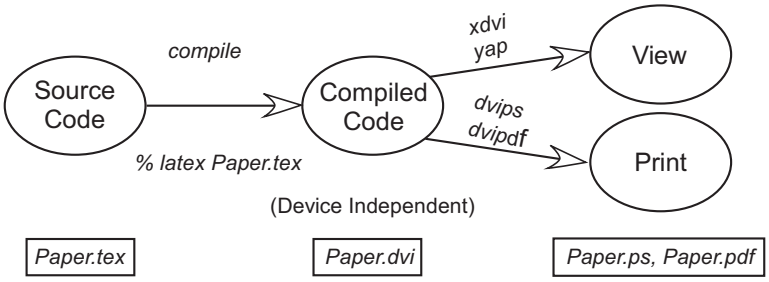


Figure 22.1 The steps followed and utilities used in preparing a document with LaTeX. The source file is `Paper.tex`, the device-independent file is `Paper.dvi`, and the file ready for printing/posting is `Paper.ps` or `Paper.pdf`.

The theory behind LaTeX is that authors should worry about the contents of a document and leave the format up to the computer. In practice there are a number of steps to follow before authors see their documents typeset. This is in contrast to WYSIWYG (what you see is what you get), in which the formulas appear on the computer screen as you enter them. We personally prefer a markup-language approach for complicated documents, especially if that document is to be placed in a digital library. We find that not having to concentrate on how things look until after the writing is over lets us get the hard part of creation done first. However, there are a number of LaTeX commands to learn, and debugging complicated equations can be challenging. Nevertheless, a thing of beauty is a joy forever.

22.2 STRUCTURE OF A LaTeX DOCUMENT

LaTeX is a member of a class of languages known as *markup languages*.[1] The term "markup" comes from the printing trade, where an editor would "markup" a document by inserting symbols to indicate properties such as font size and spacing. This same purpose is served by the LaTeX commands, only in this case they are not for a human typesetter to follow, but for a computer. Inasmuch as LaTeX is a compiled, in contrast to an interpreted, language, when you are done with the writing, the compiler will read your entire document, place the tables, figures, and so forth, in the best places throughout your document, and then cross-reference the whole thing. Like other compilers, you may have to remove all errors before the compiler produces anything, and the compiler may need to make several passes through the document to get it all right.

22.3 SAMPLE INPUT FILE (SAMPLE.TEX)

A LaTeX source file has two basic parts:

[1] Others include HTML, XML, GML, and MATHML.

```
\documentclass[options]{classname}
    ...
        Part I: preamble
    ...
    \begin{document}
    ...
        Part II: actual document
    ...
     %%\end{document}
```

The first part is the preamble, and it starts with a statement of the type of document your document is to be. The choices, `classname = article, report, book, slides,` or `letter`, seem to cover most everything. Older versions of LaTeX started with a `\documenttype` command, but the newer LaTeX 2e uses the `\documentclass` command. The `[options]` argument might include the font sizes (in points)—10 pt (default), 11 pt, or 12 pt—as well as `onecolumn` (default), or `twocolumn`, `oneside` or `twoside`, or `landscape`. With `\documentclass{article}` you obtain a document in 10 pt font. The preamble might also include macros (commands) that you have defined for your own use, and commands for including special packages, for example, for dealing with graphics.

To give you a real example, here, and on the CD is `Sample.tex`, an edited version of a problem set with a comment following the `%`:

```
\documentclass{article}
\begin{document}

\section{Sample \LaTeX\ Output}\index{Comments!Latex@\LaTeX}

\noindent This document consists of bits and pieces of a
final examination given at \textbf{\large Oregon State
University}. The material is from the text
\emph{Computational Physics} \cite{CP}, which does not cost
much \$\$.

We have left off anything fancy, like title page, from the
preamble and used an enumerated list environment
(\tell{enumerate}), with mathematics environment inserted as
needed. An \emph{Environment} usually begins with
\verb+\begin{name}+ and ends with \verb+\end{name}+.

% This is a one-line comment beginning with %; it does not get printed.

\begin{enumerate}

\item Given $N=6$ random numbers distributed between 0 and
10,
\[ 3, 7, 3, 1, 8, 2 \]

\begin{enumerate}
    \item Use this sequence to simulate a random walk and determine
          if the distance from the origin $R = \sqrt{N}$.
```

```
    \item Use this sequence to determine an approximate value for the
          double integral

    \begin{equation}
    \int_0^1 \int_0^1 {dx\,dy\over  x^2+y^2}
    \end{equation}

\end{enumerate}

\item Given the Schr\"{o}dinger equation for two coupled
wave functions $\psi_1$ and $\psi_2$,
    \begin{eqnarray}
 {-\hbar^2 \over 2m_1} {d^2 \psi_1 (x)\over dx^2} + V(x)\psi_1(x) +
              U(x)\psi_2(x)   &=& E \psi_1(x) \label{eq.1} \\
{-\hbar^2 \over 2m_2} {d^2 \psi_2 (x)\over dx^2} +
V(x)\psi_2(x) +
                  U(x)\psi_1(x)&=& E \psi_2(x) \label{eq.2}
    \end{eqnarray}
Express equations (\ref{eq.1}) and (\ref{eq.2}) in the
dynamical form appropriate for numerical integration:
\[
{d \textbf{y}(t)\over dt} = \textbf{f(y},t)
\]
\end{enumerate}
%\end{document}
```

When we apply LaTeX to these paragraphs we produce the text:

22.4 SAMPLE LaTeX OUTPUT

This document consists of bits and pieces of a final examination given at **Oregon State University**. The material is from the text *Computational Physics* [CP 05], which does not cost much $$.

We have left off anything fancy, like title page, from the preamble and used an enumerated list environment (`enumerate`), with mathematics environment inserted as needed. An *Environment* usually begins with `\begin{name}`and ends with `\end{name}`.

i) Given $N = 6$ random numbers distributed between 0 and 10,

$$3, 7, 3, 1, 8, 2$$

(a) Use this sequence to simulate a random walk and determine if the distance from the origin $R = \sqrt{N}$.

(b) Use this sequence to determine an approximate value for the double integral

$$\int_0^1 \int_0^1 \frac{dx\,dy}{x^2 + y^2} \tag{22.1}$$

ii) Given the Schrödinger equation for two coupled wave functions ψ_1 and ψ_2,

$$\frac{-\hbar^2}{2m_1}\frac{d^2\psi_1(x)}{dx^2} + V(x)\psi_1(x) + U(x)\psi_2(x) = E\psi_1(x) \tag{22.2}$$

$$\frac{-\hbar^2}{2m_2}\frac{d^2\psi_2(x)}{dx^2} + V(x)\psi_2(x) + U(x)\psi_1(x) = E\psi_2(x) \tag{22.3}$$

Express equations (22.2) and (22.3) in the dynamical form appropriate for numerical integration:

$$\frac{d\mathbf{y}(t)}{dt} = \mathbf{f}(\mathbf{y}, t)$$

Brief Look at Input File

Most of the characters in our input file `Sample.tex` are the same as in the output. However, mathematical symbols, Greek, and some special characters are entered with commands: the math symbols and Greek, because they do not appear on the keyboard, the special characters because LaTeX uses them within its own commands.

Look at the commands in `Sample.tex`. You will notice that LaTeX commands usually consist of a backslash \ followed by a word. As an instance, we use `\item`, `\begin{equation}`, and `\emph{Computation Physics}` respectively. The Greek letter ψ was produced with `\psi`, and the mathematical symbol $\sqrt{N}$ with `\sqrtN`. As we also see in these examples, the special characters { and } are used to enclose the argument to a LaTeX command. To illustrate, the $\sqrt{}$ is applied to N via `\sqrtN`, and both the words *Computational* and *Physics* are emphasized by `\emph{Computational Physics}`. Of special importance in this file is the dollar sign `$`. When used as part of the text to indicate a dollar sign, it is entered as `\$`. When used as part of LaTeX command to switch between text and math modes, it is entered as `$`, for example:

```
coupled wave functions $\psi_1$ and $\psi_2$
```

coupled wave functions ψ_1 and ψ_2

Although we talk more about this in a separate section, we produced mathematical equations that were separated off from the text ("displayed") by placing them between the commands `\[` and `\]`, or between `\begin{equation}` and `\end{equation}`, or between `\begin{eqnarray}` and `\end{eqnarray}`.

Special Characters

The characters

```
# $ % & ~ _ ^ \ { }
```

are called "special characters" because LaTeX uses them for its own purposes. This means that you cannot use them unless you do something special. If you simply want the special character to be printed just as any other letter, include a \ in front of the character:

```
$ \$
```

There are two exceptions to this rule. You cannot produce \ via \\, because LaTeX uses \\ to indicate the end of a line. So produce it by typing `$\backslash$`:

```
\ $\backslash$
```

The second exception is the tilde. Seeing that `\~` means "place a tilde accent over the following letter," a ~ is produced by `\~{}`, that is, having nothing there (empty braces) for the tilde to go over:

~name `\~{}name`

Paragraphs, Spaces, and Breaks

Look now at the two short paragraphs below the headings in `Sample.tex`. A new paragraph is indicated by one (or more) skipped lines. LaTeX then automatically indents the paragraph as appropriate for its place in the document, for example, it does not indent at the beginning of a new section. To ensure that our first paragraph after the table is *not* indented, we placed a `\noindent` command there.

Ordinary text is entered with no special commands. In general you do not even have to worry about the spacing between words and lines, or where lines begin and end. LaTeX treats any number of spaces as equivalent to a single space, and views the carriage return (↵ Enter) at the end of a line as though it were just another space. If you want to insist on an extra blank space being inserted at some point, then you insert a `\` command, that is, a blank space preceded by a *backslash*. This means that you do not have to worry about spaces in your input, although you should give it some structure with spaces and new lines to make the content clear, as you would with any coding.

Though you should not have to do this for most of your work, sometimes you do need to insert a horizontal or vertical blank space in your document. You request that LaTeX does that (a "request" because LaTeX has its own ideas about spacing, and it gets the last word) with the `\hspace{}` and `\vspace{}` commands. You become more insistent by using the `\hspace*{}` and `\vspace*{}` forms. These commands take the length of the space as an argument in the braces. The length is specified with a number of dimensional units that include:

`ex em pt pc in cm mm`

Here, `ex` and `em` are the sizes of a typeset x and m, `pt` and `pc` are the typesetting measures of points and picas, and the others are obvious.

```
Here \hspace{4ex} is a 4ex horizontal space.
Here \hspace{9mm} is a 9mm horizontal space.
Negative space & overstriking\hspace{-3em}---
A 3ex vertical space between here \vspace{3ex}

and this line.
```

Here is a 4ex horizontal space.
Here is a 9mm horizontal space.
Negative space & overstriking---
A 3ex vertical space between here

and this line.

If you want to insist on a line ending someplace and a new paragraph beginning, then you insert a double *backslash* `\\` to end the line. In addition, you add some extra blank space after the line by including an argument to `\\`. To illustrate, we place `\\*[3ex]` here,

and you see that it skips three ex's of space before printing this line.

Usually LaTeX decides where to start a new page according to its own internal rules. As an example, it tries not to split up tables or place the last line of a paragraph on a new page. If you disagree with LaTeX's decision and wish to force a page break, use:

`\pagebreak`	hint to LaTeX if embedded, command if on separate line
`\newpage, \clearpage`	no attempt to fill page

Though you would never "double-space" a book or journal article, there are occasions when you want to double-space a draft, or to make a business letter fill the page. If you want to vary the interline spacing, you need to change the LaTeX environmental variable `\baselinestretch` from its default value of 1. Generally you do this in the *preamble* with:

```
\renewcommand{\baselinestretch}{1.75}
```

where we have set the spacing to one and three-quarter lines. The command then holds for the entire document. (You may try to reset the spacing locally, but that may mess up the length of the page.)

Quotation Marks and Dashes

Despite most of us never worrying about such things, LaTeX is fairly serious when it comes to quotation marks and dashes. This is a consequence of its being a typesetting program, and typesetters are very serious about such things. When writing in LaTeX, we usually place a word in quotation by placing two left quotes on the left side and two right quotes on the right. As a case in point, the quotes in

"golfballs appear to 'drop out of the sky' near the end of their trajectories"

is produced from the input

```
``golfballs appear to 'drop out of the sky' near the end of
their trajectories''
```

Scrutinize how the left double quote " is entered as *two* left single quotes in succes-

sion, ‘ and ‘, and the right double quote is entered as *two* right single quotes, ’ and ’. As a general rule, one should not use the (undirected) ‘double quote’ character `"` on the keyboard. (Some text editors, like emacs, automatically substitute either “ or ” when you enter `"` if they know you are entering a LaTeX document.)

LaTeX and typesetters deal with dashes of three lengths:

intra-word dash or hyphen, as in intra-word	`intra-word dash or hyphen, as in intra-word`
double dash indicates range, as in 3–8	`double dash indicates range, as in 3--8`
triple or punctuation dash—popular in news	`triple or punctuation dash---popular in news`

22.5 FONTS FOR TEXT

The basic theory behind LaTeX is that the user should worry about content and let the program worry about presentation. So it follows that you should tell LaTeX whether you want some material to be *emphasized*, or **emboldened**, or presented as mathematics, but you should not have to worry about the name or size of the actual font being used. To give an instance, you *emphasize* text, and LaTeX usually reacts by placing that text in italic font. However, LaTeX uses a different italic font for text than it does for mathematics; for example, $this\ (math)$ is different from *this (text)*.

By default, LaTeX uses the *Computer Modern* fonts, which give a characteristic look to LaTeX documents. If you really want some material to have a certain presentation that differs from LaTeX’s standard, then you may have to change some things “by hand” or develop some macros. However, those occasions should be rare and should be done only at the last minute before final presentation. The words you wish to have in a specific format are usually given as the arguments, for example:

Emphasized text	`\emph{Emphasized text}`
Bold text	`\textbf{Bold text}`
`Teletype (monospaced) text`	`\texttt{Teletype (monospaced) text}`
SMALL CAPITAL LETTERS	`\textsc{small capital letters}`
$f = md^2x/dt^2$ (math)	`$f = m d^2 x/dt^2$` (math)

Explicit fonts are obtained with commands of the sort:

Roman font	`\textrm{Roman font}`
Italic font	`\textit{Italic font}`
Sans serif font	`\textsf{\small Sans serif font}`
Slanted, but not italics	`\textsl{Slanted, but not italics}`

If you are working with someone else’s document, you may run into some old style commands for font changes. Though not strictly LaTeX 2e, these still work:

Emphasized	`{\em Emphasized}`
Bold	`{\bf Bold}`
`Teletype`	`{\tt Teletype}`

Type Sizes

I'm tiny	`\tiny{I'm tiny}`
I'm footnotesize	`\footnotesize{I'm footnotesize}`
I'm normalsize	`\normalsize{I'm normalsize}`
I'm Large	`\Large{I'm Large}`
I'm huge	`\huge{I'm huge}`

I'm scriptsize	`\scriptsize{I'm scriptsize}`
I'm small	`\small{I'm small}`
I'm large	`\large{I'm large}`
I'm LARGE	`\LARGE{I'm LARGE}`
I'm Huge	`\Huge{I'm Huge}`

Fancy Text Accents

English appears to be an underachiever when it comes to accents. However, if you wish to include some foreign text in your documents, then you may well need some of the accents LaTeX provides:

`\'{o}`	ó	`\`{o}`	ò	`\^{o}`	ô	`\"{o}`	ö	`\~{o}`	õ	`\={o}`	ō
`\.{o}`	ȯ	`\u{o}`	ŏ	`\v{o}`	ǒ	`\H{o}`	ő	`\t{oo}`	o͡o	`\c{o}`	o̧
`\d{o}`	ọ	`\b{o}`	o̲	`\'{i}`	í	`\'{j}`	ǰ				

If you want to accent mathematics, then different commands are needed yet, and we will get to that.

Math Symbols in Text

Placing mathematics in text, such as $\int f(x)dx$, is done by placing the mathematics between dollar signs, for example, `$\int f(x)dx$`. Other symbols are created in math mode with the commands:

#	`\#`	$	`\$`	%	`\%`	&	`\&`
\	`$\backslash$`	\	`\verb'\'`	^	`\verb'^'`	^	`\char94`
^	`\^{}`	~	`\char126`	{	`\{`	}	`\}`
_	`\_`	œ, Œ	`\oe, \OE`	æ, Æ	`\ae, \AE`	*r*a, *r*A	`\aa, \AA`
ω, Ω	`\o, \O`	ł, Ł	`\l, \L`	ß	`\ss`	¿	`?'`
¡	`!'`	†	`\dag`	‡	`\ddag`	§	`\S`
¶	`\P`	©	`\copyright`	£	`\pounds`		

22.6 ENVIRONMENTS

A LaTeX *Environment* usually begins with `\begin{name}` and ends with `\end{name}`. Indeed, since a LaTeX document begins and ends as `\begin{document}` ...`\end{document}`, the entire source file is in a *document* environment. Other environments include:

`array`	math arrays	`center`	centered text or tables
`description`	definition list	`document`	the full LaTeX document
`enumerate`	numbered and lettered list	`eqnarray`	aligned equations
`figure`	floating figure	`itemize`	bullet list
`list`	custom list	`minipage`	page within a page
`picture`	basic diagram drawn by LaTeX	`quotation`	long quotation
`quote`	short quotation	`tabbing`	define and use tabs
`table`	floating and referenced table	`tabular`	in-place table
`verbatim`	set as entered in \texttt	`\verb|text|`	in-line verbatim

22.7 LISTS

LaTeX supports three types of lists:

enumerate: for numbered lists;

itemize: for bullet lists;

description: for definition lists like this;

list: custom lists.

```
\LaTeX supports three types of lists:
\begin{description}
\item [enumerate] for numbered lists, \item [itemize ] for
bullet lists, \item [description ] for definition lists like
this. \item [list ] Custom lists
\end{description}
\end{enumerate}
```

The numbers or bullets are provided automatically by LaTeX , so all the user has to do is separate items by an `\item` command:

1. This is a numbered list, but you need not enter numbers.
2. This second item contains math $\sin^2 x + \cos^2 x = 1$.

```
\begin{enumerate}
\item This is a numbered list, but no numbers are entered.
\item This second item contains math $\sin^2 x+\cos^2 x =1$.
\end{enumerate}
```

Sublists are created just by starting another list within an existing list. LaTeX generates yet different symbols for sublists:

1. First item in list
   a. First item in sublist
   b. Second item in sublist
2. Second item in list

```
\begin{enumerate}
\item First item in list
    \begin{enumerate}
     \item First item in sublist
     \item Second item in sublist
     \end{enumerate}
\item Second item in list
\end{enumerate}
```

Text Tables

Tables are produced in the LaTeX text mode by entering the **tabular** environment. It is entered via `\begin{tabular}{...}`, and exited via `\end{tabular}`. The table is placed in the text at the location in which it is entered. If you want a table that is numbered, that has a caption, and that can be referenced, then you insert your `\tabular` commands between a set of `\table` commands. We describe that later.

Here is a table from one of our Java chapters:

Description	Data Types	Size	Examples
single character	char	2B	a, e, I, $, 6
integer	byte, short, int, long	1, 2, 4, 8 B	12, -30, -128, 127
floating point	float, double	4, 8 B	9.34F, 7.2867
logical	boolean ($\leq, \geq$)	1 bit	true or false

It is the result of:

```
\begin{tabular}{|l||c|c|r|}
\hline\hline \textbf{Description} &  \textbf{Data Types} &
\textbf{Size} &  \textbf{Examples} \\*[2ex] \hline
single character & \tell{char} & 2B & {a, e, I, \$, 6} \\
integer & \tell{byte, short, int, long} & {1, 2, 4, 8 B} & 12, -30, -128, 127\\
floating point & \tell{float, double} & {4, 8 B} & {9.34F, 7.2867}\\
logical & \tell{boolean} ($\leq,\geq$) & 1 bit&true or false\\
\hline
\end{tabular}
```

The argument in braces after the `\begin{tabular}` command specifies the format for the columns in the table. The `l` indicates that the first column is left-justified.

The two `c`'s indicates that the next two columns are centered, and the last `r` indicates that the fourth column is right-justified. The vertical bars | are optional and instruct LaTeX to place vertical bars in the columns indicated. The fact that we have bars before the first column and after the last means that we want bars on the outsides of the table.

Peer at how we have two `\hline` commands before we enter the data, a single `\hline` command after we enter the column headings, and a single `\hline` command after we have entered all the data. These commands cause the horizontal lines in the table to be drawn. They too are optional and must be placed *after* a row separator `\\` or before the rows begin.

Data are entered into the table one row at a time. Each row ends with a double backslash \\, and the columns are separated by an ampersand character `&`. Mathematical symbols or equations are placed in the table by switching to math mode, which we did with $\leq$ and $\geq$.

You do not have to specify the number of rows in your tables; LaTeX will know when you are done from the `\end{tabular}` command. In the present example we end the last row with a `\\`. Normally that is not necessary, but we include it because we have a `\hline` command to follow. If you need some more vertical space between two rows, include a space option such as `\\*[2ex]` to end a row. We did that after the heading.

Floating Tables

In published books and journals, tables and figures often "float" on the page. This means that they are placed where they fit in best, which is often not at the place where they are first mentioned. Clearly, this becomes more important as your table or figure gets larger and harder to fit in someplace. Table 22.1 is an example of a floating table. It is produced with the commands:

```
\begin{table}\caption{A sample floating table with a number and a caption.}
\begin{tabular}{|c|ccc|c|}
\hline\hline
Binary $abc$ & $=  a\times 2^2$ & $+ b\times2^1$ & $+c \times 2^0$ & = Decimal \\
\hline
000 & 0 & 0 & 0 & 0\\
001 & 0 & 0 & 1 & 1\\
010 & 0 & 1 & 0 & 2\\
111 & 1 & 1 & 1 & 7\\ \hline
\end{tabular}
\label{tab.binary2}
\end{table}
```

Table 22.1 A sample floating table with a number and a caption.

Binary abc	$= a \times 2^2$	$+b \times 2^1$	$+c \times 2^0$	= Decimal
000	0	0	0	0
001	0	0	1	1
010	0	1	0	2
111	1	1	1	7

Inspect how the table is made by inserting the standard `\begin{tabular}` and `\end{tabular}` pair within a `\begin{table}` and `\end{table}` pair. In addition, there are `\caption{}` and `\label{}` commands within the table environment but outside of the tabular environment. The argument to `\caption` contains the caption that appears under the table. (If the caption command appears before the tabular command, the caption will be above the table.) The argument to `\label` is a tag that is used to reference the table with the `\ref` command (the numbering is done automatically):

Table 22.1 is referenced `Table~\ref{tab.binary2} is referenced`

This same scheme of labeling and referencing is used with equations.

22.8 SECTIONS

and

Subsections

and

Subsubsections

LaTeX permits you to have sections, subsections, and subsubsections in your documents. You give each a title, and LaTeX gives each a sequential number, which changes automatically as other sections are inserted or removed, and a different sized heading. By way of example, the section, subsection, and subsubsection right above this paragraph were produced with the commands:

```
\section{Sections} and  \subsection{Subsections} and \subsubsection{Subsubsections}
```

Placing an asterisk before the title of the section or subsection will suppress LaTeX 's automatic numbering, as in `\section*{A Section without a Number}`.

Quotations and Footnotes

LaTeX contains two commands for setting off text from its surroundings. The **quote** environment is used for short quotations, while the **quotation** environment is appropriate for longer ones. They differ in indentation:

> This is a quote. Look at how this first paragraph is indented. The **quote** environment is used for short quotations.
>
> This is still the quote Observe how this second paragraph is indented. To repeat, a **quote** environment is for short quotations.

> This is a quotation. Observe how this first paragraph is indented. The **quotation** environment is appropriate for longer quotations.
>
> This is still the quotation. Check how this second paragraph is indented and spaced. A **quotation** environment is for longer quotations.

```
\begin{quote} ... \end{quote}

\begin{quotation} ... \end{quotation}.
```

Footnotes are created with the `\footnote{text}` command. You place the command where you want a reference to the footnote, and LaTeX automatically places a reference there and sets the note at the bottom of the page. For instance, we place a footnote here[2] with `\footnote{Sample note with its own period.}`. You should find it below and below a horizontal line.

[2] Sample note with its own period.

Chapter Twenty Three

LaTeX for Mathematics

23.1 ENTERING MATHEMATICS: MATH MODE

Typesetting mathematics properly is a challenge for humans and their computer counterparts. In order for mathematical symbols to convey their meanings properly, their size and vertical and horizontal placements must adhere exactly to accepted mathematical notation. LaTeX does a beautiful job at mathematical typesetting, better than any other program, in our opinion, and is almost as good as a human typesetter.

To input an equation or mathematical symbol you enter *math mode*, and then leave math mode when you want to enter text again. The equations or symbols may be embedded in text or displayed (set off) between lines of text. In-text mathematics, such as $R = \sqrt{N}$, is surrounded by dollar signs `$R = \sqrt{N}$`, the first one indicating a switch from text to math mode, the second one indicating a switch back from math to text:

R varies as $R \propto \sqrt{N}$ for `$R$ varies as $R \propto \sqrt{N}$ for`

In addition to dollar signs, you may also use `\(` and `\)` to mark the beginning and the end of a mathematical formula or symbol embedded in text:

R varies as $R \propto \sqrt{N}$ `\(R\) varies as \(R \propto \sqrt{N}\)`

Though probably just a matter of taste, we find it easier to read and edit a LaTeX source file when there are dollar signs in the text; to us, the `\(` and `\)` look too much like part of the equation.

Mathematics that is *displayed* (separated from the text) looks like:

$$\frac{-\hbar^2}{2m}\frac{d^2\psi(x)}{dx^2} + V(x)\psi(x) = E\psi(x). \tag{23.1}$$

Check that this equation has a number on the extreme right (automatically assigned by LaTeX) and a period for punctuation (put there because we prefer equations with

punctuation). This equation was obtained with the LaTeX input:

```
\begin{equation}
   {-\hbar^2 \over 2m} {d^2 \psi (x)\over dx^2} + V(x)\psi(x) = E \psi(x).
\end{equation}
```

The `\begin{equation}` and `\end{equation}` commands define the environment for numbered mathematical equations. If you do *not* want the equation to be numbered, that is, so it appears as:

$$\frac{-\hbar^2}{2m}\frac{d^2\psi(x)}{dx^2} + V(x)\psi(x) = E\psi(x),$$

then use `\[` and `\]` to begin and end the displayed equation environment:

```
\[
   {-\hbar^2 \over 2m} {d^2 \psi (x)\over dx^2} + V(x)\psi(x) = E \psi(x).
\]
```

23.2 MATHEMATICAL SYMBOLS AND GREEK

Most characters have their standard meaning in math mode. However, letters used as symbols appear in a special mode of italic, and function names and subscripts appear in Roman. The special characters, already discussed in §22.3, are entered in math mode as:

#	`\#`	$	`\$`	%	`\%`	&	`\&`	_	`\_`	{	`\{`	}	`\}`	\	`\backslash`

The spacings between math symbols will be different than the spacings in text mode and are designed to make the mathematics clearer. As always, LaTeX ignores extra spaces or carriage returns in the your source file, unless you force the issue. This means that you may insert extra spaces in your source file for clarity, or distribute your equations over several lines. LaTeX will put it all together and make it look nice. If you insist on inserting spaces (not worth the trouble unless you are publishing the document). For example, if you enter `$xy$`you get xy, otherwise:

$x\ y$	`x\ y`	normal space	$x\:y$	`x\:y`	medium space	$x\,y$	`x\,y`	thin space
$x\;y$	`x\;y`	thick space	$x\!y$	`x\!y`	back space	$x\ \ y$	`x\ \ y`	double space

Greek Letters

Greek letters are produced in math mode by preceding the full name of the letter by a backslash \. For example, $C = 2\pi r$ is input as `$C = 2 \pi r$`. The lowercase Greek letters are obtained as:

α	\alpha	β	\beta	γ	\gamma	δ	\delta	ϵ	\epsilon
ε	\varepsilon	ζ	\zeta	η	\eta	θ	\theta	ϑ	\vartheta
ι	\iota	κ	\kappa	λ	\lambda	μ	\mu	ν	\nu
ξ	\xi	o	o (text o)	π	\pi	ρ	\rho	ϱ	\varrho
σ	\sigma	ς	\varsigma	τ	\tau	υ	\upsilon	ϕ	\phi
φ	\varphi	χ	\chi	ψ	\psi	ω	\omega		

For those that are different from their Roman counterparts, uppercase Greek letters are obtained by capitalizing the first character of their names:

Γ	\Gamma	Δ	\Delta	Θ	\Theta	Λ	\Lambda	Ξ	\Xi	Π	\Pi
R	R (text R)	Σ	\Sigma	Υ	\Upsilon	Φ	\Phi	Ψ	\Psi	Ω	\Omega

Relations

$\leqq$	\leq	$\leq$	\le	$\geq$	\geq	$\geqq$	\ge	$\equiv$	\equiv
$\prec$	\prec	$\succ$	\succ	$\sim$	\sim	$\preceq$	\preceq	$\succeq$	\succeq
$\simeq$	\simeq	$\ll$	\ll	$\gg$	\gg	$\asymp$	\asymp	$\subset$	\subset
$\supset$	\supset	$\approx$	\approx	$\subseteq$	\subseteq	$\supseteq$	\supseteq	$\cong$	\cong
$\sqsubseteq$	\sqsubseteq	$\sqsupseteq$	\sqsupseteq	$\bowtie$	\bowtie	$\in$	\in	$\ni$	\ni
$\ni$	\owns	$\propto$	\propto	$\vdash$	\vdash	$\dashv$	\dashv	$\models$	\models
$\smile$	\smile	$\mid$	\mid	$\vert$	\vert	$\doteq$	\doteq	$\frown$	\frown
$\parallel$	\parallel	$\Vert$	\Vert	$\perp$	\perp	:	\colon	$\wedge$	\wedge
$\wedge$	\land	$\vee$	\vee	$\vee$	\lor	$\cdots$	\cdots	...	ldots

Negative Relations

Most negative relations are formed by including a \not command in front of the normal relation:

$\not\leq$ \not\leq $\not\equiv$ \not\equiv

This is just \not placed in front of \leq and \equiv. In addition, the symbol $\neq$ may be formed in a number of ways:

$\neq$ \neq \ne \not\equal

Binary (and Other) Operators

$\pm$	\pm	$\cap$	\cap	$\vee$	\vee	$\mp$	\mp	$\cup$	\cup
$\wedge$	\wedge	$\setminus$	\setminus	$\uplus$	\uplus	$\oplus$	\oplus	$\cdot$	\cdot
$\sqcap$	\sqcap	$\ominus$	\sqcap	$\times$	\times	$\sqcup$	\sqcup	$\otimes$	\otimes
$*$	\ast	$\triangleleft$	\triangleleft	$\oslash$	\oslash	$\star$	\star	$\triangleright$	\triangleright
$\odot$	\odot	$\diamond$	\diamond	$\wr$	\wr	$\dagger$	\dagger	$\circ$	\circ
$\bigcirc$	\bigcirc	$\ddagger$	\ddagger	$\bullet$	\bullet	$\bigtriangleup$	\bigtriangleup	$\amalg$	\amalg
$\div$	\div	$\bigtriangledown$	\bigtriangledown						

Arrows

$\leftarrow$	\leftarrow	$\rightarrow$	\rightarrow	$\longleftarrow$	\longleftarrow
$\longrightarrow$	\longrightarrow	$\Leftarrow$	\Leftarrow	$\Rightarrow$	\Rightarrow
$\Longleftarrow$	\Longleftarrow	$\Longrightarrow$	\Longrightarrow	$\leftrightarrow$	\leftrightarrow
$\Leftrightarrow$	\Leftrightarrow	$\swarrow$	\swarrow	$\longleftrightarrow$	\Longleftrightarrow
$\hookleftarrow$	\hookleftarrow	$\hookrightarrow$	\hookrightarrow	$\leftharpoonup$	\leftharpoonup
$\rightharpoonup$	\rightharpoonup	$\leftharpoondown$	\leftharpoondown	$\downarrow$	\downarrow
$\uparrow$	\uparrow	$\rightharpoondown$	\rightharpoondown	$\Uparrow$	\Uparrow
$\Downarrow$	\Downarrow	$\updownarrow$	\updownarrow	$\Updownarrow$	\Updownarrow
$\nearrow$	\nearrow	$\nwarrow$	\nwarrow	$\searrow$	\searrow
$\longleftrightarrow$	\longleftrightarrow	$\mapsto$	\mapsto	$\longmapsto$	\longmapsto

Math Parentheses

$[$	\lbrack	$]$	\rbrack	$\lfloor$	\lfloor	$\rfloor$	\rfloor	$\lceil$	\lceil	$\rceil$	\rceil
$\{$	\lbrace	$\}$	\rbrace	$\langle$	\langle	$\rangle$	\rangle	$\{$	\{ (alt)	$\}$	\} (alt)

Miscellaneous Math Symbols

$\aleph$	\aleph	$\prime$	\prime	$\forall$	\forall	$\hbar$	\hbar	$\emptyset$	\emptyset
$\exists$	\exists	$\imath$	\imath	∇	\nabla	$\neg$	\neg	$\jmath$	\jmath
$\surd$	\surd	$\bot$	\bot	ℓ	\ell	$\top$	\top	$\natural$	\natural
$\wp$	\wp	$\flat$	\flat	$\sharp$	\sharp	$\angle$	\angle	$\|$	|
$\triangle$	\triangle	Im	\Im	Re	\Re		\backslash	∂	\partial
	\clubsuit	$\heartsuit$	\heartsuit	$\spadesuit$	\spadesuit	$\diamondsuit$	\diamondsuit	∞	\infty

"Big" Operators

$\sum$	\sum	$\bigcap$	\bigcap		\bigodot	$\prod$	\prod	$\bigcup$	\bigcup
$\bigotimes$	\bigotimes	$\coprod$	\coprod	$\bigsqcup$	\bigsqcup	$\bigoplus$	\bigoplus	$\int$	\int
$\bigvee$	\bigvee	$\biguplus$	\biguplus	$\oint$	\oint	$\bigwedge$	\bigwedge		

23.3 MATH ACCENTS

$\underline{e}$	\underline{e}	$\overline{e}$	\overline{e}	$\hat{e}$	\hat{e}	$\check{e}$	\check{e}	$\tilde{e}$	\tilde{e}	$\acute{e}$	\acute{e}
$\grave{e}$	\grave{e}	$\dot{e}$	\dot{e}	$\ddot{e}$	\ddot{e}	$\breve{e}$	\breve{e}	$\bar{e}$	\bar{e}	$\vec{e}$	\vec{e}

23.4 SUPERSCRIPTS AND SUBSCRIPTS

A subscript is created by adding an underscore _ to a symbol and following that with the subscript, as in `I_x`. A superscript is created by adding a caret ^ to a symbol and following that with the superscript, as in `x^2`. And by doing both, a symbol may contain both a superscript and subscript:

$$I_x = m_1 x_1^2 + m_2 x_2^2$$

is obtained with either order of super- and subscripts:

`I_x = m_1 x_1^2 + m_2 x_2^2`, or `I_x = m_1 x^2_1 + m_2 x^2_2`.

Inasmuch as the sub- and superscripts here are above each other, the two ways of inputting are equivalent. If the sub- or superscript contains more than one character, then the characters must be placed together within braces:

$$I_{11} = m_a x_{a,1}^2 + m_b x_{b,1}^2$$

is obtained by with either order of super- and subscripts:

`I_{11} = m_a x_{a,1}^2 + m_b x_{b,1}^2` or `I_{11} = m_a x^2_{a,1} + m_b x^2_{b,1}`
.

23.5 CALCULUS AND SUMS

$\frac{dx}{dt}$	{dx\over dt}	$\frac{d^2x}{dt^2}$	{d^2 x\over dt^2}
$\frac{\partial u}{\partial t}$	{\partial u\over \partial t}	$\lim_{e\to 0}$	\lim_{e \to 0}
$\frac{\partial^2 \psi}{\partial x^2}$	{\partial^2\psi\over \partial x^2}	$\sum_{i=1}^{N} x^i/i!$	\sum_{i=1}^N x^i/i!
$\int_a^b f(x)\,dx$	\int_a^b f(x)\,dx	$\int\int f(z)\,dx\,dy$	\int\int f(z) \,dx\,dy

23.6 CHANGING MATH FONTS

We have seen that LaTeX sets mathematics in an italic font that is different from the italics of text mode. You obtain Roman and bold fonts within math mode with the commands `\mathrm` and `\mathbf`:

$\mathbf{S} = \mathbf{E} \times \mathbf{B}$ `\mathbf{S} = \mathbf{E} \times \mathbf{B}`

If you start running out of letters, why not try the uppercase calligraphic letters in math mode: `\,\cal{A}`:

$$\mathcal{A}\mathcal{B}\mathcal{C}\mathcal{D}\mathcal{E}\mathcal{F}\mathcal{G}\mathcal{H}\mathcal{I}\mathcal{J}\mathcal{K}\mathcal{L}\mathcal{M}\mathcal{N}\mathcal{O}\mathcal{P}\mathcal{Q}\mathcal{R}\mathcal{S}\mathcal{T}\mathcal{U}\mathcal{V}\mathcal{W}\mathcal{X}\mathcal{Y}\mathcal{Z}.$$

Sometimes you need to include text in an equation. You do that by making a box with `\mbox{text}` and placing the text in the box:

$x = 3,\ \text{for}\ y > 12$ `x = 3, \mbox{ for } y > 12`

In view of the fact that LaTeX removes single blank spaces in math mode but not in text mode, the two single spaces we placed within the box above will remain. Alternatively, we could have inserted spaces with `\` as `x = 3,\ \mbox{for}\ y > 12`.

23.7 MATH FUNCTIONS

The basic mathematical functions are obtained by entering a backlash `\` before the name. To cite an instance, $x^3 \sin y^2$ is obtained from `x^3 \sin y^2`. Look at how you need to leave a space after the function name as a separator, unless another backslash follows. Available functions include:

```
\arccos   \arcsin   \arctan    \arg      \cos     \cosh   \cot   \coth
  \csc      \deg      \det      \dim      \exp     \gcd    \hom    \inf
  \ker       \lg      \lim    \liminf   \limsup    \ln    \log    \max
  \min       \Pr      \sec      \sin      \sinh    \sup    \tan   \tanh
```

Other mathematical function names are obtained by switching to Roman math font, and then writing the function name; for example, $\mathrm{cosec}(A)$ as `$\mathrm{cosec}(A)$`.

23.8 FRACTIONS

To make a fraction with `num` over `denom`, use the `\frac{num}{denom}` command:

$v = \frac{dx}{dt}$ `v = \frac{dx}{dt}`

Even though this LaTeX command works fine, we prefer the simpler `\over` TeX command:

$v = \frac{dx}{dt}$ `v = {dx \over dt}`

23.9 ROOTS

The square root symbol $\sqrt{N}$ is obtained with the `\sqrt{N}` command. LaTeX automatically adjusts the height and lengths of root symbols to fit into the equation and to span the argument:

$\sqrt{x} = \frac{a \pm \sqrt{b^2 - 4ac}}{z}$ `\sqrt{x} = {a \pm \sqrt{b^2-4ac} \over z}`

Higher roots are obtained by including an optional order parameter in square brackets to `\sqrt[n]{}`:

$\sqrt[3]{x} = \frac{a + \sqrt[5]{b^2 - 4ac}}{z}$ `\sqrt[3]{x} = {a + \sqrt[5]{b^2-4ac} \over z}`

23.10 BRACKETS (DELIMITERS)

Except for braces, which are used for commands, most normal-sized brackets or delimiters are entered into mathematics as typed:

(( )) [[ ]] { \{ } \} | | ‖ \|

As an instance: $\{a\} = (b) * [c] + |d| \times \|e\|$ `\{a\} = (b)*[c] + |d| \times \|e\|`

Just as it does for roots, LaTeX scales the size of these delimiters to match the size of the rest of the equation. However, since an equation may have a number of nested brackets, LaTeX needs some help in deciding just which brackets you want to match. Thus, you give it help by tacking on the commands `\left` and `\right` to the delimiters that you wish to match up:

$$\left[\left| \frac{1}{\sqrt{z}} \right| = \{a + \aleph\} \right]$$

```
\left[ \left|{1\over\sqrt{z}} \right|= \left\{ a + \aleph \right\} \right].
```

In order for LaTeX to do all the work for you in sizing and placing these limiters, it demands that all the `\right`s be balanced by an equal number of `\left`s. LaTeX does not demand that the types match up, just the number of rights and lefts. So, for example, `\left(` may be paired off with `\right]`. Although this all makes

sense, it introduces a problem if you try to create something like

$$\int_a^b \frac{d^2 f}{dx^2} dx = \left. \frac{df}{dx} \right|_a^b,$$

where there is no delimiter to match the left |. For this purpose, LaTeX contains the *null delimiters* `\left.` and `\right.` that count as delimiters but do not show up when typeset. As an instance, the above formula was obtained with:

```
$\int_a^b {df^2\over dx^2} dx = \left. {df\over dx} \right|_a^b$
```

23.11 MULTILINE EQUATIONS

Having LaTeX arrange multiline equations is essentially the same as working with the *tabular* environment for text. You place each equation on a separate line (row), and then align the equal signs in each equation by placing them in the same column. To cite an instance, `Sample.tex` on the CD creates

$$\frac{-\hbar^2}{2m_1}\frac{d^2\psi_1(x)}{dx^2} + V(x)\psi_1(x) + U(x)\psi_2(x) = E\psi_1(x) \tag{23.2}$$

$$\frac{-\hbar^2}{2m_2}\frac{d^2\psi_2(x)}{dx^2} + V(x)\psi_2(x) + U(x)\psi_1(x) = E\psi_2(x) \tag{23.3}$$

```
\begin{eqnarray}
 {-\hbar^2 \over 2m_1} {d^2 \psi_1 (x) \over dx^2} + V(x)\psi_1(x) +
               U(x)\psi_2(x)   &=& E \psi_1(x)\\
{-\hbar^2 \over 2m_2} {d^2 \psi_2 (x) \over dx^2} +
V(x)\psi_2(x) +
               U(x)\psi_1(x)   &=& E \psi_2(x)
\end{eqnarray}
```

Observe here that we display these equations with the *equation-array* environment `eqnarray` rather than the *equation* environment that we used for single-line equations. We begin with `\begin{eqnarray}` and end with `\end{eqnarry}`. These commands produce numbered equations. If unnumbered, multiline equations are desired, they are produced with the `\begin{eqnarray*}` and `\end{eqnarry*}` forms. The equations are entered into an `eqnarray` environment in the same way as they would be with `{equation}`, except that the equal signs are enclosed by ampersands `&=&`, and each line of the equation ends with a double backslash `\\`. As also occurs in the tabular environment, this produces a column with all the equal signs aligned, and permits there to be as many lines with equations as you enter.

Because `eqnarray` is just a tabular environment, it is possible to align symbols other than the equal sign, or to use this environment to split a very long equa-

tion onto two lines, with the continuation being indented:

$$\begin{aligned} -\frac{d^2\psi_1(x)}{dx^2} - \frac{d^2\psi_2(x)}{dx^2} &+ V(x)\psi_1(x) + V(x)\psi_2(x) \\ &+ U(x)\psi_2(x) + U(x)\psi_1(x) \\ &= \quad E\psi_1(x) + E\psi_2(x) \end{aligned}$$

```
\begin{eqnarray*}
 - {d^2 \psi_1 (x)\over dx^2}  -  {d^2 \psi_2 (x)\over dx^2}
                            &+&  V(x)\psi_1(x) + V(x)\psi_2(x) \\
                            &+&  U(x)\psi_2(x)  + U(x)\psi_1(x) \\
= && E \psi_1(x) +E \psi_2(x)
\end{eqnarray*}
```

23.12 MATRICES AND MATH ARRAYS

The **array** environment is used to produce matrices and mathematical arrays:

$$\left(\begin{array}{ccc} x & x^2 & y \\ \sqrt{y} & x^y & z \\ z^a & z^{-1} & i \end{array} \right) \Rightarrow \left[\begin{array}{ccc} \alpha - \mu & \beta & \Gamma \\ \delta & \beta^3 & \alpha + \gamma \\ -x & y & \lambda \end{array} \right]$$

```
\left( \begin{array}{ccc}
         x & x^2 & y \\
         \sqrt{y} & x^y & z \\
          z^a & z^{-1} & i \end{array} \right)     \Rightarrow
          \left[ \begin{array}{ccc}
                 \alpha - \mu & \beta & \Gamma \\
                 \delta & \beta^3 & \alpha + \gamma \\
                  -x & y & \lambda
                  \end{array} \right]
```

This is essentially identical to how tables are constructed for text. All that is new is the use of large right and left delimiters to produce a matrix symbol that fits the array. The alignment characters `{ccc}` are the same as with tabular.

Another approach to matrices, which we personally find easier and more compact than the LaTeX standard, is the TeX command `\pmatrix`:

$$\begin{pmatrix} \frac{mb^2}{12} & 0 & 0 \\ 0 & \frac{ma^2}{12} & 0 \\ 0 & 0 & m\frac{a^2+b^2}{12} \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 4 & 0 \\ 0 & 0 & 5 \end{pmatrix}$$

```
\pmatrix{{mb^2\over 12} & 0 & 0 \cr
           0 & {ma^2\over 12} & 0 \cr
           0 & 0 & m{a^2+b^2\over 12}}
 = \pmatrix{1 & 0 & 0 \cr
```

```
0 & 4 & 0 \cr
0 & 0 & 5}
```

We see that `pmatrix` takes care of its own delimiters and uses the carriage return command `\cr` to end the rows.

The array environment is also used to produce an equation containing cases:

$$\theta(x) = \begin{cases} 1, & \text{if } x \geq 0, \\ 0, & \text{if } x < 0. \end{cases}$$

```
\theta(x) = \left\{ \begin{array}{ll}
                      1 & \mbox{if $x \geq 0$};\\
                      0 & \mbox{if $x < 0$}.\end{array} \right.
```

Here too, our preference is the TEX command `\cases`, which we find simpler:

$$\theta(x) = \begin{cases} 1, & \text{if } x \geq 0, \\ 0, & \text{if } x < 0. \end{cases}$$

```
\theta(x) = \cases{1, & if $x \geq 0$, \cr
                   0, & if $x < 0$.    \cr}
```

23.13 INCLUDING GRAPHICS

Most drawing and graphing programs have a number of options to determine the file format in which to save graphics. Because LATEX is used to produce publication quality typesetting, if you include graphics, then they too should be publication quality. PostScript (`.ps`) and Encapsulated PostScript (`.eps`) figures are of publication quality, so it should not be a surprise that there are nice packages to include these formats in LATEX documents.

We will discuss how to include `.eps` figures, with the same commands used for `.ps` figures. Given a choice, use encapsulated figures, since it may be easier to adjust their size to fit the document. If your figure is of a different type, you should be able to open it in a drawing program and save it as an `.eps` figure. In addition, we recommend that you embed the fonts used in your figure with the figure; this does make for a larger file, but also avoids the possibility of some inappropriate local font being substituted when the document is printed (especially likely if you scale the figure).

We recommend the *graphics* package for including figures, although we have also had success with the *psfig* package, especially for placing figures at desired points on the page. If a package is not part of your LATEX installation, it

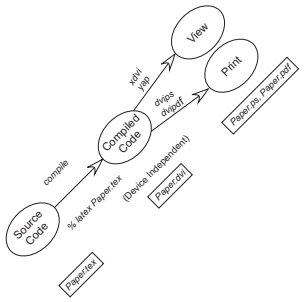


Figure 23.1 A scaled-down and rotated Figure 9.1.

may be added. Before you use a package, you must include it in your document's preamble (the part before the `\begin{document}`command):

```
\usepackage[dvips]{graphicx, color}
```

On the CD is the sample graphics file `Latex_Compile.eps` used to create Figure 22.1. It is redrawn on this page as Figure 23.1 with the commands

```
\begin{figure}
\begin{center}
\includegraphics[angle=45, scale=0.35]{Latex_Compile.eps}
\caption{A scaled-down and rotated
Figure~\ref{fortran_compile.eps}. \label{sample.fig}}
\end{center}
\end{figure}
```

The `figure` environment is standard LaTeX. It is also used to draw pictures with LaTeX, or to leave room for and label figures to be pasted in later (use `\vspace` to make the room). The `\includegraphics` command is placed within the figure environment. To ensure that the figure is centered on the page, we place the figure in a `center` environment. The `angle=45, scale=0.25` specification in square brackets is optional and leads to a figure that is slanted and scaled to 25% of its original size.[1] The name of the figure `Latex_Compile.eps` is given in curly braces, with a path indication if it is not in current directory. Scrutinize how after the name of the figure we issue the `\caption` command containing the caption as its argument. Within the caption's argument (still within the curly braces) we have placed the label command `\label{sample.fig}`. We then refer to this figure as `\ref{sample.fig}`, as we do in the sample `\includegraphics` command above, and let LaTeX take care of its actual number.

[1] We are able to scale and rotate with no loss in quality because PostScript figures are not bit maps, but instead contain a text description of the figure and the fonts.

As an example of \psfig, examine Figure 15.1 (p. 290) with its multiple parts:

```
\begin{figure}
\psfig{file=FIGS/planet.eps, height=1.85in} \hspace{6ex}
\psfig{file=FIGS/PlanetApplet.eps, width=1.5in}
\caption{\emph{Left: } The gravitational ....)}
\label{planet.eps}
\end{figure}
```

23.14 EXERCISE: PUTTING IT ALL TOGETHER

1. Enter Sample.tex as given in §22.3. Use a text editor and save the file as Sample.tex. We recommend *WinEdt* and *MikTex* for Windows, and *gnu xemacs* for Unix/Linux.

2. Compile the file by pressing the *LaTeX* button or by issuing a command from a shell:

```
> latex Sample.tex
```
Compile latex source Sample.tex

If there are no errors in your file, you should get a fairly verbose response with statements of the sort

```
This is TeX, Version 3.14159 (MiKTeX 2 UP 1) 2 NOV 2003
10:44 1478 **Sample.tex (Sample.tex LaTeX2e <2000/06/01>
Babel <v3.6Z> ... C:\Program
Files\MiKTeX\tex\latex\base\article.cls Document Class:
article No file Sample.aux. LaTeX Warning: Reference 'eq.1'
on page 1 undefined on input line 37. LaTeX Warning:
Reference 'eq.2' on page 1 undefined on input line 37.
 [1] (sample.aux)
Output written on sample.dvi (1 page, 2676 bytes).
```

3. The warnings that there is No file Sample.aux and that Reference 'eq.1' ('eq.2') undefined are to be expected on first compilation. The Sample.aux file contains the information LaTeX needs to cross-reference the source Sample.tex. Yet LaTeX does not write it until the end of compilation, and, consequently, it does not yet know that you have defined references eq.1 and eq.2.

4. Compile the source file again. This time the compiler should find a Sample.aux file to use for cross-references and not warn you about undefined references.

5. The last part of LaTeX's message tells you that the files sample.aux and sample.dvi were written, and that the typeset document is one page long. If you have changed some labels or references, then LaTeX may request that you compile the source file again in order to incorporate the updated .aux file.

6. If there were no errors, your working directory should now contain the files

`Sample.tex`	source file
`sample.aux`	auxiliary working file with reference numbers and such
`sample.log`	log of LaTeX messages; similar to messages flashed on screen
`sample.dvi`	device-independent, viewable version of typeset document

7. If you are lucky enough to get a `sample.dvi` file, then look at it to admire your work. On Unix you do this with:

```
> xdvi sample.dvi                                    Preview .dvi file
```

On Windows use *Yap* (yet another previewer) or the DVI button.

8. If you get complaints that LaTeX was not happy with the way a line fits on the page, just ignore them unless you are really going to publish the paper. If there are real errors, correct them one at a time and proceed. If you cannot figure out where the error is, try forcing LaTeX to skip it by continually entering `Enter`. Alternatively, you isolate the location of an error by moving the `%\end{document}` up in your document until LaTeX compiles with no errors. Then move it down gradually, recompiling at each stage until the error is uncovered.

9. To see how LaTeX responds to errors, make these mistakes:

Correct	Change to incorrect
`\item`	`\iten`
`$N=6$`	`$N=6`

The first error should illicit the response

```
 ! Undefined control sequence.
l.19     \iten
               Use this sequence to simulate a random walk and
! LaTeX Error: Something's wrong--perhaps a missing \item.
```

This error message indicates with a carriage return where LaTeX notices something is wrong, in this case, that it has never heard of `\iten`. Below that, LaTeX indicates that it thinks you meant to write `\item`. In the second case where a `$` is left off, LaTeX responds with

```
! LaTeX Error: Bad math environment delimiter. l.17 \[
        3, 7, 3, 1, 8, 2 \]
```

We see that LaTeX knows something is wrong in the math environment (we left off the second `$`, but it does not know it until the *next* math environment begins with a `\[` on line 17. Consequently, you need to look at line 17 in the source file, and work back from there to see where there is a missing math delimiter.

10. Once you have produced a complete `.dvi` file, you convert it into the PostScript file `sample.ps` for printing with the `dvips` command:

```
> dvips -o sample.ps sample.dvi        Convert .dvi file to .ps file
> dvips -o sample.ps sample            Also converts .dvi file to .ps file
```

Here the `-o` option places the output into the file whose name follows. If you leave off the `-o` option, then the `.ps` file might well be sent directly to some printer. In `WinEdt`, the `dvips` command is issued by pushing the *dvi* $\hookrightarrow$ *ps* button. You should get an output statement showing you all the page numbers that have been converted, and indicating that the output is placed in the file `sample.ps`. There is also the *pslatex* version of LaTeX that uses PostScript fonts in place of LaTeX's standard *Computer Modern* fonts. This does produce a smaller `.ps` file in the end.

11. If you want a `.pdf` file, you may issue the `pdfLATEX` command in the first place, or use the `dvipdf` command to convert your `.div` file. However, if the `.eps` figures in your file do not get placed in the `.pdf` file properly, we recommend the use of Adobe *Distiller* to convert the entire `.ps` file to a `.pdf` one.

12. If your document does not contain any embedded PostScript figures, then you may be able to print it successfully from the `.dvi` previewer. Otherwise, we have found it best to create a `.ps file` of your document and to print that.

13. Now print your `sample.ps` file. You do that from a *GhostScript* viewer, such as *GSview*, or with Unix printer commands:

```
> lpr -Pps497 sample.ps                Print to printer ps497
> lp -d  ps497 sample.ps               Print to device (printer) ps497
```

14. Now let us go back and add some more features to `Sample.tex`:

a. Include the sample PostScript figure `Latex_Compile.eps` from the disk or Web into your document.

b. Include Table 22.1 into your document (the commands are all given in the text). First keep the table in one place by using just the `tabular` environment, and then let it float by placing the `tabular` environment within a `table` environment.

c. Include a line in your document that uses the `\ref` command to refer to the floating table by its label.

d. Create a title page by placing your personalized versions of these commands into the preamble:

```
\title{My Title}
\author{My Name \and My Friend\\ Our Institution}
\date{Someday}
\thanks{To those who have supported this noble effort with cash.}
```

After placing these commands in the preamble, you need to create the title in the body of the document with:

```
\maketitle
```

Appendix A

Glossary

absolute value Value of a quantity expressed as a positive number, *e.g.*, $|f(x)|$.

accuracy The degree of exactness provided by a description or a theory. *Accuracy* usually refers to an absolute quality, while *precision* usually refers to the number of digits used to represent a number.

address The numerical designation of a location in memory. An identifier, such as a label, that points to an address in memory or a data source.

algorithm A set of rules for solving a problem in a finite number of steps. Usually independent of the software or hardware.

allocate To assign a resource for use, often memory.

alphanumeric The combination of alphabetic letters, numerical digits, and special characters, such as %, $, and /.

analog The mapping of a continuous physical observable to numbers. As an instance, a car's speed to its speedometer.

animation A process in which motion is simulated by presenting a series of slightly different pictures (frames) in succession.

append To add on, especially to the end of an object or word.

application A self-contained executable program containing tasks to be performed by a computer, usually for a practical purpose.

architecture The overall design of a computer in terms of its major components: memory, processor, I/O, and communication.

archive To copy programs and data to an auxiliary medium or file system for long-term and compact storage.

argument A parameter passed from one program part to another, or to a command.

arithmetic unit Part of the central processing unit that performs arithmetic.

array (matrix) A group of numbers stored together in rows and columns that may be referenced by one or more subscripts. Each number in an array is an array element.

assignment statement Command that sets a value to a variable or symbol.

B Abbreviation for byte (8 bits).

b Abbreviation for bit or baud (1 bit/sec).

background (1) A technique of having a programming run at low priority ("in background") while a higher-priority program runs "in foreground." (2) The part of video display not containing windows.

base The radix of a number system. (10 is the radix of the decimal system.)

basic machine language Instructions telling the hardware to do basic operations such as store or add binary numbers.

batch The running of programs without user interaction; often in background.

baud 1 bit per second.

binary Related to the number system with base 2.

BIOS Basic Input/Output System.

bit Contraction of "binary digit"; the digits 0 or 1 used in binary representation.

Boolean algebra A branch of symbolic logic dealing with logical relations as opposed to numerical values.

boot To "bootstrap"; to start a computer by loading the operating system.

branch To pick a path within a program based on the value of variables.

bug A mistake in a computer program or operating system; a malfunction.

bus A communication channel (bunch of wires) used for transmitting information quickly among computer parts.

byte Eight bits of storage. Java uses two bytes to store a single character in extended unicode.

byte code Compiled code that is read by all computer systems, but still needs to be interpreted (or recompiled) on each system. Contained in class file.

cache Small, very fast part of memory used as temporary storage between the very fast CPU registers and main memory.

calling sequence The data and setup needed to call a method or a subprogram.

central processing unit (CPU) The part of a computer that accepts and acts on instructions; where calculations are done and communications controlled.

checkpoint A statement within a program that stops normal execution and provides some output to assist in debugging.

checksum The summation of digits or bits used to check the integrity of data.

child Object created by presently existing parent object.

class A group of objects or methods having a common characteristic. Collection of data types and associated methods. An instance of an object. Byte code version of a Java program.

clock Electronics that generate the periodic signal to begin execution.

code A program or the writing of a program.

column The vertical line of numbers in an array.

column-major order The method used by Fortran to store matrices in which the leftmost subscript varies most rapidly and attains its maximum value before the subscript to the right is incremented. (Java uses row-major order.)

command A computer instruction. A control signal.

command key A keyboard key, or combination of keys, that performs a predefined function.

compilation Translation of a program written in a high-level language into (more) basic machine language.

compiler A program that translates source code from a high-level computer language to machine language or object code.

concatenate To join together two or more strings, head to tail.

concurrent processing Same as parallel processing; simultaneous execution of several related instructions.

conditional statement Statement to be executed only under certain conditions.

control character A character that modifies or controls the running of a program (e.g., *control +c*).

control statement A statement within a program that transfers control to another section of the program.

copy To transfer data *without* removing the original.

CPU See central processing unit.

crash The abnormal termination of a program or a piece of hardware due to some malfunction.

cycle time (clock speed) Time it takes CPU to execute simplest instruction.

data Information stored in numerical form; plural of datum.

data type Definitions that permits proper interpretation of character string.

debug To detect, locate, and remove mistakes in a program or hardware.

default The assumption made when no specific directive is given.

delete To remove and leave no record.

digital Representation of quantities in discrete form; contrast analog.

dimension of array Number of subscripts needed to access single array element.

directory A collection of files given their own name.

disc, disk A circular magnetic medium used for storage.

discrete Related to distinct elements.

double precision Use of two memory words to store a number.

download To transfer data *from* a remote computer *to* a local computer.

driver A set of instructions needed to transmit data to/from external device.

dump Data resulting from listing all information in memory.

dynamic RAM Computer memory that must be refreshed at frequent intervals.

E A symbol for exponent. To illustrate, `1.97E2` $= 1.97 \times 10^2$.

element An item of data within an array; a component of a language.

enable To make a computer part operative.

ethernet A high-speed local area network (LAN) composed of specific cable technology and communication protocols.

executable program A set of instructions that can be loaded into the computer's memory and executed.

executable statement A statement that causes some computational action, such as assigning a value to a variable.

fetch To locate and retrieve information from storage.

floating point Representation of numbers as mantissa times base raised to a power. Scientific notation.

FLOP Floating Point Operation.

foreground Running high-priority programs before low.

Fortran An acronym for **for**mula **tran**slation.

fragmentation File storage in many small, dispersed pieces.

G Abbreviation for giga; one billion in USA; 10^9.

garbage Meaningless numbers, usually the result of error or improper definition. Obsolete data in memory waiting to be removed ("collected").

giga Prefix indicating one billion, 10^9, of something (USA).

GUI Graphical user interface; a window environment.

hard disk A circular, spinning, storage device using magnetic memory.

hardware Physical components of a computer system.

hashing A transformation that converts keystrokes to data values.

heuristic A trial-and-error approach to problem solving.

hexadecimal Base 16; {0,1,2,3,4,5,6,7,8,9,A,B,C,D,E,F}.

hidden line surface Part of a graphics object normally hidden from view.

high-level language Programming language similar to normal language.

host computer A central computer providing services to terminals.

icon Small on-screen symbol that activates an application.

increment The amount added to a variable, especially an array index.

index The symbol used to locate a variable in an array, the subscript. A reference table kept in memory.

infinite loop The endless repeating of a set of instructions.

input Introduction of data from an external device into main storage.

instructions Orders to the hardware to do basic things such as fetch and add.

instruction stack Group of instructions currently in use.

interpolation Finding values between known values.

interpreter A language translator that converts each line of source code into machine code and immediately executes each line.

interrupt A command that stops execution of a program when some abnormal condition is encountered.

iterate To repeat a series of steps automatically.

jump A departure from the linear processing of code; branch, transfer.

just-in-time compiler A procedure that takes a Java class file that would ordinarily be interpreted and recompiles it into a more efficient machine code.

K Abbreviation for KILO, one thousand, 10^3.

kernel The central part of a large program or operating system that does not get significantly modified when run on different computers.

kill To delete or stop a process.

language Rules, representations, and conventions used to convey information.

library (lib) A collection of programs or methods usually on a related topic.

linking The connecting of separate pieces of code to form an executable program.

literal A symbol that defines itself, such as the letter *A*.

load To read information into the computer's memory.

load module A program that is loaded into memory and run immediately.

log in (on) To sign onto the computer, to begin a session.

loop A set of instructions executed repeatedly as long as some condition is met.

low-level language Machine-related programming not easy for humans to read.

machine language The set of instructions understood by the computer hardware.

machine precision The maximum positive number that, when added to the number stored as 1, does not change it.

macro A single, higher-level statement resulting in several lower-level ones.

main method Part of application program where execution begins; may call other methods but cannot be called by them.

main storage The fast, electronic memory; physical memory.

mantissa The significant digits in a floating-point number; e.g., `1.2` in `1.2E3`.

mega, M A prefix denoting a million, or $1,048,576 = 2^{20}$.

metalanguage A language used to define other languages.

method A subroutine used to calculate a function or manipulate data.

modular programming The technique of writing program with many, reusable, and independent parts.

modulo (mod) Function that yields only remainder after division of numbers.

multiprocessors Computers with more than one processor.

multitasking The system by which several jobs reside in a computer's memory simultaneously; may run in parallel or sequentially.

nesting Embedding a group of statements within another group.

object A software component with properties like physical objects. A combination of data (variables, properties) and methods (behaviors) to interact with the data; and abstract data type containing multiple parts.

object-oriented programming A modular programming style focused on classes of data objects and associated methods to interact with the objects.

object program (code) A program in basic machine language produced by compiling a high-level language.

octal Base 8; easy to convert to or from binary.

operating system (OS) The program that controls the computer and runs applications, processes I/O, and shells.

optimization The modification of a program to make it run more quickly.

overflow A number that is larger than the largest number a computer can store accurately.

package A collection of related programs or classes.

page A segment of disk memory that gets read into central memory in one block.

parallel (concurrent) processing Simultaneous or independent processing in different CPUs.

parallelization Rewriting an existing program to run on a parallel computer.

partition The section of memory assigned to a program during its execution.

physical memory The fast, electronic memory of a computer; main memory; contrast to *virtual memory*.

physical record The physical unit of data for input or output that may contain a number of logical records.

pipeline (segmented) arithmetic units Assembly-line approach to central processing in which CPU simultaneously gathers, stores, and processed data.

pixel A picture element, a dot on the screen. See also voxel.

portable document format, pdf A document format developed by Adobe that is of high quality and still readable within a Web browser.

PostScript, ps A standard language developed by Adobe for sending text and graphics to printers.

precision The degree of exactness with which a quantity is presented. High-precision numbers are not necessarily *accurate*.

program A set of instructions that a computer interprets and executes.

protocol A set of rules or conventions.

pseudocode A mixture of normal language and coding that provides a symbolic guide to a program.

queue An ordered group of items waiting to be acted upon in turn.

radix The base number in a number system that gets raised to powers.

RAM Random access (central) memory that is reached directly.

random access Reading or writing memory independent of storage order.

record A collection of data items treated as a unit.

recurrence (recursion) The use of a loop to produce new values of a variable computed in previous iterations.

registers Very high-speed memory used by the central processing unit.

reserved words Words that cannot be used in an application program.

RISC Reduced Instruction Set Computer; a CPU design that increases arithmetic speed by decreasing the number of instructions the CPU must follow.

row-major order The method used by Java to store matrices in which the right-most subscript varies most rapidly and attains its maximum value before the subscript to the left is incremented.

run To execute a program.

scalar A data value or number, for example, π.

serial/scalar processing Calculations in which numbers are processed in sequence. Contrast to vector and parallel processing.

shell The command line interpreter; the part of the operating system in which the user enters commands.

simulation The modeling of a real system by a computer program. The use of one system to represent or model another one.

single precision The use of one computer word to store a variable.

software Programs or instructions.

source code Program in high-level language needing compilation to run.

stochastic A process in which there is an element of chance.

string A connected sequence of characters treated as a single object.

structure The organization or arrangement of a program or a computer.

subprogram The part of a program invoked by another program unit, a *method*.

supercomputer The class of fastest and most computers available.

syntax The rules governing the structure of a language.

telnet Protocols for computer–computer communications.

tera, T 10^{12}, or $2^{30} = 1,073,741,824$.

top-down programming Designing a program from the most general view of the problem, down to the specific subroutines.

unary An operation that uses only one operand; monadic.

underflow A result that is smaller than the smallest number that a computer stores properly.

unit A device having a special function.

upload To transfer data *from* a local *to* a remote computer; opposite of download.

utility programs Programs to enhance other programs or do chores.

vector A group of N numbers in memory arranged in one-dimensional order.

vector processing Calculations in which an entire vector of numbers is processed with one operation.

virtual memory Memory on the slow, hard disk and not in fast RAM.

visualization The production of two- and three-dimensional pictures or graphs of the numerical results of computations.

volume A physical unit of a storage medium, such as a disk.

word A unit of main storage, usually 1, 2, 4, 6, or 8 bytes.

word length The amount of memory used to store a computer word.

Appendix B

Fortran Quick Reference

Compilation & Interpretation of Application

To compile your Fortran file, consult the documentation on your specific compiler.

Application Program Structure

```
!Comments begin with the exclamation mark, '!'.
!After '!', the compiler ignores the remainder of the line.

!Declare an external procedure to add two integers:
Integer Function c_add(x,y)  !Declare the function type
  Integer :: x,y             !Declare dummy variables
  c_add = x+y                !Perform function operations
End Function c_add           !Properly end EACH procedure

!Declare the main program:
Program Hello
  !Create an interface to use the 'c_add' function
  Interface
    Contains
    Integer Function c_add(x,y)
      Integer :: x,y
    End Function c_add
  End Interface
  Implicit None     ! Requires ALL variables to be declared
  Integer :: i=1,j=2  !Declare with initial values
  Real*8 :: b,c
  !Assign any other needed initial values
  b = 1.57
  c = sin(b)      !Intrinsics require no other references
  !Writing to the screen with the default '*'
  write(*,*) 'sin(b)= ',c   !Display string and value of c
  write(*,*) 'i+j= ',c_add(i,j)   !invoke function c_add
End Program Hello
```

Data (Variable) Types

Description	Type	Size/Format
Integer	`integer`	1B (=8b) Byte, 2B Short, 4B integer, 8B long
Floating Point	`real, complex`	Single or Double precision (*4, or *8)
Character (String)	`character(len=<len>)`	16-bit (2B)*len
Logical	`logical`	1 bit, true or false

Sample Data Representations

Representation	Meaning	Representation	Meaning
`i = 10`	integer	`i = 3.1e0`	Scientific
`i = 10.`	decimal (real*4)	`i = 10.0_8; 10.0d0`	real*8

Naming Convention

1. No more than 31 characters.
2. A letter must be the first character, any remaining characters must be either a letter, a number, or the underscore.
3. Names are *not* case-sensitive.
4. Fortran does not have specified reserved words, that is, words that are restricted from use as variable names.

Array References

`integer :: i(10)`	Declare & allocate memory, integer array
`real :: x(10)`	Declare & allocate memory, real array
`real*8 :: y(10,15)`	Declare a double-precision 2D array
`b_array(i,j) = 45.`	Assign a value to array element b_array(i,j)
`h=size(b_array,DIM=2)`	Extract length of the 2nd dimension of b_array

Arithmetic Operators

Operator	Example	Description
`+`	`x + y`	Add `x` and `y`
`-`	`x - y`	Subtract `y` from `x`
`*`	`x * y`	Multiply `x` by `y`
`/`	`x / y`	Divide `x` by `y`
`**`	`x**3`	raise `x` to power 3

Relational Operators

Operator	Example	Return true if
`>`	`x > y`	`x` is greater than `y`
`>=`	`x >= y`	`x` is greater than or equal to `y`
`<`	`x < y`	`x` is less than `y`
`<=`	`x <= y`	`x` is less than or equal to `y`
`==`	`x == y`	`x` and `y` are same *object*
`/=`	`x /= y`	`x` and `y` are not equal

Logical Operators

Operator	Example	Name: Return true if
`.AND.`	`x .AND. y`	**Logical and:** `x` and `y` both true, conditionally evaluates `y`
`.OR.`	`x .OR. y`	**Logical or:** either `x` or `y` true, conditionally evaluates `y`
`.NOT.`	`.NOT. x`	**Not:** `x` is false

Order of Precedence

1. left to right 2. RHS, then LHS 3. parentheses 4. **
5. `*` or `/` 6. `+` or `-` 7. `=`

Mathematical Function Library [Use: name(argument)]

sin	cos	tan	asin	acos	atan	achar
exp	log	iachar	sqrt	random	abs	max
min	ceil	floor	nint	mod	log	ln

Flow Control

`x = x + 1`	line to be evaluated, *etc.*
if `( (x < 3) .AND. (y == 12) )` **then** `z = y * x`	Evaluate once if true
if `( x <= 0. )` **then** `y = y * y` **else** `y = 2 * y`	Can have one or more lines Only one `else` permitted (catchall)
if `( score >= 90 )` **then** `grade = 'A'` **else if** `( score >= 80 )` **then** `grade = 'B'` **else if** `( score >= 70 )` **then** `grade = 'C'`	The "if" condition Any number of `else if`'s OK Inaccessible if earlier `else` satisfied
select `(month)` **case** `(1) s = 'Jan'` **case** `(12) s = 'Dec'` **case default** `s = 'undefined'` **end select**	 Can have as many cases as needed Usually need a default case End the select statement

do `i=0,n,1`
`<statement block>`
end do
	(initial value, max value, increment)
	Code executed repetitively
	Brackets the end of the loop

do while `( <boolean> )`
`<statement block>`
end do
	Goes through if true
	Code executed in a loop
	Brackets the end of loop

Input and Output, Screen & Keyboard

`write(*,*) 'count = ', j`	Screen output
`print *, 'value is ', b`	Screen output
`read(*,*) c`	Read from keyboard, store in variable c
`read(*,*) r`	Read from keyboard, store in variable r
`read(*,20) x`	Read keyboard (using format 20), store in x

Input and Output, Files

`open(UNIT=1,'filename',status='REPLACE')`	nNew unit for 'filename' with stated status
`read(1,40) s`	Read from Unit 1 using Format 40, store in s
`40 format(F12.4)`	Provide any number of format specifications
`write(1,*) s`	Write value for s to Unit 1 using free-formatting
`close(Unit=1,status='KEEP')`	Closes 1 and retain

B.1 TRANSFERRING FILES FROM THE CD

The Maple and Fortran worksheets and programs given on the CD do not need any special installation. You just need to pick a directory/folder on your computer where you want them stored, and copy the appropriate folder from the disk into that directory. Of course you will need to insert the CD into the CD drive, and open that drive to see the contents of the CD. (On Windows computers you get to that drive by clicking on the My Computer icon.)

B.2 USING OUR FORTRAN PROGRAMS

The Java programs on the CD are almost all source (`.java`) files and therefore need to be compiled before they will execute. There are a number of ways in which you may work with and execute our Java programs. The least painful is probably to use a complete programming workbench like *Code Warrior* or the *Forte Suite*. With these you need only punch some buttons to compile, debug, and execute programs. We view these programming environments as powerful tools that will increase the productivity of professional programmers. However, we believe that the more basic approach is preferable for the purpose of this introductory book.

The basic approach to Java programming that we prefer is to use an editor to modify the Java source file, *save the file*, and then execute the resulting class file from the prompt (command line) of a shell. In this way the beginning programmer learns that there are just two basic types of files to deal with, and that they are universal for all operating systems. In contrast, the programming environments often produce an assortment of files containing formatting and other such information, with the user not sure just what is truly needed to write and run a program.

On Windows computers you may use a text editor such as *Notepad* to modify Java source file. You should not, in contrast, use a word processor such as *MS Word*, since it formats text by inserting control characters that are invisible to you but stop the Java compiler from getting its job done. We prefer using the *WinEdt* [WinEdt] editor on Windows, which also provides immediate access to a shell in the same directory as the source (it is also excellent for LaTeX).

On Unix systems we recommend the free *gnu emacs* (especially *xemacs*) editor [Gnu]. It is powerful and permits compilation and execution from within the editor. It is excellent too with LaTeX. In addition, the *jEdit* Java editor is an excellent and free Java editor that is written in Java. Thus, it runs identically on all operating systems and provides some very useful color coding of codes.

Once you have your `.java` file open in an editor, you need a shell or command line from which to issue the `javac` and `java` commands. On Unix this is easy if you first change to the appropriate directory and open your editor from there. It is equally easy on Windows computers if you open a shell from your editor. However, if you use the Start > Run > Open > Command route, then you will have to navigate the Window's file directory structure with the `dir` and `cd` commands.

Bibliography

MAPLE AND MATHEMATICA

[Corl 02] CORLESS, R. M. (2002), *Essential Maple 7, An Introduction for Scientific Programmers,* Springer, New York.

[Gass 98] GASS, R. (1998), *Mathematica for Scientists and Engineers*, Prentice-Hall, Upper Saddle River, NJ.

[Hass 03] HASSANI, H. (2003), *Mathematical Methods Using Mathematica*, Springer, New York.

[Heck 03] HECK, A. (2003), *Introduction to Maple, Third Ed.*, Springer, New York.

[Meissner 95] MEISSNER, L. P. (1995), *Maple, Fortran 90,* PWS Publishers, Boston.

[N&W 96] NICOLAIDES, R., AND N. WALKINGTON (1996), *Maple, A Comprehensive Introduction,* Cambridge University Press, Cambridge.

[R&M 01] ROMANO, A., AND A. MARASCO (1996), *Mathematica Navigator, Second Ed.*, Elsevier/Academic Press, New York.

[Rusk 04] RUSKEEPAA, H. (2001), *Scientific Computing with Mathematica*, (ODEs) Birkhäuser, Boston.

[Wolf 99] WOLFRAM, S. (1999), *The Mathematica Book,* Wolfram Media, Champaign, and Cambridge University Press, Cambridge.

[Zimm 02] ZIMMERMAN, R. L., AND F. I. OLNESS, (2002), *Mathematica for Physicists*, Addison-Wesley, San Francisco.

JAVA AND FORTRAN

[Bron 03] BRONSON, G. J. (2003), *Java for Engineers and Scientists,* Thompson, Brooks/Cole, Toronto.

[Chap 04] CHAPMAN, S. J. (2004), *Java for Engineers and Scientists*, Second Ed., Pearson, Prentice Hall, Upper Saddle River, NJ.

[ChapF 04] CHAPMAN, S. J. (2004), *Fortran 90/95 for Engineers and Scientists*, Second Ed., McGraw-Hill, New York.

[Dav 99] DAVIES, R. (1999), *Introductory Java for Scientists and Engineers*, Addison-Wesley, Harlow, UK.

[Eck 02] ECK, D. (2002), *Introduction to Programming Using Java, Version 4.0*, (a free textbook), http://math.hws.edu/javanotes.

[Flan 97] FLANAGA, D. (1997), *Java in a Nutshell, Desktop Quick Reference*, Second Ed., O'Reilly, Sebastopol, CA.

[F90_CSEP] THE COMPUTATIONAL SCIENCE EDUCATION PROJECT, *Fortran 90 and Computational Science*, http://csep1.phy.ornl.gov/pl/pl.html.

[F90_UL] MARSHALL, A. C. *The University of Liverpool Fortran 90 Course Notes*, http://www.liv.ac.uk/HPC/F90page.html.

[Smit 91] SMITH, D. N. (1991), *Concepts of Object-Oriented Programming*, McGraw-Hill, New York.

COMPUTATIONAL SCIENCES

[CP 05] LANDAU, R. H., AND M. J. PÁEZ (1997), *Computational Physics, Problem Solving with Computers*, Wiley, New York;
LANDAU, R. H., M. J. PÁEZ, AND C. C. BORDEIANU (2005), *Computational Physics, Problem Solving with Computers*, Second Ed., Wiley, New York.

[G&T 96] GOULD, H., AND J. TOBOCHNIK (1996), *An Introduction to Computer Simulation Methods*, Second Ed., Addison-Wesley, Reading, PA.

[Gar 00] GARCIA, A. L. (2000), *Numerical Methods for Physics*, Prentice Hall, Upper Saddle River, NJ.

[Knuth 86] KNUTH, D. E. (1986), *The TeXbook*, Addison-Wesley, Reading, PA.

[L&F 93] LANDAU, R. H., AND P. J. FINK (1993), *A Scientist's and Engineer's Guide to Workstations and Supercomputers*, Wiley, New York.

[Lap 85] LAMPORT, L., (1986), *LaTeX: A Document Preparation System*, Second Ed. Addison-Wesley, Reading, PA.

[LAP 95] ANDERSON, E., Z. BAI, C. BISCHOF, J. DEMMEL, J. DONGARRA, J. DU CROZ, A. GREENBAUM, S. HAMMARLING, A. MCKENNEY, S. OSTROUCHOV, AND D. SORENSEN (1995), *Lapack Users' Guide*, Second Ed., SIAM, Philadelphia; http://netlib.org.

[Press 94] PRESS, W. H., B. P. FLANNERY, S. A. TEUKOLSKY, AND W. T. VETTERLING (1994), *Numerical Recipes*, Cambridge University Press, Cambridge, UK.

[UCES] UNDERGRADUATE COMPUTATIONAL ENGINEERING AND SCIENCE, http://www.krellinst.org/UCES/.

[Wilk 95] WILKINS, D. R., *Getting Started with LaTeX*, Second Ed., http://www.maths.tcd.ie/~dwilkins/LaTeXPrimer/, (1995).

[Zach 96] ZACHARY, J. L. (1993), *Introduction to Scientific Programming, Computational Problem Solving using Maple and C*, Springer-Telos, Santa Clara, CA.

MATHEMATICS

[A&S 64] ABRAMOWITZ, M., AND I. A. STEGUN (1964), *Handbook of Mathematical Functions*, U.S. Govt. Printing Office, Washington.

[B&R 92] BEVINGTON, P. R., AND D. K. ROBINSON (1992), *Data Reduction and Error Analysis for the Physical Sciences*, McGraw-Hill, New York.

[Fral 76] FRALEIGH, J. B. (1976), *A First Course in Abstract Algebra*, Second Ed., Addison-Wesley, Reading, PA.

[Krey 88] KREYSZIG, E. (1988), *Advanced Engineering Mathematics*, Sixth Ed., Wiley, New York.

SCIENCE

[D&A 00] DEGAUDENZI, M. E., AND C. M. ARIZMENDI, *Wavelet-Based Fractal Analysis of Electrical Power Demand*, Fractals, **8,** no. 3 (2000) 239–245.

[Feig 79] FEIGENBAUM, M. J. (1979), *J. Stat. Physics* **21**, 669.

[Fein 76] G. FEINBERG, Phys. Rev. **159** (1976) 1089–1105.

[Feyn 63] FEYNMAN, R. P., R. B. LEIGHTON, AND M. SANDS, (1963), *The Feynman Lectures on Physics* §9.7.

[M&T 88] MARION, J. B., AND S. T. THORNTON (1988), *Classical Dynamics of Particles and Systems*, Third Ed., Harcourt Brace Jovanovich, Orlando, FL.

[R & M 93] REITZ, J. R., F. J. MILFORD, AND CHRISTY, R. W. (1993), *Foundations of Electromagnetic Theory*, Fourth Ed., Addison-Wesley, Reading, PA.

[Rash 90] RASBAND, S. N. (1990), *Chaotic Dynamics of Nonlinear Systems*, Wiley, New York.

[Ser 00] SERWAY, R. A., AND R. J. BEICHNER (2000), *Physics for Scientists and Engineers*, Fifth Edition, Saunders, Orlando, FL.

[Smith 65] SMITH, J. H. (1965), *Introduction to Special Relativity*, Benjamin, New York.

[USN&WR] *US News & World Report*, 1/28/99.

WEB RESOURCES

Web pages change so often that we expect some of these URLs to fail. If so, we suggest searching on the name.

[Black] *The Java-Linux Porting Project*, http://www.blackdown.org/.

[CPlets] *Interactive Computational Physics Applets*, http://www.physics.orst.edu/~rubin/nacphy/CPapplets/, http://www.physics.orst.edu/~rubin/TALKS/CPtalk/DEMO/samples.html.

[Dislin] H. Michels, MPI fuer Sonnensystemforschung, *Dislin Scientific Plotting Software*, http://www.mps.mpg.de/dislin/.

[DX] *OpenDX, Data Explorer* (formerly IBM's DataExplorer), http://www.opendx.org/.

[Energy] *Edison Electrical Institute, Climatron Research Technical Reports*, http://www.climatron.com; *California Independent System Operator*, http://www.caiso.com; *Platte River Power Authority*, http://www.prpa.org.

[Game] *Gamelan Repository of Java Code*, http://www.gamelan.com/.

[Gnu] *The GNU Project, Emacs*, http://www.gnu.org/.

[Gnuplot] A command-line interactive datafile and function plotting utility, http://www.gnuplot.info/.

[Grace] A WYSIWYG 2D plotting tool (descendant of ACE/gr, Xmgr), http://plasma-gate.weizmann.ac.il/Grace/.

[Jama] HICKLIN, J., C. MOLER, P. WEBB, R. F. BOISVERT, B. MILLER, R. POZO, AND K. REMINGTON, *JAMA: Java Matrix Library*, http://math.nist.gov/javanumerics/jama/.

[Jampack] *Java Matrix Package*, http://www.mathematik.hu-berlin.de/~lamour/software/JAVA/Jampack/Doc/00_Manual.html.

[jEdit] *jEdit*, http://www.jedit.org.

[JF] *NPAC JavaForce*, http://www.npac.syr.edu/projects/javaforce/.

[Jgran] *Java Grande Forum*, http://www.javagrande.org/.

[JHPC] *Java for High Performance Computing*, http://www.jhpc.org/.

[JNL] VISUAL NUMERICS, *Java Numerical Library*, http://www.vni.com/products/imsl/jmsl.html.

[Jopt] *Java Optimization*, http://www-2.cs.cmu.edu/~jch/java/optimization.html.

[JsimS] *Physics Simulation with Java*, http://www.particle.kth.se/~fmi/kurs/PhysicsSimulation/.

[Jtut] *Sun's Java Tutorial*, http://java.sun.com/docs/books/tutorial.

[MapWeb2] *Indiana Maple WWW Tutorial*, http://www.indiana.edu/~statmath/math/maple.

[MMM] *Mathematica, Maple, Matlab, IDL, Translations*, http://amath.colorado.edu/computing/mmm/.

[Net] *Netlib*, http://www.netlib.org/.

[NIST] *Java Numerics at National Inst of Sci and Tech*, http://math.nist.gov/javanumerics/.

[OSS] *Open Source Software*, http://www.opensource.org/.

[Printf] BENGTSSON, H., *Java Printf Package*, http://www.braju.com/.

[PtPlot] Ptolemy plotting project, http://ptolemy.eecs.berkeley.edu/java/ptplot/.

[SunJ] *Sun Microsystems Java Home Page*, http://java.sun.com/.

[UnixWeb] LANDAU, R. H., P. J. FINK, AND M. JOHNSON, *Unix Survival Guide*, http://www.physics.orst.edu/~rubin/nacphy/UNIX/.

[Visad] *VisAD, Java Component Library for Interactive and Collaborative Visualization and Analysis of Numerical Data*, http://www.ssec.wisc.edu/~billh/visad.html.

[WinEdt] SIMONIC, A., *WinEdt*, http://www.winedt.com/.

[WinZip] *WinZip Extractor and Compactor*, http://www.winzip.com/.

Index