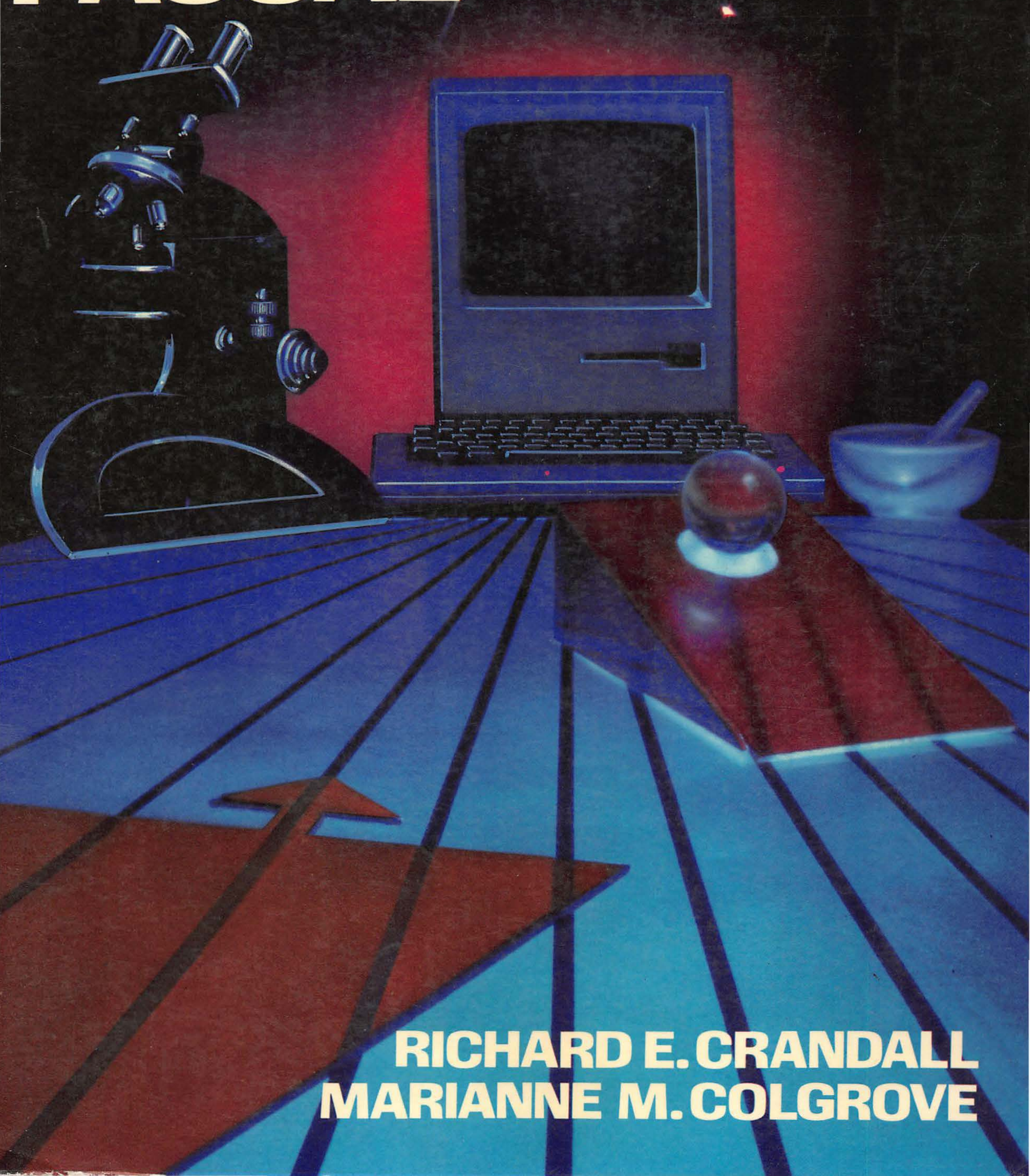
SCIENTIFIC PROGRAMMING WITH MACINTOSH PASCAL



RICHARD E. CRANDALL
MARIANNE M. COLGROVE

[redacted]

Scientific Programming with Macintosh Pascal

Scientific Programming with Macintosh Pascal

Richard E. Crandall
Marianne M. Colgrove

A Wiley Press Book
John Wiley & Sons, Inc.
New York • Chichester • Brisbane • Toronto • Singapore

Publisher: Stephen Kippur
Editor: Theron Shreve
Managing Editor: Katherine Schowalter
Electronic Book Production Services: The Publisher's Network

Library of Congress Cataloging-in-Publication Data

Crandall, Richard E., 1947-
  Scientific programming with Macintosh PASCAL.

  1. Macintosh (Computer)—Programming.
2. PASCAL (Computer program language)
3. Science—Data processing. I. Colgrove,
Marianne M. II. Title.
QA76.8.M3C73 1986 005.2′65 85-19083

ISBN 0-471-82176-4

Printed in the United States of America

86 87 10 9 8 7 6 5 4 3 2 1

Contents

Preface

Scientific Programming with Macintosh Pascal is intended for Macintosh users who are interested in using their machines to produce scientific results. By results we mean not only calculations but graphics, sound, and general input/output. To this end the book includes extensive example programs and exercises to cover what we perceive are the most important basic aspects of scientific programming. Where possible, we incorporate graphics and animation techniques—certainly Macintosh specialties—to convey scientific principles.

Users who enjoy computer modeling will find the example programs of special interest, chiefly because of the unique origin of those examples. The programs and ideas found within grew out of experiences in the science classrooms of Reed College, where the use of Pascal in courses dates back to 1978. In such settings emphasis is placed on ideas as opposed to programs, with the latter used to support the learning and research. Accordingly, this book is not a computer science book. It is a book of scientific ideas, and we have chosen tried and true ideas which we know to be interesting to science students. There is no better testing ground for examples and exercises than a class of sharp, motivated beginning programmers. In many cases we modified older programs to take advantage of what is best about the Macintosh, doing this in response to feedback from teachers and students.

We are indebted to the many faculty and students who lent their fine hands to the testing and modification of the programming ideas presented in the book. Special acknowledgements are due to Richard Wood and Peter Shirley whose advanced programs we have used in later chapters. Other contributors are Neil Alexander, Zoe Mendell, Nic McPhee, Scott Gillespie, and Shep Doeleman. The book strategy was markedly improved through discussions with Professor Robert Reynolds who teaches some active, well-populated introductory Pascal laboratories. We appreciate also the helpful encouragement and support from our friends at THINK Technologies and Apple Computer. We could not have done the project without the inestimable clerical talents of Rebecca Kilgore.

Richard E. Crandall
Marianne M. Colgrove

Portland, Oregon
February 1986

1 Introduction

THEME: This chapter provides an introduction to the philosophy and format of *Scientific Programming with Macintosh Pascal.* It discusses the prerequisites and reference materials recommended for effective use of the book as well as the suggested approach to learning scientific programming.

GOALS: You will be able to delve into practice programs and exercises with an awareness of Macintosh Pascal's strengths and weaknesses. Each chapter is summarized so those readers who have specific programming projects in mind will be able to select the appropriate chapters for study.

REFERENCE MATERIALS: Novice programmers and Macintosh users will probably want to study an introductory Pascal text, the *Macintosh User Manual,* and/or the *Macintosh Pascal User's Guide*. See the end of this chapter for selected references.

Why Macintosh Pascal?

Pascal was developed by Niklaus Wirth in 1971 to fill the need for a simple-to-learn, easily implemented programming language which could be used to teach elementary programming concepts. Pascal, like other high-level languages, provides programmers, and those studying Pascal programming, with the tools to communicate specific, un-ambiguous instructions to the computer. According to Cooper & Clancy (*Oh! Pascal!*, p. xvii), the two primary goals of Pascal were (and are):

1. To provide a teaching language that would bring out concepts common to all languages, while avoiding inconsistencies and unnecessary detail.
2. To define a truly standard language that would be cheap and easy to implement on any computer.

Judging by Pascal's current popularity, both in educational and research settings, these objectives have been, for the most part, achieved.

Macintosh Pascal, designed by THINK Technologies exclusively for the Apple Macintosh, serves to enhance Pascal's fundamental ease of use and educational value. Much of the credit for this belongs to the Apple Macintosh itself: its innovative devices, visual interface and general ease of use provide a base that is consistent with Pascal's ideals. With the addition of Macintosh Pascal's interpretive environment and strong error-checking and debugging features, the result is an easy-to-use, high-level programming language with great educational utility.

Macintosh Pascal should not, however, be reserved for the classroom setting only. Its potential value as a tool for the sciences cannot be overlooked. Indeed, many of the special features of the Macintosh and Macintosh Pascal that make it a valuable teaching medium also make it a viable scientific tool. This book demonstrates many scientific problems for which the speed of solution is not the primary requirement: for example, a graphics display of a scientific model that is to be drawn on the screen. This is especially true when factoring the time needed to design and edit the required algorithms into a project's total development time. After just a little experience with the self-checking Macintosh Pascal Edit window, one saves valuable time at refining and modifying the overall project. An extreme example of these considerations is the spectacular ray-tracing demonstration by Peter Shirley, discussed in Chapter 9 as program Scene. Mr. Shirley was a student of one of the authors (REC) at the time the program was written. Though one must wait hours (literally) for the final graphics, it is a fact that Shirley wrote the program in one or two days. When he ran the first tests, he would simply do something else (besides programming) for the long waiting period, coming back eventually to view the Macintosh screen. Thus the Macintosh Pascal interpreter, although very slow in execution compared with compiled systems, did not take away an undue portion of his personal time. The simple realization that his program was very fast in the writing lends support to the proficiency of the Macintosh Pascal approach for many problems of science.

It is for users who can benefit from the advantages of the interpretive environment that *Scientific Programming with Macintosh Pascal* was designed. The dedicated learner who is willing to work through the sample programs and exercises will not only emerge with a strong understanding of basic scientific programming concepts, but should also be able to develop applications with considerable practical utility. In addition, the serious reader should be able to develop the problem-solving strategies that enable one to approach future programming problems creatively and elegantly.

Before embarking on *Scientific Programming*, the reader should be familiar with the Macintosh visual interface and basic operations such as cutting, pasting, and text editing with mouse-oriented editors. If you have not previously used a Macintosh, refer to the Macintosh owner's guide for details on basic techniques. In addition, you should have a working knowledge of the fundamentals of the Pascal language. This book is not intended to teach you Pascal, rather, it is meant to help you expand the range of problems to which you can apply your programming skills. Familiarity with the Macintosh Pascal programming environment—using the Observe window to debug a program, for example—is desirable, though not required. If you need additional preparation in any of the above areas, check the selected bibliography at the end of this book for appropriate texts and reference materials.

The best way to learn the techniques discussed in this book is to sit down at a Macintosh and practice programming: work through the exercises at the end of each chapter; experiment with program changes and see what happens. This is not to say, however, that you should abandon paper and pencil in favor of long hours in front of a computer. It can be invaluable to draft a flow chart or a simple sketch of program logic before using the computer. In this way, frustrating bugs can often be anticipated or avoided altogether. You will undoubtedly develop a balance of paper and computer work time that most suits your particular needs. A combination of these methods will promote more efficient learning and programming than the exclusive use of either method.

In addition to working with the chapter exercises, you should devise relatively large-scale projects to work on. Projects will help you develop problem-solving strategies as they involve many levels of difficulty. Solutions for each subproblem must fit together into a unified program. Devising your own project also insures that it will incorporate your particular area of interest and will more likely produce an application of personal utility. Lastly, a carefully defined project will have a very clear outcome, making it painfully obvious when a program doesn't run as intended. Working on such large-scale tasks will force you to look at the 'big picture' of program strategy and concept flow. The included exercises, on the other hand, provide detailed glimpses of specific commands and their results. Thus, projects and exercises should be used to complement each other.

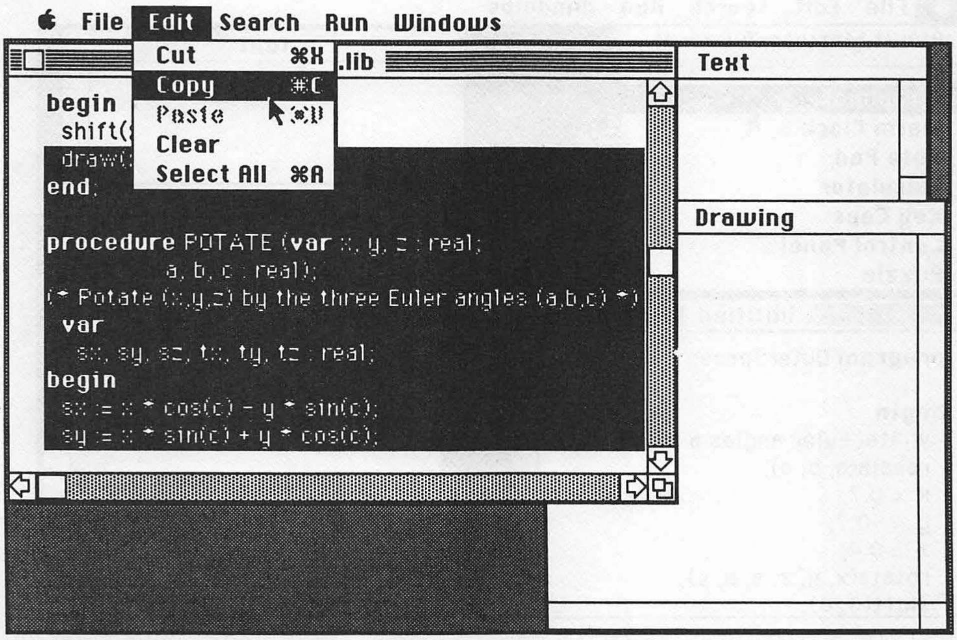
Any programming language has limitations and, therefore, trade-offs must be made. Any programmer, before embarking on a large scientific project, should be aware of the strengths and weaknesses of Macintosh Pascal. As an interpreted language, Macintosh Pascal provides reliable error-checking and feedback. As a result, program speed is sacrificed. Another trade-off arises from the use of specialized Mac features such as the mouse and high-resolution QuickDraw graphics routines. These are the very capabilities that make Macintosh Pascal unique; however, they can reduce the portability (to other machines or systems) of a given program. The user should also keep in mind the memory limitations of the 128K Macintosh: Macintosh programs are limited to approximately 33K in size on a 128K machine, making it most suited for medium to small applications. You can select 'About Macintosh Pascal' from the Apple Menu to see what the memory situation is for any program about to be run.

Procedural libraries (listed in Appendix A) are used extensively in the book. There are a number of reasons for this emphasis on generalized libraries. First, such libraries help you maintain programmatical modularity and consistency. Second, because the libraries are grouped according to general concepts (e.g., graphics , mathematics, etc.) they help the programmer approach a given problem from a more conceptual, strategic perspective. Third, using common libraries facilitates sharing applications with other people and makes it easier to refer to old programs and recall their logical structure. Fourth, if several programs share a common library, you can avoid storing the same information in several different files and thus conserve disk space.

Though the chapters tend to proceed from relatively easy to more difficult programming applications, each chapter is designed to be modular so you can easily concentrate on topics that suit your particular needs. Chapter 2, *Programming the Macintosh Devices*, discusses how to utilize the devices unique to the Macintosh and, as such, provides a good introduction to more purely scientific applications discussed in later chapters. Special attention is given to screen, mouse-handling, timing, and sound generator routines. Chapter 3 is dedicated to numerical analysis programs such as integer computations, differential calculus, integral calculus and linear equations. Chapter 4 is dedicated to initial experimentation with graphics problems. Chapter 5 discusses a subject ubiquitous in scientific settings: probability and statistics. This area is important to all scientists who need to process experimental data. Chapter 6 covers three-dimensional techniques, with emphasis on the library (3D.lib) found in this book. Chapter 7 specializes in dynamic models—models in which quantities, often graphically displayed, change in time. Chapter 8 provides optional treatment of the serial port of the Macintosh. Among other things, you will see how to program a terminal emulator which turns the Macintosh into a simple terminal for logging in by modem or direct line at baud rates up to 9600 baud. Chapter 9 is comprised of a set of special topics chosen to exemplify what is best about the Macintosh Pascal approach. These have the flavor of actual real-world projects, and for the most part, were taken from the authors' experience at Reed College in the matter of classroom and research work.

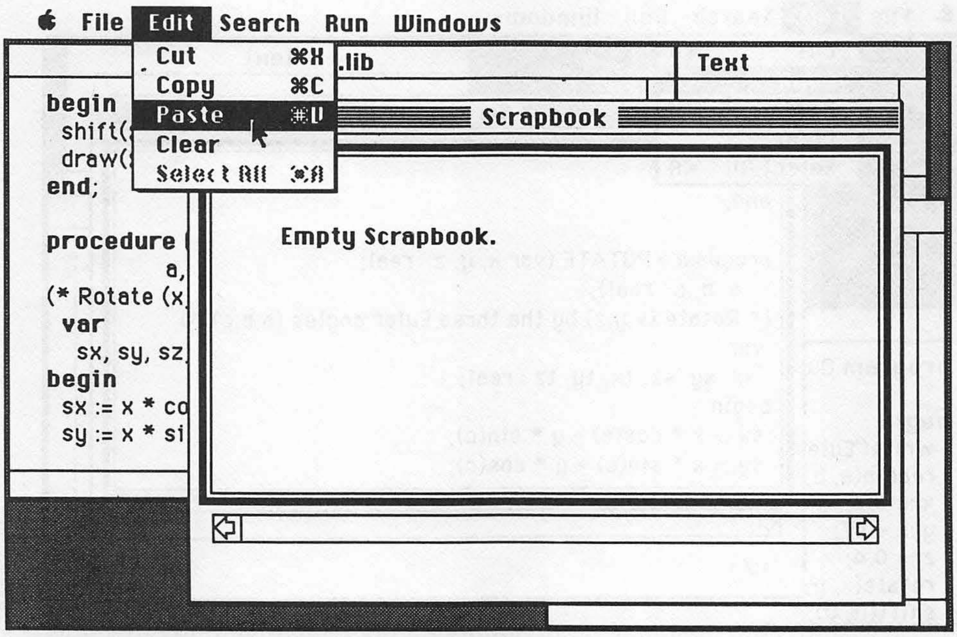
A final word is in order concerning the Pascal libraries. Once you have libraries on disk, there is a good way to continually paste them, or segments of same, into your programs. The idea is to use the Macintosh Scrapbook and Clipboard in tandem. Refer to Figures 1.1 through 1.5 for a tour of the method.

This method becomes quite easy with practice, and you may always go back at any time and undo mistakes by selecting text and clearing. Often it is good to put several whole libraries onto the Scrapbook, which can hold several independent entries.



```
begin
  shift(
  draw(
end;

procedure ROTATE (var x, y, z : real;
           a, b, c : real);
(* Rotate (x,y,z) by the three Euler angles (a,b,c) *)
  var
    sx, sy, sz, tx, ty, tz : real;
begin
  sx := x * cos(c) - y * sin(c);
  sy := x * sin(c) + y * cos(c);
```

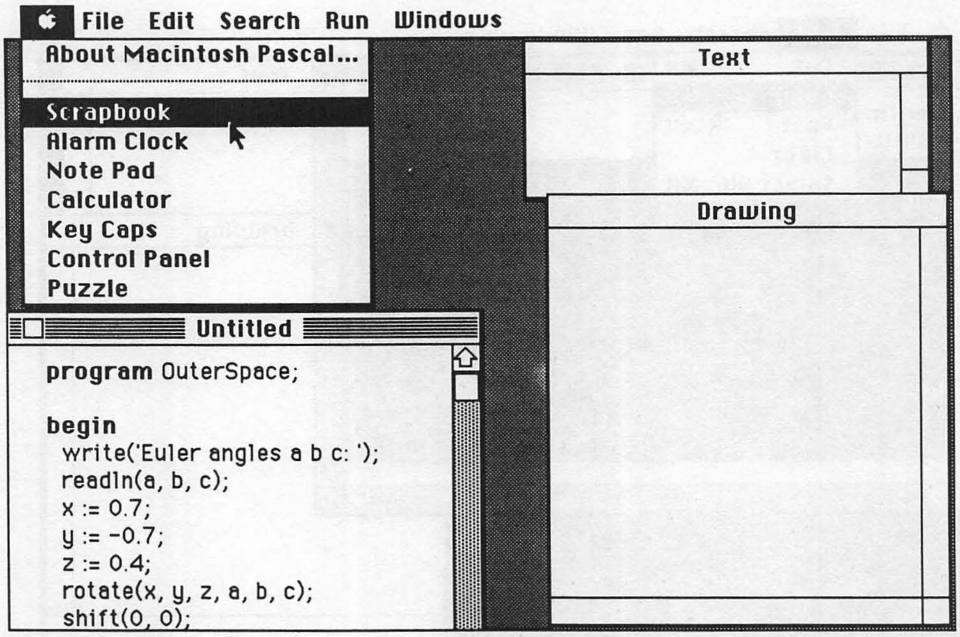
Figure 1.1 The user has opened the 3-D graphics library (3D.lib) and selected particular procedures, including ROTATE which will rotate the 3-space view of objects, and is about to Copy this segment to the Clipboard.



```
begin
  shift(
  draw(
end;

procedure (
       a,
(* Rotate (x,
  var
    sx, sy, sz,
begin
  sx := x * co
  sy := x * si
```

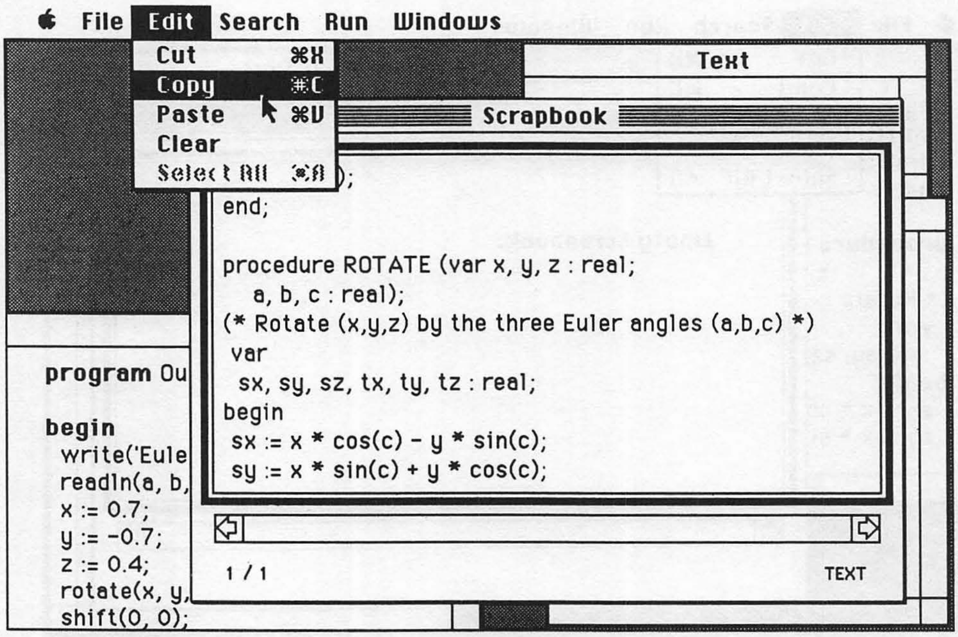
Empty Scrapbook.

Figure 1.2 The Scrapbook has been opened and is currently empty. The user is about to Paste the program segment into the Scrapbook.



```
program OuterSpace;

begin
  write('Euler angles a b c: ');
  readln(a, b, c);
  x := 0.7;
  y := -0.7;
  z := 0.4;
  rotate(x, y, z, a, b, c);
  shift(0, 0);
```

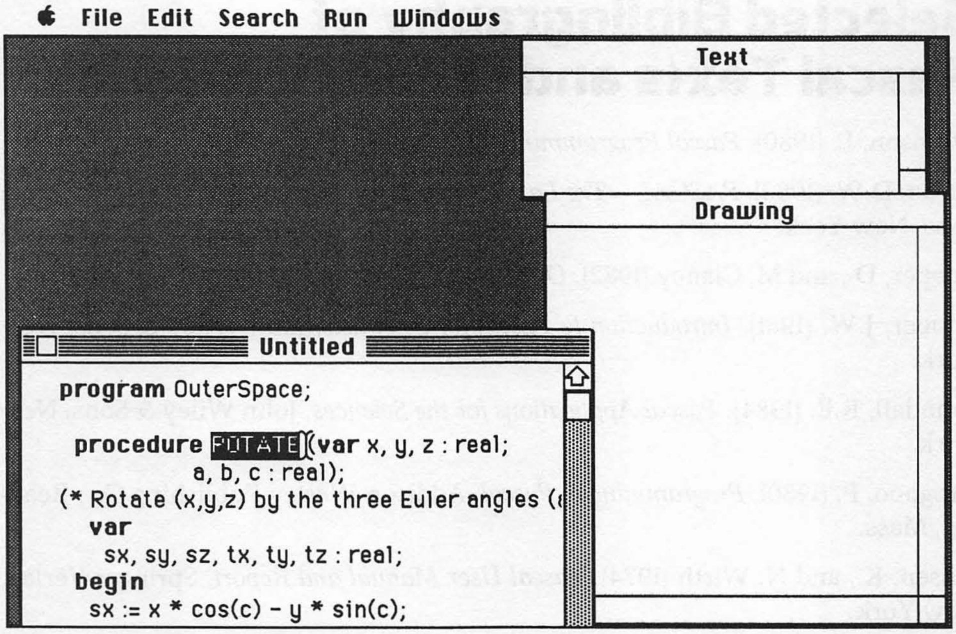
Figure 1.3 The Scrapbook is assumed correct, and (possibly much later, even if the Macintosh has been turned off) the user opens a new Edit window and starts out a new program which will use the 3-D procedures of interest.



```
);
end;

procedure ROTATE (var x, y, z : real;
   a, b, c : real);
(* Rotate (x,y,z) by the three Euler angles (a,b,c) *)
 var
  sx, sy, sz, tx, ty, tz : real;
begin
 sx := x * cos(c) - y * sin(c);
 sy := x * sin(c) + y * cos(c);
```

```
program Ou

begin
 write('Eule
 readln(a, b,
 x := 0.7;
 y := -0.7;
 z := 0.4;
 rotate(x, y,
 shift(0, 0);
```

Figure 1.4 The Scrapbook is again selected, and Copy is used again to get the procedures onto the Clipboard.



```
program OuterSpace;

  procedure ROTATE (var x, y, z : real;
               a, b, c : real);
(* Rotate (x,y,z) by the three Euler angles (
  var
    sx, sy, sz, tx, ty, tz : real;
  begin
    sx := x * cos(c) - y * sin(c);
```

Figure 1.5 The user has moved the text cursor to a point underneath the program header, selected Paste from the Edit menu, and put away the Scrapbook. Result: the new program has the 3-D procedure ROTATE included.

Keep this rule in mind:

BOOK LIBRARY PROCEDURE NAMES APPEAR IN ALL CAPITALS THROUGHOUT THE BOOK.

If you see a procedure spelled out in a program listing like so:

procedure CLEAR

it means that CLEAR comes from one of your book libraries. This particular one, CLEAR, occurs often and has the effect of conveniently sizing, placing, and clearing the Drawing window of Macintosh Pascal prior to actual drawing.

Selected Bibliography of Pascal Texts and Reference Books

Atkinson, L. (1980). *Pascal Programming*, John Wiley & Sons, New York.

Baron, D.W. (1980). *PASCAL—The Language and its Implementations*, John Wiley & Sons, New York.

Cooper, D., and M. Clancy (1982). *Oh! Pascal!*, W.W. Norton & Co., New York.

Cooper, J.W. (1981). *Introduction to Pascal for Scientists*, John Wiley & Sons, New York.

Crandall, R.E. (1984). *Pascal Applications for the Sciences*, John Wiley & Sons, New York.

Grogono, P. (1980). *Programming in Pascal*, Addison-Wesley Publishing Co., Reading, Mass.

Jensen, K., and N. Wirth (1974). *Pascal User Manual and Report*, Springer-Verlag, New York.

Moll, R. and R. Folsom (1985). *Macintosh Pascal*, Houghton Mifflin Company, Boston.

Moore, J.B. (1982). *PASCAL*, Reston Publishing Co., Reston, VA.

Schneider, G.M., and S.C. Bruell (1981). *Advanced Programming and Problem Solving with Pascal*, John Wiley & Sons, New York.

Schneider, G.M, S.W. Weingart, and D.M. Perlman (1978). *An Introduction to Programming and Problem Solving with Pascal*, John Wiley & Sons, New York.

2 Programming Macintosh Devices

THEME: This Chapter covers the devices unique to the Macintosh and their associated procedures and functions. The serial port, being less commonly used and more technically difficult, is dealt with in Chapter 9.

GOALS: You will learn how to use the Macintosh devices for general scientific problems, especially where striking visual/auditory output or convenient manual input is desired.

LIBRARIES USED: Graphics.lib (subset)

REFERENCE MATERIALS: *The Macintosh Pascal Reference Manual.*

Description of Devices

The Macintosh's devices are surprisingly easy to program. The following paragraphs describe the Mac Devices in detail:

Screen This is a large array of pixels, with Pascal's Text and Drawing windows being set up as subsets of the whole screen. The procedures differ depending on which window you wish to access.

Keyboard This is essentially a standard computer keyboard, and except for a few small differences, is treated in Pascal in the conventional manner.

Mouse This is an important input device, and you can obtain its coordinate position and button state from within your programs.

Sound Generator This device can be accessed from within Macintosh Pascal, and there are options ranging from the simple (pure notes) to the complex (special waveforms).

Printer This is usually the Apple ImageWriter and is handled, as with various other Pascal systems, as a "file."

Disk This is handled with conventional Pascal I/O (Input/Output) procedures.

Timers The Macintosh has internal timing hardware that allows you to access either fast (1/60th second) counting or date and time information from within programs.

Serial (Modem) Port This device enables you to access other systems using the Macintosh as a terminal. This is of interest only to users who have this special requirement, and its use is somewhat technical, so the means for serial access are reserved for Chapter 9.

Each of these devices can be accessed literally from inside your programs, and all you need to know from the programming point of view is the manner in which one uses the procedures and functions. This chapter covers a subset of procedures and functions which is intentionally incomplete, but is designed to give you examples of almost all of the options you need for scientific programming.

To get the best results out of Macintosh you must first learn how these devices function. The intricacies of the hardware will not be covered; however, this text shall instead concentrate on the programming point of view. Eventually you will begin to think of the device procedures and functions as natural and elegant extensions of conventional Pascal.

The Screen

The screen windows called 'Text' and 'Drawing' differ as follows. The Text window is essentially a conventional Pascal output device, with the standard reading and writing procedures of the language accessing that window. The Drawing window is for graphic output and involves several new, Macintosh-specific procedures. The key procedures are summarized in the following list. Remember that here and elsewhere, we do not cover exhaustively all possible procedures. We have simply chosen for you a set with which you can do almost any scientific programming project. You can refer to the *Macintosh Pascal Reference Manual* for complete details on all available procedures.

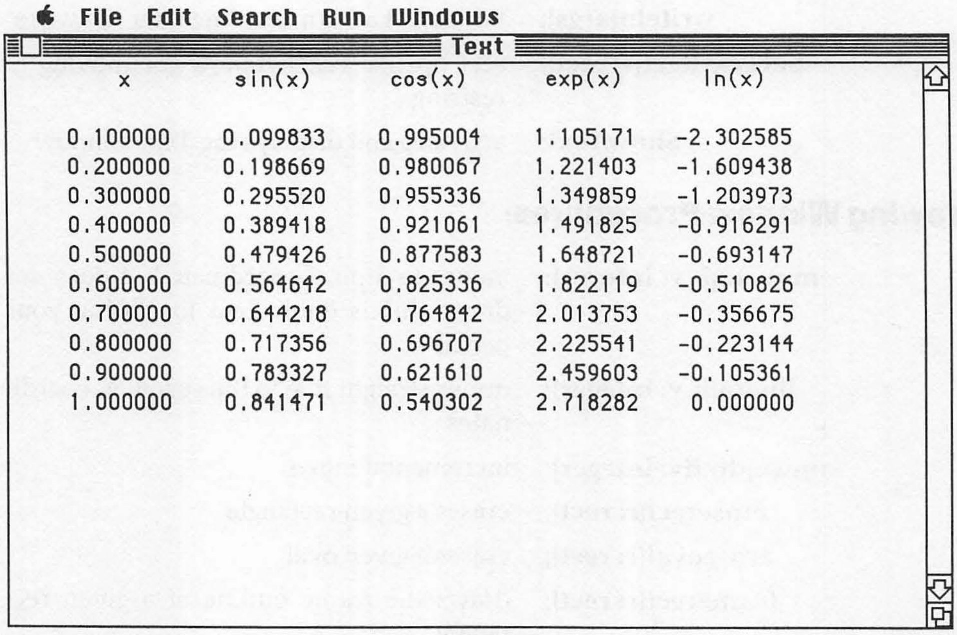
Text Window Output Procedures:

write(args);	writes characters or numbers to the Text window

writeln(args);	writes but adds a new line after the write
SetTextRect(r: rect);	sets up the Text window for moving or resizing
ShowText;	activates and displays the Text window

Drawing Window Procedures:

moveto(h,v: integer);	moves to a pixel coordinate but does not draw; this is equivalent to 'lifting your pencil'
lineto(h,v: integer);	draws straight line to the specified coordinates
move(dh,dv: integer);	incremental move
eraserect(r: rect);	erases a given rectangle
eraseoval(r: rect);	erases a given oval
framerect(r: rect);	draws the frame outline of a given rectangle
frameoval(r: rect);	draws the frame outline of a given oval
paintrect(r: rect);	fills in a given rectangle
paintoval(r: rect);	fills in a given oval
invertrect(r: rect);	reverses the color of a given rectangle
invertoval(r: rect);	reverses the color of a given oval
writedraw(args);	like write for the Text window, except that font can be manipulated, and special positioning is possible
pensize(w,h: integer);	chooses new size for the drawing pen
penpat(pat: pattern);	chooses new pen pattern
penmode(mode: integer);	chooses the 'drawing' logic
backpat(pat: pattern);	chooses a background pattern; useful for animation where you alternately blank and draw
textsize(size: integer);	chooses font size
textface(face: style);	chooses style such as bold or italic
SetRect(var r: rect; left, top, right, bottom: integer);	defines a rectangle (also the contained oval) by setting a rect variable.
SetDrawingRect(r: rect);	sets up the Drawing window for moving or resizing



```
     x         sin(x)      cos(x)      exp(x)      ln(x)

  0.100000   0.099833   0.995004   1.105171  -2.302585
  0.200000   0.198669   0.980067   1.221403  -1.609438
  0.300000   0.295520   0.955336   1.349859  -1.203973
  0.400000   0.389418   0.921061   1.491825  -0.916291
  0.500000   0.479426   0.877583   1.648721  -0.693147
  0.600000   0.564642   0.825336   1.822119  -0.510826
  0.700000   0.644218   0.764842   2.013753  -0.356675
  0.800000   0.717356   0.696707   2.225541  -0.223144
  0.900000   0.783327   0.621610   2.459603  -0.105361
  1.000000   0.841471   0.540302   2.718282   0.000000
```

Figure 2.1 | WriteTable Output

ShowDrawing;	activates and displays the Drawing window
PtInRect(pt: point; r: rect): boolean;	returns a Boolean value = true if a Point is in a Rectangle (actually a function, not a procedure)

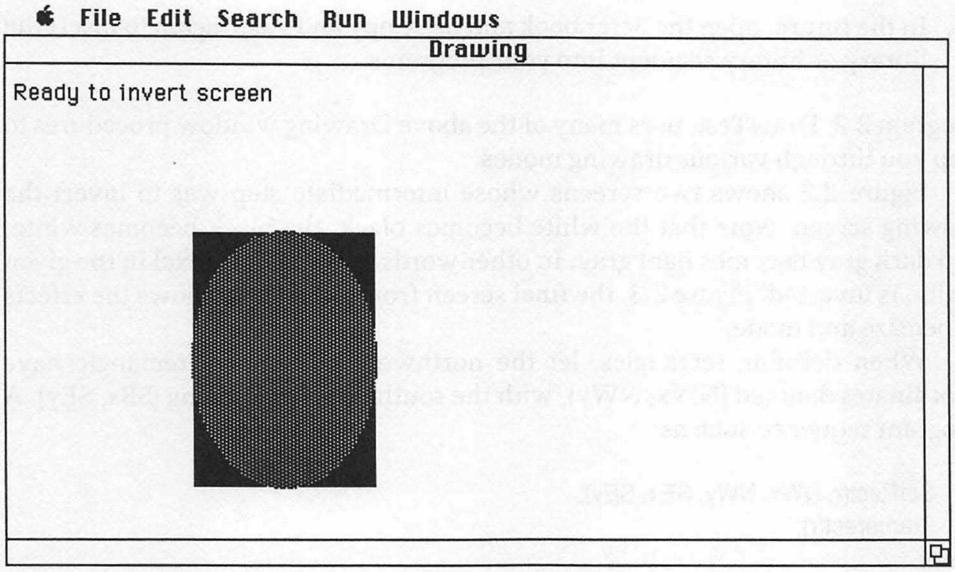
Program 2.1, WriteTable, shows how Text window output can be formatted when numerical data is involved. The output of the program is shown in Figure 2.1.

Note that a statement:

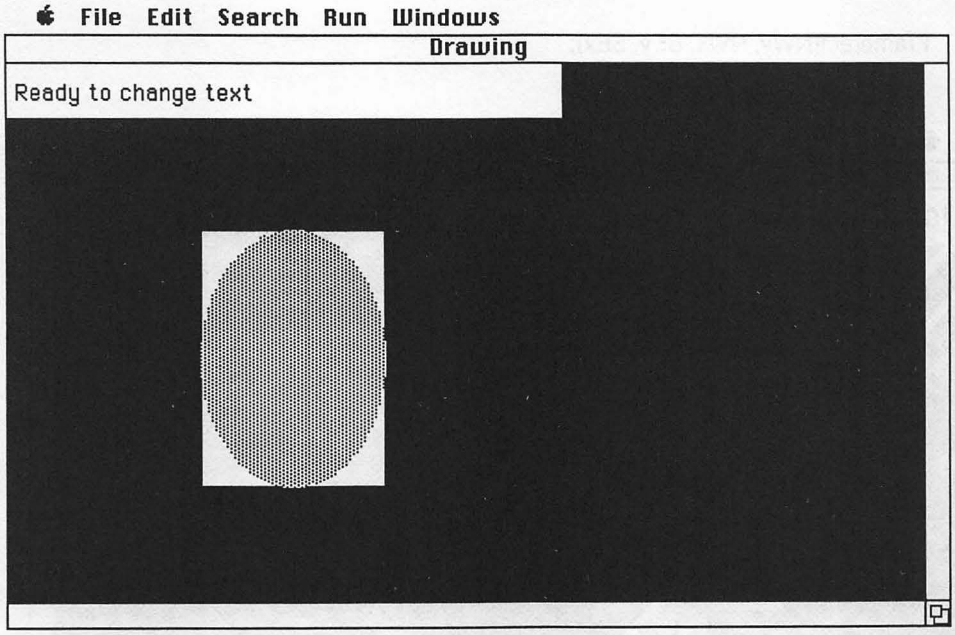
```
writeln(x:6:12,...);
```

has the effect of expanding the real number output to 6 places in a field of 12 spaces. Note also that the procedure 'erase,' which uses a modification of the procedure 'CLEAR' in library 'graphics.lib', makes sure that the Text window is correctly placed and sized. Though this procedure appears to be an unnecessary addition to the program, it is a good idea in many applications to know precisely where the windows will be. This will prevent parts of graphs from being hidden accidentally. A useful procedure for pasting in library segments of this kind is:

1. Open the library (graphics.lib on your disk, for example) or type in the correct segment into the normal Pascal Editor window.



Ready to invert screen



Ready to change text

Figure 2.2 | Successive output from DrawTest program 2.2

2. Use the Edit menu to Copy the desired text to the Clipboard
3. Open the Scrapbook.
4. Use the Edit menu to Paste the desired text into the Scrapbook.

5. In the future, open the Scrapbook and use Copy and Paste again to insert the library or library segment into your programs.

Program 2.2, DrawTest, uses many of the above Drawing window procedures to step you through various drawing modes.

Figure 2.2 shows two screens whose intermediate step was to invert the drawing screen. Note that the white becomes black, the black becomes white, and dark gray becomes light gray. In other words, every single pixel in the given region is inverted. Figure 2.3, the final screen from DrawTest, shows the effects of pensize and mode.

When defining rectangles, let the northwest corner of a rectangle have coordinates denoted (NWx, NWy), with the southeast corner being (SEx, SEy). A program sequence such as:

```
SetRect(r, NWx, NWy, SEx, SEy);
Framerect(r);
```

will correctly draw the desired rectangle; but so will the single statement:

```
Framerect(NWy, NWx, SEy, SEx);
```

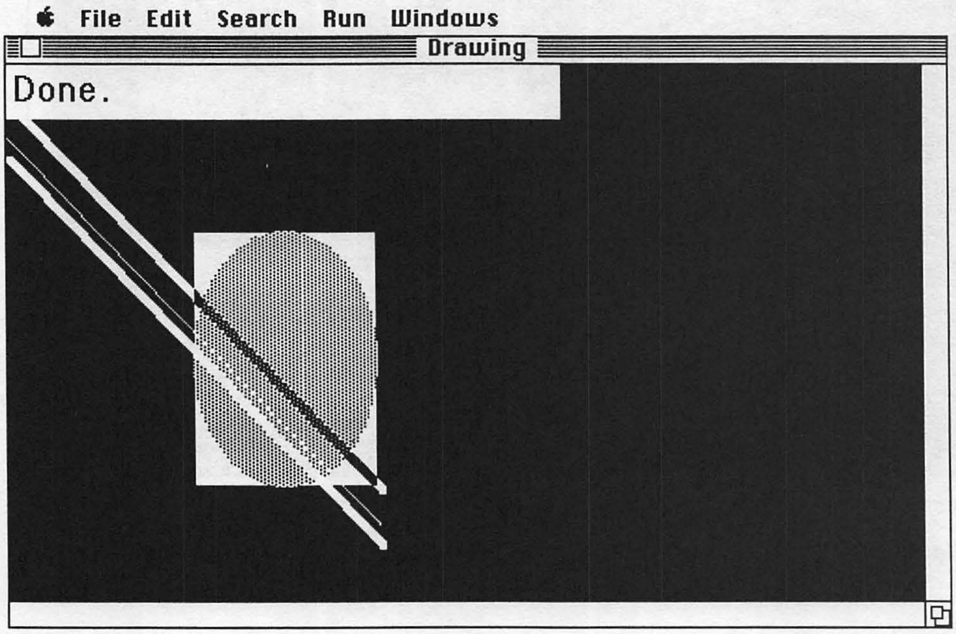


Figure 2.3 | Final output from DrawTest program showing pensize and mode

where you should observe that x,y coordinates are swapped but the rectangle's corners are not.

It is easy to remember the difference between procedures such as backpat and penpat. The backpat procedure chooses essentially the background color, while penpat refers to the foreground. Thus:

```
backpat(white);
eraseoval(r);
```

will do the same thing as:

```
penpat(white);
paintoval(r);
```

Thus 'erasing' puts in background pattern while 'painting' puts in foreground (pen) pattern. Later in this book we do animation—the visual movement of objects on the screen; naturally, 'backpat' and related procedures will be used to erase a previous image for a moving object.

When you get to actual scientific programming tasks, there are a few more things to keep in mind. First, eraserect(r) will not only set the interior of the region to background, it will also erase the frame lines. Second, text labels written onto the screen with writedraw are the only ones affected by the procedures textsize and textface; these procedures do not affect the Text window output. Third, it is sometimes convenient to use animation procedures of your own. For example, you might create a procedure called 'show' and one called 'vanish' which take care of color settings. If you are just animating lines on the normal, white screen, you can do:

```
procedure vanish;
begin
    penpat(white);
end;

procedure show;
begin
    penpat(black);
end;
```

Then the following calls will behave according to the indicated comments:

```
show;
framerect(r);                    (* Rectangle appears *)
vanish;
framerect(r);                    (* Rectangle disappears *)
```

Care must be taken if the screen background changes during program execution. Such eventualities require more sophisticated animation procedures which take due note of the possible background.

The Keyboard

The Macintosh keyboard is straightforward, and its use involves four Pascal constructs:

read(args);	reads characters or numbers from keyboard. These also are being echoed to the Text window.
readln(args);	reads similarly but expects a final <Return>
eof;	this Boolean = true when you hit the <Enter> key
eoln;	this Boolean = true if the next character is a <Return>

The two procedures for reading information are the same as in conventional Pascal. The function eof, which stands for 'end-of-file,' is a common one in Pascal versions, but the difference for Macintosh is the <Enter> key. There are handy tricks for pausing in a program, one of which is simply to do:

```
write    ('Hit <Return> to continue');
readln;
```

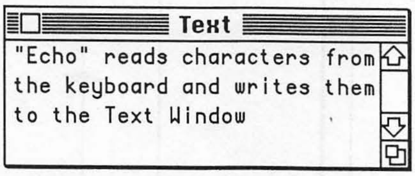
and the other, which applies when there is to be just one pause, is:

```
write    ('Hit <Enter> to start');
while not eof do;
```

This while loop will cycle until such time as the <Enter> key is pressed, but then eof will be true from that point on. When you learn disk access, the function will take the form eof(f), where f is essentially the name of the disk file in question.

A common requirement for a programmer is to be able to read in characters from the keyboard and put them into an array. This is one of those Pascal tasks which can cause considerable grief, so the following is one correct way to do this. Start with declarations such as:

```
var letters: packed array[1..200] of char;
  size:integer;
```



```
"Echo" reads characters from
the keyboard and writes them
to the Text Window
```

Figure 2.4 | Typical Echo output

and eventually do a loop which puts keyboard input into the array 'letters.' Program 2.3 shows Echo, which will perform typical keyboard input operations. Typical output is shown in Figure 2.4.

The key loop is the repeat structure in the program. It might appear to be rather involved, but that is because of Pascal's peculiar way of handling I/O. The reason for the various appearances of eoln and eof is to take care of strange situations such as hitting the <Enter> key in the middle of a line. The same loop construct is used for disk file I/O described below, in which case booleans such as eoln(f) and eof(f) will be used.

The Mouse

There are two entities you should understand with regard to the Mouse: one is the screen point at which the Mouse cursor sits; the other is the boolean value of the mouse button (is it pressed ?). These are returned as an integer pair (x,y) in the first instance, and a boolean value in the second.

The Mouse procedures and functions are as follows:

getmouse(x, y: integer);	sets up values of current cursor position
button;	= true if and only if button is down when function is called

One elegant way to use the getmouse procedure is to declare:

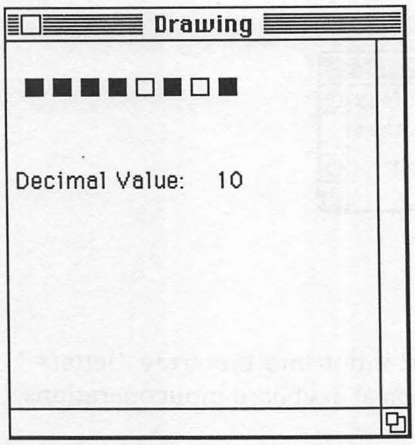
```
var p: Point;
```

where Point is a Macintosh Pascal record two integers long. Then do:

```
getmouse(p.h, p.v);
```

The key extensions .h and .v refer to horizontal and vertical coordinates, respectively. This technique is not mandatory: for example, you can always do:

```
getmouse(x,y);
```



```
Decimal Value:   10
```

Figure 2.5 | Typical Switches output

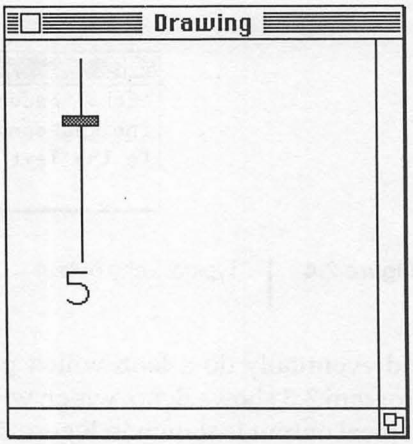


Figure 2.6 | SlidePot output

where x and y are normal integers; but the Point record should be used for reference to such functions as PtInRect. These and other considerations for the Mouse are dealt with in the demonstration programs which are described below.

Program 2.4, Switches (typical output shown in Figure 2.5), shows how to use the Mouse to turn on and off iconic controls. The current color states of the eight switches determine a binary and hence a decimal value, with the latter printed on the screen. This is a good example of using erasure followed by writedraw to update a panel display.

Program 2.5, SlidePot and the screen output in Figure 2.6 show how to move an object with the Mouse.

The Sound Generator procedure, sysbeep, is discussed in the following section. The key program segment here is the 'while button do' loop, which animates the small potentiometer knob as long as you hold down the Mouse button. Note that the function downandin, which determines whether the mouse is both pressed and inside the knob, could also have been done using rectangles and the function PtInRect, as was done in the Switches program.

One of the most important uses of the Mouse in scientific programming is the input of dynamical values, such as initial direction and velocity of a moving object. This kind of application is covered in later chapters (for example, program Lunacy in Chapter 5).

The Sound Generator

The Macintosh Sound Generator is easy to use if you stick to the simpler procedures, such as those summarized in the following list.

note(freq, amp, dur: integer);	plays a note of given frequency, amplitude, and duration
SetSoundVol (level: integer);	sets the volume of the Macintosh Speaker
SysBeep (duration: integer);	sound Macintosh beep for a given duration

There are other sound procedures, but they are not used in this book. You should know that there is an option for arbitrary waveform synthesis, and that working knowledge of record types is required to make use of that option.

The procedure note is a useful one. One would do:

```
var amplitude, duration: integer;
  frequency: Longint;      (* Note: this frequency var is not an integer *)
```

Then, after setting the variables, the simple call:

```
note (frequency,amplitude,duration);
```

will sound the desired tone. Program 2.6, Piano, shows how to compute and play the notes of the well-tempered scale.

The pitch of middle C is assumed to be 256 Hertz. A note exactly x half-steps (the smallest scale steps) away will have the frequency:

$$f = 256 * 2^{\left(\frac{x}{12}\right)}$$

In other words, every musical scale step is an extra factor of the twelfth root of two. Of course, such powers must be computed in Pascal with expressions such as the one in procedure init of the program Piano. Note that the Longints for the scale are only computed once, then later referred to as array elements. Figure 2.7 shows the appearance of the piano.

The Printer

There are no special procedures for the printer, but it has a special name in Macintosh Pascal. This name is 'Printer:' and it is treated like a file. Disk files are discussed next, so for now, the procedure:

```
rewrite(f,'Printer:');
```

is the way to set up the printer for output of text or numerical data. For the actual printing you use:

```
write(f,...);
writeln(f,...);
```

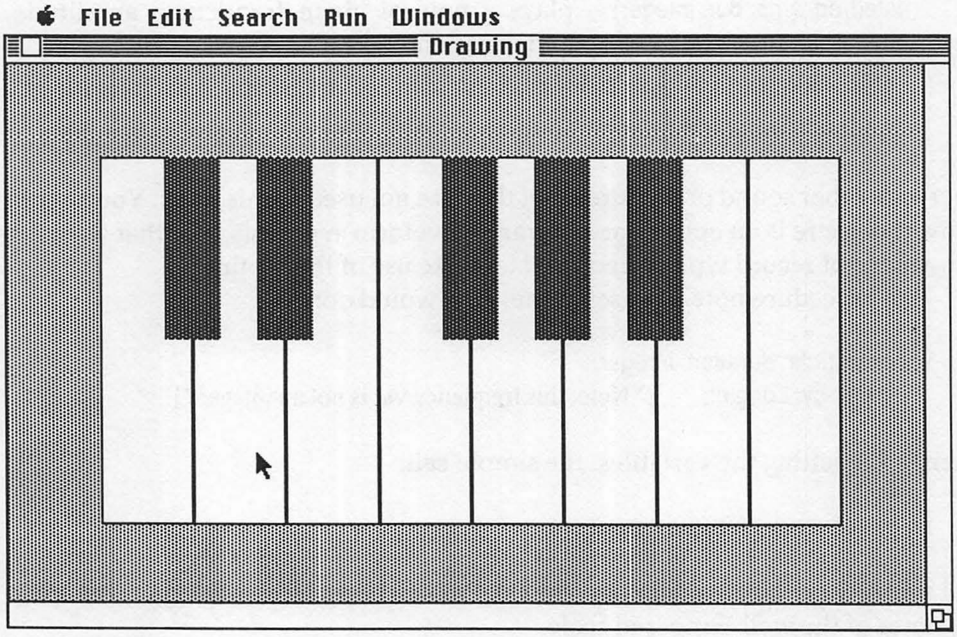


Figure 2.7 | Piano output

where f now refers to the printer because of the rewrite statement. Program 2.7, PrintTable, shows a version of Program 2.1, WriteTable, except that now the results go to the printer.

The Disk

The Macintosh disk can be used in scientific applications for storage and retrieval of data. The formats of data written to or read from the disk can be the same as any allowed formats for the keyboard and screen. In fact, you have already seen that other devices can be treated in certain senses like files. In all file access programming, you use declarations such as:

```
var f:text;
```

Then the key file access procedures are:

reset(f: text);	set up a new file to be read
rewrite(f: text);	set up a new file to be written to
close(f: text);	gracefully terminate file operations

There are some interesting ways to use these procedures, as shown in Programs 2.8, 2.9, and 2.10 with accompanying output examples in Figures 2.8, 2.9, 2.10 respectively. These programs can be summarized as follows:

SelfPrint Accesses its own disk image and sends this to the printer. Note the simultaneous use of reset and rewrite, although the files should be different ones.

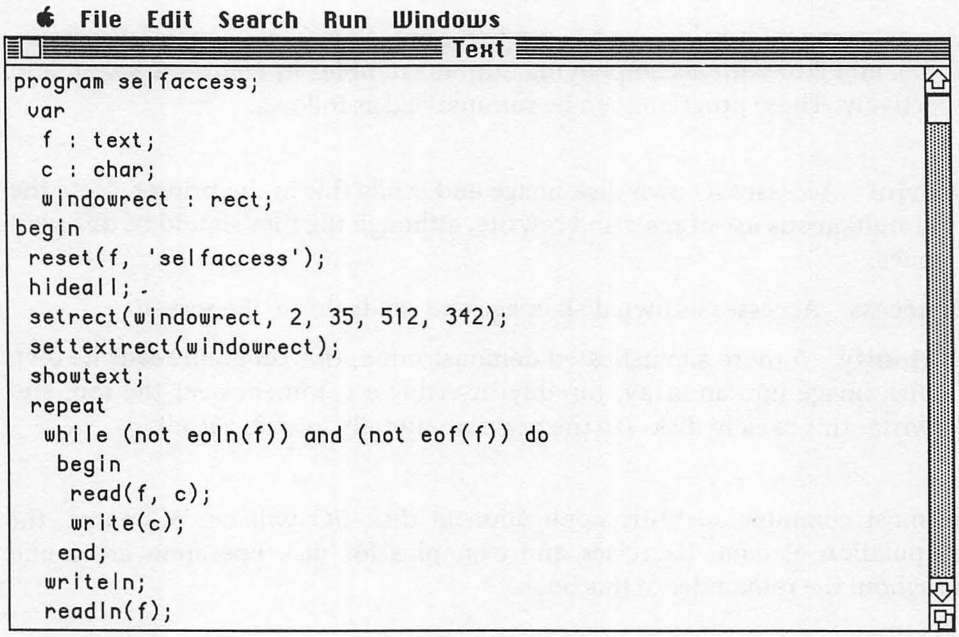
SelfAccess Accesses its own disk image and sends this to the screen.

SelfModify A more sophisticated demonstration, this program reads its own disk image into an array, forcibly inserting a comment near the top, and writes this back to disk—so the program literally modifies itself.

The most common scientific application of disk I/O will be, of course, the manipulation of data. Exercises and examples for disk operation are found throughout the remainder of this book.

```
program selfprint;
(* SENDS ITSELF TO IMAGEWRITER *)

 var
  f, g : text;
  c : char;

begin
 reset(g, 'selfprint');
 rewrite(f, 'Printer:');
 repeat
  while (not eoln(g)) and (not eof(g)) do
   begin
    read(g, c);
    write(f, c);
   end;
  if not eof(g) then
   begin
    readln(g);
    writeln(f);
   end;
 until eof(g);
 close(f);
 close(g);
end.
```

Figure 2.8 SelfPrint output



```
program selfaccess;
 var
  f : text;
  c : char;
  windowrect : rect;
begin
 reset(f, 'selfaccess');
 hideall;
 setrect(windowrect, 2, 35, 512, 342);
 settextrect(windowrect);
 showtext;
 repeat
  while (not eoln(f)) and (not eof(f)) do
   begin
    read(f, c);
    write(c);
   end;
  writeln;
  readln(f);
```

Figure 2.9 | SelfAccess output

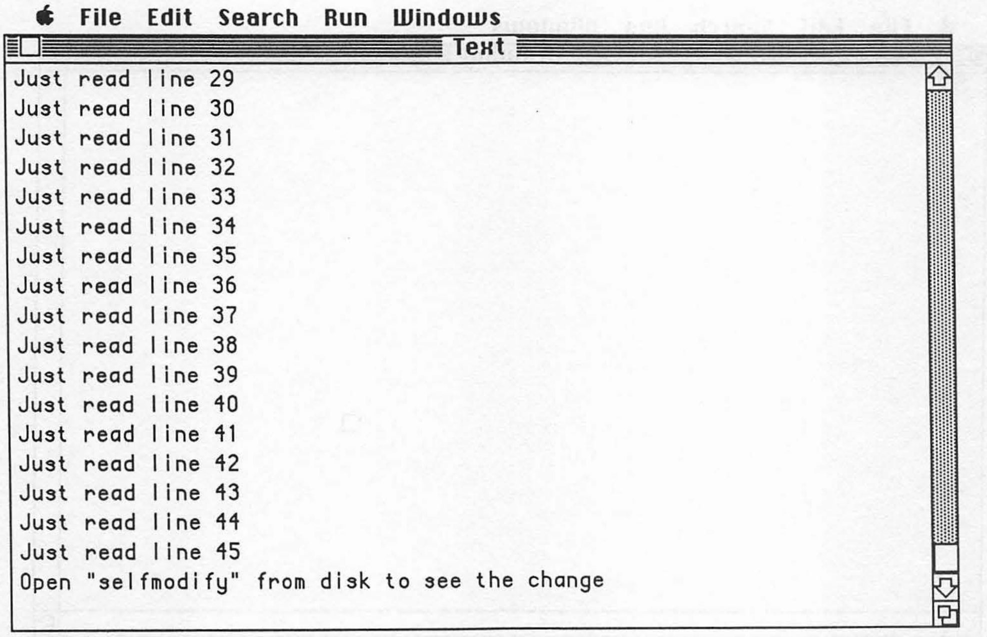
The Timers

There are two timing options: you can access the date and time-of-day, or you can do timing at a 1/60th second resolution. The Timers' procedures and functions can be summarized as follows:

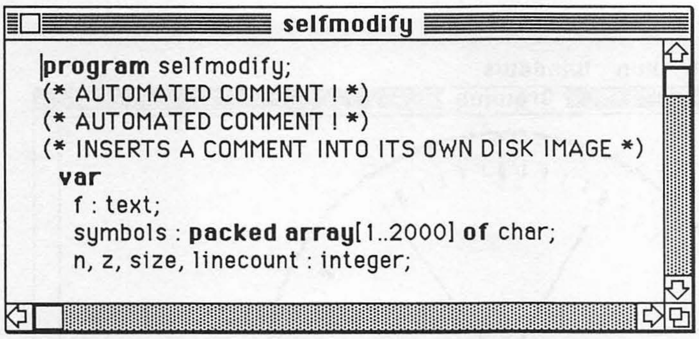
GetTime(DateTime: rec);	gets date and time-of-day record
SetTime(DateTime: rec);	sets system clock
TickCount;	(function) returns a Longint of the number of 1/60th second 'ticks' since an (virtually) unknown time in the past
Synch;	synchronizes the screen display with the tick clock.

The TickCount function is easy to use. You can, for example, wait for half a second by doing:

```
var t:Longint;
...
    t: = TickCount;
    while TickCount-t < 30 do;
```



```
Just read line 29
Just read line 30
Just read line 31
Just read line 32
Just read line 33
Just read line 34
Just read line 35
Just read line 36
Just read line 37
Just read line 38
Just read line 39
Just read line 40
Just read line 41
Just read line 42
Just read line 43
Just read line 44
Just read line 45
Open "selfmodify" from disk to see the change
```



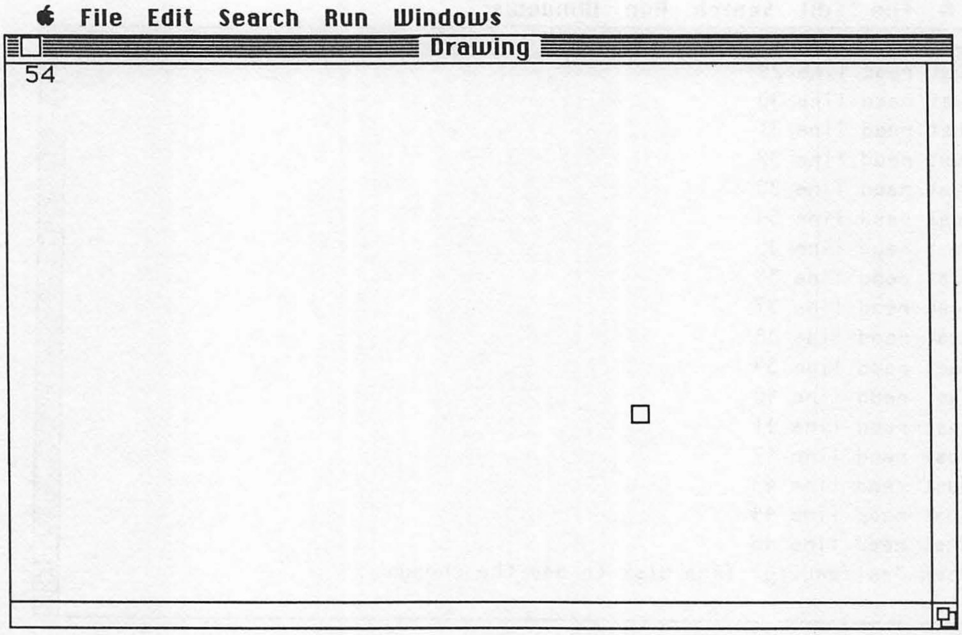
```
program selfmodify;
(* AUTOMATED COMMENT ! *)
(* AUTOMATED COMMENT ! *)
(* INSERTS A COMMENT INTO ITS OWN DISK IMAGE *)
  var
    f : text;
    symbols : packed array[1..2000] of char;
    n, z, size, linecount : integer;
```

Figure 2.10 | SelfModify output

This works because every time the while loop references TickCount, the difference (TickCount-t) has grown correspondingly.

Program 2.11, ReactionTime, uses random numbers and TickCount function to assess the time it takes the user to move the Mouse into a small, randomly positioned box as in Figure 2.11.

Program 2.12, Clock, and the clock appearance, Figure 2.12, shows how to use the date and time-of-day records.



```
54
```

Figure 2.11 | ReactionTime output

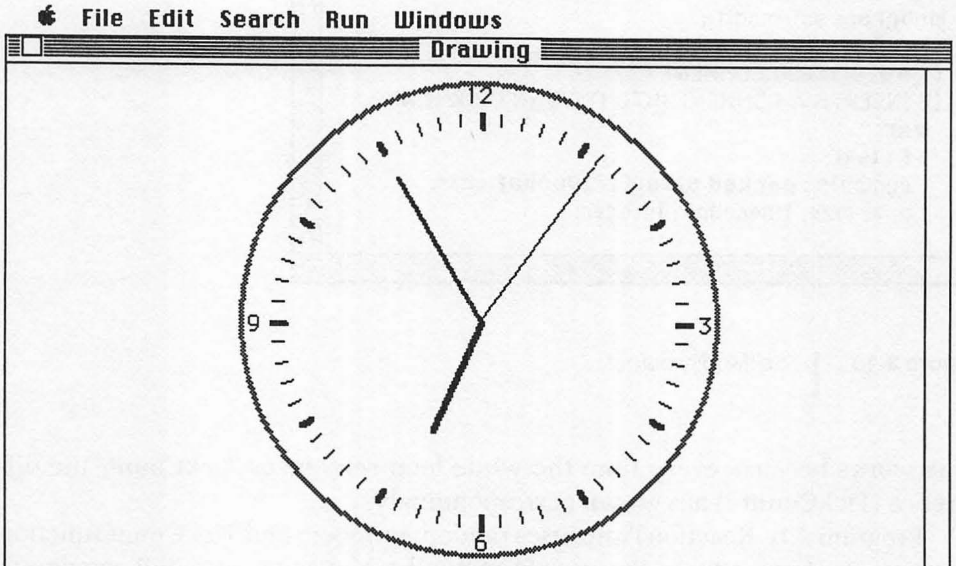


Figure 2.12 | Clock output

Using Devices Efficiently

This chapter concentrated on Macintosh devices to give you an appreciation for the many device options. But it is important not to get bogged down in device play exclusively, for this can get in the way of the scientific approach. You should use devices for situations such as:

1. Storage of calculated data (e.g., a large table of base-ten logarithms) on disk.
2. Graphical display of dynamic phenomena, especially for problems which are not tractable analytically and there is a genuine need for visual assessment.
3. Mouse input when you want to select among a great many alternatives quickly (e.g., if you want to point to a section of a curve).
4. Processing of previously calculated data which has been typed in or otherwise placed on disk. An example would be a Fourier Transform of numerical data. Then the Text window or printer or Drawing window can be used for output.
5. Sound generation to signify when a process is done, or when a suspicious value has been calculated, or when a debugging statement is passed.
6. Any use of a device to aid in learning and demonstration of scientific principles: a geiger counter that clicks using specially computed random times, for example.

In general, use the Macintosh's devices when you really need them, which means when more conventional output is not sufficient. To underscore this point, the next topic, numerical analysis, is often regarded as relatively dry when compared to bells, whistles, and visuals. However, when graphics is used along with analysis, there is something to be gained from the combination.

Exercises

1. Write a program similar to Echo but which ensures that every letter written to the Text window be transformed to upper-case (capitalized). There are two ideas to use: one is that a character c is lower-case if and only if (ord(c) > 96) and (ord(c) < 123); the other is that upper-case characters are obtained from lower-case by subtraction of 32 decimal from the the lower-case ord value.
2. Modify program Switches to print out the hexadecimal value of the 8-bit byte. This output will range from 00 through FF.
3. Modify the program SlidePot so that the slider is shaded progressivley more

toward white (black shading at the very bottom) as it is pushed upward. The colors available include black, dark gray, gray, light gray, white.

4. Make the program Piano put out a fuller keyboard; that is, more octaves.
5. Add a fine-tuning control of the piano using the ideas of program SlidePot so that the entire piano can be tuned to a standard. Many telephone dial tones hum at or near 440 A pitch.
6. Using the ideas behind programs SlidePot and Piano, construct a tone generator which has both an amplitude slide and a frequency slide.
7. Write a program similar to SelfAccess which prints out the number of characters in some file. Compare the number you get with the number given at the Macintosh desktop level under Get Info file option.
8. Write a program which reads in numbers from a text file, using something like:

   ```
   readln (f,x);
   ```

   and sums over all the x, putting the numerical sum in the Text window. To create a test file for this, just set up a new Edit window and type in pure numbers. Then save this file as if it were a program—which it isn't. It can still be accessed with file I/O statements. Later in the book this technique is used to test data processing and signal analysis programs.
9. The program ReactionTime has the drawback that the mouse might be very near to the next random placement of the box. This means that 'luck' is a variable in getting short times. Modify the program so that the next test does not occur until the mouse is taken over to some region, say the far left of the screen, and the box shows up only in a region other than the mouse-restart region.
10. Using the ideas behind the program Clock, create a stopwatch with a 'pushbutton' that must be ticked with the mouse to start, stop, and reset, in that order. Use the function tickcount to get 1/60ths and hence 1/10ths-second resolution. You will not be able to move the 1/10th-second hand in real time, but you should be able to move it every once in a while, readjusting it to the fractional seconds after a stop press.

Answers

1. Near the end of program Echo is an assignment c: = letters[n]. It is there that you may put a function which transforms case:

   ```
   c: = transform(letters[n]);
   ```

The function is written as:

```
function transform(d:char) :char;
begin
    n: = ord(d);
    if (n>96) and (n<123) then transform: = chr(n - 32) else
      transform: = d;
end
```

2. One way to print the hexidecimal value of a decimal number x is to assign:

   ```
   h: = x div 16;
   k: = x mod 16;
   ```

   Note both h and k take on values 0,...,15. To print out 0,...,9,A,B,C,D,E,F for this range, one nice trick is to use:

   ```
   h: = 48 + h + 7 * (h div 10));
   ```

   and the same for k. Then:

   ```
   write (chr(h),chr(k));
   ```

   will print out the hexadecimal.

3. The thing to change is the backpat(gray) call, with the pattern depending on the value of variable yf.
4. This is a straightforward extension of the text program. All you need to do is define the keys periodically. The only real work is to change the init procedure and permute function.
5. This should use techniques similar to those of SlidePot, in which you change some frequency offset with a manual slider; and where Piano calls note(f[i],...)], you want to call note(f[i] + offset,...).
6. The solution to this is to create another SlidePot, then keep calling the note procedure, with frequency and amplitude being results of the two manual sliders.
7. Every time SelfAccess reads in a character, you want a variable such as size to be incremented by one. If the characters do not compare with the Get Info response at Mac desktop level, this may be because you did not count <Returns>.
8. This is straightforward. You should have a block something like:

```
readln(f,x);
sum:= sum + x;
```

9. The process starting with waitsome should not begin until the mouse is in the starting region. It is a good idea to do a sysbeep when the region is entered, so at least you will know that the random waiting time will begin to cycle out.

10. Such a stopwatch is actually useful, especially if you don't rely too much on animation (which hurts the resolution of the timer). The easiest implementation is to use the track stopwatch mode in which the sequence 'start-freeze-reset—start-freeze-reset—...' is assumed.

Program 2.1

```
program WriteTable;
(* DEMONSTRATES Screen Output OF CALCULATIONS *)
 var
  n : integer;
  x : real;

 procedure erase;
(* Activates and expands Text Window to fill screen *)
  var
   windowrect : rect;
 begin
  hideall;
  setrect(windowrect, 2, 35, 512, 342);
  settextrect(windowrect);
  showtext;
 end;

begin
 erase;
 writeln('       x        sin(x)      cos(x)      exp(x)      ln(x)');
 writeln;
 for n := 1 to 10 do
  begin
   x := n / 10;
   writeln(x : 12 : 6, sin(x) : 12 : 6, cos(x) : 12 : 6, exp(x) : 12 : 6,
    ln(x) : 12 : 6);
  end;
end.
```

Program 2.2

```
program DrawTest;
(* DEMONSTRATES A VARIETY OF GRAPHICS ROUTINES *)

 var
  r, textrect : rect;

 procedure CLEAR
(* Activates and expands Drawing Window to fill screen *)
  var
   windowrect : rect;
 begin
  hideall;
  setrect(windowrect, 2, 35, 512, 342);
  setdrawingrect(windowrect);
  showdrawing;
 end;
```

(continued)

```
 procedure message;
(* Clears away old message and prepares to write a new one *)
 begin
  backpat(white);
  eraserect(0, 0, 30, 300);
  moveto(4, 20);
 end;

begin
 CLEAR;
 showtext;
 writeln('Hit <Return> to continue');
 setrect(r, 100, 90, 200, 230);
 message;
 writedraw('Ready to frame');
 readln;
 framerect(r);
 showdrawing;
 message;
 writedraw('Ready to paint black oval');
 readln;
 penpat(black);
 paintoval(r);
 message;
 writedraw('Ready to invert rect');
 readln;
 invertrect(r);
 message;
 writedraw('Ready to paint LtGray oval');
 readln;
 backpat(LtGray);
 eraseoval(r);
 message;
 writedraw('Ready to invert oval');
 readln;
 invertoval(r);
 message;
 writedraw('Ready to invert screen');
 readln;
 invertrect(0, 0, 300, 511);
 message;
 writedraw('Ready to change text');
 readln;
 textsize(18);
 textface([bold]);
 message;
 writedraw('Ready to draw white line');
 readln;
 penpat(white);
 moveto(0, 40);
 lineto(200, 250);
 message;
 writedraw('Ready to draw thick line');
 readln;
 pensize(4, 4);
 moveto(0, 50);
 lineto(200, 260);
 message;
 writedraw('Ready to draw "xor" line');
 readln;
 penpat(black);
 penmode(patXor);
 moveto(0, 20);
 lineto(200, 230);
 message;
 writedraw('Done.');
end.
```

Program 2.3

```
program Echo;
(* READS KEYBOARD CHARS INTO AN ARRAY AND WRITES THEM *)
(* BACK OUT *)

 var
  letters : packed array[1..200] of char;
  size, n : integer;
  c : char;

begin
 hideall;
 showtext;
 size := 0;
 repeat
  while (not eoln) and (not eof) do
   begin
    size := size + 1;
    read(c);
    letters[size] := c;
   end;
  if (not eof) then
   begin
    size := size + 1;
    readln;
    letters[size] := chr(13);
   end;
 until eof;  (* Loop exits when <Enter> key is pressed *)
```

(continued)

```
   for n := 1 to size do
    begin
     c := letters[n];
     if ord(c) <> 13 then
      write(c)
     else
      writeln;
    end;
  end.
```

Program 2.4

```
program Switches;
(* DISPLAYS A 'PANEL' OF 8 'SWITCHES' WHICH REPRESENT BINARY *)
(* BITS *)

 var
  finger : point;
  lamp : array[0..7] of rect;
  on : array[0..7] of boolean;
  total, bit, summand : integer;
  change : boolean;

 procedure update (r : rect;
         state : boolean);
 begin
  if state then
   backpat(white)
  else
   backpat(black);
  eraserect(r);
  framerect(r);
 end;

begin
 hideall;
 showdrawing;
 moveto(4, 80);
 writedraw('Decimal Value:');
 for bit := 0 to 7 do
  begin
   on[bit] := false;
   setrect(lamp[bit], 10 + (7 - bit) * 15, 20, 20 + (7 - bit) * 15, 30);
   update(lamp[bit], on[bit]);
  end;
 repeat
  getmouse(finger.h, finger.v);
  change := false;
  for bit := 0 to 7 do
```

```
   begin
    if PtInRect(finger, lamp[bit]) and button then
     begin
      change := true;
      on[bit] := not on[bit];
      update(lamp[bit], on[bit]);
      while button do
       ;
     end;
   end;
  if change then
   begin
    summand := 1;
    total := 0;
    for bit := 0 to 7 do
     begin
      if on[bit] then
       total := total + summand;
      summand := 2 * summand;
     end;
    backpat(white);
    eraserect(70, 110, 90, 150);
    moveto(112, 80);
    writedraw(total : 1);
   end;
 until false;
end.
```

Program 2.5

```
program SlidePot;
(* DEMONSTRATES HOW TO MOVE AN OBJECT *)
(* WITH THE MOUSE *).

 var
  x, y, pot, oldpot, n : integer;

 procedure drawscale;
 begin
  moveto(40, 10);
  lineto(40, 120);
 end;

 function downandin (x, y, pot : integer) : boolean;
 begin
  if ((button) and (x >= 30) and (x <= 50) and (y >= pot - 3) and (y <= pot +
  3)) then
   downandin := true
  else
```

(continued)

```
   downandin := false;
 end;

 function volume (y : integer) : integer;
 begin
  volume := (115 - y) div 14;
 end;

 procedure processpot (var yf, yi : integer);
(* Animates the 'volume control' *)
 begin
  if (yf < 15) then
   yf := 15;
  if (yf > 115) then
   yf := 115;
  backpat(white);
  eraserect(yi - 3, 30, yi + 3, 50);
  drawscale;
  backpat(gray);
  eraserect(yf - 3, 30, yf + 3, 50);
  framerect(yf - 3, 30, yf + 3, 50);
  yi := yf;
  backpat(white);
  eraserect(121, 20, 160, 80);
  moveto(30, 144);
  writedraw(volume(yf) : 1);
 end;

begin
 hideall;
 showdrawing;
 textsize(24);
 pot := 110;
 oldpot := pot;
 processpot(pot, oldpot);
 repeat
  getmouse(x, y);
  if downandin(x, y, pot) then
   begin
    while button do
     begin
      getmouse(x, pot);
      processpot(pot, oldpot);
     end;
    setsoundvol(volume(pot));
    sysbeep(20);
   end;

   until false;
  end.
```

Program 2.6

```
program Piano;
(* SIMULATES A PIANO KEYBOARD OCTAVE *)

 var
  r : array[0..13] of rect;
  f : array[0..12] of longint;
  i, j : integer;
  pt : point;

 procedure CLEAR;
(* Activates and expands Drawing window to fill screen *)
  var
   windowrect : rect;
 begin
  hideall;
  setrect(windowrect, 2, 35, 512, 342);
  setdrawingrect(windowrect);
  showdrawing;
  backpat(ltgray);
  eraserect(0, 0, 600, 600);
 end;

 procedure init;
  var
   i : integer;
 begin
  for i := 0 to 7 do
   setrect(r[i], 50 + 50 * i, 50, 100 + 50 * i, 250);
  for i := 8 to 13 do
   setrect(r[i], 85 + 50 * (i - 8), 50, 115 + 50 * (i - 8), 150);
  for i := 0 to 13 do
   begin
    if (i < 8) then
     begin
      backpat(white);
      eraserect(r[i]);
      framerect(r[i]);
     end;
    if ((i > 7) and (i <> 10)) then
     begin
      backpat(dkgray);
      eraserect(r[i]);
     end;
   end;
  for i := 0 to 12 do
   f[i] := trunc(512 * exp(i * ln(2) / 12));
 end;
```

(continued)

```
 function permute (j : integer) : integer;
  var
   i : integer;
 begin
  case j of
   0 :
    i := 0;
   1 :
    i := 2;
   2 :
    i := 4;
   3 :
    i := 5;
   4 :
    i := 7;
   5 :
    i := 9;
   6 :
    i := 11;
   7 :
    i := 12;
   8 :
    i := 1;
   9 :
    i := 3;
   11 :
    i := 6;
   12 :
    i := 8;
   13 :
    i := 10;
  end;
  permute := i;
 end;

 procedure play (i : integer);
 begin
  note(f[i], 200, 10);
 end;

begin
 CLEAR;
 init;
 repeat
  while not button do
   getmouse(pt.h, pt.v);
  j := -1;
  i := 13;
  repeat
   while ((j < 0) and (i >= 0)) do
```

```
    begin
     if ((PtInRect(pt, r[i])) and (i <> 10)) then
      j := i;
     i := i - 1;
    end;
   if (j <> -1) then
    begin
     i := permute(j);    (* gets actual key number 0-11 *)
     play(i);
    end;
  until not button;
  until false;
 end.
```

Program 2.7

```
program PrintTable;
(* DEMONSTRATES Printer Output OF CALCULATIONS*)

 var
  f : text;
  n : integer;
  x : real;

begin
 rewrite(f, 'Printer:');
 writeln(f, '    x      sin(x)    cos(x)    exp(x)    ln(x)');
 for n := 1 to 10 do
  begin
   x := n / 10;
   writeln(f, x : 12 : 6, sin(x) : 12 : 6, cos(x) : 12 : 6,exp(x) : 12 : 6,
   ln(x) : 12 : 6);
  end;
end.
```

Program 2.8

```
program SelfPrint;
(* SENDS ITSELF TO IMAGEWRITER *)

 var
  f, g : text;
  c : char;

begin
 reset(g, 'selfprint');
 rewrite(f, 'Printer:');
 repeat
  while (not eoln(g)) and (not eof(g)) do
```

(continued)

```
   begin
    read(g, c);
    write(f, c);
   end;
  if not eof(g) then
   begin
    readln(g);
    writeln(f);
   end;
 until eof(g);
 close(f);
 close(g);
end.
```

Program 2.9

```
program SelfAccess;
(* ACCESSES ITSELF AND WRITES ITSELF TO Text Window *)

 var
  f : text;
  c : char;
  windowrect : rect;

begin
 reset(f, 'selfread');
 hideall;
 setrect(windowrect, 2, 35, 512, 342);
 settextrect(windowrect);
 showtext;
 repeat
  while (not eoln(f)) and (not eof(f)) do
   begin
    read(f, c);
    write(c);
   end;
  writeln;
  readln(f);
 until eof(f);
 close(f);
end.
```

Program 2.10

```
program SelfModify;
(* INSERTS A COMMENT INTO ITS OWN DISK IMAGE *)

 var
  f : text;
```

```
   symbols : packed array[1..2000] of char;
   n, z, size, linecount : integer;

 begin
  reset(f, 'selfmodify');
  showtext;
  n := 0;
  linecount := 0;
  writeln('Now reading from "selfmodify"...');
  repeat
   while (not eoln(f)) and (not eof(f)) do
    begin
     n := n + 1;
     read(f, symbols[n]);
    end;
   n := n + 1;
   linecount := linecount + 1;
   symbols[n] := chr(13);
   writeln('Just read line ', linecount : 1);
   readln(f);
  until eof(f);
  size := n;
  close(f);
  rewrite(f, 'selfmodify');
  linecount := 0;
  for n := 1 to size do
   begin
    z := ord(symbols[n]);
    if z <> 13 then
     write(f, symbols[n])
    else
     begin
      linecount := linecount + 1;
      writeln(f);
      if linecount = 1 then
       writeln(f, '(* AUTOMATED COMMENT ! *)');
     end;
   end;
  close(f);
  writeln('Open "selfmodify" from disk to see the change');
 end.
```

Program 2.11

```
program ReactionTime;
(* USES 'TICKCOUNT' TO TIME USER MOUSE ACTION *)

 var
  t : Longint;
```

(continued)

```
  x, y, xx, yy : integer;

 procedure CLEAR;
(* Activates and expands Drawing Window to fill screen *)
  var
   windowrect : rect;
 begin
  hideall;
  setrect(windowrect, 2, 35, 512, 342);
  setdrawingrect(windowrect);
  showdrawing;
 end;

 procedure waitsome;
(* Waits a random amount of time *)
  var
   n : integer;
   u : Longint;
 begin
  n := random mod 600;
  u := tickcount;
  while tickcount - u < n do
 end;

 procedure drawsome (var x, y : integer);
(* Draws a randomly placed square *)
 begin
  x := random mod 360 + 100;
  y := random mod 260;
  framerect(y, x, y + 10, x + 10);
 end;

begin
 CLEAR;
 repeat
  eraserect(0, 100, 300, 511);
  waitsome;
  drawsome(x, y);
  sysbeep(5);
  t := tickcount;
  repeat
   getmouse(xx, yy);
  until (xx >= x) and (yy >= y) and (xx <= x + 10) and (yy <= y + 10);
  eraserect(0, 0, 90, 90);
  moveto(10, 10);
  writedraw(tickcount - t - 2 : 1);
 until false;
end.
```

Program 2.12

```
program Clock (output);
(* DRAWS A CLOCK WITH ANIMATED HOUR, MINUTE *)
(* AND SECOND HANDS *)

 const
  pi = 3.14159265359;

 var
  time : datetimerec;
  second, minute, hour, oldsecond, oldminute, oldhour : integer;

 procedure CLEAR;
(* Activates and expands Drawing Window to fill screen *)
  var
   windowrect : rect;
 begin
  hideall;
  setrect(windowrect, 2, 35, 512, 342);
  setdrawingrect(windowrect);
  showdrawing;
 end;

 procedure MAP (x, y : real;
         var hor, ver : integer);
(* Translates real values into integer coordinates *)
 begin
  if abs(x) > 1.9 then
   x := x / abs(x) * 1.9;
  hor := 255 + trunc(x * 130);
  if abs(y) > 1 then
   y := y / abs(y);
  ver := 138 - trunc(y * 130);
 end;

 procedure DRAW (x, y : real);
(* Draws a line from current position to point x, y *)
  var
   h1, v1 : integer;
 begin
  MAP(x, y, h1, v1);
  lineto(h1, v1);
 end;

 procedure SHIFT (x, y : real);
```

(continued)

```
(* Moves pen to x, y without drawing *)
  var
   h, v : integer;
 begin
  MAP(x, y, h, v);
  moveto(h, v);
 end;

 procedure CIR (x, y, r : real);
(* Draws a circle centered at x, y with radius r *)
  var
   h2, v2, h1, v1 : integer;
 begin
  MAP(x - r, y + r, h2, v2);
  MAP(x + r, y - r, h1, v1);
  frameoval(v2, h2, v1, h1);
 end;

 procedure clockface;
(* Draws the clockface *)
  const
   r = 0.85;
   d = 0.10471975512;
  var
   angle : real;
   n : integer;
 begin
  pensize(3, 3);
  penpat(gray);
  CIR(0, 0, 1);
  SHIFT(-0.97, -0.05);
  writedraw('9');
  SHIFT(-0.07, 0.91);
  writedraw('12');
  SHIFT(0.91, -0.05);
  writedraw('3');
  SHIFT(-0.03, -0.97);
  writedraw('6');
  penpat(black);
  for n := 0 to 59 do
   begin
    angle := pi * (15 - n) / 30;
    if n mod 5 = 0 then
     pensize(3, 3)
    else
     pensize(1, 1);
    SHIFT(0.82 * cos(angle), 0.82 * sin(angle));
    DRAW(0.87 * cos(angle), 0.87 * sin(angle));
   end;
 end;
 procedure hand (handtype, count : integer;
         visible : boolean);
(* Animates the clock hands *)
  var
   angle, len : real;
 begin
  case handtype of
   0 :
    begin
     pensize(1, 1);
     len := 0.8;
    end;
   1 :
    begin
     pensize(2, 2);
     len := 0.7;
    end;
   2 :
    begin
     pensize(3, 3);
     len := 0.5;
    end;
  end;
  angle := pi * (15 - count) / 30;
  if visible then
   penpat(black)
  else
   penpat(white);
  SHIFT(0, 0);
  DRAW(len * cos(angle), len * sin(angle));
 end;

 procedure newface;
(* Updates clock time *)
 begin
  minute := time.minute;
  hour := trunc(5 * time.hour + minute / 12);

  if (oldhour <> hour) then
   hand(2, oldhour, false);
  if (oldminute <> minute) then
   hand(1, oldminute, false);
  hand(0, oldsecond, false);
  oldhour := hour;
  oldminute := minute;
  oldsecond := second;
  if (hour - minute) * (minute - second) = 0 then
   hand(1, minute, false);
  if (minute - hour) * (hour - second) = 0 then
   hand(2, hour, false);
```

(continued)

```
   if (minute - second) * (second - hour) = 0 then
    hand(0, second, false);
   hand(0, second, true);
   hand(1, minute, true);
   hand(2, hour, true);
  end;

begin
 CLEAR;
 clockface;
 gettime(time);
 oldminute := time.minute;
 oldhour := trunc(5 * time.hour + oldminute / 12);
 oldsecond := time.second;
 repeat
  gettime(time);
  second := time.second;
  if second <> oldsecond then
   newface;
 until false;
end.
```

3 Numerical Programs

THEME: You inspect, modify, and create various programs whose primary function is to calculate. In this chapter the text becomes rather technical, but the exercises contain many simple projects.

GOALS: You will learn how to incorporate standard concepts of numerical analysis into your Pascal programs, and how to solve those numerical problems for which a computer is well suited.

LIBRARIES USED: Math.lib, Num.lib, Matrix.lib.

REFERENCE MATERIALS: Textbooks in the areas of numerical and real analysis, calculus, differential equations, algebra, number theory.

Sequences of Integers

The following are some simple integer sequences and some suggestions for writing programs which compute these sequences. The sequence of factorials:

```
1, 2*1, 3*2*1, 4*3*2*1, . . .
```

of which the n-th term is denoted n! (n factorial), is a tried and true beginning sequence in programming studies. One way to generate the sequence is to establish two variables. Call one variable *n* and the other *factorial.* Then go through the following steps:

Step 1 Set n: = 1 and factorial: = 1.
Step 2 Multiply factorial by n.
Step 3 Increment n by 1.
Step 4 Go back to step 2.

This logic is simple but contains the seeds from which much more complicated calculations can grow. The logic steps are a form of *recursion*—in that they go back to a previous step. There is a way to generate factorials with what is called a *recursive function*; that is, one that calls itself. The two methods of generating factorials—steps 1-4 above and a recursive function—are shown in Program 3.1(a) Factorial and 3.1(b) RecFactorial. No output is shown for these programs since it is just the familiar list of factorials: 1,2,6,24,120,. . . .

The next sequence is prime numbers. These are numbers:

2, 3, 5, 7, 11, 13, 17, . . .

characterized by having exactly two divisors: themselves and 1. There are several ways to generate prime numbers. In fact the primes are all the more interesting because there are so many ways of generating them.

Method 1	Find out, for given integer p, whether it has any divisors greater than 1 but not exceeding sqrt(p)—the square root of p. If there are no such divisors, then p is prime.
Method 2	Start with a block of consecutive integers and strike out all the multiples of 2 (except 2 itself), all the multiples of 3 (except 3), and so on. What is left is the set of primes in the original block. This is called the Sieve of Eratosthenes.
Method 3	Use a theorem from the theory of numbers to test for primality. There are various elegant theorems of this type.

For Method 1, you will need a loop something like:

```
(* p is under test for primality *)
divisor: = 2;
while (p mod divisor< >0) and (divisor < p/divisor) do
  if divisor=2 then divisor: = divisor + 1 else
    divisor: = divisor + 2;
```

Notice that the use of the sqrt function is avoided by simply testing whether:

```
divisor < p/divisor
```

since this condition becomes false when divisor reaches sqrt(p). Another implementation of Method 1 is that of recursion to the same kind of divisor checks. These two approaches are embodied in Program 3.2(a) Primes and 3.2(b) RecPrimes.

For Method 2, you need an array of some sort in which to store the fact of a number being 'sieved out.' You might start with:

```
const max = 500;      (* Look for all primes < max *)
var sieve: array [1..max] of boolean;

  for n: = 1 to max do sieve[n]: = true;
  divisor: = 2;
  repeat
    k: = 2*divisor;
    while k=max do
      begin
      sieve[k]: = false;
      k: = k + divisor;
      end;
    repeat
      divisor: = divisor + 1;
    until sieve[divisor];
until divisor> sqrt(max);
```

After this segment executes, you would print out those values of divisor for which sieve[divisor] is true. Method 2 is embodied in Program 3.3 Sieve.

For Method 3, you can use a host of primality checks. One of these is Fermat's Theorem: if p is prime, then:

$$2^{(p-1)} \bmod p = 1$$

Unfortunately, the converse is not true. There are composite (non-prime) numbers p having the property. In the number theory library of this book, a procedure for computing the general expression:

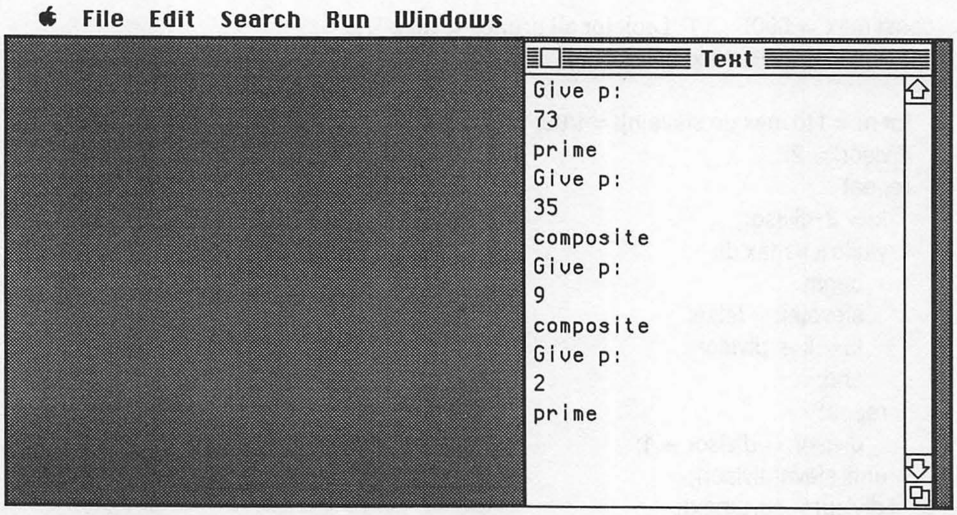
$$x^y \bmod p$$

is given as function PMOD. This can be used for the Fermat test, which you must remember is not a total test; you can only conclude that a number is not prime (is composite) if $2^{(p-1)} \bmod p <> 1$. The example used in this chapter for Method 3 involves Wilson's Theorem, which is a total, if relatively inefficient, test for primality. This states that p is prime if and only if:

$$(p-1)! \bmod p = p-1$$

For example, if $p = 7$, then $6! = 6*5*4*3*2*1 = 720$, and since 721 is divisible by 7, this factorial mod p is -1, so 7 is prime. This example has been chosen to illustrate a recurring theme in computer calculations: that of continual reduction. Instead of computing:

$$(p-1)! = (p-1)*(p-2)*(p-3)* \ldots *2*1$$



```
Give p:
73
prime
Give p:
35
composite
Give p:
9
composite
Give p:
2
prime
```

Figure 3.1 | Typical Wilson output

which gets very large very fast, you compute the modulus at each step, since mod p is to be taken for the factorial. Thus, it is much better to compute in the style of Program 3.4, Wilson,in which the ideas of recursive factorial and primality test have been joined. Typical output is shown in Figure 3.1.

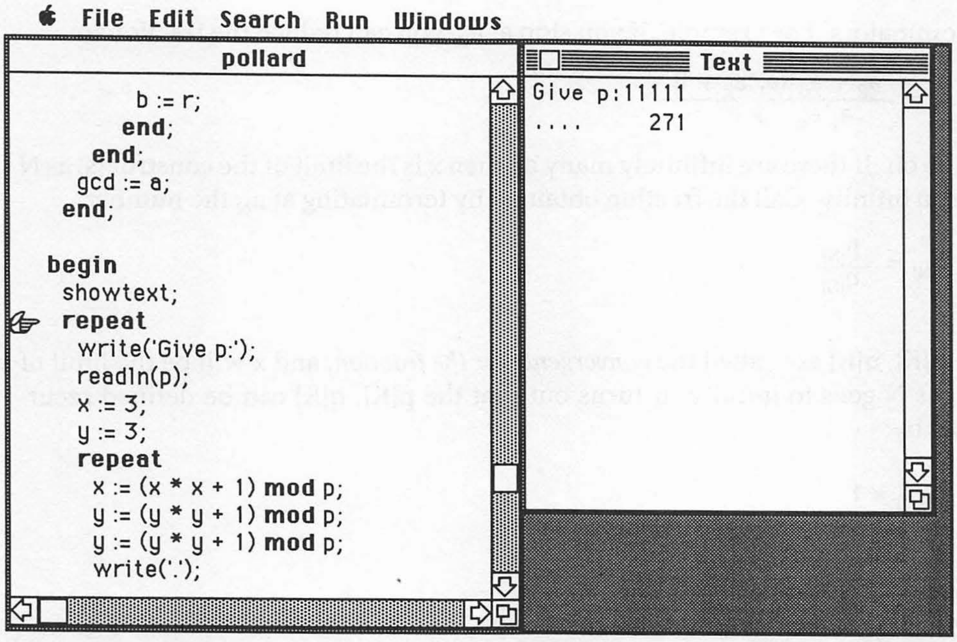
Closely related to studies of the sequence of prime numbers is the problem of factoring integers into prime factors. As with prime generation, there are many methods for finding factors. For a number n to be factored, two such methods are:

Method 1	Scan through the numbers less than n, finding some p such that n mod p = 0.
Method 2	Use a theorem or technique from the theory of numbers which will yield factors of n.

The example provided in Program 3.5 refers to Method 2. The technique is called the *Pollard Rho* factoring, and uses a curious integer sequence defined by:

```
x[1]: = 3;
x[m+1]: = x[m]*x[m] + 1        for m = 1,2, . . .
```

This sequence first computes the greatest common divisor of $x^{2m} - x^m$ and the number n to be factored. When this common divisor is greater than 1 but less than n, it is a factor. The method is somewhat difficult to understand but very fast. The greatest common divisor algorithm (actually, Euclid's Algorithm) is embodied in



```
        b := r;
      end;
    end;
    gcd := a;
  end;

begin
  showtext;
  repeat
    write('Give p:');
    readln(p);
    x := 3;
    y := 3;
    repeat
      x := (x * x + 1) mod p;
      y := (y * y + 1) mod p;
      y := (y * y + 1) mod p;
      write('.');
```

```
Give p:11111
....      271
```

Figure 3.2 | Pollard output

the function GCD of the Num.lib. Figure 3.2 shows a typical output in which the number 11111 = 41 * 271 is factored.

Of course, these programs are not ideal for Macintosh Pascal, where speed has been traded for greater ease of use. But it is valuable to be able to work with sequences, for they underlie the ideas of calculus and differential equations which are critical components to the modeling of physical phenomena.

Another class of integer sequences are those for which an additive recursion relation is given, as opposed to the multiplicative one for the factorials and the complicated, obscure one for primes. The famous Fibonacci numbers are such a sequence (see exercises at end of chapter). A more general case are the so-called convergents to a continued fraction. One may expand any positive real number x as a continued fraction:

$$x = a_0 + \cfrac{1}{a_1 + \cfrac{1}{a_2 + \cfrac{1}{a_3 + \dots}}}$$

The way to evaluate such a construct is to stop temporarily at the term a_N and compute the numerator and denominator by successive multiplication by denominators. For example, if you stop at a_2, you can reduce the fraction to:

$$x = \frac{a_2 + a_0*(a_1*a_2 + 1)}{a_1*a_2 \ + 1}$$

and so on. If there are infinitely many a_K then x is the limit of the constructs, as N goes to infinity. Call the fraction obtained by terminating at a_N the number:

$$x_{[N]} = \frac{p_{[N]}}{q_{[N]}}$$

The p[K], q[K] are called the *convergents for the fraction*, and x will be the limit of x[N] as N goes to infinity. It turns out that the p[K], q[K] can be defined recursively by:

$$\begin{aligned} p_{[-1]} &= 1 \\ q_{[-1]} &= 0 \\ p_{[0]} &= a_0 \\ q_{[0]} &= 1 \end{aligned}$$

$$\begin{aligned} p_{[K]} &= a_K * p_{[K-1]} + p_{[K-2]} \qquad K = 1,2,\ldots \\ q_{[K]} &= a_K * q_{[K-1]} + q_{[K-2]} \end{aligned}$$

There are two approaches to programming with continued fractions:

Approach 1. Given a real number x, find the elements a_0, a_1, a_2, . . . of its continued fraction.

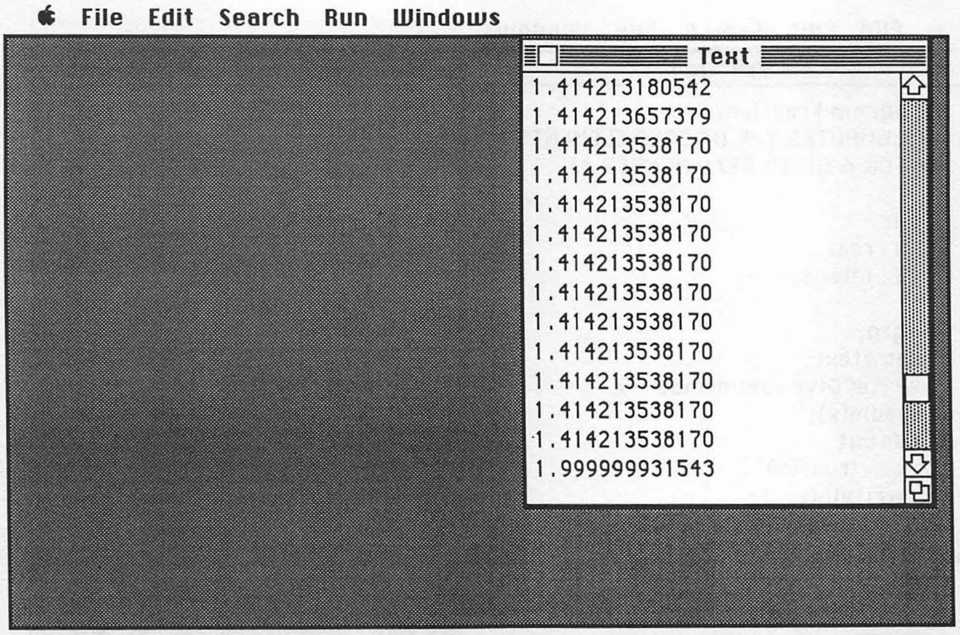
Approach 2. Given the integer elements a_0, a_1, a_2, . . . calculate a value for the continued fraction x.

An interesting thing about these fractions is that they are very efficient in Approach 2. This is especially true if there is a clear pattern to the integer sequence of elements a_0, a_1, ... If so, tight programs may be written which calculate important numbers. The programmer uses some algorithm to calculate the Kth element a_K, then uses the elements to compute the convergents $p_{[K]}$, $q_{[K]}$; and finally estimates the number x with terms:

$$\frac{p_{[K]}}{q_{[K]}}$$

Program 3.6, RootOfTwo, uses Approach 2 to compute the square root of 2, which has the infinite continued fraction expansion:

$$\text{sqrt}(2) = 1 + \cfrac{1}{2 + \cfrac{1}{+2 + \cfrac{1}{2 + \ldots}}}$$



```
1.414213180542
1.414213657379
1.414213538170
1.414213538170
1.414213538170
1.414213538170
1.414213538170
1.414213538170
1.414213538170
1.414213538170
1.414213538170
1.414213538170
1.414213538170
1.999999931543
```

Figure 3.3 RootOfTwo output

The program computes that part of the fraction having all the twos, then adds one to get the correct root. Figure 3.3 shows typical output from this program.

Program 3.7, Fraction,uses Approach 1 and finds, for a given real number, the elements of the continued fraction. It is of interest that many important numbers have patterns in their expansions. Figure 3.4 shows the pattern for the number 'e'. You could, in principle, use the integer sequence output from program Fraction as input for a program of Approach 2 to recover the original real number.

Limits and Sums

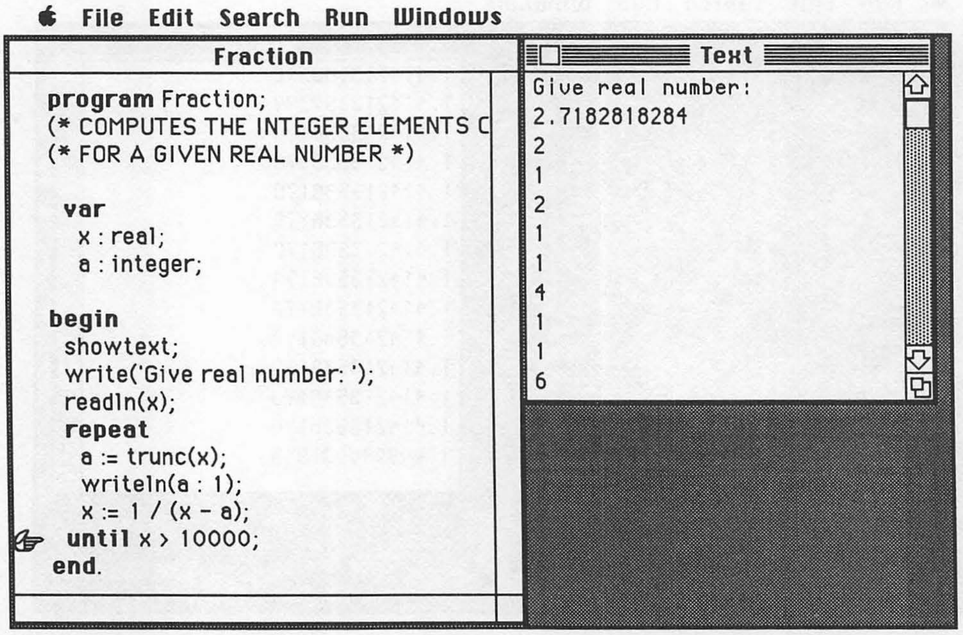
Having looked at some integer sequences, we now turn to rational sequences. These are distinctly different from integer sequences in that they can have limits. For example, the sequence of rationals:

$$\frac{2}{1} \quad \frac{3}{2} \quad \frac{4}{3} \quad \frac{5}{4} \quad \ldots \quad \frac{(n+1)}{n} \quad \ldots$$

has the limit 1, or:

$$\lim_{n->\infty} \frac{(n+1)}{n} = 1$$

$$e = \lim_{n->\infty} 1+\frac{1}{n}^{n}$$



```
program Fraction;
(* COMPUTES THE INTEGER ELEMENTS C
(* FOR A GIVEN REAL NUMBER *)

  var
   x : real;
   a : integer;

 begin
  showtext;
  write('Give real number: ');
  readln(x);
  repeat
   a := trunc(x);
   writeln(a : 1);
   x := 1 / (x - a);
  until x > 10000;
 end.
```

```
Give real number:
2.7182818284
2
1
2
1
1
4
1
1
6
```

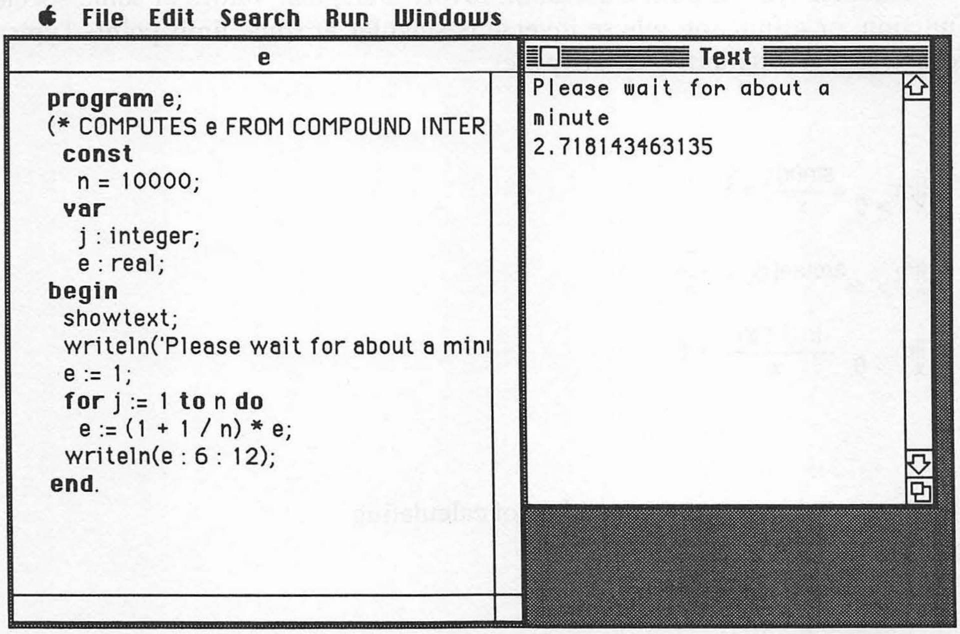
Figure 3.4 | Fraction output

Another example is the limit of a continued fraction as you saw before, where the ratio of convergents $\frac{p[K]}{q[K]}$ approaches the limit x. Consider a famous formula which defines the number 'e' in terms of a limit of a rational sequence:

$$e = \lim_{n->\infty} 1+\frac{1^n}{n}$$

You may see how this works by writing out a few cases, remembering that e = 2.718 . . .:

n	$\frac{(n+1)}{n}$	$\frac{(N+1)^n}{n}$
1	$\frac{2}{1}$	2
2	$\frac{3}{1}$	2.25
3	$\frac{4}{1}$	2.37
.	.	.
.	.	.
.	.	.
99	$\frac{100}{99}$	2.70



```
program e;
(* COMPUTES e FROM COMPOUND INTER
 const
  n = 10000;
 var
  j : integer;
  e : real;
begin
 showtext;
 writeln('Please wait for about a min
 e := 1;
 for j := 1 to n do
  e := (1 + 1 / n) * e;
 writeln(e : 6 : 12);
end.
```

```
Please wait for about a
minute
2.718143463135
```

Figure 3.5 | e output

The task of programming this calculation—taking a sufficiently large n to get a good estimate for e—can be done in several ways. There are at least three methods for doing so:

Method 1 (**Dangerous**) Compute $(n+1)^n$ and n^n and divide the former by the latter. This is fine in theory, but your computer might 'crash' with high powers like n^n.

Method 2 (**Inelegant**) Observe that $\frac{(1+1)}{n^n}$ = exp(n*ln $\frac{(1+1)}{n}$) and compute the required transcendental functions. This is also fine in theory, but if you were allowed to use recourse to the exp function itself you might as well have computed just exp(1). One should proceed by treating the limit for what it is: the limit of a sequence of rational numbers, and avoid the use of special functions for this problem.

Method 3 (**Simple and Direct**) Compute $\frac{(n+1)}{n}$ and multiply n of these identical terms together.

The last method has the features you want for limit calculations: it is easy to state, and no numbers involved will be especially large. A program to approximate "e" using Method 3 is shown in Program 3.8, E, with output as Figure 3.5.

Another type of limit calculation involves singular values of some special function, or a function whose inverse is singular at some limit point. Typical formulas that programs can verify are:

$$\lim_{x \to 0} \frac{\sin(x)}{x} = 1$$

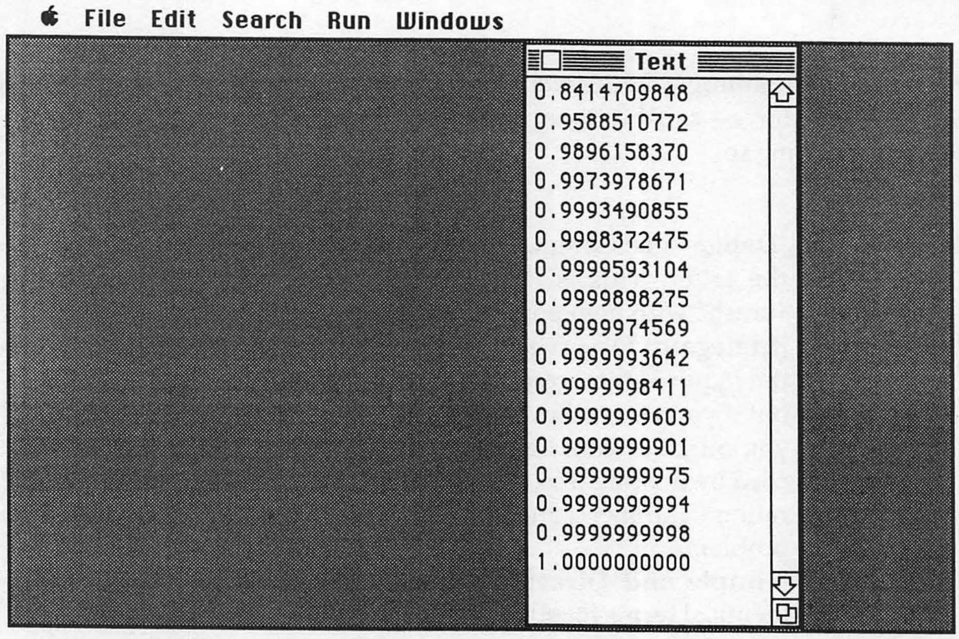
$$\lim_{x \to \infty} \arctan(x) = \frac{\pi}{2}$$

$$\lim_{x \to 0} \frac{\ln(1+x)}{x} = 1$$

Program 3.9, Limit, shows a method of calculating:

$$\frac{\sin(x)}{x}$$

as x approaches zero. Output is shown in Figure 3.6.



```
0.8414709848
0.9588510772
0.9896158370
0.9973978671
0.9993490855
0.9998372475
0.9999593104
0.9999898275
0.9999974569
0.9999993642
0.9999998411
0.9999999603
0.9999999901
0.9999999975
0.9999999994
0.9999999998
1.0000000000
```

Figure 3.6 | Limit output

Note that the initial x is consecutively halved until a condition such as:

$x < 1e-18$

is met. It is important to be careful with loop conditions; for example, if you are not sure whether x will be negative, a proper end of loop statement would be:

```
until abs(x) < epsilon
```

where epsilon is a small positive constant. Generally speaking, the nature of program loops depends greatly on the type of limit desired. But one thing to keep in mind is that machine constraints, such as the smallest allowed real, are relevant. For Macintosh Pascal, which supports the SANE Floating Point libraries for its real-number calculations, the exponent range exceeds:

$1e-1000$ to $1e+1000$

and maximum output is 12 significant figures. These features are sufficient for most scientific applications.

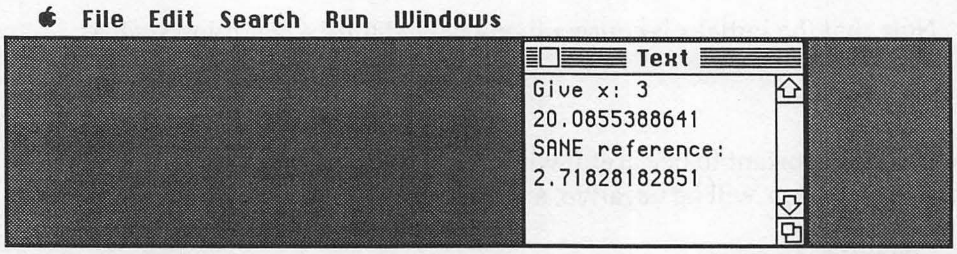
Closely related to limits are infinite sums. This is because an infinite sum, if properly defined and convergent, can be though of as the limit of larger and larger numbers of terms summed. Perhaps the simplest and most illustrative example of a useful sum is the series for exp(x):

$$\exp(x) = 1x + \frac{x^2}{2!} + \frac{x^3}{3!} + \ldots$$

Note that this provides an alternative definition for e as the number exp(1). As often happens with these numerical calculations, there is more than one way that programmers tend to proceed.

Method 1	(**Wrong**) For various n, compute x^n and n!, divide the former by the latter, and add all these together. This method has the same difficulty that was mentioned before: terms can blow up.
Method 2	(**Right**) Call the first term, 1, the 0-th term. Then multiply by $\frac{x}{n}$ to get the n-th term, and add up all the terms.

Method 2 is exemplified in cases like this: the term $\frac{x^3}{3!}$ is obtained from the term $\frac{x^2}{2}$ by multiplying the latter by $\frac{x}{3}$. Program 3.10, Expo, shows how e can be calculated with a series, and how this compares with exp[1], which is a floating-point direct calculation from the SANE library. The resident function exp[1] is computed in a manner very similar to the sum being analyzed. The output is shown in Figure 3.7.



```
Give x: 3
20.0855388641
SANE reference:
2.71828182851
```

Figure 3.7 | Expo output

One word of caution: there are many ways of terminating approximations to infinite sums. Criteria include: stop when the sum is sufficiently near to a known book value, stop when the next summand is sufficiently near zero, stop when N terms have been summed, and so on. The exercises for this chapter involve various criteria for stopping summations.

Differential Calculus

The fundamental entity of differential calculus—the derivative—is a special kind of limit, and observations from the last section apply. If f(x) is a function, the derivative (when it exists) is the limit:

$$\frac{df}{dx} = f' = \lim_{dx->0} \frac{f(x+dx) - f(x)}{dx}$$

Familiar derivatives are:

$$\frac{d(x^n)}{dx} = n\,x^{n-1} \quad ; n \text{ constant}$$

$$\frac{d(\exp(ax))}{dx} = a \exp(ax) \quad ; a \text{ constant}$$

$$\frac{d(\ln(x))}{dx} = \frac{1}{x}$$

$$\frac{d(\sin(x))}{dx} = \cos(x)$$

$$\frac{d(\cos(x))}{dx} = -\sin(x)$$

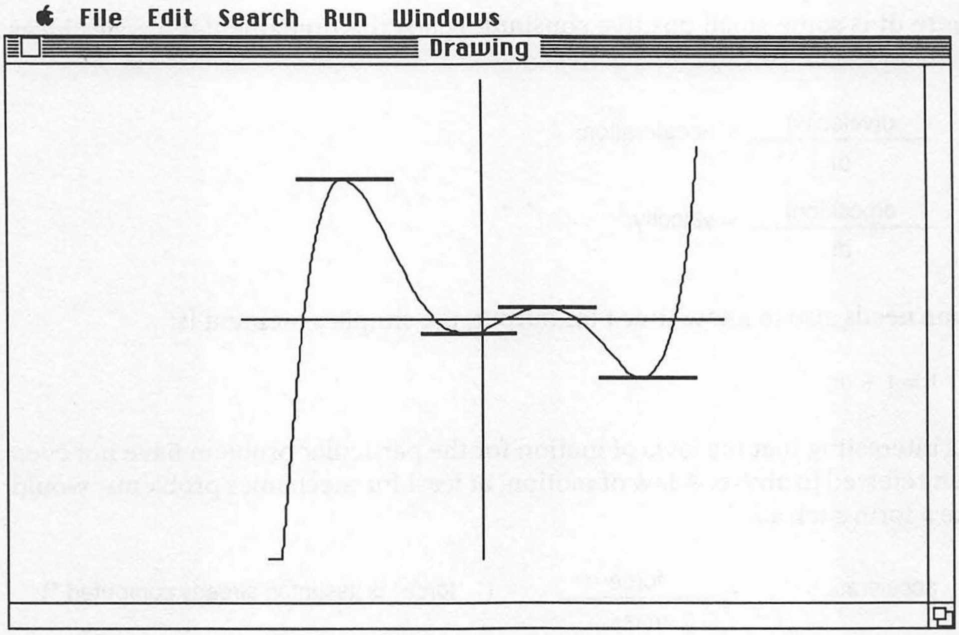


Figure 3.8 | PolyPlot output

Each of these can be shown by using techniques of limit analysis. There are two main approaches to the incorporation of differential calculus in programs.

Approach 1. Directly calculate directly the derivative of a Pascal function, either a key function or, more commonly, a user-defined function.

Approach 2. Use differential properties of some phenomenon, and embody these properties in program statements. This programming usually involves what are called *differential equations*.

For Approach 1, an example is Program 3.11, PolyPlot, which draws a fifth-degree polynomial on the graphics screen and also indicates the critical points (points at which $\frac{df}{dx} = 0$). The graphics output is shown in Figure 3.8.

Approach 2 is by far the more common one for programming differential calculus. An important case is that of dynamical modeling, in which quantities change with respect to a time variable. The basic idea is that if you have an acceleration that is computable for a given time t, then the program segment:

```
velocity:= velocity + acceleration * dt;
position:= position + velocity * dt;
```

where dt is some small positive constant, echos the fundamental physics ideas that:

$$\frac{d(\text{velocity})}{dt} = \text{acceleration};$$

$$\frac{d(\text{position})}{dt} = \text{velocity};$$

If one needs also to know time t for output, the simple statement is:

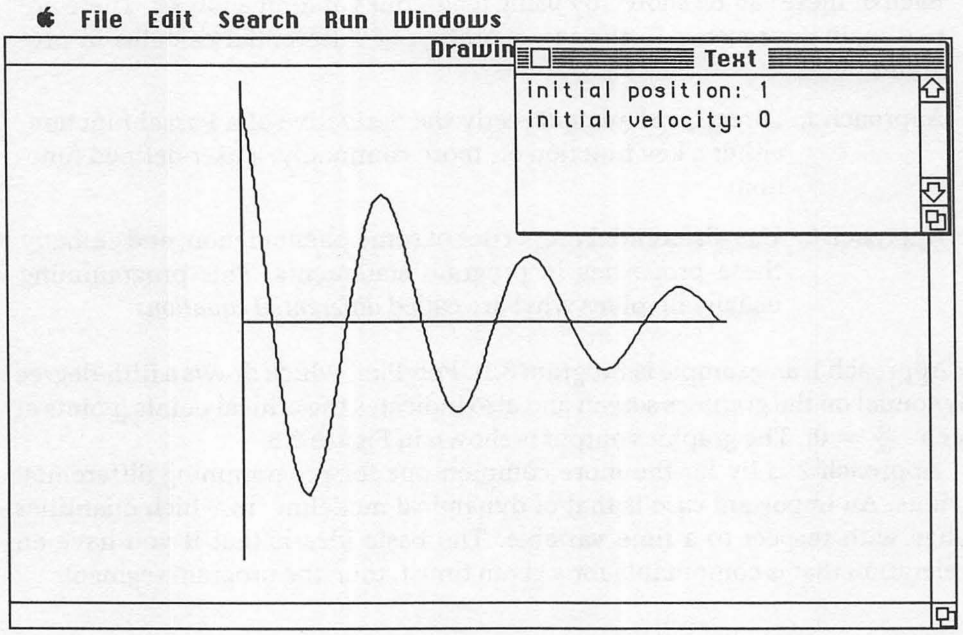
```
t: = t + dt;
```

It is interesting that the laws of motion for the particular problem have not even been referred to above. A law of motion, at least for mechanics problems, would take a form such as:

$$\text{acceleration} = \frac{\text{force}}{\text{mass}}; \qquad (\text{* 'force' is assumed already computed *})$$

$$\text{acceleration} = f(\text{position}); \qquad (\text{* f is some function *})$$

$$\text{acceleration} = f(\text{time}); \qquad (\text{* f is some function *})$$



```
initial position: 1
initial velocity: 0
```

Figure 3.9 | Oscillator output

and it should be remembered that the calculus segment involving dt multiplications is not part of the law of motion *per se*.

Program 3.12, Mechanics, gives a useful skeleton upon which to build differential equation solvers.

Program 3.13, Oscillator, shows how to solve the differential equation:

$$m\left(\frac{d}{dt}\right)\left(\frac{d}{dt}\right)x + 2b\frac{dx}{dt} + kx = 0$$

which is the equation for a damped oscillator: mass m on a spring of Hooke's constant k, with damping coefficient b. Figure 3.9 shows the output of this program. The graphics.lib has been used to graph real numbers onto the screen. This technique, which amounts to mapping the region:

$$-1 < x < 1$$
$$-1 < y < 1$$

into a convenient square in the middle of the screen, is covered in more detail in Chapter 4.

Newton's Method

There is an ingenious method, due to Isaac Newton, of solving an equation by successive applications of the derivative concept. The idea is to use a function's own derivative to estimate the rate of change of that function, and thereby predict other function values. Say you wish to solve the equation:

$$f(x) = 0$$

where the function f is known, but might be of a transcendental class which prevents analytic solutions. If, in absence of known analytic solutions, you choose an initial x[0], it is likely that:

$$f(x_0) < > 0$$

Otherwise, you would have a solution. But the actual value $f(x_0)$ can be used together with the derivative:

$$f' = \frac{df}{dx}$$

to estimate a better x value, call it x_1. The essential algebra is to define:

$$x_1 = x_0 - \frac{f(x_0)}{f'(x_0)}$$

and in general, to relate the (n + 1)-st estimate to the n-th estimate by:

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}$$

For reasonable choices of initial x_0, the sequence:

$$x_0, x_1, x_2, x_3, \ldots x_n$$

converges to a solution of the equation $f(x) = 0$. That is to say, $f(x_n)$ gets closer and closer to zero as you continue to iterate the equation above for x_{n+1}.

There are several ways to see how this works. One is to observe that if you actually had an x_n with $f(x_n) = 0$ (or, since this is extremely unlikely for most problems, just a very good guess x_n with f being very close to zero) the iteration equation for x_{n+1} would essentially read:

$$x_{n+1} = x_n$$

so the guess would be stable. This is not a proof that Newton's Method finds solutions, but is suggestive: if the initial choice x_0 is sufficiently shrewd, the approximate equality $x_{n+1} = x_n$ will get very good very rapidly as both of these terms approach the correct answer. Many numerical analysis texts discuss the method more rigorously.

Another way to see how Newton's Method works is to note that the iteration formula for x_{n+1} involves computation of the tangent slope line to $f(x_n)$, and 'shooting a tangent ray' with the slope $f'(x_n)$ to intersect the x-axis. The new guess is at this point of intersection.

For an example of Newton's Method, consider the following question.

Can you divide two real numbers using operations other than divide? The answer is yes, you can, by doing Newton's Method. Clearly, it is enough to be able to compute the reciprocal $\frac{1}{z}$ without recourse to division *per se*. Then any quotient of reals, say $\frac{y}{z}$, can be written:

$$\frac{y}{z} = y * \frac{1}{z}$$

Thus, you only need to analyze a reciprocating function. You need to solve, for a given z, the equation:

$$f(x) = \frac{1}{x} - z = 0$$

Now the derivative is:

$$f' = \frac{-1}{(x*x)}$$

so that the Newton iteration is:

$$x_{n+1} = x_n \quad \frac{\dfrac{1}{x_n} - z}{\dfrac{-1}{(x_n * x_n)}}$$

$$= 2 * x_n - z * x_n * x_n$$

For appropriate initial guesses x_0 (see exercises at the end of this chapter) x_n will converge to $\frac{1}{z}$, and from the last equation, you can see that this is done using only multiplication and subtraction.

Program 3.14, Newton, shows how to solve f(x) = 0 for arbitrary user-defined functions f. Note that the derivative df is computed by force, using limit ideas. This is because the program skeleton is written without yet knowing the user function. When you choose an f, especially a simple one, it is better to place the analytical derivative into the function block for df. For example, if f(x) = x*sin(x) − 1, the skeleton program Newton will work, but better accuracy is obtained by replacing the limit calculation within the df block by the line:

```
df:= sin(x) + x * cos(x);
```

Program 3.15, Solver, finds solutions to the equation:

$$f(x) = x{*}\sin(x) - 1 = 0$$

There are an infinite number of solutions, so the one to which Newton's Method converges depends on the initial choice x_0. Output for program Solver is shown in Figure 3.10.

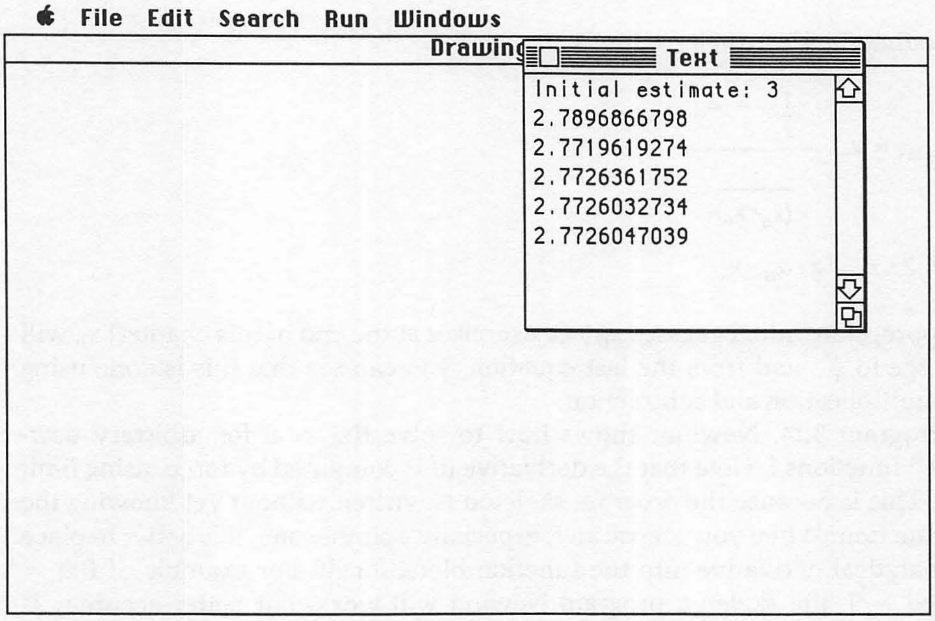
Integral Calculus

Just as differential calculus involves concepts of limits, integral calculus involves concepts of sums (and also limits). The essential idea is that an integral:

$$I = \int_a^b f(x) \quad dx$$

can be thought of (for sufficiently smooth f) as a limit:

$$I = \lim_{n \to \infty} \sum_{i=0}^{n-1} f(x_i)\ dx$$



```
Initial estimate: 3
2.7896866798
2.7719619274
2.7726361752
2.7726032734
2.7726047039
```

Figure 3.10 | Solver output

where $dx = \frac{b-a}{n}$ is one n-th of the x interval of integration, and the x_i are spaced along the interval according to:

$x_i := a + i * dx$

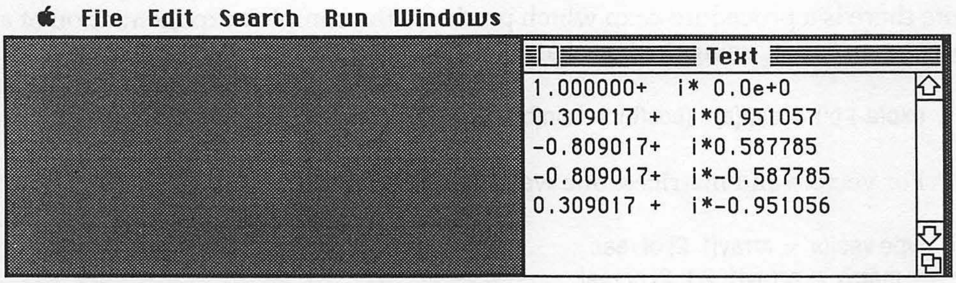
Therefore integrals can be computed by correctly setting up summations and taking a fine enough grid (small enough dx).

Program 3.16, Integral, is another skeleton program which serves to approximate integrals. No output is displayed in the book for this program, but exercises for this chapter involve integrals for which the program can be used.

It should be pointed out that this basic integration technique can be vastly improved by alterations in the summation. Techniques such as quadrature, trapezoidal approximation, and so on are covered in numerical analysis texts. These more sophisticated methods usually improve the approximation by summing over regions which are not precisely rectangular.

Complex Numbers, Vectors and Matrices

Pascal is well suited for multidimensional arrays in scientific programming, the only real defect being that once an array is sized in a declaration, it cannot be re-



```
1.000000+   i* 0.0e+0
0.309017 +  i*0.951057
-0.809017+  i*0.587785
-0.809017+  i*-0.587785
0.309017 +  i*-0.951056
```

Figure 3.11 | UnityRoots output

sized within the program. We have put complex numbers, vectors, and matrices into this section because the ideas are so closely related: a complex number is something like a 2-vector, a matrix is a collection of vectors, and so on.

Options abound for how to declare and manipulate these objects. One can set up programs so that a matrix M has its (i,j)-th element accessible as either:

```
M[i,j]
```

or

```
M[i][j]
```

or 'record' types can be used, and so on. Complex numbers are a good example of a time when you might want to use record types. For example, the declarations:

```
type complex = record
    re: real;
    im: real;
  end;
var z: complex;
```

allow you to access real and imaginary parts of the complex number z as z.re and z.im, respectively. Program 3.17, UnityRoots, (corresponding output Figure 3.11) computes the five fifth roots of unity as complex numbers. These are the values:

$$\exp \frac{2\pi ik}{5} \; ; k = 0..4$$

Note there is a procedure cexp which produces the complex exponentiation of a complex number, using the fact that:

$$\exp(a+bi) = \exp(a) * (\cos(b) + i \sin(b))$$

For vectors and matrices, one way to proceed is to declare:

```
type vector = array[1..2] of real;
  matrix = array[1..2,1..2] of real;

var v:vector;
  m:matrix;
```

and refer to vector components as v[i] and matrix components as m[i,j]. Another way, as suggested above, is to replace the second type declaration with:

```
matrix = array[1 . . 2] of vector;
```

in which case matrix elements are accessed as m[i][j].

Program 3.18, Ray, rotates a ray on the graphics screen by using an insert from the graphics.lib and calculates 2-space rotations according to:

$$v' = Mv$$

where v is a vector, v′ is the rotated vector, and M is a rotation matrix:

$$M = \begin{bmatrix} \cos(a) & -\sin(a) \\ \sin(a) & \cos(a) \end{bmatrix}$$

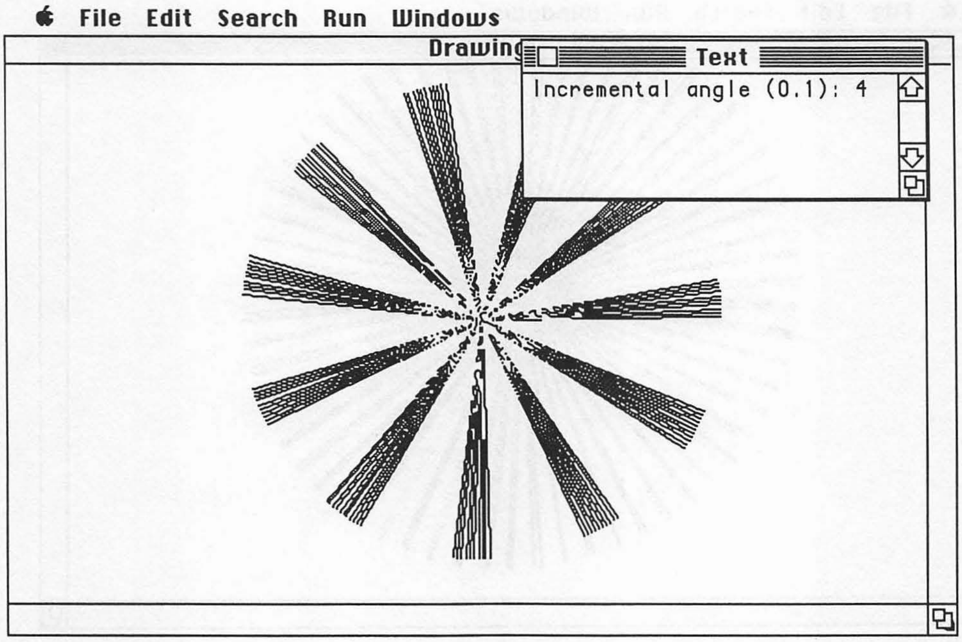
(a = the angle of rotation)

The typical output is shown in Figure 3.12. Later, when you apply animation techniques, such a ray can be made to rotate with past-blanking, causing rotational motion to appear.

There are several ways to perform graphics rotations. One way is to use the real-valued rotation matrix but specify integer-valued vectors. Then multiplication Mv would be truncated (in all vector components) so that v′ components are again integer valued. This speeds up the graphics. The fastest way to rotate is to stick with integers always, avoid the matrix, and instead, use incremental rotations as in Program 3.19, Spin, which uses the fact that:

$$x := x - \frac{y}{n};$$

$$y := y + \frac{x}{n};$$



```
Incremental angle (0.1): 4
```

Figure 3.12 | Ray output

Large N is an incremental rotation (rotation matrix generator). The second line reference to the var x (which has been changed already in the first line) is intentional: it turns out that this approximation of rotations is relatively stable. Typical output is shown in Figure 3.13, and comes out significantly faster than does the full, real-number method of program Ray.

One of the powerful features of Pascal is the capability for multidimensional modeling of dynamical trajectories. Such applications are covered in later chapters. For the moment, we give a skeleton upon which to build models which require ensembles (of particles, for example) to be updated in terms of velocity and position at each time t. Program 3.20, VMechanics,shows how to incorporate these updates in simple procedures.

The update-procedure calls near the end of the program comprise the vector equivalent of the usual calculus of motion:

```
vel:= vel + acc*dt;
pos:= pos + vel*dt;
```

The symmetry of these relations, whether for single or multidimensional dynamical variables, allows you to use just one procedure in which you update the variable, knowing its time derivative.

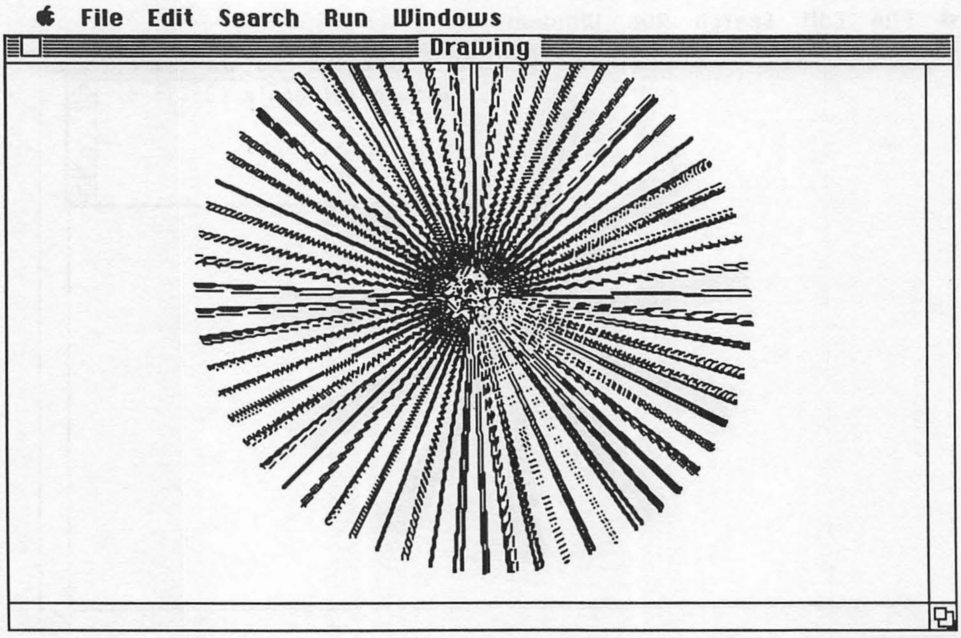
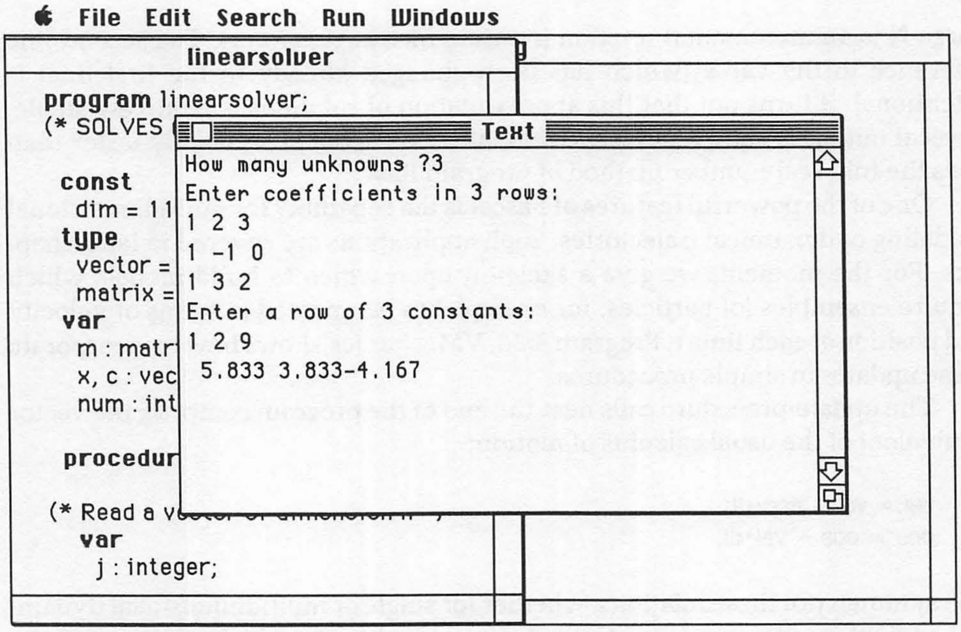


Figure 3.13 | Spin output



```
program linearsolver;
(* SOLVES (
  const
    dim = 11
  type
    vector =
    matrix =
  var
    m : matr
    x, c : vec
    num : int

  procedur

 (* Read a v
   var
     j : integer;
```

```
How many unknowns ?3
Enter coefficients in 3 rows:
1 2 3
1 -1 0
1 3 2
Enter a row of 3 constants:
1 2 9
  5.833 3.833-4.167
```

Figure 3.14 | LinearSolver output

Linear Equations

A set of n simultaneous linear equations in n unknowns X1, X2, . . ., Xn can be cast in the form:

```
M11 X1 + M12 X2 + ... + M1n Xn = C1
M21 X1 + ...                   = C2
...
Mn1 X1 + ...          + Mnn Xn = Cn
```

so that the *coefficients matrix M* is an n-by-n square matrix, and the *constants column c* is an n-dimensional vector. When the determinant of M is non-zero, there will be precisely one simultaneous solution, thought of as a vector X1,. . .,Xn. The familiar Cramer's Rule allows you to solve for the unique solution. A program which solves such a system is Program 3.21, LinearSolver, with typical output shown in Figure 3.14.

Other aspects of the matrix.lib are covered in later chapters.

Exercises

1. Write a program to generate the sequence of Fibonacci Numbers which start

   0,1,1,2,3,5,8,13, . . .

   obeying the recurrence relation that each term is the sum of the previous two. You should use type Longint to get maximum size.

2. Is any Fibonacci number besides 1 a perfect square? To test if a number N is a perfect square, you can see if sqrt(N) is sufficiently near to an integer (how near ?), or you can test to see whether all primes dividing N appear to even powers. Thus 1125 = 3*3*5*5*5 is not a perfect square because 5 appears to an odd power (3). Here is another curious criterion for squares: n is a square if and only if n has an odd number of divisors, e.g. n = 9 is divisible by 1, 3, 9.

3. Write a program to test the celebrated 3X + 1 Problem. This beautiful and still mysterious problem runs like so: Take an integer n > 1. If it is odd, multiply by 3 and add 1. If it is even, divide by two. Continue applying these rules until you reach 1. For example, starting with n = 3, you get:

   3, 16, 8, 4, 2, 1

   whereas n = 27 takes over 100 steps to reach 1! A first program should allow you to input the initial n and perform the loop until n = 1, typing out the intermediate numbers along the way. No one knows if there are any initial n which do not go down to 1, yet no one can prove that all numbers will so decay. It is known that at least all numbers up to about 1 e 12 will decay to 1.

4. Write a program that factors input numbers N using the principle that if N + sqr(m) is a perfect square q*q, then N = q*q − m*m = (q+m)*(q−m).

5. In the Math.lib there is a function COMBO(n,m) which computes the combinatorial bracket:

$$\binom{n}{m} = \frac{n!}{m!\,(n-m)!}$$

Write a program that displays Pascal's Triangle, which appears thus:

```
      1
      11
     121
    1331
    14641
  15101051
    . . .
```

and has the numbers combo(n,m) listed horizontally (m = 0,. . .,n) along the n-th row (0-th row is the first, singleton 1). You should use the writedraw procedure to place the numbers in the Drawing window.

6. Verify with a program that the sum of combo (n,m) over m = 0,. . .,n is an exact power of two.

7. Find all twin prime pairs less than 10,000. A twin prime pair is a consecutive set of odd primes, such as (3,5) or (101,103). The program Sieve is probably the fastest way to do this—one simply finds the consecutive pairs remaining after the sieve is performed on the main array.

8. Write a program to find a solution x to the modular quadratic equation:

x*x = A mod p

where p is prime. An A for which a solution x can be found is called a *quadratic residue* of the prime p. For various p, how many of the numbers (0,1,. . .,p − 1) are quadratic residues?

9. Find the elements of the continued fraction for the real number:

$$\frac{(e-1)}{(e+1)}$$

where e = exp(1). Then use these elements to go backward and estimate the value of e.

10. Find the first few elements of the continued fraction for π. No one knows what the pattern, if any, for this number might be, but it is remarkable that just a few terms of the fraction give a surprising number of accurate decimals for π. With a program, comment on the accuracy of π if you take just the first 4 elements a_0, a_1, a_2, a_3.

11. There is a beautiful result from the theory of continued fractions: that the elements a_0, a_1, a_2,. . . eventually fall into a periodic sequence if and only if the real positive number x to be expanded is q *quadratic surd*; i.e., has the form:

$$x = \frac{(A + - \text{sqrt}(B))}{C}$$

where A, B, and C are integers. With a program, find the elements, and guess the pattern for, the numbers:

sqrt(3)

$$\frac{(1 + \text{sqrt}(13))}{2}$$

$$\frac{(1+\text{sqrt}(5))}{2;}$$

Try to determine on paper some inverse problems, such as what is the exact value of the continued fraction for element sequences such as:

$(a_0,a_1,a_2,. . .) = (1,2,3,1,2,3,1,2,3,1,2,3, . . .)$ or $(5,5,5,5,5,5,5,5,. . .)$

12. Most of the elementary mathematical functions can be computed as certain kinds of continued fractions. A good example is:

$$\tan(x) = \cfrac{x}{1 - \cfrac{x^2}{3 - \cfrac{x^3}{5 - \dots}}}$$

Use this fact to define your own tan, sin, and cos functions (call them something else so as not to clash with the Pascal keywords), and try it out by computing some standard values such as tan(π/4), sin(π/2), and so on.

13. Use the observations of the last problem to compute values of exp(x), using the fact that:

$$\tanh(x) = \frac{(\exp(x) - \exp(-x))}{(\text{esp}(x) + \text{esp}(-x)}$$

and that the continued fraction for tanh(x) is the same as for tan(x) except that all – (minus) signs are replaced by + (plus) signs.

14. Here is an exercise in the art of computing limits. It turns out that the limit of the expression:

$$\frac{n! * (\frac{e}{n})^n}{(\text{sqrt}(2n))}$$

as n goes to infinity is an interesting number. Find a numerical estimate for this limit, square the estimate, and guess what the limit is.

15. The celebrated Riemann Zeta Function is given by:

$$\zeta(s) = \sum n^{-s}$$

Estimate π by two different means:

$$\zeta(2) = \frac{\pi^2}{6}$$

$$\zeta(4) = \frac{\pi^4}{90}$$

and report which method is more efficient by finding out how many tick-counts pass for each method up to some guess value of abs(estimate – π), perhaps 1e – 4. It is a good idea to only compute the sqr( ) or sqr(sqr( )) functions every so often—in fact, you do not need to compute these in principle until the sum for the Zeta Function is performed.

16. Find the numerical limit of the operations:

$$\sqrt{2}^{\sqrt{2}^{\sqrt{2}^{\sqrt{2}^{\cdot^{\cdot^{\cdot}}}}}}$$

The exact result can easily be obtained in your head if you observe that sqrt(2) to the answer is equivalent to the answer.

17. Write a program to find all of the real zeros of a general quartic polynomial:

$$A x^4 + B x^3 + C x^2 + D x + E = 0$$

where the coefficients A, B, C, D, E are input at run-time. There can be, at most, four real zeros, and various criteria exist for finding how many there can be, how large they can be, and so on.

18. Write a program to find the period for one full cycle of the simple pendulum motion:

$$\frac{d^2x}{dt^2} = \frac{-g}{L} x$$

and verify the theoretical result that $P = 2 * \pi * \text{sqrt } \frac{L}{g}$. One way to do this is to simply print out the values of x vs. time t and note when x recurs to its original value. Theory says that the period for this system is independent from the initial (non-zero) x.

19. The pendulum equation for the last exercise is not exact. The true pendulum obeys:

$$\frac{d^2x}{dt^2} = \frac{-g}{L} * \sin(x)$$

for which the simplification uses the near equality of x and sin(x) for small angles x. Write a program to find the period, which now depends on the initial x and $\frac{dx}{dt}$, and verify the theoretical result that for initial angle $x(0) = \frac{\pi}{2}$, and initial rest $\frac{dx(0)}{dt} = 0$:

$$P = (\text{absolute constant}) * 2 * \pi * \text{sqrt}\left(\frac{L}{g}\right)$$

The constant is greater than one and can be derived from the theory of elliptic integrals.

20. Find the real fifth root of two using Newton's Method.

21. Using the method in the text—doing division with only multiplication and subtraction—write a program which takes two input real numbers and, by starting with initial guess = 1 in Newton's Method, appears to converge on the correct ratio. Note that the magnitude of the denominator is essential: it can only be on one side of the value unity. Can you suggest a way to get around this?

22. Use Newton's method to find square roots of numbers z, using the formula:

$$f(x) = x^2 - z = 0$$

You will want to define your own Pascal function version of sqrt which will take any reasonable input z. A good guess for initial x is 1, or $\frac{z}{2}$, etc.

23. Find an approximate value for the integral:

    exp(- x*x) dx

24. Write a program which animates a rotating 'wheel' with four spokes, using the expedient of approximate integer-based rotations as in program Spin.
25. Write a program which uses Matrix.lib to solve the two simultaneous equations:

    $2x + 4y = 64$
    $13x - 2y = -4$

26. Using procedures for computing the dot product and magnitude of two vectors and one vector, respectively, write a program which outputs the angle between two input vectors, according to:

    $$\cos(\text{theta}) = \frac{(u \cdot v)}{|u|\,|v|}$$

27. A parallelepiped can be defined by three 3-space vectors A, B, C. The figure fits into the 'socket' formed by these three vectors emanating from the origin. The volume of the parallelepiped is given by:

    $V = |(A \times B) \cdot C|$

    Write a program which incorporates dot-product and cross-product procedures to compute volumes of such figures. Test this out on a unit cube, whose A, B, C form an orthonormal system.

Answers

1. The recursion should look like this:

```
c:= a + b;
b:= a;
a:= c;
```

2. The only such Fibonacci is 144.
3. The recursion should look like this:

```
if n mod 2 = 0 then n:= n div 2 else n:= 3*n+1;
```

4. You could choose a 'perfect square' test by defining a function:

   ```
   function isperfect(n:integer):boolean;
   begin
     x:= sqrt(n);
     if abs(x – n) < $\frac{1}{4*n}$ then isperfect:= true else isperfect:=false;
   end;
   ```

   so that, for example, isperfect(9) will be true and isperfect(10) false. Then you would add the test loop:

   ```
   isperfect(N + sqr(m))
   ```

   If true, then the factors of N are:

   ```
   round(sqrt(N)) ± m
   ```

5. This is straightforward. The drawing position for displaying combo(n,m) can be, for example:

   ```
   x:= 200 - a*n + 2*a*m;
   y:= 20 + b*n;
   ```

   for appropriate constants a and b.

6. This is straightforward. Something like:

   ```
   sum:=0;
   for m:= 0 to n do sum:= sum + combo(n,m);
   ```

   will do. If you print out $\frac{\ln(sum)}{\ln(2)}$ you will get a real number very close to the power of two involved.

7. After sieving, just check for each prime p whether the (p+2)-th array element reads prime.

8. The number of quadratic residues is $\frac{(p+1)}{2}$ for odd primes p. The number p – 1 is a quadratic residue of an odd prime p only if p mod 4 = 1.

9. The elements form just about the simplest arithmetic progression one can imagine.

10. The first few are 3,7 but very soon there is a relatively large element. If you stop there, the approximation is especially efficient.

11. The exact expansions are:

    sqrt(3): 1,1,2,1,2,1,2,1,2,. . .

    $\frac{(1 + \text{sqrt}(13))}{2}$: 2,3,3,3,3,3,3,. . .

    $\frac{(1 + \text{sqrt}(5))}{2}$: 1,1,1,1,1,1,1,. . .

    The inverse solutions are:

    1,2,3,1,2,3,1,2,3,. . . : $\frac{(4 + \text{sqrt}(37))}{7}$

    5,5,5,5,. . . : $\frac{(5 + \text{sqrt}(29))}{2}$

12. You will want to use the fact that sin and cos are related to tan by:

    $\sin(x) = \frac{\tan(x)}{A}$; $\cos(x) = \frac{1}{A}$
    where $A = \pm$ sqrt(1 + sqr(tan(x)))

13. The relation between exp and than can be inverted to give:

    $\exp(x) = \text{sqrt}\ \frac{(1+\tanh(x))}{(1-\tanh(x))}$

14. The limit is sqrt(π). This comes from Sterlings formula for approximating factorials.

15. The power-four sum is more efficient by most criteria. Though the power-four case requires about twice as many operations for each summand (two squarings of n and a divide), the error after T terms is of order $\frac{1}{T3}$, as compared with $\frac{1}{T}$ for the power-two case.

16. The value of the infinite ladder is 2. The recursion is particularly simple. In fact, the block:

```
x:= sqrt(2);
repeat
 x:= exp (x)/2 *1n (2));

until . . .
```

    will suffice.

17. This is an exploratory problem. There are many approaches to this, and you should consult a numerical analysis text for the best approaches.

18. A loop should follow the general rule for dynamic calculations. One solution is to loop on:

```
v:= v - (g/L) * dt;
x:= x + v*dt;
t:= t + dt;
```

and print out x values versus t values. One trick for finding a precise period is to find the times when velocity v changes sign. There are two such times per period.

19. The period for the stated initial conditions is about 1.2 times the period for the simplified problem.

20. To solve $f(x) = x^5 - 2 = 0$, observe that $f'(x) = 5x^4$ so that the recursion is:

$$x_{n+1} = x_n - \frac{x_n{}^5 - 2}{5x_n{}^4}$$

which can be simplified to be:

```
x:= 1;      (* or some other initial guess *)
repeat
    x:= 4*x/5 + 2/(5*sqr(x))
until . . .
```

The criterion (until . . .) can be $\text{abs}(x^5 - 2) < 1e-10$, or some such similar check for accuracy.

21. A program block to compute the ratio y/z without explicit division should look like this:

```
readln(y,z);
x:= 1;
repeat
 x:= 2*x - z*x*x;
    until abs(z*x-1) < 1e-10;
writeln(y*x);
```

The method works if z is greater than unity. There are various ways around this constraint. One of them is to multiply both y and z by the same large real to begin with, for this does not affect the ratio.

22. One definition would be:

```
function myroot(z:real):real;
var x:real;
begin
    x:= z/2;
    repeat
      x:= x/2 + z/(2*x);
    until abs(z - sqr(x)) < 1e-10;
    myroot:= x;
end;
```

23. The exact value is sqrt(π). A clean approach is to do a straightforward sum of exp(− sqr(x)) over a wide range and multiply the sum by dx.

24. Start by declaring x,y as Longints. Then initialize x:= 10000; y:= 0; and also xold:= x; yold:= y.

    Then loop on:

```
x:= x + y div 100;
y:= y - x div 100;
animate(xold,yold,x,y);
```

The animate procedure is to erase the spokes defined by the line from (0,0) to (xold,yold) and its three 'counterparts'. Then draw in the new spokes from (0,0) to (x,y). To draw the spokes, you can add the following within the animate procedure:

```
penpat(white);
moveto(250,125);
lineto(250+xold,125-yold);
moveto(250,125);
lineto(250-yold,125-xold);
moveto(250,125);
lineto(250-xold,125+yold);
moveto(250,125);
lineto(250+yold,125+xold);
penpat(black);
(* do the same 8 procedures but with x,y instead *)
```

You ought to be able, from the symmetry of the operations, to compact this procedure further by having yet another procedure which does the moveto; lineto sequence.

25. First set the matrix dimension to 2, set up the 2 X 2 matrix a, and set up the 2-column c. Then the simple call:

```
solve(2,a,c,x);
```

should give the unique solution (x,y) as x[1] = 2, x[2] = 15.

26. A typical dot product can be done with a function, such as:

```
function dot(u,v:vector):real;
var n:integer;
  temp:real;
begin
  temp:=0;
  for n:=1 to 3 do temp:= temp + x[n]*y[n];
 dot:= temp;
end;
```

27. There should be a procedure called cross, which sets up a vector variable D as AXB:

```
procedure cross(A, B:vector; var D:vector);
begin
D[1]:= A[2]*B[3] - A[3]*B[2];
D[2]:= A[3]*B[1] - A[1]*B[3];
D[3]:= A[1]*B[2] - A[2]*B[1];
end;
```

The dot product can be defined as a function as in the previous answer. Then the volume can be obtained from:

```
cross(A,B,D);
volume:= sqrt(dot(D,C));
```

Program 3.1(a)

```
program Factorial;
(* COMPUTES FACTORIALS *)

 var
  n : integer;
  factorial : Longint;

begin
 n := 1;
 factorial := 1;
 repeat
  factorial := factorial * n;
  writeln(factorial : 1);
  n := n + 1;
 until factorial > 1e8;
end.
```

Program 3.1(b)

```
program RecFactorial;
(* RECURSIVE CALCULATION OF FACTORIALS *)

 var
  n : integer;

 function factorial (n : integer) : integer;
 begin
  if (n > 1) then
   factorial := n * factorial(n - 1);
 end;

begin
 repeat
 readln(n);
 writeln(factorial(n));
until false;
end.
```

Program 3.2(a)

```
program Primes;
(* STRAIGHTFORWARD PRIME TESTER *)

 var
  p, divisor : integer;
```

```
begin
 showtext;
 repeat
  write('Give p: ');
  readln(p);
  divisor := 2;
  while (p mod divisor <> 0) and (divisor < p / divisor) do
   begin
    if divisor = 2 then
     divisor := 3
    else
     divisor := divisor + 2;
   end;
  if (p mod divisor <> 0) or (p = divisor) then
   writeln('prime')
  else
   writeln('divisible by ', divisor :
 until eof;
end.
```

Program 3.2(b)

```
program RecPrimes
(* GENERATES PRIME NUMBERS VIA A RECURSIVE FUNCTION *)

 var
  p : Longint;

 function f (n : integer;
         p : longint) : boolean;
 begin
  if sqr(n) > p then
   f := true
  else
   begin
    if p mod n = 0 then
     f := false
    else if n > 2 then
     f := f(n + 2, p)
    else
     f := f(n + 1, p);
   end;
 end;

begin
 showtext;
 repeat
```

(continued)

```
  write('Give p: ');
  readln(p);
  if f(2, p) then
   writeln('prime')
  else
   writeln('composite');
 until eof;
end.
```

Program 3.3

```
program sieve;
(* THE SIEVE OF ERATOSTHENES *)

 const
  max = 500;

 var
  sieve : array[1..max] of boolean;
  n, k, divisor : integer;

begin
 showtext;
 for n := 1 to max do
  sieve[n] := true;    (* Start out assuming all are prime *)
 divisor := 2;
 repeat
  k := 2 * divisor;
  while k <= max do
   begin
    sieve[k] := false;  (* strike all proper multiples of *)
                                  (*  divisor *)
    k := k + divisor;
   end;
  repeat
   divisor := divisor + 1;
  until sieve[divisor];  (* Go find the next prime *)
 until divisor > sqrt(max);
 for k := 2 to max do
  if sieve[k] then
   writeln(k : 1);
end.
```

Program 3.4

```
program wilson;
(* PRIMALITY TEST VIA WILSON'S THEOREM *)

 var
  p, factorial : Longint;
  n : integer;

begin
 showtext;
 repeat
  writeln('Give p: ');
  readln(p);
  factorial := 1;
  for n := 1 to p - 1 do
   factorial := (n * factorial) mod p;
  if factorial + 1 = p then
   writeln('prime')
  else
   writeln('composite');
 until eof;
end.
```

Program 3.5

```
program pollard;
(* FACTORS LONG INTEGERS USING THE 'POLLARD RHO' METHOD *)

 var
  p, x, y, g : longint;

 function GCD (a, b : longint) : longint;
  var
   r, q : longint;
 begin
  if b < 0 then
   b := -b;
  if a < 0 then
   a := -a;
  if a > 0 then
```

(continued)

```
   begin
    b := b mod a;
    r := 1;
    if b = 0 then
     r := 0;
    while r > 0 do
     begin
      q := a div b;
      r := a - q * b;
      a := b;
      b := r;
     end;
   end;
  GDC := a;
 end;

begin
 showtext;
 repeat
  write('Give p:');
  readln(p);
  x := 3;
  y := 3;
  repeat
   x := (x * x + 1) mod p;
   y := (y * y + 1) mod p;
   y := (y * y + 1) mod p;
   write('.');
   g := GDC(x - y, p);
  until (g > 1);
  writeln(g);
 until eof;
end.
```

Program 3.6

```
program RootOfTwo;
(* CONTINUED FRACTION METHOD - CALCULATES ROOT OF TWO *)

 var
  pPast, pPresent, pFuture, qPast, qPresent, qFuture : Longint;
  x : real;

begin
 showtext;
 pPast := 0;
 qPast := 1;
 pPresent := 1;
 qPresent := 2;
 repeat
```

```
   pFuture := 2 * pPresent + pPast;
   qFuture := 2 * qPresent + qPast;
   x := pFuture / qFuture + 1;
   pPast := pPresent;
   qPast := qPresent;
   pPresent := pFuture;
   qPresent := qFuture;
   writeln(x : 6 : 12);
  until pFuture > 1e7;
  writeln(sqr(x) : 6 : 12);
 end.
```

Program 3.7

```
program Fraction;
(* COMPUTES THE INTEGER ELEMENTS OF THE CONTINUED FRACTION *)
(* FOR A GIVEN REAL NUMBER *)

 var
  x : real;
  a : integer;

begin
 showtext;
 write('Give real number: ');
 readln(x);
 repeat
  a := trunc(x);
  writeln(a : 1);
  x := 1 / (x - a);
 until x > 10000;
end.
```

Program 3.8

```
program e;
(* COMPUTES e FROM COMPOUND INTEREST FORMULA *)

 const
  n = 10000;

 var
  j : integer;
  e : real;

begin
 showtext;
 writeln('Please wait for about a minute');
```

(continued)

```
  e := 1;
  for j := 1 to n do
   e := (1 + 1 / n) * e;
  writeln(e : 6 : 12);
 end.
```

Program 3.9

```
program limit;
(* COMPUTES LIMIT OF  sin(x)/x  AS  x  APPROACHES ZERO  *)

 var
  x : real;

begin
 showtext;
 x := 1;
 repeat
  writeln(sin(x) / x : 8 : 10);
  x := x / 2;
 until x < 1e-5;
end.
```

Program 3.10

```
program expo;
(* COMPUTES exp(x) FROM POWER SERIES *)

 var
  x, term, sum : real;
  n : integer;

begin
 showtext;
 sum := 1;
 write('Give x: ');
 readln(x);
 term := 1;
 sum := 0;
 n := 0;
 repeat
  sum := sum + term;
  n := n + 1;
  term := term * x / n;
 until abs(x / n) < 1e-2;
 writeln(sum : 8 : 10);
 writeln('SANE reference: ', exp(x) : 8 : 10);
end.
```

Program 3.11

```
program polyplot;
(* GRAPHS QUINTIC POLYNOMIAL AND ITS CRITICAL POINTS *)

 const
  delta = 0.03;

 var
  x : real;
  oldsgn : integer;

 function sgn (x : real) : integer;
 begin
  if x < 0 then
   sgn := -1
  else if x > 0 then
   sgn := 1
  else
   sgn := 0;
 end;

 function f (x : real) : real;
 begin
  f := 10 * (x - 0.8) * (x - 0.4) * (x - 0.1) * (x + 0.2) * (x + 0.75);
 end;

 function df (x : real) : real;
  const
   epsilon = 1e-6;
 begin
  df := (f(x + epsilon) - f(x)) / epsilon;
 end;

 procedure CLEAR;
(* Activates and expands Drawing Window to fill screen *)
  var
   windowrect : rect;
 begin
  hideall;
  setrect(windowrect, 2, 35, 512, 342);
  setdrawingrect(windowrect);
  showdrawing;
 end;

 procedure MAP (x, y : real;
         var hor, ver : integer);
(* Returns hor,ver as pixel integers for given real coords (x,y) *)
 begin
```

(continued)

```
  if abs(x) > 1.9 then
   x := x / abs(x) * 1.9;
  hor := 255 + trunc(x * 130);
  if abs(y) > 1 then
   y := y / abs(y);
  ver := 138 - trunc(y * 130);
 end;

 procedure DRAW (x, y : real);
(* Draws to (x,y) *)
  var
   h1, v1 : integer;
 begin
  MAP(x, y, h1, v1);
  lineto(h1, v1);
 end;

 procedure SHIFT (x, y : real);
(* moves invisible to (x,y) *)
   var
    h1, v1 : integer;
  begin
   MAP(x, y, h1, v1);
   moveto(h1, v1);
  end;

 begin
  CLEAR;
  SHIFT(1, 0);
  DRAW(-1, 0);
  SHIFT(0, -1);
  DRAW(0, 1);
  x := -0.9;
  oldsgn := sgn(df(x));
  SHIFT(x, f(x));
  repeat
   DRAW(x, f(x));
   if sgn(df(x)) <> oldsgn then
    begin
     pensize(1, 2);
     SHIFT(x - 0.2, f(x));
     DRAW(x + 0.2, f(x));
     SHIFT(x, f(x));
     pensize(1, 1);
    end;
   oldsgn := sgn(df(x));
   x := x + delta;
  until x > 0.9;
 end.
```

Program 3.12

```
program mechanics;
(* SKELETON SOLVER FOR MECHANICS PROBLEMS *)

 const
  dt = 0.001; (* this is your small time increment *)
 var
  pos, vel, acc : real;
  t : real;

begin
 showtext;
 write('initial position: ');
 readln(pos);
 write('initial velocity: ');
 readln(vel);
 t := 0;
 repeat
  t := t + dt;
    (* insert mechanics formula for 'acc' components here *)
  vel := vel + acc * dt;
  pos := pos + vel * dt;
    (* do your output here as desired *)
 until false;  (* Or some appropriate condition *)
end.
```

Program 3.13

```
program oscillator;
(* DAMPED OSCILLATOR*)

 const
  dt = 0.03; (* this is your small time increment *)
  m = 1;
  k = 100;
  b = 1;

 var
  pos, vel, acc : real;
  t : real;

 procedure CLEAR;
(* Activates and expands Drawing Window to fill screen *)
  var
   windowrect : rect;
```

(continued)

```
 begin
  hideall;
  setrect(windowrect, 2, 35, 512, 342);
  setdrawingrect(windowrect);
  showdrawing;
 end;

 procedure MAP (x, y : real;
         var hor, ver : integer);
(* Returns hor,ver as pixel integers for given real coords (x,y) *)
 begin
  if abs(x) > 1.9 then
   x := x / abs(x) * 1.9;
  hor := 255 + trunc(x * 130);
  if abs(y) > 1 then
   y := y / abs(y);
  ver := 138 - trunc(y * 130);
 end;

 procedure DRAW (x, y : real);
(* Draws to (x,y) *)
  var
   h1, v1 : integer;
 begin
  MAP(x, y, h1, v1);
  lineto(h1, v1);
 end;

 procedure SHIFT (x, y : real);
(* moves invisible to (x,y) *)
  var
   h1, v1 : integer;
 begin
  MAP(x, y, h1, v1);
  moveto(h1, v1);
 end;

begin
 CLEAR;
 SHIFT(-1, 1);
 DRAW(-1, -1);
 SHIFT(-1, 0);
 DRAW(1, 0);
 showtext;
 write('initial position: ');
 readln(pos);
 write('initial velocity: ');
 readln(vel);
 t := 0;
 repeat
```

```
  acc := -k * pos / m - 2 * b * vel / m;
  vel := vel + acc * dt;
  pos := pos + vel * dt;
  if t = 0 then
   SHIFT(t - 1, pos)
  else
   DRAW(t - 1, pos);
  t := t + dt;
 until t > 2;  (* Or some appropriate condition *)
end.
```

Program 3.14

```
 program newton;
 (* GENERAL NEWTON'S METHOD SOLUTION SEEKER *)

  var
   x, newx : real;

  function f (x : real) : real;
  begin
    (* Define user function here, f:= ... *)
  end;

  function df (x : real) : real;
   const
    epsilon = 1e-6;
  begin
   df := (f(x + epsilon) - f(x)) / epsilon;
  end;

 begin
  showtext;
  write('Initial estimate: ');
  readln(newx);
  repeat
   x := newx;
   newx := x - f(x) / df(x);
   writeln(newx : 8 : 10);
  until abs(x - newx) < 1e-9;
 end.
```

Program 3.15

```
program solver;
(* SOLVES TRANSCENDENTAL EQUATION  x*sin(x) -1 = 0 *)

 var
  x, newx : real;
```

(continued)

```
  function f (x : real) : real;
  begin
   f := x * sin(x) - 1;
  end;

  function df (x : real) : real;
   const
    epsilon = 1e-6;
  begin
   df := (f(x + epsilon) - f(x)) / epsilon;
  end;

 begin
  showtext;
  repeat
   write('Initial estimate: ');
   readln(newx);
   repeat
    x := newx;
    newx := x - f(x) / df(x);
    writeln(newx : 8 : 10);
   until abs(x - newx) < 1e-5;
  until eof;
 end.
```

Program 3.16

```
program integral;
(* CALCULATES GENERAL INTEGRALS *)

 const
  grain = 100;     (* This is the number of pieces in interval *)
 var
  a, b, x, int, dx : real;
  i : integer;

 function f (x : real) : real;
 begin
(* Define user function to be integrated, f:= ... *)
 end;

begin
 showtext;
 write('Limits a b: ');
 readln(a, b);
 dx := (b - a) / grain;
 int := 0;
 for i := 0 to grain - 1 do
```

```
  int := int + f(x + i * dx);
 writeln(int * dx : 8 : 10);
end.
```

Program 3.17

```
program unityroots;
(* COMPUTES THE FIVE FIFTH ROOTS OF UNITY *)

 const
  pi = 3.1415926535897932;

 type
  complex = record
    re : real;
    im : real;
   end;

 var
  z, u : complex;
  n : integer;

 procedure cexp (var u, z : complex);
(* u becomes exp(z) *)
 begin
  u.re := cos(z.im) * exp(z.re);
  u.im := sin(z.im) * exp(z.re);
 end;

begin
 showtext;
 for n := 0 to 4 do
  begin
   z.re := 0;
   z.im := 2 * pi * n / 5;
   cexp(u, z);
   writeln(u.re : 3 : 6, '+  i*', u.im : 3 : 6);
  end;
end.
```

Program 3.18

```
program ray;
(* SPINS A 2-VECTOR USING REAL ARITHMETIC *)

 type
  vector = array[1..2] of real;
  matrix = array[1..2, 1..2] of real;
```

(continued)

```
var
 v : vector;
 m : matrix;
 a : real;


procedure mulvect (var v : vector;
        m : matrix);
 var
  i, j : integer;
  u : vector;
begin
 for i := 1 to 2 do
  begin
   u[i] := 0;
   for j := 1 to 2 do
    u[i] := u[i] + m[i, j] * v[j];
  end;
 for i := 1 to 2 do
  v[i] := u[i];
end;

procedure CLEAR;
 var
  windowrect : rect;
begin
 hideall;
 setrect(windowrect, 2, 35, 512, 342);
 setdrawingrect(windowrect);
 showdrawing;
end;

procedure MAP (x, y : real;
        var hor, ver : integer);
begin
 if abs(x) > 1.9 then
  x := x / abs(x) * 1.9;
 hor := 255 + trunc(x * 130);
 if abs(y) > 1 then
  y := y / abs(y);
 ver := 138 - trunc(y * 130);
end;

procedure DRAW (x, y : real);
 var
  h1, v1 : integer;
begin
 MAP(x, y, h1, v1);
 lineto(h1, v1);
end;
```

```
procedure SHIFT (x, y : real);
 var
  h1, v1 : integer;
begin
 MAP(x, y, h1, v1);
 movetc(h1, v1);
end;

begin
 penmode(patxor);
 CLEAR;
 showtext;
 write('Incremental angle (0.1): ');
 readln(a);
 v[1] := 1;
 v[2] := 0;
 m[1, 1] := cos(a);
 m[1, 2] := -sin(a);
 m[2, 1] := -m[1, 2];
 m[2, 2] := m[1, 1];
 repeat
  SHIFT(0, 0);
  mulvect(v, m);
  DRAW(v[1], v[2]);
 until false;
end.
```

Program 3.19

```
program spin;
(* SPINS A 2-VECTOR USING INTEGER ARITHMETIC *)

 var
  x, y : longint;

 procedure CLEAR;
(* Activates and expands Drawing Window to fill screen *)
  var
   windowrect : rect;
 begin
  hideall;
  setrect(windowrect, 2, 35, 512, 342);
  setdrawingrect(windowrect);
  showdrawing;
 end;

begin
 CLEAR;
 pensize(2, 2);
 penmode(patxor);
```

(continued)

```
 x := 1500;
 y := 0;
 repeat
  x := x - y div 10;
  y := y + x div 10;
  moveto(250, 125);
  lineto(250 + x div 10, 125 + y div 10);
 until false;
end.
```

Program 3.20

```
program vmechanics;
(* SOLVER FOR MULTI-DIMENSIONAL MECHANICS *)

 const
  dim = 2;  (* this is your number of space dimensions *)
  dt = 0.001; (* this is your small time increment *)

 type
  vector = array[1..dim] of real;

 var
  pos, vel, acc : vector;
  t : real;

 procedure READVECT (var a : vector);
  var
   i : integer;
 begin
  for i := 1 to dim do
   read(a[i]);
 end;

 procedure update (var a : vector;
         b : vector);
  var
   i : integer;
 begin
  for i := 1 to dim do
   a[i] := a[i] + dt * b[i];
 end;

begin
 write('initial pos components: ');
 READVECT(pos);
 write('initial vel components: ');
 READVECT(vel);
 repeat
  t := t + dt;
```

```
    (* insert mechanics formula for 'acc' components here *)
  update(vel, acc);
  update(pos, vel);
    (* do your output here as desired *)
 until 0 = 1;
end.
```

Program 3.21

```
program linearsolver;
(* SOLVES n LINEAR EQUATIONS IN n UNKNOWNS *)

 const
  dim = 11;    (* There can be at most 10 = dim-1 unknowns *)
 type
  vector = array[1..dim] of real;
  matrix = array[1..dim, 1..dim] of real;
 var
  m : matrix;
  x, c : vector;
  num : integer;

 procedure READVECT (n : integer;
         var x : vector);
(* Read a vector of dimension n *)
  var
   j : integer;
 begin
  for j := 1 to n do
   read(x[j]);
  readln;
 end;

 procedure READMAT (n : integer;
         var m : matrix);
(* Read an nXn matrix *)
  var
   i, j : integer;
 begin
  for i := 1 to n do
   begin
    for j := 1 to n do
     read(m[i, j]);
    readln;
   end;
 end;

 procedure WRITEVEC (n : integer;
         x : vector);
(* Output the vector x of dimension n *)
```

(continued)

```
 var
  i, j : integer;
begin
 for i := 1 to n do
  write(x[i] : 6 : 3);
 writeln;
end;

function DET (n : integer;
        a : matrix) : real;
(* Return the determinant of nXn matrix a *)
 var
  ii, jj, kk, ll, ff, nxt : integer;
  piv, cn, big, temp, term : real;
begin
 ff := 1;
 for ii := 1 to n - 1 do
  begin
   big := 0;
   for kk := ii to n do
    begin
     term := abs(a[kk, ii]);
     if term - big > 0 then
      begin
       big := term;
       ll := kk
      end
     end;
    if ii - ll <> 0 then
     ff := -ff;
    for jj := 1 to n + 1 do
     begin
      temp := a[ii, jj];
      a[ii, jj] := a[ll, jj];
      a[ll, jj] := temp
     end;
    piv := a[ii, ii];
    nxt := ii + 1;
    for jj := nxt to n do
     begin
      cn := a[jj, ii] / piv;
      for kk := ii to n + 1 do
       a[jj, kk] := a[jj, kk] - cn * a[ii, kk]
     end
   end;
  temp := 1;
  for ii := 1 to n do
   temp := temp * a[ii, ii];
  DET := temp * ff
 end;
```

```
  procedure SOLVE (n : integer;
          a : matrix;
          c : vector;
          var x : vector);
   var
    k : integer;
    d : real;

   procedure swap (n, k : integer;
           var a : matrix;
           var c : vector);
    var
     e : real;
     j : integer;
   begin
    for j := 1 to n do
     begin
      e := c[j];
      c[j] := a[j, k];
      a[j, k] := e
     end
   end;

  begin
   d := DET(n, a);
   for k := 1 to n do
    begin
     swap(n, k, a, c);
     x[k] := DET(n, a) / d;
     swap(n, k, a, c)
    end
  end;

begin
 showtext;
 write('How many unknowns ?');
 readln(num);
 writeln('Enter coefficients in ', num : 1, ' rows: ');
 READMAT(num, m);
 writeln('Enter a row of ', num : 1, ' constants: ');
 READVECT(num, c);
 SOLVE(num, m, c, x);
 WRITEVEC(num, x);
end.
```

4 Graphics and Animation

THEME: You will learn how to draw and animate in two dimensions. This chapter prepares you for later, more topical chapters including a chapter on 3D graphics.

GOALS: To be able to model physical phenomena, either by animation or by static visualization.

LIBRARIES USED: Graphics.lib
Math.lib

REFERENCE MATERIALS: Textbooks and references in the areas of interest in your discipline that contain problems you may want to solve with the computer.

Real-valued Coordinates

In this chapter, the Graphics.lib figures prominently. The library was created on the basis of 'most-needed' graphics procedures. Note that Graphics.lib is useful whenever you have to plot real numbers and may be avoided whenever you need only integer plots. The correspondence between the library procedures and the built-in QuickDraw procedures is summarized in the following list:

Built-in QuickDraw	Graphics.lib
moveto(m,n:integer);	SHIFT(x,y:real);
lineto(m,n:integer);	DRAW(x,y:real);
getmouse(m,n:integer);	REALMOUSE(x,y:real);
frameoval(m − r,n − r,m + r,n − r);	CIR(x,y,r:real);
(clearing of Graphics Window)	CLEAR;

In addition, the MAP and UNMAP procedures allow translation between integer coordinates and real coordinates, and vice versa. The two procedures are called as follows:

MAP(x,y,m,n); (* forces (m,n) to be the integer coordinates for given (x,y) *)

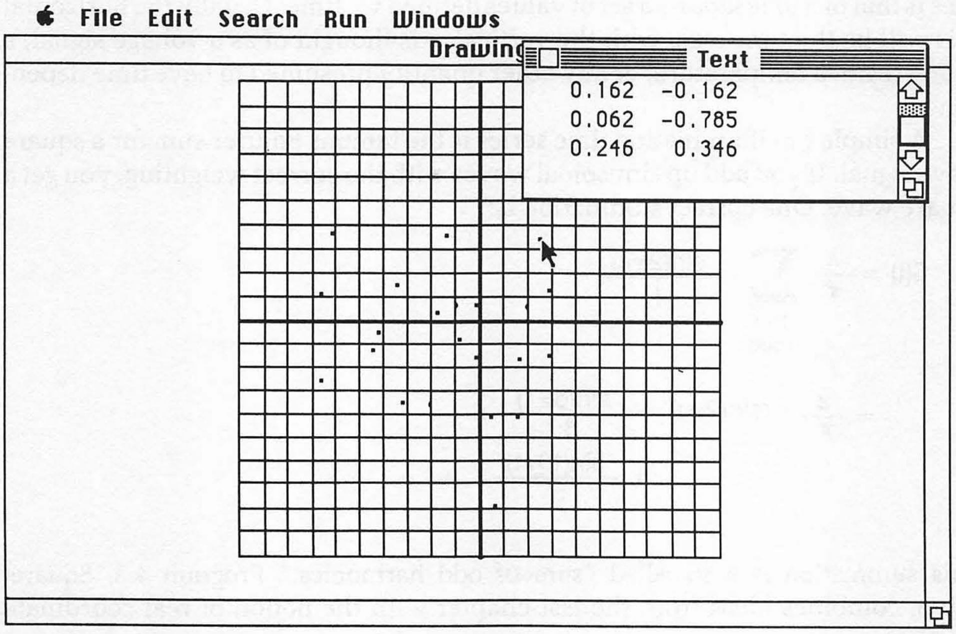
UNMAP(m,n,x,y); (* forces (x,y) to be the real coordinates for given (m,n) *)

Program 4.1, Grid, shows how to interpret the graphics screen real coordinates. The user may press the mouse and get real-valued coordinates in the Text window. Notice that the convenient coordinate ranges:

$-1 < x < +1$
$-1 < y < +1$

define a central square region of the Drawing window. Figure 4.1 shows typical behavior of the program.

It is true that any program you can write with the real-valued coordinates could be done with integers alone. Integer graphics are faster as well. But it is often convenient to think in terms of reals, and avoid problems, with dynamic range for example, which occur for integers that are too positive or too negative. The enormous range (powers of ten exceeding ±,1000) of Macintosh Pascal reals means you almost never have to worry about scaling and overflow problems if



```
0.162  -0.162
0.062  -0.785
0.246   0.346
```

Figure 4.1 | Grid output

you do all graphs in terms of real coordinates. Keep in mind that Graphics.lib is simply a translation library that finds the correct integers for you.

A good technique for using libraries or pieces of them is to get them onto the Macintosh Scrapbook, and then Paste them into your source program as described in Chapter 1.

Graphing of Functions

A good beginning for learning graphics is to plot simple functions against two axes. Here is a simple example: choose a function of x and plot it for some interval of interest on the x-axis. Let us try graphing the famous Bessel functions:

$$J_0(x), J_1(x), J_2(x) \ldots$$

which can be calculated using the appropriate function from the Math.lib. These functions arise in the theory of differential equations of the second order. They represent, among other things, the amount of warp in an oscillating, circular membrane (e.g., a drumhead) as a function of distance from the membrane's center. Program 4.2, Bessel, and associated output Figure 4.2 show the basic technique of graphing a function.

Another common example of a function you may want to graph against two axes is that of a *time series*—a set of values defined vs. time. Usually the horizontal axis will be the time axis, with the vertical axis thought of as a voltage signal, a data stream, a temperature, or any other quantity presumed to have time dependence.

A simple but illuminating time series is the famous Fourier sum for a square wave signal. If you add up sinusoidal waves with the correct weighting, you get a square wave. One correct summation is:

$$S(t) = \frac{4}{\pi} \sum_{\substack{j=1 \\ j\,\text{odd}}} \frac{\sin(2\pi jt)}{j}$$

$$= \frac{4}{\pi} * (\sin(2\pi t) + \frac{\sin(6\pi t)}{3} + \frac{\sin(10\pi t)}{5} + \ldots$$

This summation is a so-called 'sum of odd harmonics.' Program 4.3, Squarewave, combines ideas from the last chapter with the notion of real coordinate plotting.

The typical output, Figure 4.3, shows how a square wave is approximated. Note the 'overshoot' near the origin t = 0. This is called *Gibb's phenomenon*, and

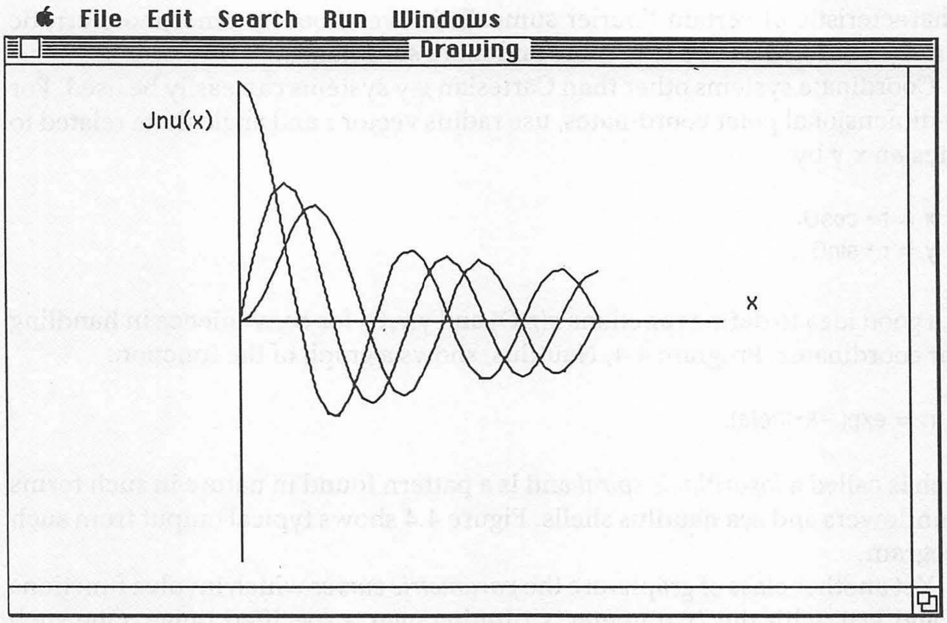
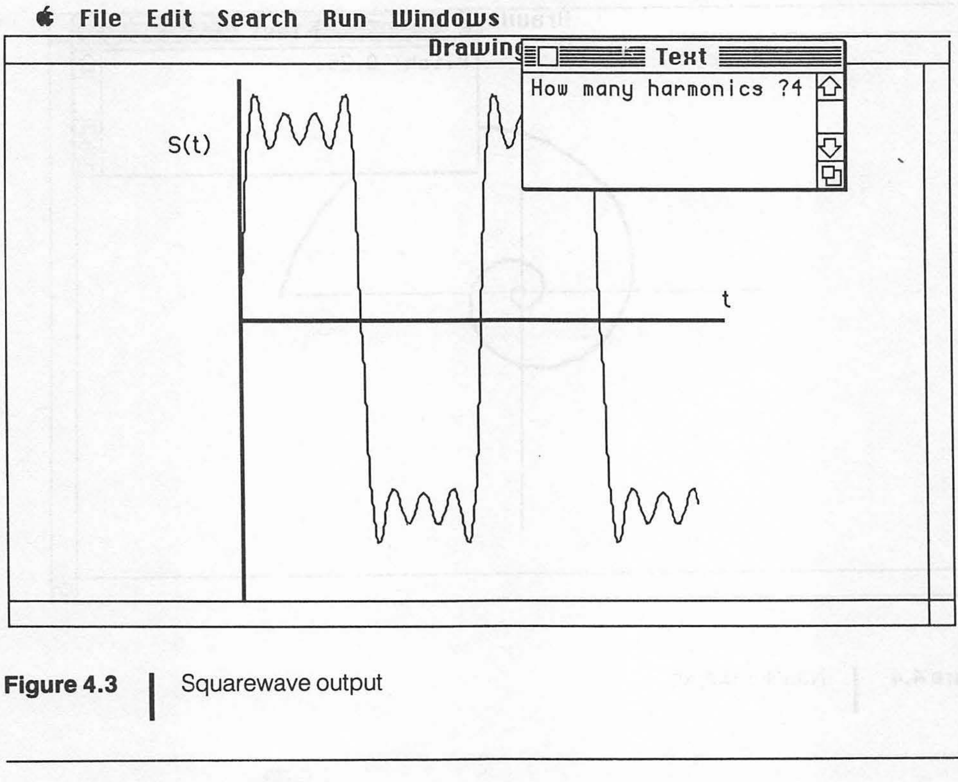


Figure 4.2 | Bessel output



```
How many harmonics ?4
```

Figure 4.3 | Squarewave output

is characteristic of certain Fourier sums. This overshoot becomes more erratic but more confined as you take more and more summands.

Coordinate systems other than Cartesian x-y systems can easily be used. For two-dimensional polar coordinates, use radius vector r and angle theta related to Cartesian x,y by:

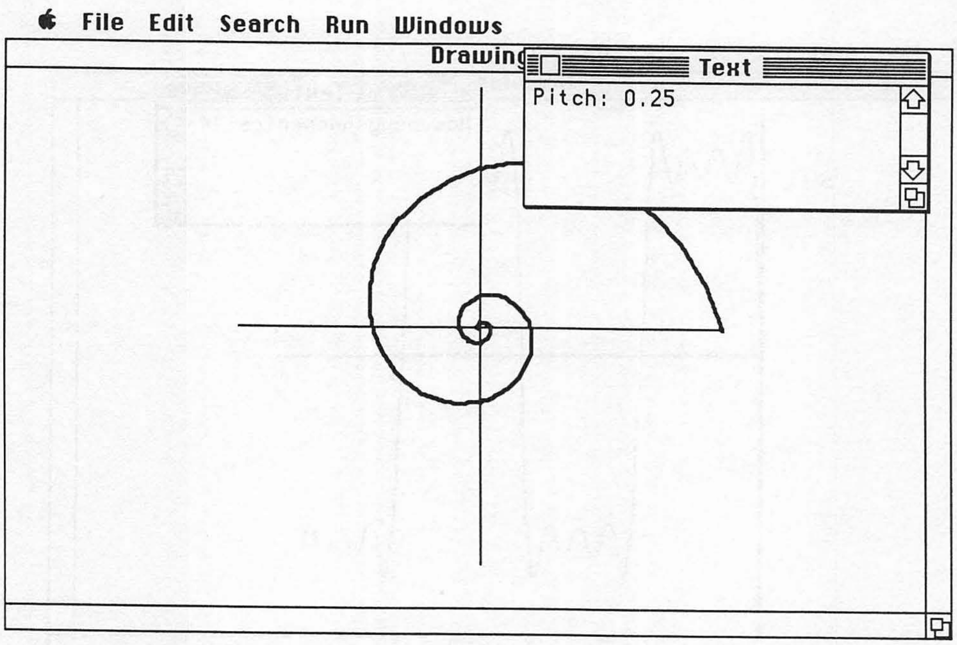
```
x = r * cosΘ
y = r * sinΘ
```

It is a good idea to define functions x(r,Θ) and y(r,Θ) for convenience in handling polar coordinates. Program 4.4, Nautilus, shows a graph of the function:

```
r: = exp( - k*theta);
```

which is called a *logarithmic spiral* and is a pattern found in nature in such forms as sunflowers and sea nautilus shells. Figure 4.4 shows typical output from such a program.

Yet another class of graphs are the *parametric curves* which involve functions x(s) and y(s), with the 'parameter's running over a specified range. One such curve is called the *cycloid*, and is the curve traced by a point on the rim of a rolling wheel. The parametric equations for the cycloid are:



```
Pitch: 0.25
```

Figure 4.4 | Nautilus output

```
x: = a * (s - sin(s));
y: = a * (1 - cos(s));
```

where a is the constant radius of the rolling wheel. Program 4.5, Cycloid, with corresponding output as Figure 4.5, shows how to set up such a problem.

Later, in Chapter 6, considerations such as the above are applied at a more difficult level to three-dimensional coordinate systems.

Graphic Solutions

Not all graphing tasks involve a known function. Indeed there are problems which lend themselves to graphic solution—those in which some number or function is sought as a goal of the programming task. Consider the famous problem of Fermat: "If light leaves point A and is to arrive at point B, what is the path of least time for the transit?"

The answer is a straight line, unless there is a shift in the speed of light due to a change in medium somewhere between A and B. When point A is, for example, in air and B is in glass, strange and wonderful things happen if the Fermat Principle of Least Time is to hold. What is more, the happening actually occurs at the glass-air interface. Program 4.6 Fermat, shows a graphic solution to the problem.

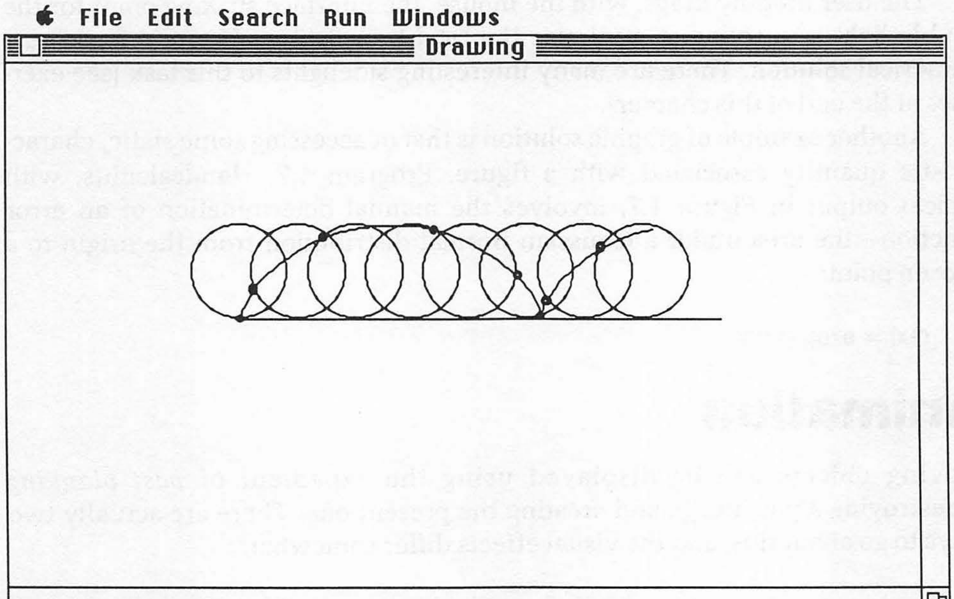
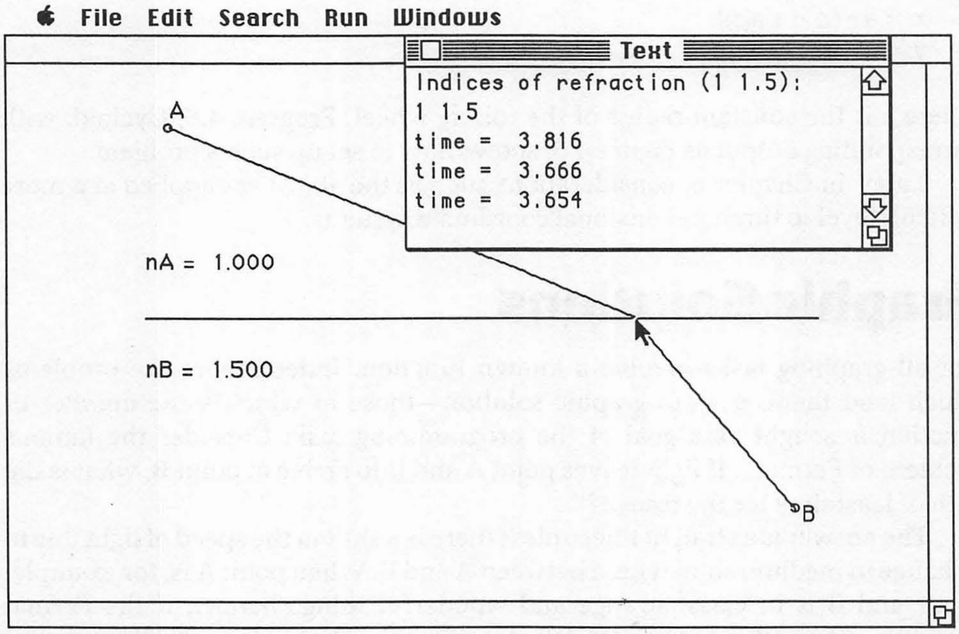


Figure 4.5 | Cycloid output



```
Indices of refraction (1 1.5):
1 1.5
time =  3.816
time =  3.666
time =  3.654
```

Figure 4.6 | Fermat output

The user literally drags, with the mouse, the interface striking point for the double light ray, trying to minimize the total transit time. Figure 4.6 shows a numerical solution. There are many interesting sidelights to this task (see exercises at the end of this chapter).

Another example of graphic solution is that of accessing some static, characteristic quantity associated with a figure. Program 4.7, Handcalculus, with typical output in Figure 4.7, involves the manual determination of an error function—the area under a Gaussian normal distribution from the origin to a chosen point:

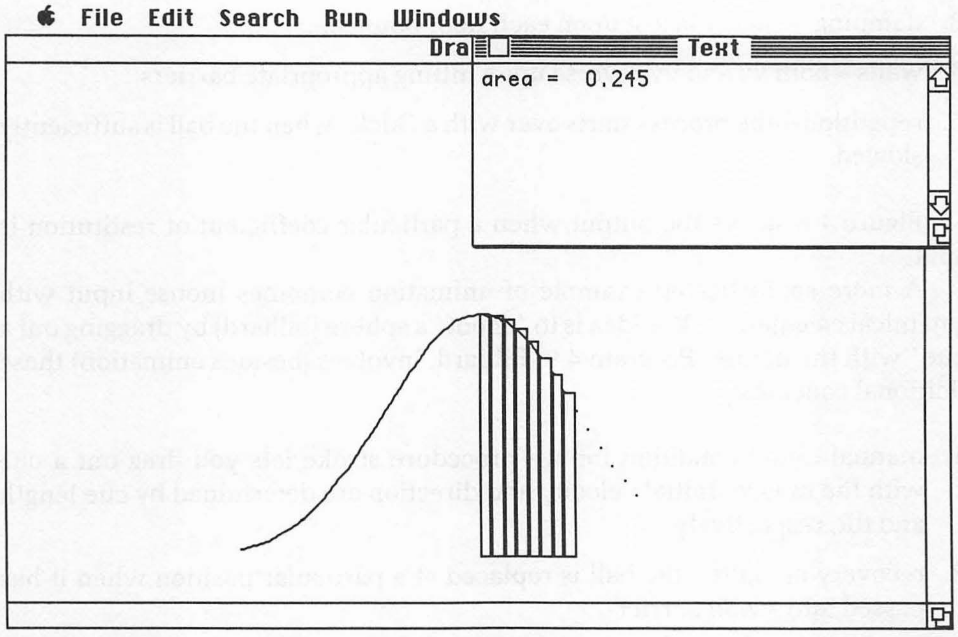
```
p(x) = exp(-x*x);
```

Animation

Moving objects can be displayed using the expedient of *past blanking* —destroying a past image and creating the present one. There are actually two ways to go about this, and the visual effects differ somewhat:

Method 1	Erase the past image and then put up the next image.
Method 2	Put up the next image, then erase the past one.

In either case you loop on such processes to give the effect of motion. Method 1



```
area =   0.245
```

Figure 4.7 | Handcalculus output

suffers from the defect that there will be times when nothing is displayed, so that animated objects appear to 'blink.' Method 2 suffers from the fact that erasure of the past image may well erase part of the present one, leaving holes in the image. In this book we use Method 1 exclusively, on the premise that blinking is not as bad as distortion. However, you should be aware that by using more sophisticated techniques—QuickDraw region definition, system calls for copying bit regions, or the synch procedure for relating time of drawing to the video retrace—you can attempt to 'smooth out' the motion. Throughout this book we use one basic technique and assume that the desired effect should carry scientific meaning without, perhaps, being as visually clean as possible.

The idea of a 'bouncing ball' finds its way into many textbooks, including this one, for the bouncing ball has most of the features of interest for the serious animator. Program 4.8, Superball, involves concepts of:

1. animation—the ball is successively blanked and redrawn in the procedure animate.
2. dynamic calculations—the equations of ballistics, for which the following:

   ```
   vy: = vy - g*dt;
   (* vx is constant between walls *)
   x: = x + vx*dt;
   y: = y + vy*dt;
   ```

   are solved numerically by the program.

3. damping—energy is lost upon each floor bounce.
4. walls—both vx and vy reverse upon hitting appropriate barriers
5. repetition—the process starts over with a 'kick' when the ball is sufficiently slowed.

Figure 4.8 shows the output when a particular coefficient of restitution is input.

A more sophisticated example of animation combines mouse input with dynamical calculation. The idea is to 'shoot' a sphere (billiard) by dragging out a 'cue' with the mouse. Program 4.9, Billiard, involves (besides animation) these additional concepts:

1. manual initial condition input—procedure stroke lets you drag out a cue with the mouse. Initial velocity and direction are determined by cue length and tilt, respectively.
2. recovery at walls—the ball is replaced at a particular position when it has passed into a wall barrier
3. using a function to return a physical parameter—in this case, the function rail returns 0 if no rail (wall) has been struck, else the wall number (1-4) is returned.

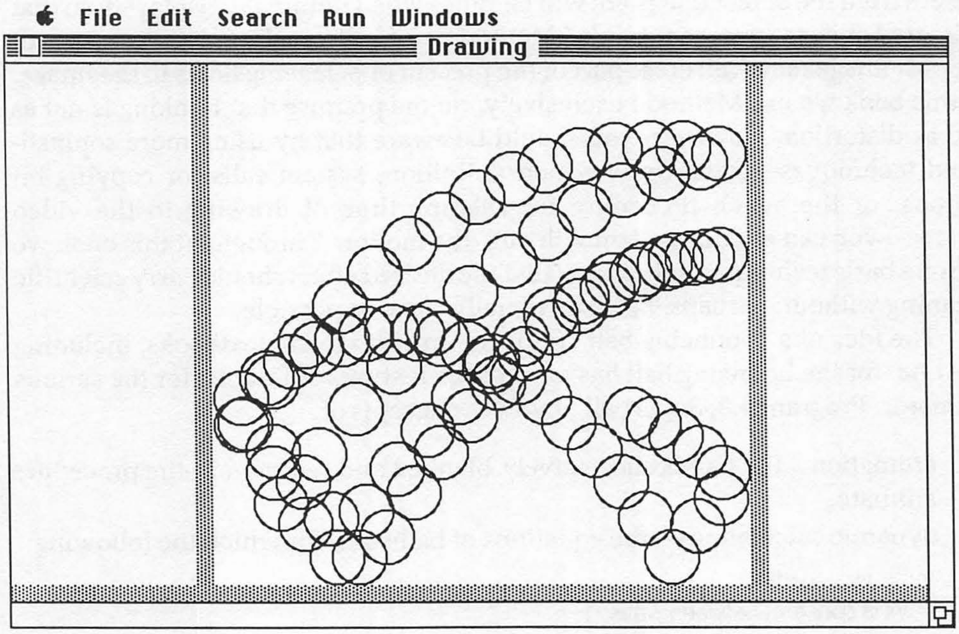


Figure 4.8 | Superball output

Figure 4.9 shows the typical appearance of the billiard table. Note that the cue is made to disappear after a shot is performed. When the ball comes to rest again, a fresh cue may be created, and so on.

In Chapter 7, we investigate more serious dynamical models using animation concepts.

Exercises

1. Write a program similar to Bessel but which lets you press the mouse to get a printout (in the Text window) of zeros (places where the graphs cross the x-axis). You will want to call READMOUSE( ) and if (button) then write both x and Jnu(x) to the Text window. Compare results with standard tables of Bessel functions. How would you increase the accuracy of this graphic method?
2. The GAMMA Function (see Math.lib) is an analytic continuation of the factorial function. In fact:

   Gamma(n+1) = n!

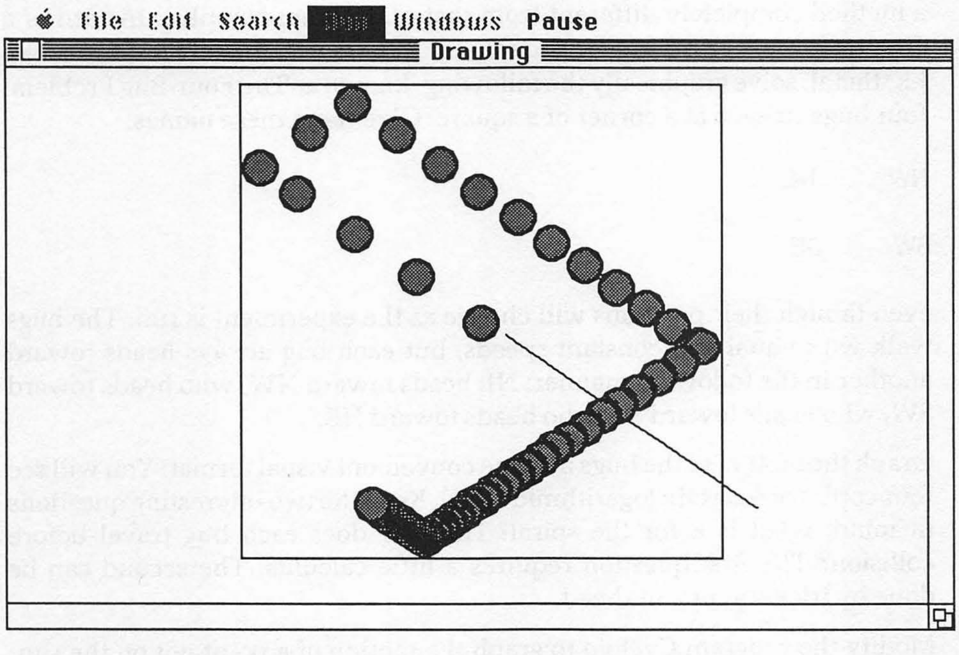


Figure 4.9 | Billiard output

for integers n, but Gamma is defined for all complex numbers having a positive real part (so it is defined as *all positive reals*), and simply passes through the integer factorial values. Write a program which graphs GAMMA(z) for real z on the interval (0,3), plotting also the points:

(1,0!), (2,1!), (3,2!)

through which Gamma must pass. Give also, as a byproduct of your program, a value for z at which Gamma is a minimum.

3. Graph the triangle wave obtained from the Fourier expansion:

$$\text{Tri(t)} = \sum_{j=1} \frac{\cos(2*\pi*j*t)}{(j*j)}$$

This is essentially the integral of the square wave given in the text which is easy to see since the derivative of a triangle wave is clearly a square wave (slopes alternate between two constants).

4. Graph the function tan(x) for x in the interval $(0,\pi)$. You will have to avoid the singularity at $x = \frac{\pi}{2}$.

5. Here you are asked to draw a graph of a certain logarithmic spiral, but using a method completely different from that of program Nautilus. Instead of a "brute-force" graphing approach in which you just calculated exp (-k*theta), solve graphically the following, known as The Four-Bug Problem: four bugs sit each at a corner of a square. Give them these names:

NW	NE
SW	SE

even though their positions will change as the experiment is run. The bugs walk with equal and constant speeds, but each bug always heads toward another in the following manner: NE heads toward NW, who heads toward SW, who heads toward SE, who heads toward NE.

Graph the motion of the bugs in some convenient visual format. You will see four copies of a certain logarithmic spiral. Keep the two interesting questions in mind: what is k for the spiral? How far does each bug travel before collision? The first question requires a little calculus. The second can be done by trickery, in your head.

6. Modify the program Cycloid to graph the motion of a point not on the rim, but rather placed somewhat inward on the wheel. The curve is called an *epicycloid*.

7. Using the program Cycloid, verify numerically the following theoretical facts about this beautiful curve:

   a. The arc length of one arch is 8 times the wheel radius.
   b. The area of one arch is 3 times the wheel area.
   c. The speed of the moving point is precisely sinusoidal in time (assuming the wheel rolls at a constant rotational speed).

8. Interesting sidelights abound for the Fermat's Principle demonstration of program Fermat. The law of refraction known as Snell's Law is a celebrated corollary of the principle. This says that in the minimum-time ray configuration, the angles of incidence (each measured between the ray and a 'normal,' or perpendicular, to the interface—that is the angle between the ray and the y-axis) THETAa and THETAb are related to the indices of refraction Na and Nb by:

   $$\frac{\sin(\Theta)}{\sin(\Theta)} = \frac{Nb}{Na}$$

   Verify Snell's Law numerically by appropriate additions to program Fermat.

9. Here is another sidelight to the refraction problem. Recalling that the speed of propagation in a medium is proportional to $\frac{1}{index}$, write a program in which the ray is actually drawn with the appropriate speed, and report the time—not via algebra, as in the text version of Fermat—by using the tickcount function. This is an example of an even more realistic model of the refraction phenomenon.

10. The famous 'brachistochrone' problem connects the seemingly separate ideas of the cycloid curve and the refraction problem. The problem is: down what shape ramp will an object slide from A to B in minimum time? Write a program to draw the appropriate curve from A to B by using Isaac Newton's ingenious method of solution: observe that, since an object having slid through a cumulative height of h (from original point A) has a speed, along the ramp, proportional to sqrt(h), the problem is the same as that of a light ray bending through a medium whose index of refraction behaves as $\frac{1}{sqrt(h)}$. Thus successive applications of Snell's Law (Exercise 8) should draw a good brachistochrone, which is actually a segment of a cycloid.

11. Write a program which reports the arc length of a user-defined function f(x) between two mouse-selected points. The integral for arc length is:

    $$S = \int sqrt\left(1 + sqr\left(\frac{df}{dx}\right)\right) dx$$

where the limits of integration are the two x-coordinates defining the arc endpoints. Test this on the function known as a catenary—the curve in which a chain will hang—given by:

$$f(x) = \cosh(x) = \frac{\exp(x) + \exp(-x)}{2}$$

12. Write a program to 'juggle' three balls. The initial configuration should be:

$$\frac{\text{A B}}{\text{Left}} \qquad \frac{\text{C}}{\text{Right}}$$

in the Graphics window, with balls A, B, C initially at rest. The left and right 'hands' are just short lines drawn on the screen and need not move (although it is fascinating to try and give them mouse control!). Then the juggling mode known as a cascade works like this:

a. Launch ball A in a parabola that will land it on the right hand.
b. When A has just about reached the top of its trajectory, launch ball B in a parabola destined for the left hand.
c. Just when ball C is near the top of its parabola, launch ball B in a third parabola.
d. Continue like this, the next step being that ball A should launch when ball B is near its top, and so on.

This problem is not so much a ballistics exercise as an exercise in combinatorics; a little cleverness is required to efficiently program the alternating sequence.

13. Write a graphics program which has several points of differing masses bouncing around within a one-dimensional interval, without damping. It is convenient to graph each point x[j] by a vertical stroke indicating its mass, but motion should only be along the x-axis. It turns out that it is fairly easy to assess whether or not balls have struck each other: just compute the distance x[i] − x[j] between a pair of masses and if this distance has not changed sign, then the masses have not recently touched. The far left and far right masses should, of course, reflect off of the 'walls' of the interval. When masses m[i] and m[j] collide, they simply trade velocities. With some work, you can verify the Equipartition Theorem of Thermodynamics: regardless of initial conditions (for example, one mass moving at first but no others), the mean kinetic energy, $\frac{(\text{mass} \cdot \text{sqr(velocity)})}{2}$ of each object is the same: heavier objects move more slowly on the average, assuming statistical equilibrium has been reached. This is a beautiful project which can be extended with far more work to higher dimensions, for which equipartition of energy still holds.

Answers

1. You can do, after the drawing, something like:

```
repeat
  if button then
    begin
      readmouse(x,y);
      writeln(x, j(nu,x));
    end;
until false;
```

2. This is a straightforward graphics problem, which should use the library procedures SHIFT and DRAW. The z for which gamma has a minimum is between 1 and 2.
3. This is a straightforward alteration of the Squarewave program.
4. This is a straightforward graphics program. One way to avoid the singularity is to start with a constant dx $= \frac{\pi}{100}$ set x $= \frac{dx}{2}$; then draw in increments of dx. This will cause two points nearby $\frac{\pi}{2}$ to 'straddle' the singularity.
5. The idea is to set up four velocity vectors and have them point in the right directions, but always normalizing them to have constant speed. The bugs meet smack in the center, and each one travels a distance equal to the side of the original square. Why? Because each velocity vector is perpendicular to the next, so a bug 'gains' on its destination bug at the same rate it would if the destination were not in motion.
6. The parametric equations are different:

```
x:= a*theta - r*sin(theta);
y:= a - r*cos(theta);
```

   where r is the distance to the (non-peripheral) point.
7. The first two verifications use previous techniques of integration. Note that the arc length element is:

```
ds = sqrt(sqr(dx) + sqr(dy));
```

   where

```
dx = dtheta * a * (1-cos(theta));
dy = dtheta * a * sin(theta);
```

and that speed is proportional to $\frac{ds}{dtheta}$, where constant angular velocity of the generating wheel is assumed.

8. If you have the interface striking point (x,y), then the angles can be obtained from the rule:

$$\tan(\Theta) = \frac{(x\text{-}xa)}{(ya\text{-}y)}$$

$$\tan(\Theta) = (y\text{-}yb)$$

and you need the relation between sine and tangent:

$$\sin(z) = \frac{\tan(z)}{\text{sqrt}(1+\text{sqr}(\tan z)}$$

9. This should be done by setting up velocity vector (vx,vy), which is designed to point at the interface, with magnitude $\frac{1}{index}$. This can be done by finding Δx, Δy as (x-xa), (y-ya) and normalizing:

```
vx:= Δx;
vy:= Δy;
norm:= sqrt(sqr(Δx) + sqr(Δy));
```

$$vx := \frac{vx}{(norm*index)}$$

$$vy := \frac{vy}{(norm*index)}$$

and iterating the equations of motion according to:

```
x:= xa;
y:= ya;
repeat
x:= x + vx*dt;
y:= y + vy*dt;
t:= t + dt;
until abs(y-ya) < epsilon;
```

where the loop termination condition amounts to the fact of striking the interface. A similiar method will get the outgoing ray, but the destination is now (xb,yb) and the index is the second index. If you keep track of tick-counts, you will find that the actual minimum time (now real time, of course) is obtained for the correct refractive path.

10. The idea is to find the correct differential equations on the basis of Newton's idea. Then the iteration is a straightforward example of numerical calculus. Since the sine of the trajectory's angle with respect to the vertical will be proportional to the speed (see Exercise 8), we have, for arc element ds:

```
dx: = K * sqrt(ya - y) * ds;
dy: = sqrt(1 - K2*(ya-y));
```

for some constant K and initial height ya. Then you fix a constant ds and iterate:

```
x: = xa;
y: = ya;
SHIFT(x,y);
repeat
(* calculate dx,dy here as above, then: *)
x: = x + dx;
y: = y + dy;
DRAW(x,y);
until dy<0
```

It is of interest that for some initial and final points the solution actually turns upward before terminating. It is best, though, to ignore the final point for this program—it is very tough to find the solution which meets a given endpoint, but every trajectory is the brachistochrone solution for some endpoint.

11. The arc length for f(x) = cosh(x) from x = 0 to x = a is given exactly by:

    $$S = \sinh(a) = (\exp(a) = \frac{\exp(-a)}{2}$$

    which should be compared to your numerical integral.

12. The ballistics problem is straightforward; you find the initial vx and vy such that a ball will land on the other hand. The combinatorics problem is best solved by calling the balls by 0, 1, 2 (mod 3) and launching, for ball n in the right air position, ball (n + 1) mod 3.

13. Two masses j and k 'contact' when the sign of the difference (x[j]-x[k]) flips, at which point you should swap the velocities.

Program 4.1

```
program grid;
(* SHOWS MEANING OF REAL-VALUED COORDINATES *)

 var
  x, y : real;
  m : integer;

 procedure CLEAR;
(* Activates and expands Drawing Window to fill screen *)
  var
   windowrect : rect;
 begin
  hideall;
  setrect(windowrect, 2, 35, 512, 342);
  setdrawingrect(windowrect);
  showdrawing;
 end;

 procedure MAP (x, y : real;
         var hor, ver : integer);
(* Returns hor,ver as pixel integers for given real coords (x,y) *)
 begin
  if abs(x) > 1.9 then
   x := x / abs(x) * 1.9;
  hor := 255 + trunc(x * 130);
  if abs(y) > 1 then
   y := y / abs(y);
  ver := 138 - trunc(y * 130);
 end;

 procedure DRAW (x, y : real);
(* Draws to (x,y) *)
  var
   h1, v1 : integer;
 begin
  MAP(x, y, h1, v1);
  lineto(h1, v1);
 end;

 procedure SHIFT (x, y : real);
(* moves invisible to (x,y) *)
  var
   h1, v1 : integer;
 begin
  MAP(x, y, h1, v1);
  moveto(h1, v1);
 end; {shift}
 procedure UNMAP (h, v : integer;
         var x, y : real);
 begin
  x := (h - 255) / 130;
  y := (138 - v) / 130;
 end;

 procedure REALMOUSE (var x, y : real);
  var
   m, n : integer;
 begin
  getmouse(m, n);
  UNMAP(m, n, x, y);
 end;

begin
 CLEAR;
 showtext;
 for m := -10 to 10 do
  begin
   SHIFT(m / 10, 1);
   DRAW(m / 10, -1);
   SHIFT(1, m / 10);
   DRAW(-1, m / 10);
  end;
 pensize(2, 2);
 SHIFT(1, 0);
 DRAW(-1, 0);
 SHIFT(0, 1);
 DRAW(0, -1);
 repeat
  REALMOUSE(x, y);
  if (button) and (y < 1.05) then
   begin
    writeln(x : 8 : 3, y : 8 : 3);
    SHIFT(x, y);
    DRAW(x, y);
   end;
 until false;
end.
```

Program 4.2

```
program Bessel;
(* PLOTS FAMILY OF BESSEL FUNCTIONS *)

 const
  dx = 0.45;
 var
```

(continued)

```
  x : real;
  nu : integer;

 procedure CLEAR;
(* Activates and expands Drawing Window to fill screen *)
  var
   windowrect : rect;
 begin
  hideall;
  setrect(windowrect, 2, 35, 512, 342);
  setdrawingrect(windowrect);
  showdrawing;
 end;

 procedure MAP (x, y : real;
         var hor, ver : integer);
(* Returns hor,ver as pixel integers for given real coords (x,y) *)
 begin
  if abs(x) > 1.9 then
   x := x / abs(x) * 1.9;
  hor := 255 + trunc(x * 130);
  if abs(y) > 1 then
   y := y / abs(y);
  ver := 138 - trunc(y * 130);
 end;

 procedure DRAW (x, y : real);
(* Draws to (x,y) *)
  var
   h1, v1 : integer;
 begin
  MAP(x, y, h1, v1);
  lineto(h1, v1);
 end;

 procedure SHIFT (x, y : real);
(* moves invisible to (x,y) *)
  var
   h1, v1 : integer;
 begin
  MAP(x, y, h1, v1);
  moveto(h1, v1);
 end;

 function GAMMA (z : real) : real;
(* gamma function of z > 0 *)

  const
   pi = 3.14159265358979323846З;
  var
   zz : real;
```

```
begin
 if (z > 1) then
  GAMMA := (z - 1) * GAMMA(z - 1)
 else if (z = 1) then
  GAMMA := 1
 else if (z > 0.5) then
  GAMMA := pi / (sin(pi * z) * GAMMA(1 - z))
 else
  begin
    zz := 1 / z - 0.5748646 + 0.9512363 * z - 0.6998588 * z * z;
    GAMMA := zz + 0.4245549 * z * z * z - 0.1010678 * z * z * z * z;
   end;
 end;

 function J (nu, z : real) : real;
(* The Bessel function of order nu *)
  var
   n, d, f, sum : real;
   ctr : integer;
 begin
  if z = 0 then
   begin
    if nu = 0 then
     J := 1
    else
     J := 0
   end
  else
   begin
    f := 1 / GAMMA(nu + 1);
    n := -(sqr(z) / 4);
    d := nu + 1;
    sum := f;
    ctr := 1;
    repeat
     f := f * n / (ctr * d);
     d := d + 1;
     ctr := ctr + 1;
     sum := sum + f
    until abs(f) < 1e-15;
    if z > 0 then
     J := sum * exp(nu * ln(z / 2))
    else
     J := (1 - 2 * (trunc(nu) mod 2)) * sum * exp(nu * ln(abs(z / 2)))
   end
 end;

begin
 CLEAR;
 SHIFT(1, 0);
```

(continued)

```
  DRAW(-1, 0);
  SHIFT(-1, 1);
  DRAW(-1, -1);
  SHIFT(-1.4, 0.8);
  writedraw('Jnu(x)');
  SHIFT(1.1, 0.05);
  writedraw('x');
  for nu := 0 to 2 do
   begin
    x := 0;
    repeat
     if x < dx / 2 then
      SHIFT(x / 10 - 1, J(nu, 0))
     else
      DRAW(x / 10 - 1, J(nu, x));
     x := x + dx;
    until x > 15;
   end;
end.
```

Program 4.3

```
program squarewave;
(* GRAPHS PARTIAL FOURIER SUM FOR SQUARE WAVE SIGNAL *)

 const
  pi = 3.14159265359;
  dt = 0.01;

 var
  t, sum : real;
  j, n : integer;

 procedure CLEAR;
(* Activates and expands Drawing Window to fill screen *)
  var
   windowrect : rect;
 begin
  hideall;
  setrect(windowrect, 2, 35, 512, 342);
  setdrawingrect(windowrect);
  showdrawing;
 end;

 procedure MAP (x, y : real;
         var hor, ver : integer);
(* Returns hor,ver as pixel integers for given real coords (x,y) *)
 begin
```

```
if abs(x) > 1.9 then
 x := x / abs(x) * 1.9;
   hor := 255 + trunc(x * 130);
   if abs(y) > 1 then
    y := y / abs(y);
   ver := 138 - trunc(y * 130);
  end;

  procedure DRAW (x, y : real);
 (* Draws to (x,y) *)
   var
    h1, v1 : integer;
  begin
   MAP(x, y, h1, v1);
   lineto(h1, v1);
  end;

  procedure SHIFT (x, y : real);
 (* moves invisible to (x,y) *)
   var
    h1, v1 : integer;
  begin
   MAP(x, y, h1, v1);
   moveto(h1, v1);
  end;

begin
 t := 0;
 CLEAR;
 showtext;
 write('How many harmonics ?');
 readln(n);
 SHIFT(1, 0);
 pensize(2, 2);
 DRAW(-1, 0);
 SHIFT(-1, -1);
 DRAW(-1, 1);
 pensize(1, 1);
 SHIFT(-1.3, 0.7);
 writedraw('S(t)');
 SHIFT(1, 0.05);
 writedraw('t');
 repeat
  sum := 0;
  for j := 1 to n do
   sum := sum + sin(2 * pi * t * (2 * j - 1)) / (2 * j - 1);
  sum := sum * 4 / pi;
  if t < dt / 2 then
```

(continued)

```
    SHIFT(-1 + t, sum * 0.8)
   else
    DRAW(-1 + t, sum * 0.8);
   t := t + dt;
  until t > 1.9;
 end.
```

Program 4.4

```
program nautilus;
(* LOGARITHMIC SPIRAL IN POLAR COORDINATES *)

 const
  dtheta = 0.2;

 var
  r, theta, k : real;

 function x (r, theta : real) : real;
 begin
  x := r * cos(theta);
 end;

 function y (r, theta : real) : real;
 begin
  y := r * sin(theta);
 end;

 procedure CLEAR;
(* Activates and expands Drawing Window to fill screen *)
  var
   windowrect : rect;
 begin
  hideall;
  setrect(windowrect, 2, 35, 512, 342);
  setdrawingrect(windowrect);
  showdrawing;
 end;

 procedure MAP (x, y : real;
         var hor, ver : integer);
(* Returns hor,ver as pixel integers for given real coords (x,y) *)
 begin
  if abs(x) > 1.9 then
   x := x / abs(x) * 1.9;
  hor := 255 + trunc(x * 130);
  if abs(y) > 1 then
   y := y / abs(y);
  ver := 138 - trunc(y * 130);
 end;
 procedure DRAW (x, y : real);
(* Draws to (x,y) *)
  var
   h1, v1 : integer;
 begin
  MAP(x, y, h1, v1);
  lineto(h1, v1);
 end;

 procedure SHIFT (x, y : real);
(* moves invisible to (x,y) *)
  var
   h1, v1 : integer;
 begin
  MAP(x, y, h1, v1);
  moveto(h1, v1);
 end;

begin
 CLEAR;
 showtext;
 write('Pitch: ');
 readln(k);
 SHIFT(1, 0);
 DRAW(-1, 0);
 SHIFT(0, -1);
 DRAW(0, 1);
 theta := 0;
 SHIFT(1, 0);
 pensize(2, 2);
 repeat
  r := exp(-k * theta);
  DRAW(x(r, theta), y(r, theta));
  theta := theta + dtheta;
 until r < 0.02;
end.
```

Program 4.5

```
program cycloid;
(* PARAMETRIC GRAPH OF CYCLOID CURVE *)

 const
  ds = 0.2;
  a = 0.2;

 var
  s : real;
  ctr : integer;
```

(continued)

```
function x (s : real) : real;
begin
 x := a * (s - sin(s));
end;

function y (s : real) : real;
begin
 y := a * (1 - cos(s));
end;

procedure CLEAR;
(* Activates and expands Drawing Window to fill screen *)
 var
  windowrect : rect;
begin
 hideall;
 setrect(windowrect, 2, 35, 512, 342);
 setdrawingrect(windowrect);
 showdrawing;
end;

procedure MAP (x, y : real;
        var hor, ver : integer);
(* Returns hor,ver as pixel integers for given real coords (x,y) *)
begin
 if abs(x) > 1.9 then
  x := x / abs(x) * 1.9;
 hor := 255 + trunc(x * 130);
 if abs(y) > 1 then
  y := y / abs(y);
 ver := 138 - trunc(y * 130);
end;

procedure DRAW (x, y : real);
(* Draws to (x,y) *)
 var
  h1, v1 : integer;
begin
 MAP(x, y, h1, v1);
 lineto(h1, v1);
end;

procedure SHIFT (x, y : real);
(* moves invisible to (x,y) *)
 var
  h1, v1 : integer;
begin
 MAP(x, y, h1, v1);
 moveto(h1, v1);
end;
```

```
procedure CIR (x, y, r : real);
 var
  h1, v1, h2, v2 : integer;
begin
 MAP(x - r, y + r, h1, v1);
 MAP(x + r, y - r, h2, v2);
 frameoval(v1, h1, v2, h2);
end;

begin
 CLEAR;
 SHIFT(1, 0);
 DRAW(-1, 0);
 SHIFT(-1, 0);
 s := 0;
 ctr := 0;
 repeat
  DRAW(x(s) - 1, y(s));
  if ctr mod 6 = 0 then
   begin
    CIR(a * s - 1, a, a);
    pensize(2, 2);
    CIR(x(s) - 1, y(s), 0.02);
    pensize(1, 1);
   end;
  ctr := ctr + 1;
  s := s + ds;
 until x(s) > 1.75;
end.
```

Program 4.6

```
program fermat;
(* FERMAT'S PRINCIPLE OF LEAST TIME - USER FINDS *)
(* VARIATIONAL LEAST-TIME PATH *)

 const
  xa = -1.3;
  ya = 0.8;
  xb = 1.3;
  yb = -0.8;

 var
  t, x, y, xstrike, na, nb : real;
  newchoice : boolean;

 procedure CLEAR;
(* Activates and expands Drawing Window to fill screen *)
  var
```

(continued)

```
   windowrect : rect;
 begin
  hideall;
  setrect(windowrect, 2, 35, 512, 342);
  setdrawingrect(windowrect);
  showdrawing;
 end;

 procedure MAP (x, y : real;
         var hor, ver : integer);
(* Returns hor,ver as pixel integers for given real coords (x,y) *)
 begin
  if abs(x) > 1.9 then
   x := x / abs(x) * 1.9;
  hor := 255 + trunc(x * 130);
  if abs(y) > 1 then
   y := y / abs(y);
  ver := 138 - trunc(y * 130);
 end;

 procedure DRAW (x, y : real);
(* Draws to (x,y) *)
  var
   h1, v1 : integer;
 begin
  MAP(x, y, h1, v1);
  lineto(h1, v1);
 end;

 procedure SHIFT (x, y : real);
(* moves invisible to (x,y) *)
  var
   h1, v1 : integer;
 begin
  MAP(x, y, h1, v1);
  moveto(h1, v1);
 end;

 procedure CIR (x, y, r : real);
(* Draws circle of radius r centered at (x,y) *)
  var
   h1, h2, v1, v2 : integer;
 begin
  MAP(x - r, y + r, h1, v1);
  MAP(x + r, y - r, h2, v2);
  frameoval(v1, h1, v2, h2);
 end;

 procedure UNMAP (h, v : integer;
        var x, y : real);
begin
 x := (h - 255) / 130;
 y := (138 - v) / 130;
end;

procedure REALMOUSE (var x, y : real);
 var
  m, n : integer;
begin
 getmouse(m, n);
 UNMAP(m, n, x, y);
end;

procedure newrays (var xstrike : real;
        x : real);
begin
 penpat(white);
 SHIFT(xa, ya);
 DRAW(xstrike, 0.01);
 SHIFT(xstrike, -0.01);
 DRAW(xb, yb);
 penpat(black);
 SHIFT(xa, ya);
 DRAW(x, 0.01);
 SHIFT(x, -0.01);
 DRAW(xb, yb);
 xstrike := x;
end;

begin
 CLEAR;
 showtext;
 writeln('Indices of refraction (1 1.5):');
 readln(na, nb);
 CIR(xa, ya, 0.02);
 SHIFT(xa, ya + 0.04);
 writedraw('A');
 CIR(xb, yb, 0.02);
 SHIFT(xb + 0.03, yb - 0.06);
 writedraw('B');
 SHIFT(-1.4, 0);
 DRAW(1.4, 0);
 SHIFT(-1.4, 0.2);
 writedraw('nA = ', na : 6 : 3);
 SHIFT(-1.4, -0.25);
 writedraw('nB = ', nb : 6 : 3);
 xstrike := (xa * yb - xb * ya) / (yb - ya);
 newrays(xstrike, xstrike);
```

(continued)

```
newchoice := true;
repeat
 while button do
  begin
   REALMOUSE(x, y);
   if (y < 1.05) then
    begin
     newrays(xstrike, x);
     newchoice := true;
    end;
  end;
 if newchoice then
  begin
   t := sqrt(sqr(xstrike - xa) + sqr(ya)) * na;
   t := t + sqrt(sqr(xstrike - xb) + sqr(yb)) * nb;
   writeln('time = ', t : 6 : 3);
   newchoice := false;
   end;
 until false;
end.
```

Program 4.7

```
program handcalculus;
(* GIVES AREA FROM ORIGIN TO MOUSE UNDER A NORMAL DISTRIBUTION *)

 const
  bins = 20;

 var
  x, y, integral : real;
  keybin, n, h, v, hh, vv : integer;
  newintegral : boolean;

 function f (x : real) : real;
  const
   pi = 3.14159265;
   variance = 0.15;
 begin
  f := 1 / sqrt(2 * pi * variance) * exp(-sqr(x) / (2 * variance));
 end;

 procedure CLEAR;
(* Activates and expands Drawing Window to fill screen *)
  var
   windowrect : rect;
 begin
  hideall;
  setrect(windowrect, 2, 35, 512, 342);
  setdrawingrect(windowrect);
  showdrawing;
 end;

 procedure MAP (x, y : real;
         var hor, ver : integer);
(* Returns hor,ver as pixel integers for given real coords (x,y) *)
 begin
  if abs(x) > 1.9 then
   x := x / abs(x) * 1.9;
  hor := 255 + trunc(x * 130);
  if abs(y) > 1 then
   y := y / abs(y);
  ver := 138 - trunc(y * 130);
 end;

 procedure DRAW (x, y : real);
(* Draws to (x,y) *)
  var
   h1, v1 : integer;
 begin
  MAP(x, y, h1, v1);
  lineto(h1, v1);
 end;

 procedure SHIFT (x, y : real);
(* moves invisible to (x,y) *)
  var
   h1, v1 : integer;
 begin
  MAP(x, y, h1, v1);
  moveto(h1, v1);
 end;

 procedure UNMAP (h, v : integer;
         var x, y : real);
 begin
  x := (h - 255) / 130;
  y := (138 - v) / 130;
 end;

 procedure REALMOUSE (var x, y : real);
  var
   m, n : integer;
 begin
  getmouse(m, n);
  UNMAP(m, n, x, y);
 end;

begin
 CLEAR;
```

(continued)

```
 showtext;
 for n := -bins to bins do
  begin
   x := n / bins;
   if n = -bins then
    SHIFT(x, f(x) - 1)
   else
    DRAW(x, f(x) - 1);
  end;
 repeat
  newintegral := false;
  while button do
   begin
    REALMOUSE(x, y);
    keybin := round(x * bins);
    if keybin >= 0 then
     begin
      newintegral := true;
      integral := 0;
      for n := 0 to bins do
       begin
        MAP(n / bins, f(n / bins) - 1, h, v);
        MAP((n + 1) / bins, -1, hh, vv);
        if n >= keybin then
         eraserect(v, h, vv, hh)
        else
         begin
          framerect(v, h, vv, hh);
          integral := integral + f(n / bins) / bins;
         end;
       end;
     end;
   end;
  if newintegral then
   writeln('area = ', integral : 6 : 3);
  newintegral := false;
 until false;
end.
```

Program 4.8

```
program superball;
(* DEMONSTRATES ANIMATION, BALLISTICS, WALL REFLECTION, AND DAMPING *)

 const
  r = 15;
  dt = 0.04;
  g = 9.8;
 var
  x, y, xold, yold, vx, vy, restitution : real;
  h, v : integer;

 procedure CLEAR;
  var
   windowrect : rect;
 begin
  hideall;
  setrect(windowrect, 2, 35, 512, 342);
  setdrawingrect(windowrect);
  showdrawing;
 end;

 procedure MAP (x, y : real;
         var hor, ver : integer);
 begin
  if abs(x) > 1.9 then
   x := x / abs(x) * 1.9;
  hor := 255 + round(x * 130);
  if abs(y) > 1 then
   y := y / abs(y);
  ver := 138 - round(y * 130);
 end;

 procedure animate (x, y : real;
         var xold, yold : real);
  var
   h1, v1 : integer;
 begin
(*    Remove next comment marks to get animation *)
(*    MAP(xold, yold, h1, v1);  *)
(*    eraseoval(v1 - r, h1 - r, v1 + r, h1 + r);  *)
  MAP(x, y, h1, v1);
  frameoval(v1 - r, h1 - r, v1 + r, h1 + r);
  xold := x;
  yold := y;
 end;

begin
 showtext;
 write('Coefficient of restitution (0.9): ');
 readln(restitution);
 CLEAR;
 penpat(ltgray);
 MAP(-1, 1, h, v);
 paintrect(v, h - r - 10, v + 285 + r, h - r);
 MAP(1, 1, h, v);
 paintrect(v, h + r, v + 285 + r, h + r + 10);
 MAP(-2, -1.1, h, v);
 paintrect(v + r, h + r, v + 10 + r, h + 500);
 penpat(black);
 x := -1;
 y := -1;
```

(continued)

```
  xold := x;
  yold := y;
  vx := +4;
  vy := +6;
  repeat
  repeat
   vy := vy - g * dt;
   y := y + vy * dt;
   x := x + vx * dt;
   if abs(y) > 1 then
    begin
     vy := -vy * restitution;
     y := y / abs(y);
    end;
   if abs(x) > 1 then
    begin
     vx := -vx * restitution;
     x := x / abs(x);
    end;
   animate(x, y, xold, yold);
  until (abs(vy) < 0.005) and (y < -0.95);
  vy := 6;
  if (vx > 0) then
   vx := 4
  else
   vx := -4;
 until false;
end.
```

Program 4.9

```
program billiard;
(* USER USES MOUSE AS CUE TO LAUNCH SINGLE BILLIARD *)

 const
  r = 10;
  dt = 0.02;
  friction = 0.94;
  pi = 3.14159265;

 var
  x, y, xold, yold, x0, y0, x1, y1, dx, dy, rr : real;
  h, v, hh, vv : integer;

 function rail (var x, y : real) : integer;
 begin
  if y < -1 + rr then
   begin
    rail := 1;
    y := -1 + rr;
   end
   else if x > 1 - rr then
    begin
     x := 1 - rr;
     rail := 2;
    end
   else if y > 1 - rr then
    begin
     rail := 3;
     y := 1 - rr;
    end
   else if x < -1 + rr then
    begin
     rail := 4;
     x := -1 + rr;
    end
   else
    rail := 0;
  end;

 procedure CLEAR;
 (* Size and clear the Drawing Window *)
  var
   windowrect : rect;
 begin
  hideall;
  setrect(windowrect, 2, 35, 512, 342);
  setdrawingrect(windowrect);
  showdrawing;
 end;

  procedure MAP (x, y : real;
          var hor, ver : integer);
  begin
   if abs(x) > 1.9 then
    x := x / abs(x) * 1.9;
   hor := 255 + round(x * 130);
   if abs(y) > 1 then
    y := y / abs(y);
   ver := 138 - round(y * 130);
  end;

  procedure UNMAP (h, v : integer;
          var x, y : real);
  begin
   x := (h - 255) / 130;
   y := (138 - v) / 130;
  end;

  procedure stroke (var x, y, xold, yold : real);
 {animates a mouse-stroke, returns endpoint (x, y), original}
 {point (xold, yold) as reals}
```

(continued)

```
  var
   h1, h2, v1, v2 : integer;
 begin
  while not button do
   begin
   end;
  getmouse(h1, v1);
  h2 := h1;
  v2 := v1;
  while button do
   begin
    penpat(white);
    moveto(h1, v1);
    lineto(h2, v2);
    getmouse(h2, v2);
    moveto(h1, v1);
    penpat(black);
    lineto(h2, v2);
   end;
  penpat(white);
  moveto(h1, v1);
  lineto(h2, v2);
  penpat(black);
  UNMAP(h1, v1, xold, yold);
  UNMAP(h2, v2, x, y);
 end;

 procedure animate (x, y : real;
         var xold, yold : real);
{erase 'old' ball and display new one}
  var
   h1, v1 : integer;
 begin
  MAP(xold, yold, h1, v1);
  eraseoval(v1 - r, h1 - r, v1 + r, h1 + r);
  MAP(x, y, h1, v1);
  paintoval(v1 - r, h1 - r, v1 + r, h1 + r);
  xold := x;
  yold := y;
 end;

begin
CLEAR;
 x := 0;
 y := 0;
 xold := x;
 yold := y;
 repeat
  MAP(-1, 1, h, v);
  MAP(1, -1, hh, vv);
  framerect(v, h, vv, hh);
  animate(x, y, xold, yold);
```

```
   stroke(x1, y1, x0, y0);
   dx := -13 * (x1 - x0) * dt;
   dy := -13 * (y1 - y0) * dt;
   rr := (r + 1) / 130;
   repeat
    repeat
     hh := rail(x, y);
     case hh of
      0 :
      ;
     1, 3 :
      begin
       if dy <> 0 then
        x := xold + (y - yold) * dx / dy;
       dy := -dy;
      end;
     2, 4 :
      begin
       if dx <> 0 then
        y := yold + (x - xold) * dy / dx;
       dx := -dx;
      end;
    end;
   until hh = 0;
   animate(x, y, xold, yold);
   x := x + dx;
   y := y + dy;
   dx := dx * friction;
   dy := dy * friction;
  until abs(dx) + abs(dy) < 0.02;
 until false;
end.
```

5 Probability and Statistics

THEME: Statistics, especially when applied to physical data, is a ubiquitous aspect of scientific programming and so, along with the underlying theoretical concepts of probability, is given a whole chapter. Various concepts ranging from random numbers and sorting to disk file and data-array handling are covered.

GOALS: To be able to process data using basic notions of probability and statistics. You will be able to access disk files for this purpose.

LIBRARIES USED: Stat.lib, Graphics.lib

REFERENCE MATERIALS: Any text on elementary statistics covers the basic definitions and nomenclature.

Random Integers

Random integers can be generated from within Mac Pascal programs through use of the built-in random function. This function returns an integer between −32767 and +32767. In applications requiring random numbers, however, some processing of these values is usually required. One of the simplest transformations of the built-in random function is to generate a random integer between 1 and N inclusive, where N is an integer. It turns out that the expression:

```
1 + random mod N
```

is integer valued and equidistributed amongst the sequence 1,2,...,N. This notion is used in Program 5.1, PokerDeal, in which random integers are used to shuffle a deck of cards.

Typical output is shown in Figure 5.1. The program is named PokerDeal and not Poker because it does not really play a full game, but it does do the initial dealing of two hands correctly. The shuffle procedure uses the following straightforward method of shuffling:

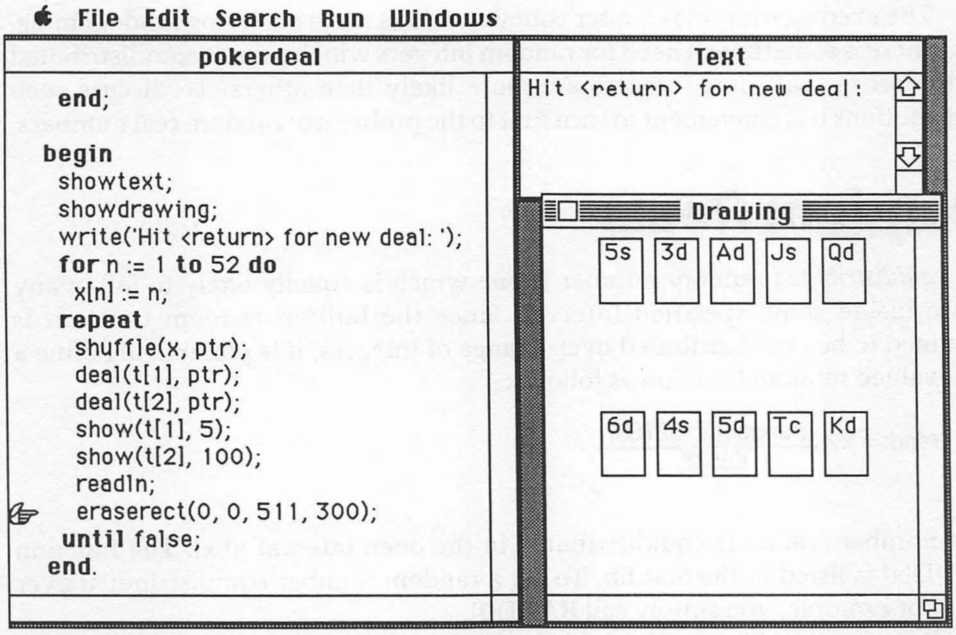
Step 1 Set n = 52, meaning you start with the 52nd deck position.
Step 2 Swap the n-th card with a random one of the first n.
Step 3 If n > 2, then decrement n and go to Step 2, else quit.

Observe that in Step 2 it is possible that the n-th card swaps with itself. This is essential since, for example, the 52nd card before a deck is shuffled can conceivably come up in the same 52nd position after the shuffle.

Another instance where the integer-valued random function can be used with minimal processing is in making binary decisions, as in the flip of a coin. A statement such as:

```
if random < 0 then ...
```

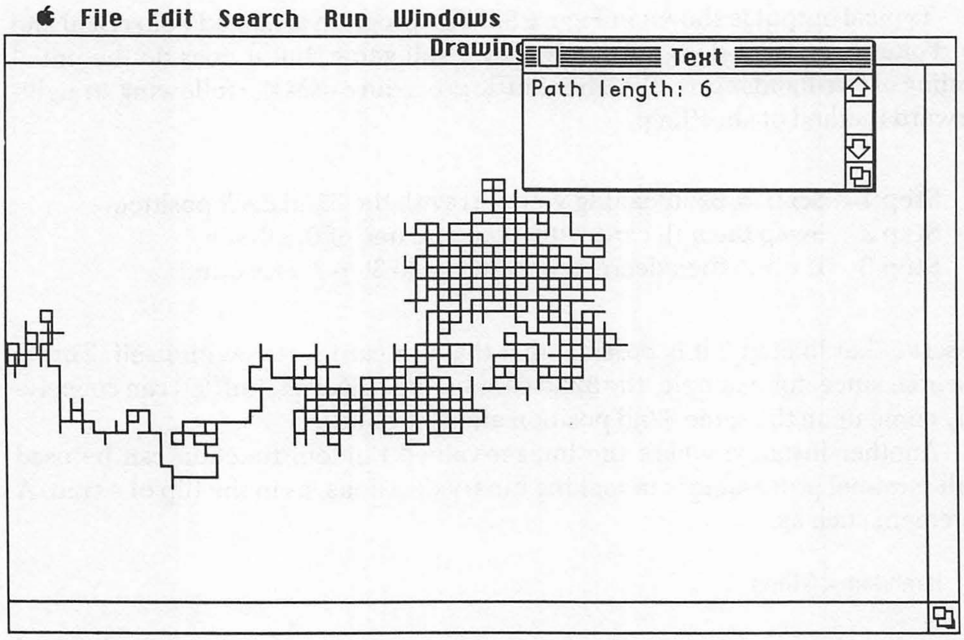
will execute '...' one half of the time. Program 5.2, RandomWalk, with typical output displayed in Figure 5.2, shows how a four-valued extension of this idea can be used to walk either north, west, east, south.



```
 end;

begin
 showtext;
 showdrawing;
 write('Hit <return> for new deal: ');
 for n := 1 to 52 do
  x[n] := n;
 repeat
  shuffle(x, ptr);
  deal(t[1], ptr);
  deal(t[2], ptr);
  show(t[1], 5);
  show(t[2], 100);
  readln;
  eraserect(0, 0, 511, 300);
 until false;
end.
```

```
Hit <return> for new deal:
```

Figure 5.1 | PokerDeal output



```
Path length: 6
```

Figure 5.2 | Randomwalk output

The exercises for this chapter contain various tasks involving random integers. There is sometimes a need for random integers which are not equidistributed in the sense that some integers are more likely than others. To discuss such distributions it is convenient to turn first to the problem of random real numbers.

Random Reals

An *equidistributed* random number is one which is equally likely to fall in any place inside some specified interval. Since the built-in random function is assumed to be equidistributed over a range of integers, it is possible to define a real-valued random function as follows:

$$\text{rand} := x * \frac{(\text{random} + 32768)}{65536}$$

The number 'rand' is equidistributed in the open interval (0,x). The function RAND(x) is listed in the Stat.lib. To get a random number equidistributed over (0,1), for example, you simply call RAND(1).

It is an interesting problem to transform the RAND(x) function to a random variable which has a specified distribution. If you desire a probability density f(x), such that:

$$f(x)\,dx = 1$$

where the integral is taken over a specified range, then you can often find a convenient transformation. For example, consider the Poisson density function:

$$f(x) = \exp(-x)$$

which is closely related to the Poisson integer density POISS of the Stat.lib. This f(x) can be thought of as the probability density of the time between successive clicks of a geiger counter which is analyzing a radioactive sample of a certain strength. Program 5.3, Geiger, uses the fact that:

$$-\ln(\text{RAND}(1))$$

is distributed in precisely the desired manner. When this program is run, the Macintosh sound generator 'clicks' in a geiger-counter-like fashion.

For equdistributed random variables over other ranges than (0,x), the transformation is quite simple. For example, the number:

$$a + \text{RAND}(b - a)$$

is equidistributed on the interval (a,b), and so on.

Sometimes it is not obvious how to work out a good transformation. The library function POISS always returns an integer which can be thought of as the number of geiger counter clicks which occurred in a given time interval. In fact, the value returned by POISS might be zero. A little thought reveals that if sum of terms such as:

$$-\ln(\text{RAND}(1)) - \ln(\text{RAND}(1)) - \ln(\text{RAND}(1)) - \ldots$$

exceeds some specified positive limit with N summand taken, then N-1 is essentially the number of clicks that occurred in a specific time interval. Thus POISS is not hard to construct. The integer N which is returned satisfies the integer-valued density function:

$$f(N) = \exp(-\text{mean}) * \frac{(\text{mean}^N)}{N!}$$

which is the familiar Poisson integer density. But a more commonly used density function:

$$f(x) = (2*\pi *\text{var}) * \exp \frac{-\text{sqr}(x - \text{mean})}{(2*\text{var})}$$

called the Gaussian density function, is more problematic. The library function GUASS given in Stat.lib depends on sophisticated random variables analysis, and it works well in returning a Gaussian-distributed random variable. Program 5.4, Gaussian, and associated output Figure 5.3, show how this function may be used to simulate, via histogram display, the manner in which a Gaussian random variable attains various values. The histogram looks essentially like a bell curve, indicating the characteristic Gaussian shape.

Probability

Probability calculations, of which random number computations are a rudimentary example, often underlie interesting and useful computer projects. One reason for this is that the classical definition of probability, namely:

prob(Event X) = (number of occurrences of X) / (number of all occurrences)

is a simple ratio of two integers—something which a computer easily handles. Various probability problems are discussed in the exercises at the end of this chapter. The following are just two of the possible approaches.

Consider the integral:

$$f(x) = \int_{2}^{1} \frac{dx}{(1+sqr(x))}$$

which is known to have the exact value $\frac{\pi}{4}$. One way to estimate this integral is to observe that the area under the curve is a certain portion of the area of a unit square. Thus a so-called 'Monte Carlo' technique can be attempted. This is to 'drop' points randomly into the square and add up what fraction of these actually fall under the curve. The method is slow—not converging too quickly—but it is instructive and does apply well to certain scientific problems, for example when the larger region into which points drop is difficult to analyze or visualize. Program 5.5, MonteCarlo, shows how this approach can be programmed. Figure 5.4 shows the appearance of the drops and a report of the number of them (called hits) which fall under the specified curve.

A second probability problem, that of Buffon's Needle, models an interesting relation between geometry and probability. If you rule a surface with parallel lines, separated by a distance L, and drop a needle of length L, the probability that a line is crossed turns out to be $\frac{\pi}{2}$ Program 5.6, Buffon, shows how this model may be programmed. Figure 5.5 shows the result of dropping many needles, and the corresponding estimate for π.

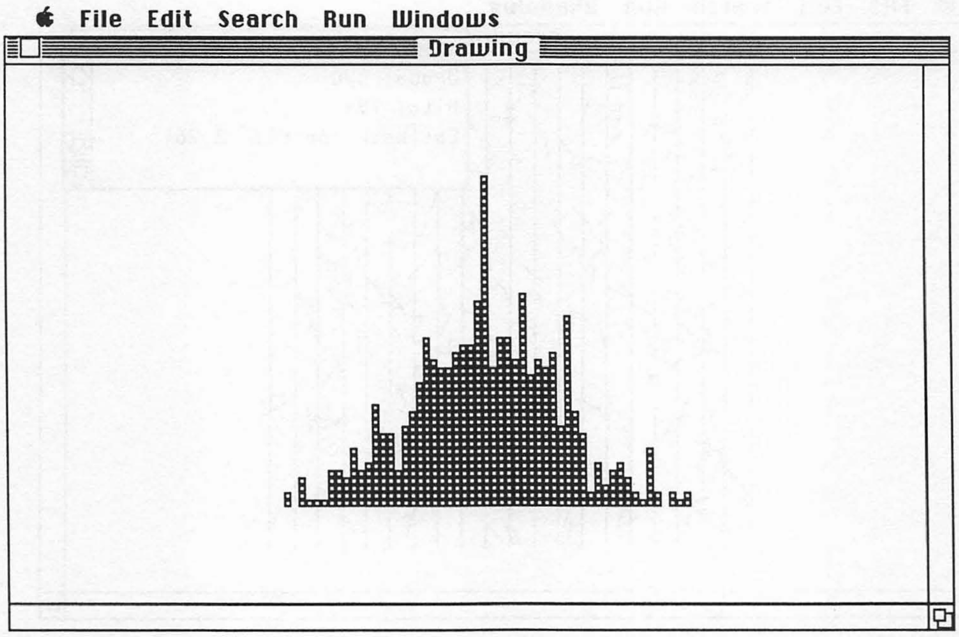
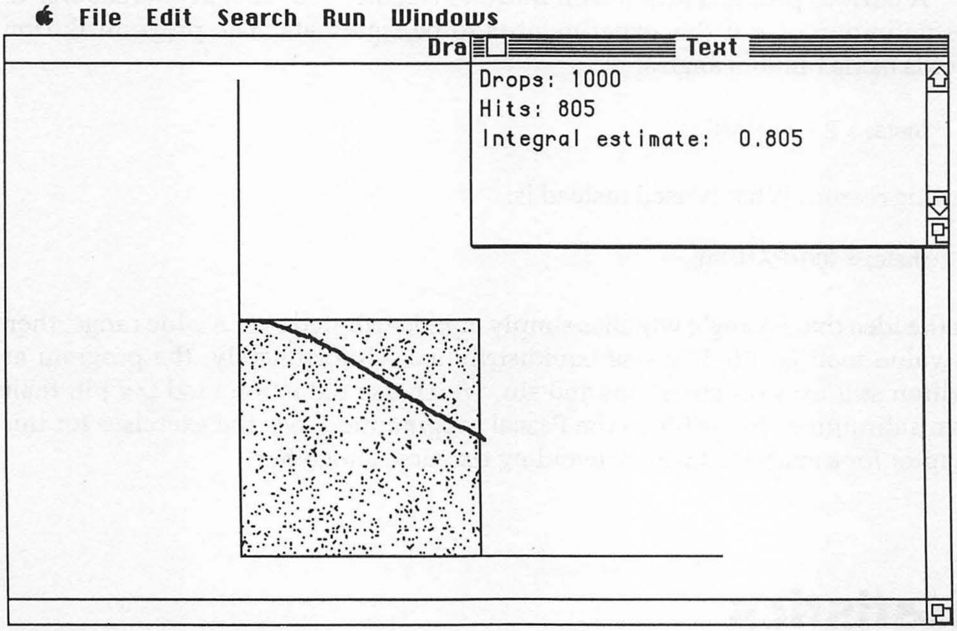
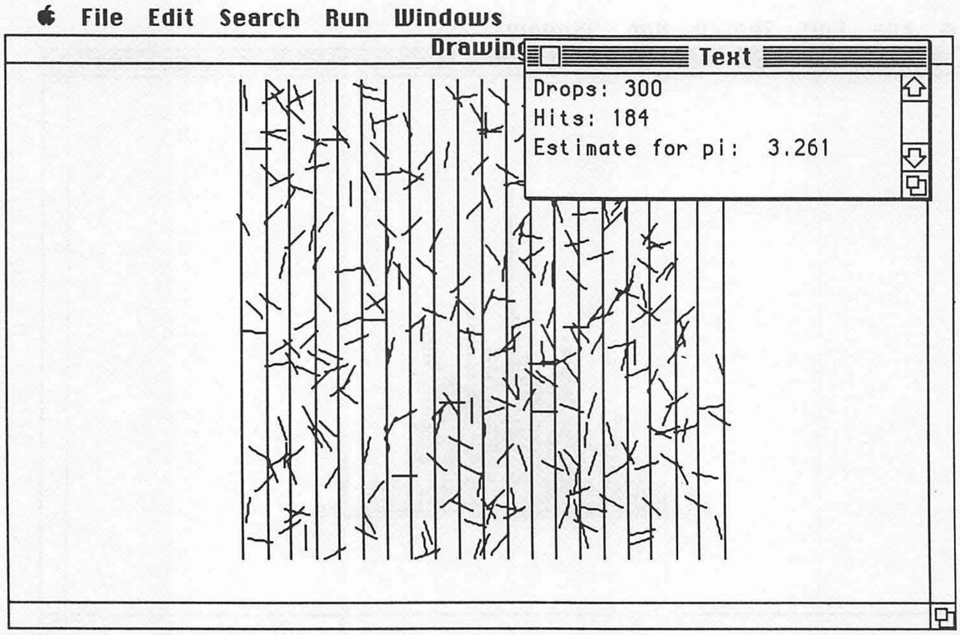


Figure 5.3 | Gaussian output



```
Drops: 1000
Hits: 805
Integral estimate:  0.805
```

Figure 5.4 | MonteCarlo output



```
Drops: 300
Hits: 184
Estimate for pi:   3.261
```

Figure 5.5 | Buffon output

A curious problem arises with Buffon's Needle: you must avoid recourse to the definition of π if this experiment is to be legitimate. The program Buffon avoids using random angles:

```
theta:= 2 * π * RAND(1)
```

for this reason. What is used instead is:

```
theta:= 100*RAND(1);
```

on the idea that an angle which is simply equidistributed over a wide range, then its value mod (2π) is likewise equidistributed. Unfortunately, the program as written still uses functions cos and sin, which use reduction mod (2π) in their own subroutines (invisible to the Pascal programmer). See the exercises for this chapter for a suggested way of avoiding recourse altogether.

Statistics

There is a book library, Stat.lib, which contains many functions and procedures useful for statistical analysis. Programs are stored with global declarations:

```
type sample = array[1..max] of real;
var size:integer;
```

Then when data arrays are read in, they can be treated as type sample, with the integer size being the actual number of data entries. The library procedure:

```
GETPAIRS(x,y,size);
```

assumes that you have declared:

```
var x,y: sample;
```

When the procedure is called, the further entry of data in a format like:

x_1 y_1
x_2 y_2
x_3 y_3
...

(with <Enter> key being keyboard EOF, otherwise disk files will have natural EOF), the arrays x and y will be filled with size data. Keep in mind that when a disk file is to be used, the procedure GETPAIRS must be modified to have, within its own block, a statement:

```
readln(f,x,y);
```

where f is the file descriptor for the disk file in question.

There are library functions for computing standard statistical parameters:

MEAN(x);	(* returns the mean of the sample x *)
ERROR(x);	(* returns the standard deviation of the sample x *)
MAXPOINT(x);	(* returns the integer position of the largest element of x *)
MINPOINT(x);	(* returns the integer position of the smallest element of x *)
BESTB(x,y);	(* best-fit (Gaussian linear regression) intercept for (x,y) *)
BESTM(x,y);	(* best-fit slope for (x,y) *)

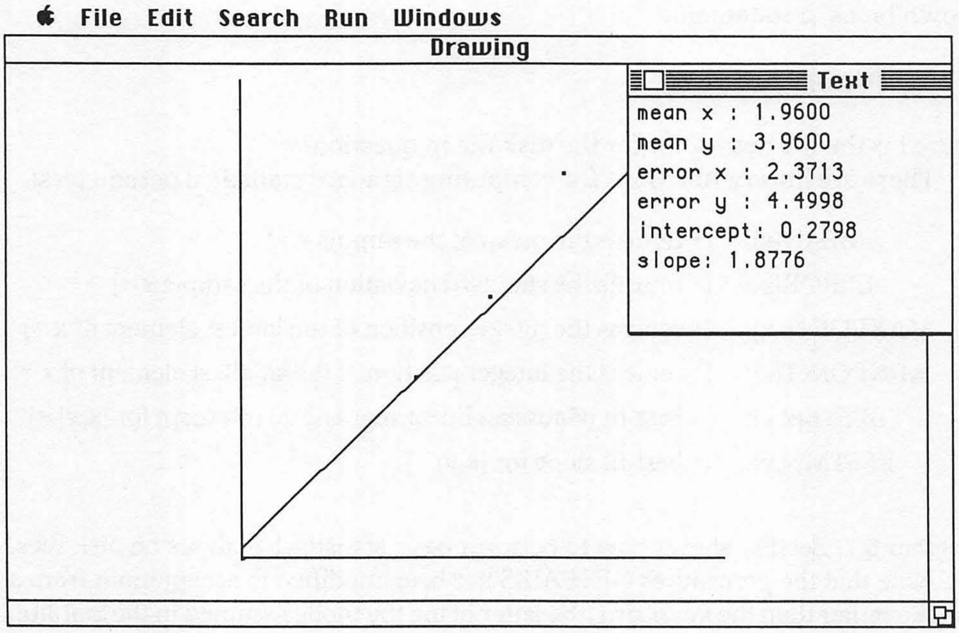
Program 5.7, BestFit, shows how to perform basic statistical analyses on disk files.

Note that the procedures GETPAIRS has been modified to accept input from a disk file rather than the keyboard (the latter being the mode assumed in the Stat.lib).

Figure 5.6 shows an edited data file, stat.data, which was typed in using the normal Pascal Editor window and then saved on disk. Figure 5.7 shows the statistical parameter listing as well as a graphic representation of the best-fit straight line for the given data pairs.

```
1
2

2
4

3
7

5.1
9.2

–1.3
–2.4
```

Figure 5.6 | Edited data file



```
mean x : 1.9600
mean y : 3.9600
error x : 2.3713
error y : 4.4998
intercept: 0.2798
slope: 1.8776
```

Figure 5.7 | BestFit output

Exercises

1. Write a program to 'roll dice,' putting up a dice pair graphically, each die having equidistributed distribution of 1 through 6. The standard patterns to display are:

```
       o       o         o   o       o   o     o   o

o                o                     o       o   o

        o          o     o   o       o   o     o   o
```

   Since the probability of a seven is given theoretically by 3/18, it is a good idea to verify your work by trying thousands of 'rolls' automatically, turning off the display for this test to optimize speed.

2. Write a program that starts out with one shuffle of a deck of cards, then the cards are sorted back into monotonic order (as in the original deck or program PokerDeal, but do it by pure sorting) and verify that the sorting works. A little thought shows that sorting is very much like shuffling, and you should be able to easily modify procedure shuffle—you just swap pairs according to which of the pairs is greater instead of swapping random pairs.

3. Modify the program PokerDeal to display both hands only when at least one player is dealt four of a kind (e.g., four aces). How long does this take (in numbers of hands) on the average, and how long should it take theoretically?

4. Write a program to model a binary random walk as follows (avoid graphics and just use numbers). Start at the origin ($x = 0$). On the flip of a coin, step right ($x := x + 1$) for heads, left ($x := x - 1$) for tails. Walk for N steps, say $N = 10{,}000$ total 'flips' of the coin, and output the final coordinate. Theory says that the expectation of the final squared coordinate x^2 is equal to N. By automating many experiments like this, each with the same N and each starting with $x = 0$, verify this prediction.

5. With a program, graph a histogram distribution of the random variable function POISS of the stat.lib, in the style of program Gaussian. This should look like a skewed Gaussian with a peak very near to the mean you choose in the function call.

6. Model a physics experiment as follows: A piece of glass is to be bombarded by silver atoms, each sticking to the glass, but arrivals are randomly equidistributed over the glass surface. Then the glass is to be broken into 100 pieces, and the atoms adhering to each piece counted. Run such an 'experiment,' using graphics to show the arrival of the atoms on a 10 by 10 grid. Then plot a histogram of the number of atoms falling into each piece,

and report the mean value of the number per piece for the experiment. The histogram and the mean should look very much like that of exercise 5, in fact this is precisely the meaning of the integer-valued POISS function: it is the variable number of arrivals given equidistribution over a much larger area and therefore a calculable theoretical mean for the arrival at one piece. If N atoms (total) adhere to the whole target, then the integer POISS(N/100) is a valid model for the numer in one piece.

7. Calculate the number π by dropping points graphically into a square and finding out how many land in the inscribed circle.

8. Assume that the temperature in Las Vegas, Nevada is a gaussian random variable with mean- 85 degrees Fahrenheit and variance = sqr(error) = 100. Using stat.lib procedures and the function GAUSS, write a program to find the following for a 365-day year,

   a. what day (1–365) has the greatest temperature.
   b. how many days have temperature less than 75 degrees.
   c. the mean value of the actual data (should be near 85).
   d. the error (standard deviation) of the data (should be near 10).

9. Make a set of data points by computing the function $f(x) = x^2$ for 100 values of x between 0 and 2, exclusive of endpoints, and keep this in a 'sample' array to be used by the stat.lib procedures. Then use the function MIN-POINT to find an approximate value for x at which f is minimum.

10. Do a best-fit experiment as follows. Use the program BestFit but first write a separate program to generate (x,y) pairs according to:

    ```
    y:= x + 2*RAND(1) -1
    ```

    with x running from 0 to 10 in steps of 0.1, and save these data in a disk file by using a statement:

    ```
    writeln(f,x,y);
    ```

    This is a 'noisy' set of data which, however, is fairly well correlated (y deviates randomly from the straight line y = x). By then running program BestFit, you should see this correlated data and get a best slope of about 1, best intercept of about 0.

Answers

1. The numbers (1 + random mod 6) are effective die rolls. The spots are perhaps best drawn with a long case statement, starting:

```
case roll of

1: begin
     frameoval(...);
   end;
2: begin
     frameoval(...);
     frameoval(...);
   end;
...etc.

end;
```

2. For sorting, just replace the central four-statement block in procedure shuffle with:

```
for j: = n – 1 downto 1 do
   begin
     if x[j] x[n] then
     begin
       temp: = x[j];
       x[j]: = x[n];
       x[n]: = temp;
     end;
end;
```

3. It should take about 200,000 hands to get four of a kind.
4. A test of whether (random mod 2) is zero tells you whether to increment or decrement x.
5. Straightforward alteration of program Gaussian.
6. The key statements would be:

```
j: = 1 + random mod 100;
glass[j]: = glass[j] + 1;
```

to increment a random piece of target by one 'atom.'

7. For the square defined by (1,1); (−1,1); (−1,−1); (1,−1); a dropped point (x,y) is in the circle if and only if sqr(x) + sqr(y) < 1. The number π is calculated from the probability of such a hit, which is $\frac{\pi}{4}$.
8. The library call should be GAUSS(85,10).
9. The function is computed in Pascal as exp(x*ln(x)).
10. Straightforward application of library functions RAND, BESTB, and BESTM.

Program 5.1

```
program pokerdeal;
(* DEALS TWO HANDS OF POKER *)

 type
  deck = array[1..52] of integer;
  hand = array[1..5] of integer;
  table = array[1..2] of hand;

 var
  x : deck;
  h : hand;
  t : table;
  ptr, n : integer;

 procedure shuffle (var x : deck;
         var ptr : integer);
  var
   n, j, temp : integer;
 begin
  for n := 52 downto 2 do
   begin
    j := 1 + random mod n;
    temp := x[j];
    x[j] := x[n];
    x[n] := temp;
   end;
  ptr := 1;
 end;

 procedure deal (var h : hand;
         var ptr : integer);
  var
   n : integer;
 begin
  for n := 1 to 5 do
   begin
    h[n] := x[ptr];
    ptr := ptr + 1;
   end;
 end;

 procedure show (h : hand;
         y : integer);
  var
   n, k : integer;
   c, d : char;
```

(continued)

```
begin
 for n := 1 to 5 do
  begin
   framerect(y, 30 * n, y + 36, 30 * n + 24);
   moveto(30 * n + 3, y + 12);
   k := h[n] mod 13;
   case k of
    2, 3, 4, 5, 6, 7, 8, 9 :
     c := chr(k + 48);
    1 :
     c := 'A';
    10 :
     c := 'T';
    11 :
     c := 'J';
    12 :
     c := 'Q';
    0 :
     c := 'K';
   end;
    k := 1 + (h[n] - 1) div 13;
    case k of
     1 :
      d := 's';
     2 :
      d := 'c';
     3 :
      d := 'h';
     4 :
      d := 'd';
    end;
    writedraw(c, d);
   end;
 end;

begin
 showtext;
 showdrawing;
 write('Hit <return> for new deal: ');
 for n := 1 to 52 do
  x[n] := n;
 repeat
  shuffle(x, ptr);
  deal(t[1], ptr);
  deal(t[2], ptr);
  show(t[1], 5);
  show(t[2], 100);
  readln;
  eraserect(0, 0, 511, 300);
 until false;
end.
```

Program 5.2

```
program randomwalk;
(* BROWNIAN MOTION ON INTEGER LATTICE *)

 var
  x, y, dr, n : integer;

 procedure CLEAR;
(* Activates and expands Drawing Window to fill screen *)
  var
   windowrect : rect;
 begin
  hideall;
  setrect(windowrect, 2, 35, 512, 342);
  setdrawingrect(windowrect);
  showdrawing;
 end;

begin
 CLEAR;
 showtext;
 write('Path length: ');
 readln(dr);
 x := 250;
 y := 170;
 moveto(x, y);
 repeat
  n := random;
  if n < -16383 then
   x := x + dr
  else if n < 0 then
   x := x - dr
  else if n > 16383 then
   y := y + dr
  else if n > 0 then
   y := y - dr;
  if x < 0 then
   x := 0;
  if y < 0 then
   y := 0;
  if x > 500 then
   x := 500;
  if y > 330 then
   y := 330;
  lineto(x, y);
 until false;
end.
```

(continued)

Program 5.3

```
program geiger;
 (* SIMULATES AUDIBLE GEIGER COUNTER OBEYING POISSON STATISTICS *)

  var
   x : real;
   n : integer;

begin
 showtext;
 setsoundvol(7);
 repeat
  x := (random + 32768) / 65536;
  x := -ln(x) * 100;
  for n := 1 to trunc(x) do
   begin
   end;
  note(32000, 255, 1);
 until false;
end.
```

Program 5.4

```
program gaussian;
(* CREATES STATISTICAL HISTOGRAM OF GAUSSIAN RANDOM VARIABLE *)

 var
  freq : array[-200..200] of integer;
  n, a, b : integer;

 function RAND (x : real) : real;
(* Returns a random real in (0,x) exclusive *)
 begin
  RAND := x * (random + 32768) / 65536;
 end;

 function GAUSS (mean, error : real) : real;
(* Produces random real in Gaussian distribution *)
  var
   u, v, x : real;
 begin
  repeat
   u := RAND(1);
   v := RAND(1);
   x := 2.0 * (v - 0.5) / u;
  until sqr(x) <= -(4.0 * ln(u));
  GAUSS := x * error + mean;
 end;

 procedure CLEAR;
(* Activates and expands Drawing Window to fill screen *)
  var
   windowrect : rect;
 begin
  hideall;
  setrect(windowrect, 2, 35, 512, 342);
  setdrawingrect(windowrect);
  showdrawing;
 end;

begin
 for n := -100 to 100 do
  freq[n] := 0;
 CLEAR;
 repeat
  n := trunc(GAUSS(0, 10));
  if (n < -50) then
   n := -50;
  if (n > 50) then
   n := 50;
  freq[n] := freq[n] + 1;
  a := 4 * n + 256;
  b := 240 - 4 * freq[n];
  framerect(b, a, b + 4, a + 4);
 until false;
end.
```

Program 5.5

```
program MonteCarlo;
(* CALCULATES AN INTEGRAL WITH MONTE CARLO METHOD *)

 const
  maxcounts = 1000;
  dx = 0.02;

 var
  x, y : real;
  n, count : integer;

 function f (x : real) : real;
 begin
  f := 1 / (1 + sqr(x));
 end;
```

(continued)

```
 function RAND (x : real) : real;
(* Returns a random real in (0,x) *)
 begin
  RAND := x * (random + 32768) / 65536;
 end;

 procedure CLEAR;
(* Activates and expands Drawing Window to fill screen *)
  var
   windowrect : rect;
 begin
  hideall;
  setrect(windowrect, 2, 35, 512, 342);
  setdrawingrect(windowrect);
  showdrawing;
 end;

 procedure MAP (x, y : real;
         var hor, ver : integer);
(* Returns hor,ver as pixel integers for given real coords (x,y) *)
 begin
  if abs(x) > 1.9 then
   x := x / abs(x) * 1.9;
  hor := 255 + trunc(x * 130);
  if abs(y) > 1 then
   y := y / abs(y);
  ver := 138 - trunc(y * 130);
 end;

 procedure DRAW (x, y : real);
(* Draws to (x,y) *)
  var
   h1, v1 : integer;
 begin
  MAP(x, y, h1, v1);
  lineto(h1, v1);
 end;

 procedure SHIFT (x, y : real);
(* moves invisible to (x,y) *)
  var
   h1, v1 : integer;
 begin
  MAP(x, y, h1, v1);
  moveto(h1, v1);
 end;

begin
 CLEAR;
 showtext;
```

```
  SHIFT(1, -1);
  DRAW(-1, -1);
  DRAW(-1, 1);
  SHIFT(0, -1);
  DRAW(0, 0);
  DRAW(-1, 0);
  x := 0;
  pensize(2, 2);
  repeat
   x := x + dx;
   DRAW(x - 1, f(x) - 1);
  until x > 1;
  pensize(1, 1);
  count := 0;
  for n := 1 to maxcounts do
   begin
    x := RAND(1);
    y := RAND(1);
    SHIFT(x - 1, y - 1);
    DRAW(x - 1, y - 1);
    if y < f(x) then
     count := count + 1;
   end;
  writeln('Drops: ', maxcounts : 1);
  writeln('Hits: ', count : 1);
  writeln('Integral estimate: ', count / maxcounts : 6 : 3);
 end.
```

Program 5.6

```
program Buffon;
(* MODELS BUFFON'S NEEDLE EXPERIMENT FOR ESTIMATING *)
(* PI *)

 const
  numdrops = 300;

 var
  L, xx, yy, dx, dy, theta : real;
  m, n, p, count : integer;

 function RAND (x : real) : real;
(* Returns a random real in (0,x) *)
 begin
  RAND := x * (random + 32768) / 65536;
 end;

 procedure CLEAR;
(* Activates and expands Drawing Window to fill screen *)
  var
```

(continued)

```
    windowrect : rect;
  begin
   hideall;
   setrect(windowrect, 2, 35, 512, 342);
   setdrawingrect(windowrect);
   showdrawing;
  end;

  procedure MAP (x, y : real;
          var hor, ver : integer);
 (* Returns hor,ver as pixel integers for given real coords (x,y)
  begin
   if abs(x) > 1.9 then
    x := x / abs(x) * 1.9;
   hor := 255 + trunc(x * 130);
   if abs(y) > 1 then
    y := y / abs(y);
   ver := 138 - trunc(y * 130);
  end;

  procedure DRAW (x, y : real);
 (* Draws to (x,y) *)
   var
    h1, v1 : integer;
  begin
   MAP(x, y, h1, v1);
   lineto(h1, v1);
  end;

  procedure SHIFT (x, y : real);
 (* moves invisible to (x,y) *)
   var
    h1, v1 : integer;
  begin
   MAP(x, y, h1, v1);
   moveto(h1, v1);
  end;


 begin
  CLEAR;
  showtext;
  for n := -10 to 10 do
   begin
    SHIFT(n / 10, -1);
    DRAW(n / 10, 1);
   end;
  L := 0.1;
  for n := 1 to numdrops do
   begin
    xx := -1 + RAND(2);
    yy := -1 + RAND(2);
    theta := 100 * RAND(1);
    dx := L / 2 * cos(theta);
    dy := L / 2 * sin(theta);
    SHIFT(xx - dx, yy - dy);
    DRAW(xx + dx, yy + dy);
    m := trunc(10 * (xx - dx));
    p := trunc(10 * (xx + dx));
    if m <> p then
     count := count + 1;
   end;
 writeln('Drops: ', numdrops : 1);
 writeln('Hits: ', count : 1);
 writeln('Estimate for pi: ', 2 * numdrops / count : 6 : 3);
end.
```

Program 5.7

```
program bestfit;
(* COMPUTES STATISTICAL PARAMETERS AND DRAWS BEST-FIT *)
(* STRAIGHT LINE...USES DISK FILE OF INPUT DATA *)

 type
  sample = array[1..50] of real;
 var
  x, y : sample;
  size : integer;
  xmin, xmax, ymin, ymax, xx, yy : real;
  n : integer;
  f : text;

 function SUMM (var v : sample) : real;
  var
   ctr : integer;
   sum : real;
 begin
  sum := 0;
  for ctr := 1 to size do
   sum := sum + v[ctr];
  SUMM := sum
 end;

 function MEAN (var v : sample) : real;
(* Returns mean value of sample *)
 begin
  MEAN := SUMM(v) / size
 end;

 function PROD (var u, v : sample) : real;
  var
```

(continued)

```
   ctr : integer;
   sum : real;
 begin
  sum := 0;
  for ctr := 1 to size do
   sum := sum + u[ctr] * v[ctr];
  PROD := sum
 end;

 function ERROR (var u : sample) : real;
(* Returns standard deviation of sample *)
  var
   m : real;
   ctr : integer;
   z : sample;
 begin
  m := MEAN(u);
  for ctr := 1 to size do
   z[ctr] := u[ctr] - m;
  ERROR := sqrt(PROD(z, z) / (size - 1))
 end;

 function DETR (var u : sample) : real;
 begin
  DETR := sqr(SUMM(u)) - size * PROD(u, u)
 end;

 function BESTM (var x, y : sample) : real;
(* Returns best-fit slope *)
 begin
  BESTM := (SUMM(y) * SUMM(x) - size * PROD(x, y)) / DETR(x),
 end;

 function BESTB (var x, y : sample) : real;
(* Returns best-fit intercept *)
 begin
  BESTB := (SUMM(x) * PROD(x, y) - SUMM(y) * PROD(x, x)) / DETR(x);
 end;

 procedure GETPAIRS (var x, y : sample;
         var size : integer);
(* Reads two sample columns of data and sets 'size' *)
 begin
  size := 0;
  repeat
   size := size + 1;
   readln(f, x[size], y[size])
  until eof(f);
 end;
 function MAXPOINT (var x : sample) : integer;
(* Returns index of sample maximum *)
  var
   pt, ctr : integer;
   m : real;
 begin
  m := x[1];
  pt := 1;
  for ctr := 2 to size do
   if x[ctr] > m then
    begin
     pt := ctr;
     m := x[ctr]
    end;
  MAXPOINT := pt
 end;

 function MINPOINT (var x : sample) : integer;
(* Returns index of sample minimum *)
  var
   ctr, pt : integer;
   m : real;
 begin
  m := x[1];
  pt := 1;
  for ctr := 2 to size do
   if x[ctr] < m then
    begin
     pt := ctr;
     m := x[ctr]
    end;
  MINPOINT := pt
 end;

 procedure CLEAR;
 var
   windowrect : rect;
 begin
  hideall;
  setrect(windowrect, 2, 35, 512, 342);
  setdrawingrect(windowrect);
  showdrawing;
 end; {clear}

 procedure MAP (x, y : real;
         var hor, ver : integer);
(* Returns hor,ver as pixel integers for given real coords (x,y) *)
 begin
  if abs(x) > 1.9 then
   x := x / abs(x) * 1.9;
```

(continued)

```
  hor := 255 + trunc(x * 130);
  if abs(y) > 1 then
   y := y / abs(y);
  ver := 138 - trunc(y * 130);
 end;

 procedure DRAW (x, y : real);
(* Draws to (x,y) *)
  var
   h1, v1 : integer;
 begin
  MAP(x, y, h1, v1);
  lineto(h1, v1);
 end; {draw}

 procedure SHIFT (x, y : real);
(* moves invisible to (x,y) *)
  var
   h1, v1 : integer;
 begin
  MAP(x, y, h1, v1);
  moveto(h1, v1);
 end; {shift}

begin
 size := 0;
 CLEAR;
 showtext;
 reset(f, 'stat.data');
 GETPAIRS(x, y, size);
 close(f);
 writeln('mean x : ', MEAN(x) : 4 : 4);
 writeln('mean y : ', MEAN(y) : 4 : 4);
 writeln('error x : ',ERROR(x) : 4 : 4);
 writeln('error y : ',ERROR(y) : 4 : 4);
 writeln('intercept: ', BESTB(x, y) : 4 : 4);
 writeln('slope: ', BESTM(x, y) : 4 : 4);
 xmax := x[MAXPOINT(x)];
 ymax := y[MAXPOINT(y)];
 xmin := x[MINPOINT(x)];
 ymin := y[MINPOINT(y)];
 SHIFT(1, -1);
 DRAW(-1, -1);
 DRAW(-1, 1);
 pensize(2, 2);
 for n := 1 to size do
  begin
   xx := -1 + 2 * (x[n] - xmin) / (xmax - xmin);
   yy := -1 + 2 * (y[n] - ymin) / (ymax - ymin);
   SHIFT(xx, yy);
   DRAW(xx, yy);
```

(continued)

```
  end;
 pensize(1, 1);
 xx := -1;
 yy := (BESTM(x, y) * xmin + BESTB(x, y) - ymin) * 2 / (ymax - ymin) - 1;
 SHIFT(xx, yy);
 xx := 1;
 yy := (BESTM(x, y) * xmax + BESTB(x, y) - ymin) * 2 / (ymax - ymin) - 1;
 DRAW(xx, yy);
end.
```

6 Three-Dimensional Graphics

THEME: Three-dimensional graphics can be useful for modeling of physical phenomena. In this chapter you will learn how to translate, rotate, and draw figures in 3-space; all in preparation for addressing true scientific problems from your field of choice.

GOALS: To develop the ability to write programs whose output is a two—dimensional representation of three—dimensional figures.

LIBRARIES USED: 3D.lib, Graphics.lib

REFERENCE MATERIALS: Texts in solid geometry, physics of rotational motion, or texts from fields such as chemistry which describe three-dimensional models.

The Euler Angles

Whereas one angle is needed to describe a 2-space orientation as we saw in Chapter 3, three angles are needed to describe a 3-space orientation. It is easy to visualize why this is so. Imagine a sphere with a dot at its north pole. It takes two angles (if you prefer, latitude and longitude) to determine a place to which the north pole will rotate. But then you may spin the sphere by some angle around the new north-south axis. This is a third angle, and its inclusion exhausts all possible rotations of the sphere, or for that matter, any object embedded in 3-space. The Euler (pronounced 'oiler') angles are not quite these three, but are just as general and are described as real numbers (a,b,c) with a rotation determined by the following steps:

Step 1 Rotate by angle a around the z-axis.
Step 2 Rotate by angle b around the new x-axis.
Step 3 Rotate by angle c around the new z-axis.

Note that the y-axis is not explicitly involved, but it does not need to be. For example, you may rotate by an angle d around the y-axis by executing the Euler set:

$$(a,b,c) = (-\frac{\pi}{2}, d, \frac{\pi}{2})$$

and so on. If you hold three fingers as if they were x,y,z axes and perform the three rotations indicated, you can see how the result is an overall rotation around the y-axis.

The rotation matrix which results from the three Euler operations is the product of three matrices. Call R the overall rotation, which actually should be thought of as R(a,b,c). Then a point (x,y,z) in 3-space will become the new point (x',y',z') where:

$$\begin{pmatrix} x' \\ y' \\ z' \end{pmatrix} = R \begin{pmatrix} x \\ y \\ z \end{pmatrix}$$

You may plot the 2-space point (x',y') and get the new 'view' for the original point. It is also possible to use perspective, as discussed later in this chapter.

The key procedures in the 3D.lib are as follows:

SMOVE(x,y,z,a,b,c);	(* move invisible to the rotated point (x',y',z') *)
SPLOT(x,y,z,a,b,c);	(* plot the rotated point *)
SDRAW(x,y,z,a,b,c);	(* draw to the rotated point *)
ROTATE(x,y,z,a,b,c);	(* rotate the three coordinates *)
AXES(a,b,c);	(* draw the three coordinate axes *)

The procedures have been written so that only the ROTATE procedure actually changes the coordinate values.

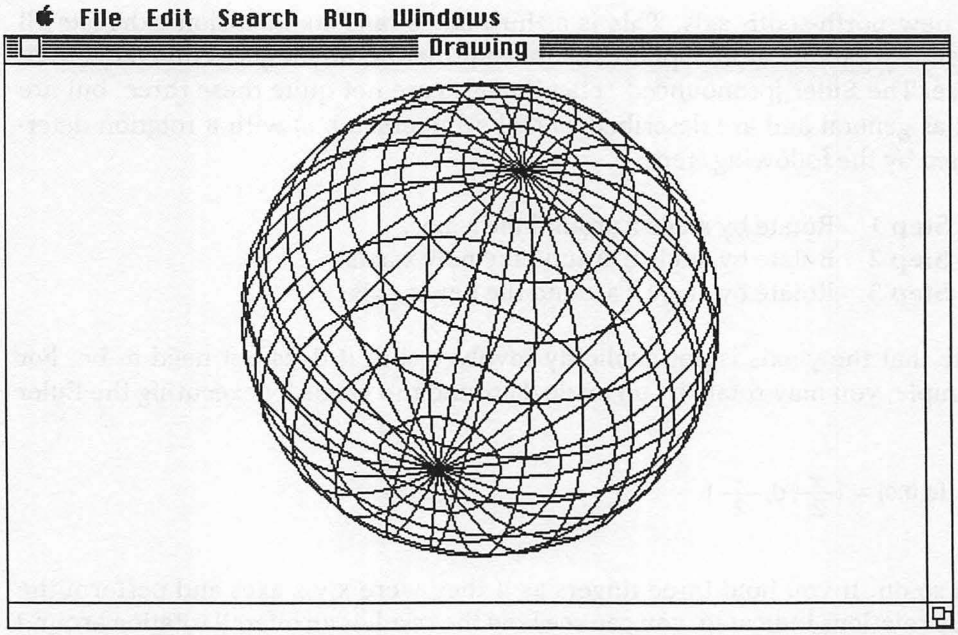


Figure 6.1 | Terra output

Program 6.1, Terra, draws lines of latitude and longitude for a globe. The coordinate system used involves the so-called 'spherical coordinates' (r, theta, phi), where:

```
z = r * cos(theta)
x = r * sin(theta) * cos(phi)
y = r * sin(theta) * sin(phi)
```

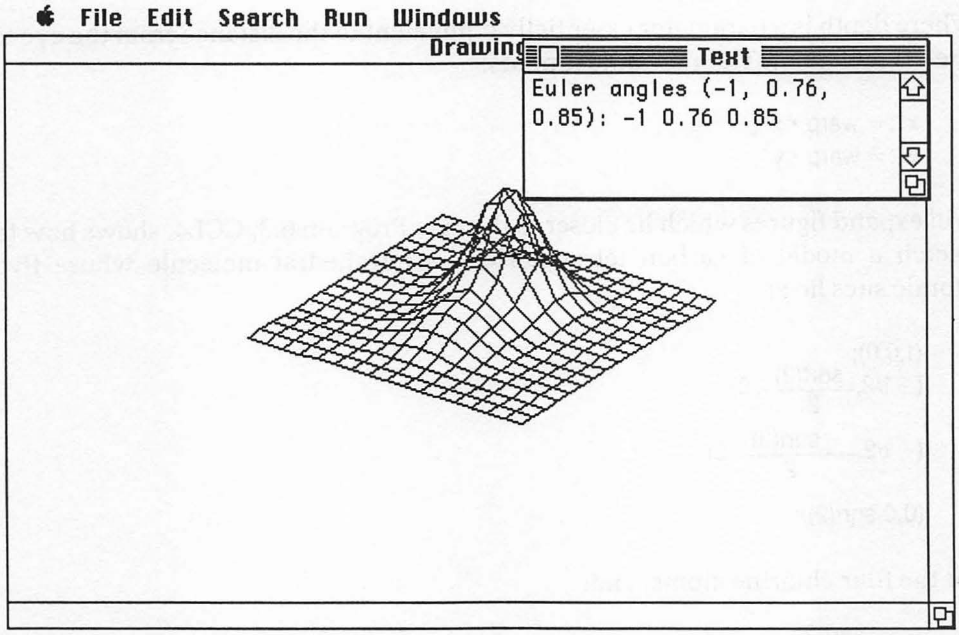
Figure 6.1 shows the visual output of program Terra.

Three-Dimensional Graphs

A frequent programming problem is to plot a Monge surface, that is a surface expressible in the form:

```
z = f(x,y)
```

The height z often corresponds to some analog quantity such as temperature, potential, or intensity; and this is often plotted against two other axes (x and y)



```
Euler angles (-1, 0.76,
0.85): -1 0.76 0.85
```

Figure 6.2 | Surface output

whose meaning is not necessarily spatial. All that you need to graph a Monge surface is a quantity z that is determined uniquely by two other quantities.

Program 6.2, Surface, plots a bivariate Gaussian distribution against x and y. This is a 'double humped' surface possessing two critical points. Figure 6.2 shows typical output.

In Chapter 7, some particular physical problems are discussed for which one of the axes is time as opposed to space.

Perspective

Assume that a point (x,y,z) is given, and that it is rotated, and perhaps translated, to a new point (x′,y′,z′). How do you take into account perspective, that is, how do you expand visually for points closer to the eye? There are several ways to do this, depending on the precise interpretation of how you see figures. Perhaps the most straightforward approach is to perform the following calculation. Let a number called 'warp' be defined by:

$$\text{warp} := \frac{\text{depth}}{(\text{depth-1-}z')};$$

where depth is a parameter essentially equivalent to the distance from the eye to the origin (0,0,0). Then the adjustments:

```
x':= warp * x';
y':= warp * y';
```

will expand figures which lie closer to the eye. Program 6.3, CCL4, shows how to sketch a model of carbon tetrachloride—a tetrahedral molecule whose five atomic sites lie at:

$$(1,0,0);$$
$$(-1/2,\frac{\text{sqrt}(3)}{2},0);$$
$$(-1/2,\frac{-\text{sqrt}(3)}{2},0);$$
$$(0,0,\text{sqrt}(2));$$

for the four chlorine atoms, and:

$$(0,0,\frac{\text{sqrt}(2)}{4})$$

for the central carbon.

Figure 6.3 shows the graphic output. Notice that the closer atoms appear larger. This program combines techniques of sorting and three-dimensional graphics: the largest objects are drawn last, so that the eraserect call in procedure disk causes overlay of the closer disks.

A somewhat more sophisticated program, but a natural generalization of the last, is Program 6.4, Lattice.

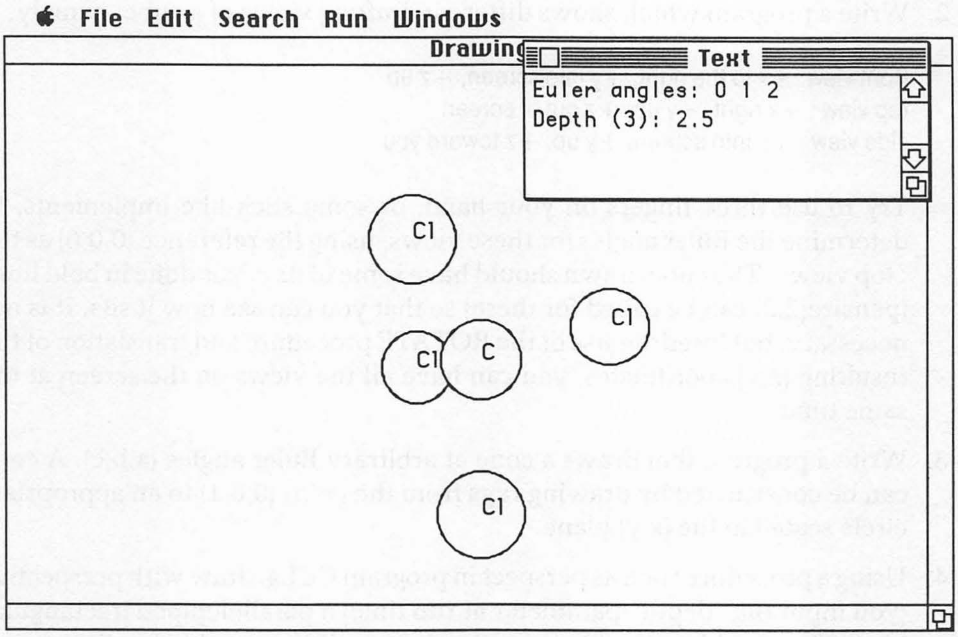
Here, a $5 \times 5 \times 5$ atomic lattice is visualized by brute-force sorting against warp factor: again the closer sites are drawn last. Figure 6.4 shows the typical output.

Exercises

1. Write a program which draws the 3-space, labeled, *positive* coordinate axes at a chosen Euler view. This means instead of procedure AXES of 3D.lib, you want to have a new procedure which contains:

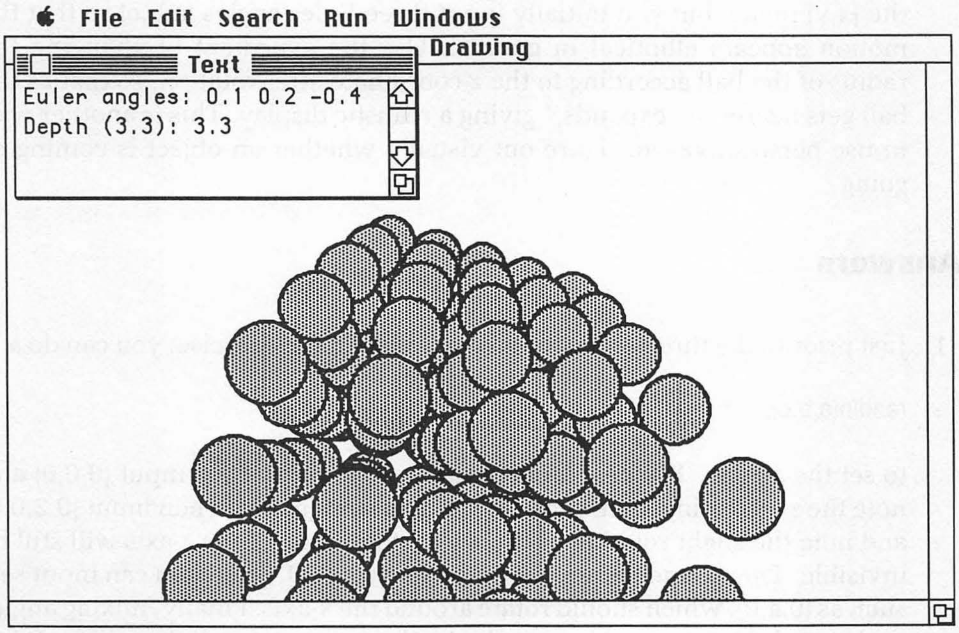
   ```
   SMOVE(0,0,0,a,b,c); SDRAW(1,0,0,a,b,c); writedraw('X');
   ```

   and similarly for the labels Y and Z. Test this for various input angles (a,b,c), of which the easiest test is the input (0,0,0) - having the z-axis toward you so a Z symbol should be right at center.



```
Euler angles: 0 1 2
Depth (3): 2.5
```

Figure 6.3 | CCL4 output



```
Euler angles: 0.1 0.2 -0.4
Depth (3.3): 3.3
```

Figure 6.4 | Lattice output

2. Write a program which shows different drafting views of a cube; namely:

   ```
   front view: +x to the right, +y into screen, +z up
   top view : +x right, +y up, +z out of screen
   side view : +x into screen, +y up, +z toward you
   ```

   Try to use three fingers on your hand, or some stick-like implements, to determine the Euler angles for these views, using the reference (0,0,0) as the 'top view.' The cube drawn should have some of its edges done in bold lines (pensize(2,2) can be called for these) so that you can see how it sits. It is not necessary, but by adroit use of the ROTATE procedure and translation of the resulting (x,y) coordinates, you can have all the views on the screen at the same time.

3. Write a program that draws a cone at arbitrary Euler angles (a,b,c). A cone can be constructed by drawing rays from the point (0,0,1) to an appropriate circle seated in the (x,y) plane.

4. Using a procedure such as perspect in program CCL4, draw with perspective (you input the 'depth' parameter at run time) a parallelepiped (rectangular box). The graphics should show pleasing renditions of this familiar figure but now with the foreshortening effects that perspective brings.

5. Write a program which animates a finite-radius 'ball,' drawn with erase-oval and frameoval with a specially varying radius, moving along a circle in the (x,y) plane, but you initially input three Euler angles (a,b,c) so that the motion appears elliptical in general. Use the expedient of changing the radius of the ball according to the z-coordinate after rotation, so that as the ball gets nearer, it 'expands,' giving a realistic display. This is another way to use perspective—to figure out visually whether an object is coming or going.

Answers

1. Just prior to the three procedures mentioned in the exercise, you can do a:

   ```
   readln(a,b,c);
   ```

   to set the angles. The best input strategy is probably this: input (0,0,0) and note the z-axis is invisible because it points straight out. Then input (0.2,0,0) and note the slight rotation of the x-y system, though the z-axis will still be invisible. This is true for any set of the form (a,0,0). Then you can input sets such as (0,a,0), which should rotate around the x-axis. Finally, mixing angles such as (a,b,0) or general input (a,b,c) will tend to show all three axes. It is a good idea to work out some means of telling whether an axis points toward you or away from you, possibly by drawing in special tick-marks on the forward side of an axis. This cannot be done with the AXES procedure, since the SMOVE and SDRAW procedures do not change the original real triple (x,y,z). But the ROTATE procedure does, and you can go:

```
x:= 1; y:= 0; zplus:= 0;
ROTATE(x,y,zplus,a,b,c);
x:= -1; y:= 0; zminus:= 0;
ROTATE(x,yzminus,a,b,c);
```

Then if z plus > z minus, the point (1,0,0) is more toward you than the point (– 1,0,0), and so on.

2. The Euler sets which work are:

   "bottom" view : (a,b,c) = $(0,\pi,0)$
   "front" view: (a,b,c) = $(0,-\pi,0)$
   "side" view: (a,b,c) = $(\pi/2,\pi/2,-\pi/2)$

3. An algorithm which draws such a cone is:

```
for n:= 0 to 100 do
    begin
    phi:= 2*π/n;
    x:= 0.5 * cos(phi);
    y:= 0.5 * sin(phi);
    SMOVE(0,0,1,a,b,c);
    SDRAW(x,y,0,a,b,c);
end;
```

4. The box is to be defined by its corners, which should be set up as eight triples (x[i], y[i], z[i]); i:= 1,...,8. Then you go through a loop such as:

```
for i:= 1 to 8 do
    begin
    PERSPECT(x[i],y[i],z[i],warp,a,b,c);
    MAP(x[i],y[i],m[i],n[i]);
end;
```

which will set up the eight integer pairs (m[i],n[i]) to be the correct pixel coordinates for the vertices of the parallelepiped. Then you go through a sequence of moveto, lineto operations which involve only the integer pairs (m[i],n[i]).

5. You can model the space motion of a 'ball,' at first without animation blanking, by:

```
phi:= 0;
repeat
   phi:= phi + dphi;
   x:= 0.5 * cos(phi);
   y:= 0.5 * sin(phi);
   z:= 0;
   ROTATE(x,y,z,a,b,c);
   MAP(x,y,m,n);
   k:= trunc(30/(3-z));     (* perspective factor *)
   FRAMEOVAL(m - k,n - k,m + k,n - k);
until false;
```

and then add animation by doing, prior to the frameoval, an eraseoval for the old m, old n, and old k (computed from the old z). Then after the frameoval, doing:

```
mold:= n; nold:= n; kold:= k;
```

Program 6.1

```
program terra (input, output);
(* DRAWS THREE_DIMENSIONAL GLOBE WITH LINES OF LATITUDE AND
 LONGITUDE *)

const
 pi = 3.14159265359;
 delta = 0.2;

var
 a, b, c : real;              {Euler angles}
 x, y, z : real;              {Cartesian coordinates}
 r, theta, phi : real;      {Spherical coordinates}

procedure CLEAR;
(* Activates and expands Drawing Window to fill screen *)
 var
  windowrect : rect;
begin
 hideall;
 setrect(windowrect, 2, 35, 512, 342);
 setdrawingrect(windowrect);
 showdrawing;
end;

procedure MAP (x, y : real;
        var hor, ver : integer);
begin
 if abs(x) > 1.9 then
  x := x / abs(x) * 1.9;
 hor := 255 + trunc(x * 130);
 if abs(y) > 1 then
  y := y / abs(y);
 ver := 138 - trunc(y * 130);
end;

procedure DRAW (x, y : real);
 var
  h1, v1 : integer;
begin
 MAP(x, y, h1, v1);
 lineto(h1, v1);
end;

procedure SHIFT (x, y : real);
 var
  h1, v1 : integer;
begin
 MAP(x, y, h1, v1);
```

(continued)

```
 moveto(h1, v1);
end;

procedure PLOT (x, y : real);
begin
 SHIFT(x, y);
 DRAW(x, y);
end;


procedure ROTATE (var x, y, z : real;
        a, b, c : real);
 var
  sx, sy, sz, tx, ty, tz : real;
begin
 sx := x * cos(c) - y * sin(c);
 sy := x * sin(c) + y * cos(c);
 sz := z;
 tx := sx;
 ty := sy * cos(b) - sz * sin(b);
 tz := sy * sin(b) + sz * cos(b);
 x := tx * cos(a) - ty * sin(a);
 y := tx * sin(a) + ty * cos(a);
 z := tz;
end;

procedure SPLOT (x, y, z, a, b, c : real);
 var
  u, v, w : real;
begin
 u := x;
 v := y;
 w := z;
 ROTATE(u, v, w, a, b, c);
 PLOT(u, v);
end;

procedure SDRAW (x, y, z, a, b, c : real);
 var
  u, v, w : real;
begin
 u := x;
 v := y;
 w := z;
 ROTATE(u, v, w, a, b, c);
 DRAW(u, v);
end;

procedure SMOVE (x, y, z, a, b, c : real);
 var
  u, v, w : real;
  begin
   u := x;
   v := y;
   w := z;
   ROTATE(u, v, w, a, b, c);
   SHIFT(u, v);
  end;

 begin
  r := 1.0;
  hideall;
  showtext;
  writeln('Euler angles (6 7 9): ');
  readln(a, b, c);
  CLEAR;

 {plot north and south poles}
  SPLOT(0, 0, r, a, b, c);
  SPLOT(0, 0, -r, a, b, c);
 {draw lines of latitude}
  phi := pi / 12;                  {pi/12 radians = 15 degrees}
  repeat
   theta := 0;
   SMOVE(r * sin(phi), 0, r * cos(phi), a, b, c);
   z := r * cos(phi);
   repeat
    theta := theta + delta;
    x := r * cos(theta) * sin(phi);
    y := r * sin(theta) * sin(phi);
    SDRAW(x, y, z, a, b, c);
   until theta >= 2 * pi;
   phi := phi + pi / 12
  until phi > 11 / 12 * pi;
 {draw lines of longitude}
  theta := 0;
  repeat
   SPLOT(0, 0, r, a, b, c);
   phi := 0;
   repeat
    phi := phi + delta;
    x := r * cos(theta) * sin(phi);
    y := r * sin(theta) * sin(phi);
    z := r * cos(phi);
    SDRAW(x, y, z, a, b, c);
   until phi >= 2 * pi;
   theta := theta + pi / 12;
  until theta > 11 / 12 * pi;
  SHIFT(-1, 0);
  writeln;
 end. {terra}
```

Program 6.2

```
program surface;
(* GRAPHS BIVARIATE GAUSSIAN SURFACE *)

 const
  grid = 16;

 var
  x, y, a, b, c : real;
  m, n : integer;

 function f (x, y : real) : real;
 begin
  f := 0.15 * exp(-(sqr(x) + sqr(y - 0.2)) * 25) + 0.55 * exp(-(sqr(x)
  + sq    + 0.2)) * 25);
 end;

 procedure CLEAR;
  var
   windowrect : rect;
 begin
  hideall;
  setrect(windowrect, 2, 35, 512, 342);
  setdrawingrect(windowrect);
  showdrawing;
 end;

 procedure MAP (x, y : real;
         var hor, ver : integer);
 begin
  if abs(x) > 1.9 then
   x := x / abs(x) * 1.9;
  hor := 255 + trunc(x * 130);
  if abs(y) > 1 then
   y := y / abs(y);
  ver := 138 - trunc(y * 130);
 end;

 procedure DRAW (x, y : real);
  var
   h1, v1 : integer;
 begin
  MAP(x, y, h1, v1);
  lineto(h1, v1);
 end;

 procedure SHIFT (x, y : real)
  var
   h1, v1 : integer;
 begin
  MAP(x, y, h1, v1);
  moveto(h1, v1);
 end;

 procedure ROTATE (var x, y, z : real;
         a, b, c : real);
  var
   sx, sy, sz, tx, ty, tz : real;
 begin
  sx := x * cos(c) - y * sin(c);
  sy := x * sin(c) + y * cos(c);
  sz := z;
  tx := sx;
  ty := sy * cos(b) - sz * sin(b);
  tz := sy * sin(b) + sz * cos(b);
  x := tx * cos(a) - ty * sin(a);
  y := tx * sin(a) + ty * cos(a);
  z := tz;
 end;

 procedure SDRAW (x, y, z, a, b, c : real);
  var
   u, v, w : real;
 begin
  u := x;
  v := y;
  w := z;
  ROTATE(u, v, w, a, b, c);
  DRAW(u, v);
 end;

 procedure SMOVE (x, y, z, a, b, c : real);
  var
   u, v, w : real;
 begin
  u := x;
  v := y;
  w := z;
  ROTATE(u, v, w, a, b, c);
  SHIFT(u, v);
 end;

begin
 CLEAR;
 showtext;
 write('Euler angles (-1  0.76  0.85): ');
 readln(a, b, c);
 for m := 0 to grid do
  begin
   for n := 0 to grid do
```

(continued)

```
    begin
     x := 0.7 * (-1 + 2 * m / grid);
     y := 0.7 * (-1 + 2 * n / grid);
     if n = 0 then
      SMOVE(x, f(x, y), y, a, b, c)
     else
      SDRAW(x, f(x, y), y, a, b, c);
    end;
  end;
 for n := 0 to grid do
  begin
   for m := 0 to grid do
    begin
     x := 0.7 * (-1 + 2 * m / grid);
     y := 0.7 * (-1 + 2 * n / grid);
     if m = 0 then
      SMOVE(x, f(x, y), y, a, b, c)
     else
      SDRAW(x, f(x, y), y, a, b, c);
    end;
  end;
end.
```

Program 6.3

```
program CCL4;
(* DRAWS CARBON TETRACHLORIDE WITH PERSPECTIVE *)

 const
  num = 5;

 var
  x : array[1..num, 1..3] of real;
  a, b, c, depth : real;
  warp : array[1..num] of real;
  order : array[1..num] of integer;
  m, n, temp : integer;

 procedure CLEAR;
  var
   windowrect : rect;
 begin
  hideall;
  setrect(windowrect, 2, 35, 512, 342);
  setdrawingrect(windowrect);
  showdrawing;
 end;

 procedure MAP (x, y : real;
         var hor, ver : integer);
```

```
begin
 if abs(x) > 1.9 then
  x := x / abs(x) * 1.9;
 hor := 255 + trunc(x * 130);
 if abs(y) > 1 then
  y := y / abs(y);
 ver := 138 - trunc(y * 130);
end;

procedure DRAW (x, y : real);
 var
  h1, v1 : integer;
begin
 MAP(x, y, h1, v1);
 lineto(h1, v1);
end; {draw}

procedure SHIFT (x, y : real);
 var
  h1, v1 : integer;
begin
 MAP(x, y, h1, v1);
 moveto(h1, v1);
end; {shift}

procedure disk (x, y, r : real);
(* Draws solid disk of radius r centered at (x,y) *)
 var
  h1, v1, h2, v2 : integer;
begin
 MAP(x - r, y + r, h1, v1);
 MAP(x + r, y - r, h2, v2);
 eraseoval(v1, h1, v2, h2);
 frameoval(v1, h1, v2, h2);
end;

procedure ROTATE (var x, y, z : real;
        a, b, c : real);
 var
  sx, sy, sz, tx, ty, tz : real;
begin
 sx := x * cos(c) - y * sin(c);
 sy := x * sin(c) + y * cos(c);
 sz := z;
 tx := sx;
 ty := sy * cos(b) - sz * sin(b);
 tz := sy * sin(b) + sz * cos(b);
 x := tx * cos(a) - ty * sin(a);
 y := tx * sin(a) + ty * cos(a);
 z := tz;
end;
```

(continued)

```
 procedure perspect (var x, y, z, warp : real;
         a, b, c, depth : real);
 begin
  ROTATE(x, y, z, a, b, c);
  warp := depth / (depth - 1 - z);
  x := x * warp;
  y := y * warp;
 end;

begin
 x[1, 1] := 1;
 x[1, 2] := 0;
 x[1, 3] := 0;
 x[2, 1] := -1 / 2;
 x[2, 2] := sqrt(3) / 2;
 x[2, 3] := 0;
 x[3, 1] := -1 / 2;
 x[3, 2] := -sqrt(3) / 2;
 x[3, 3] := 0;
 x[4, 1] := 0;
 x[4, 2] := 0;
 x[4, 3] := sqrt(2);
 x[5, 1] := 0;
 x[5, 2] := 0;
 x[5, 3] := sqrt(2) / 4;
 for m := 1 to num do
  begin
   for n := 1 to 3 do
    begin
     x[m, n] := x[m, n] * 0.2;
    end;
  end;
 CLEAR;
 showtext;
 write('Euler angles: ');
 readln(a, b, c);
 write('Depth (3): ');
 readln(depth);
 for m := 1 to num do
  begin
   perspect(x[m, 1], x[m, 2], x[m, 3], warp[m], a, b, c, depth);
   order[m] := m;
  end;
 for m := 1 to num - 1 do
  begin
   for n := m + 1 to num do
    begin
     if warp[order[m]] > warp[order[n]] then
      begin
       temp := order[m];
       order[m] := order[n];
```

```
        order[n] := temp;
       end;
     end;
   end;
   for n := 1 to num do
    begin
     m := order[n];
     x[m, 1] := x[m, 1] * warp[m];
     x[m, 2] := x[m, 2] * warp[m];
     disk(x[m, 1], x[m, 2], 0.1 * warp[m]);
     SHIFT(x[m, 1], x[m, 2]);
     if m < num then
      writedraw('Cl')
     else
      writedraw('C');
    end;
  end.
```

Program 6.4

```
program lattice;
(* DRAWS 5X5X5 ATOMIC SITE LATTICE *)


 const
  num = 125;
  side = 5;

 var
  x : array[1..num, 1..3] of real;
  a, b, c, depth : real;
  warp : array[1..num] of real;
  order : array[1..num] of integer;
  m, n, p, temp, index : integer;

 procedure CLEAR;
  var
   windowrect : rect;
 begin
  hideall;
  setrect(windowrect, 2, 35, 512, 342);
  setdrawingrect(windowrect);
  showdrawing;
 end;

 procedure MAP (x, y : real;
         var hor, ver : integer);
(* Modified library procedure *)
 begin
  hor := 255 + trunc(x * 130);
```

(continued)

```
  ver := 138 - trunc(y * 130);
 end;

 procedure DRAW (x, y : real);
  var
   h1, v1 : integer;
 begin
  MAP(x, y, h1, v1);
  lineto(h1, v1);
 end;

 procedure SHIFT (x, y : real);
  var
   h1, v1 : integer;
 begin
  MAP(x, y, h1, v1);
  moveto(h1, v1);
 end;

 procedure disk (x, y, r : real);
(* Draws solid disk of radius r centered at (x,y) *)
  var
   h1, v1, h2, v2 : integer;
 begin
  MAP(x - r, y + r, h1, v1);
  MAP(x + r, y - r, h2, v2);
  eraseoval(v1, h1, v2, h2);
  frameoval(v1, h1, v2, h2);
 end;

 procedure ROTATE (var x, y, z : real;
         a, b, c : real);
  var
   sx, sy, sz, tx, ty, tz : real;
 begin
  sx := x * cos(c) - y * sin(c);
  sy := x * sin(c) + y * cos(c);
  sz := z;
  tx := sx;
  ty := sy * cos(b) - sz * sin(b);
  tz := sy * sin(b) + sz * cos(b);
  x := tx * cos(a) - ty * sin(a);
  y := tx * sin(a) + ty * cos(a);
  z := tz;
 end;

 procedure perspect (var x, y, z, warp : real;
         a, b, c, depth : real);
 begin
  ROTATE(x, y, z, a, b, c);
  warp := depth / (depth - 1 - z);
  x := x * warp;
  y := y * warp;
 end;

begin
 for m := 0 to side - 1 do
  begin
   for n := 0 to side - 1 do
    begin
     for p := 0 to side - 1 do
      begin
       index := 1 + p + side * n + sqr(side) * m;
       x[index, 3] := 1 + p;
       x[index, 2] := -1 + 2 * n / side;
       x[index, 1] := -1 + 2 * m / side;
      end;
    end;
  end;

 for m := 1 to num do
  begin
   for n := 1 to 3 do
    begin
     x[m, n] := x[m, n] * 0.2;
    end;
  end;
 CLEAR;
 showtext;
 write('Euler angles: ');
 readln(a, b, c);
 write('Depth (3.3): ');
 readln(depth);
 for m := 1 to num do
  begin
   perspect(x[m, 1], x[m, 2], x[m, 3], warp[m], a, b, c, depth);
   order[m] := m;
  end;
 for m := 1 to num - 1 do
  begin
   for n := m + 1 to num do
    begin
     if warp[order[m]] > warp[order[n]] then
      begin
       temp := order[m];
       order[m] := order[n];
       order[n] := temp;
      end;
    end;
  end;
 backpat(LtGray);
 pensize(2, 2);
```

(continued)

```
  for n := 1 to num do
   begin
    m := order[n];
    x[m, 1] := x[m, 1] * warp[m];
    x[m, 2] := x[m, 2] * warp[m];
    disk(x[m, 1], x[m, 2], 0.07 * warp[m]);
    SHIFT(x[m, 1], x[m, 2]);
   end;
end.
```

7 Dynamical Models

THEME: Problems of the general category of dynamical (time-dependent) models are discussed.

GOALS: To achieve familiarity with the programming of dynamical models, with emphasis on graphic representation of solutions.

LIBRARIES USED: 3D.lib, Graphics.lib.

REFERENCE MATERIALS: Texts and references from your chosen field.

Space-Time Models

In space-time models the basic idea is to assign two axes to space and time, respectively. Of special interest are those problems from physics, biology, chemistry, and engineering in which some function:

```
f(x,t)
```

of space (x) and time (t) is desired. In such situations, the value of f will be a third axis. Methods from the last chapter for three-dimensional grahics apply. In Program 7.1, Pulse, a particularly convenient set of Euler angles is given as:

$$c = 0.86$$

$$b = 0.75$$

$$a = -1$$

This produces a visually comfortable set of three axes, as in Figure 7.1, where the axis going off to the left is time, the axis to the right is space, and the vertical axis for this program is pulse amplitude.

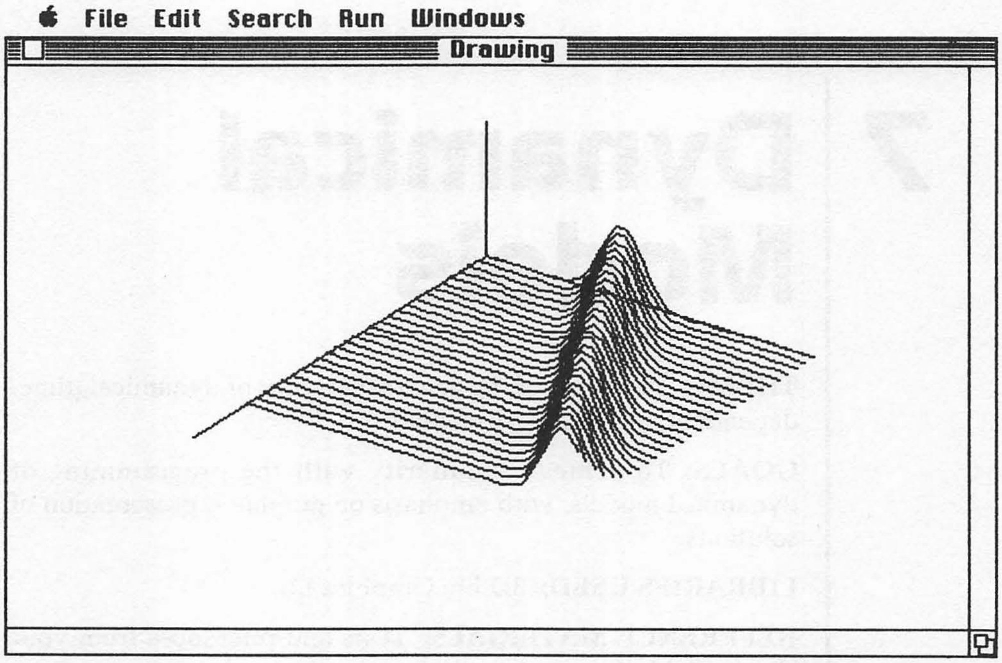


Figure 7.1 | Pulse output

What is graphed is a Gaussian wave pulse of the form:

psi(x,t) = constant * exp(– sqr(ax – bt))

The pulse propagates along the x-axis with velocity $\frac{b}{a}$. It is important to note that you may graph in this way either a 'hard function' as in program Pulse, or the solution to a dynamic differential equation. For the Pulse example, the wave is a solution to a classical wave equation for velocity $\frac{b}{a}$.

Parametric Space Curves

When time is thought of as a parameter, three functions such as:

x(t)

y(t)

z(t)

may be displayed as a single space curve which twists and winds its way through 3-space as the time t increases. Of course, the parameter need not be t, but often some physical quantities x,y,z are dynamical variables governed by differential equations; and whether the parameter of differentiation is actually time or not, the model can be thought of dynamically. In relativistic physics, for example, the space curve parameter can be 'proper time,' which is actually an invariant distance, or even a dimensionless parameter.

Program 7.2, StrangeAttractor, models the so-called 'Lorenz Attractor.' This is a system of coupled differential equations:

```
dx/dt = -sigma * (x-y);
dy/dt = -x * (z-r) - y;
dz/dt = x * y - b * z;
```

where sigma, b, and r are constants.

This model describes a complex problem in the theory of fluids. Phenomena such as chaos and attraction are exhibited by this system. The graphic output is interesting.

Figure 7.2 shows the appearance of the space curve. What is not shown is how the two attractors—the centers of the obvious vortices—keep being revisitied alternately by the trajectory. This behavior is evident when you run the program.

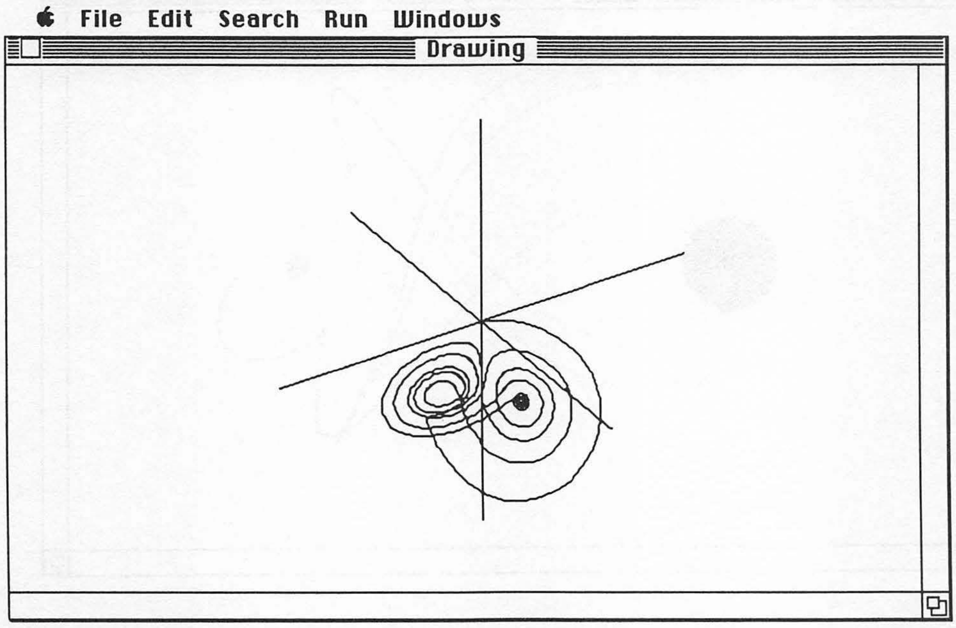


Figure 7.2 | StrangeAttraction output—space curve

Animation

The techniques of Chapter 4 in regard to animation can be applied to general dynamical problems. This approach is especially useful if the problem at hand does not have a convenient analytical solution. One example is the three-body problem of an object orbiting under the influence of both the earth and moon. The equation of motion for the position r of the orbiting projectile is:

$$\frac{d^2r}{dt^2} = \frac{-GM(r-re)}{(|r-re|)^3} - \frac{Gm(r-rm)}{(|r-rm|)^3}$$

where re, rm are the position vectors of the earth, moon, respectively; and M, m are the respective masses of these bodies. In Program 7.3, Lunacy, the equation of motion is emobodied in the four statements involving the acceleration record. Both acceleration.x and acceleration.y must be updated in such problems because acceleration resolves into its orthogonal components.

Figure 7.3 shows the results of aiming with the programmed mouse-cue method (see, for example, program Billiards of Chapter 4) in a way that enables the projectile to orbit both planet and moon.

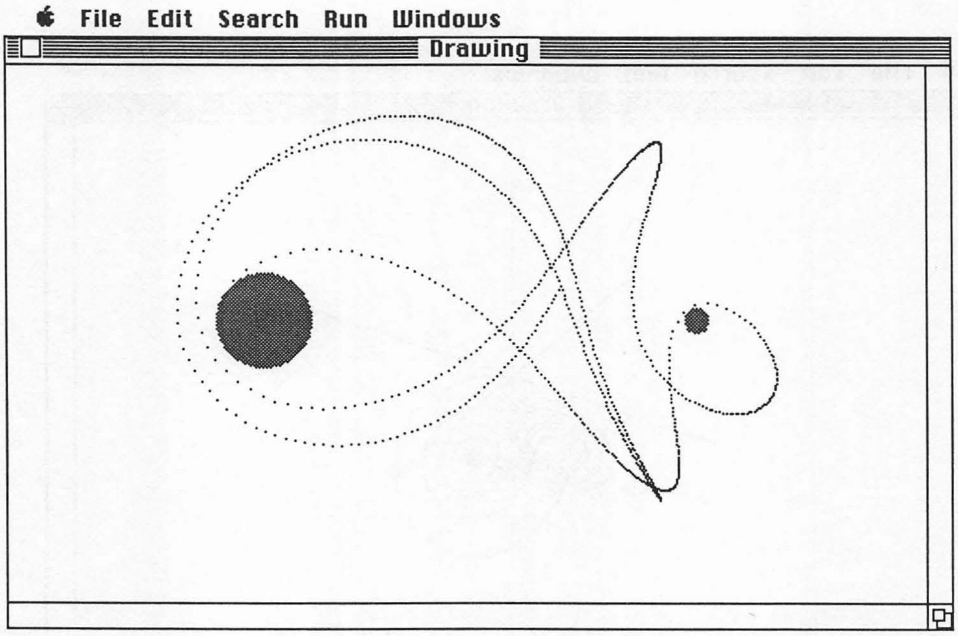


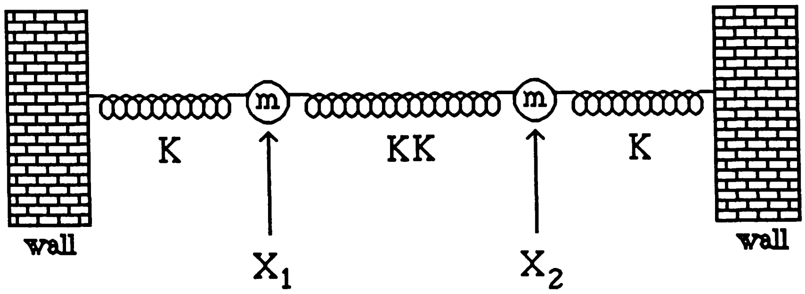
Figure 7.3 | Lunacy output—planet and moon orbit

Exercises

1. Remove the lunar gravitation pull from program Lunacy and verify that the projectile can indeed be launched into a stable, elliptical orbit around the Earth.

2. Write a program to model the behavior of a particle in the rings of Saturn, as follows. Such particles are in orbit around the main planet, but are perturbed by orbiting Saturnian moons. Assume one moon, and force it to travel in a circular orbit at radius 1, but with otherwise correct period, with Saturn at the origin. Then solve the dynamics of a ring particle which is pulled upon by both planet and moon. If you launch this ring particle in an orbit which would be circular without the moon present, it will be perturbed by the moon and the orbit will go uncircular. But the overall perturbation is a radical function of the particle's initial radius. In fact, for a certain radius, the period of the particle is about one half the period of the orbiting moon. At this radius for the ring particle, a *resonance* occurs which causes relatively great perturbation. This has been proposed as an explanation of the dark Cassini Division—a black annulus which appears to separate the main ring from a thin outer one. The division is thought to be caused by the moon Mimas, which is placed just so that a circular orbit at Cassini's Division would have half of Mimas' orbital period. A good program should show radical effects at such a radius. With some patience and the understanding that execute times will be slow, you can put many particles into the fray, and attempt to eventually come up with an actual, visible division at whose radius only a few particles subsist. You need a few hundred ring particles to see this effect on the graphics screen after scores of revolutions of the moon.

3. Model a coupled oscillator system as follows. Assume three springs tied in series, with the whole assembly affixed to two rigid walls. Two equal masses exist at the two places where springs meet:



The outer springs have equal spring constant K while the inner spring has a different constant KK. The equations of moion for the coordinates x1 and x2 are:

$$m \frac{d^2}{dt^2} x_1 = - K * x_1 - KK * (x_1 - x_2);$$

$$m \frac{d^2}{dt^2} x_2 = - K * x_2 - KK * (x_2 - x_1);$$

Write a program to solve these equations, and animate the masses on the graphics screen. Several interesting phenomena should be evident:

a. There are two normal mode frequencies that are apparent when you either start both masses going to the right initially, or when you start them out of phase (moving apart, for example) initially.
b. For sufficiently small ratio $\frac{K2}{K}$, the coupling is called 'weak,' and in this case, beats between the normal modes will be evident. In fact, if $\frac{K2}{K}$ is small, then starting one mass moving initially with the other at rest will start a fascinating process in which the originally moving mass eventually transfers virtually all of its energy to the other mass, after which the motion sloshes back to the original mass. This goes on forever at a frequency given by the difference between the normal mode frequencies of (a).

4. Write a program to model the Logistic Equation of population biology:

$$\frac{dN}{dt} = r * N * (K - N)$$

where:

N = population of organisms (number of living entities)

r = constant Malthusian growth rate

K = carrying capacity (population limit)

By starting with small initial values of N at time t = 0, you can witness the population growing exponentially as exp(rt) for some time, then being limited by the carrying capacity term and settling out to a constant value.

5. Make a space-time evolution plot of the celebrated *double soliton:*

$$f(x,t) = \frac{4\cosh(2x-8t) + \cosh(4x-64t) + 3}{(3\cosh(x-28t) + \cosh(3x-36t))^2}$$

This strange animal consists of two lumps which, starting from time t = 0, move toward each other, collide and pass through each other, then emerge intact after the collision, proceeding to surry away from each other forever thereafter. Solitons were discovered in the late 1960's by computer analysis. Generally, they represent solutions to nonlinear equations of motion which do not 'fall apart' either by collision or by age.

6. The Diffusion Equation is a good example of a differential equation involving both space and time but requiring only one set of initial data. The differential equation is:

$$\frac{d}{dt} f(x,t) = D \frac{d^2}{dx^2} f(x,t)$$

with D denoting the Diffusion Constant for the problem. Starting with a Gaussian initial condition:

```
f(x,0) = exp(-5*x*x)
```

corresponding to some substance localized in space (imagine a "splot" of ink which will diffuse through a water medium, for example), make a space-time plot of the evolution of f(x,t). It is a good idea for such problems to set up an array for f, as with:

```
var f: array[-max..max] of real;
```

from which the second derivative with respect to x is given by:

$$dd := \frac{(f[j+1] - 2*f[j] + f[j-1])}{dx}$$

where j is $<$ max and $>$ -max, and represents the spatial point $x = j*dx$. Then the evolution of f is determined by an equation of motion such as:

```
f[j]:= f[j] + dd * D * dt;
```

and the techniques of program Pulse should give a good display. This Gaussian initial condition will 'diffuse' as it should into shallower and shallower Gaussians with time. If you are ambitious, you can tackle the problem of diffusion such as this but between two reflecting walls. Then any initial distribution of matter f(x,0) should, by rights, end up after long t as a flat, featureless distribution as the matter settles between the walls. The key to handling this kind of wall-boundary condition is to force the spatial derivative $\frac{df}{dx}$ to vanish at any wall.

7. The dynamics of a circular membrane (drumhead) are beautiful indeed. It is a fairly involved task to solve the time dependent differential equations for a membrane by computer, but a good alternative is to use the known solutions

and to make 3-space surface plots of the drumhead. Let z be a zero of the n-th Bessel function Jn. Then the function defined for polar coordinates (r,theta) by:

$f(r,theta) = J_n(z*r) * \cos(n*theta)$

represents a 'snapshot' of an oscillating drumhead whose fixed rim is at radius r = 1. Write a program to make surface plots of some of these beautiful oscillation modes. One can either track along rays of constant theta, or along line loops of constant r, always plotting the membrane height as f(r,theta).

Answers

1. This involves the simple removal of the moon component of acceleration; in fact, the whole block starting 'if not crash then . . . ' should be removed.
2. The whole problem revolves around a Kepler's Law, which states that if a body orbits at a distance r from the Saturn mass Msat, then the angular velocity of the stable circular orbit is:

   $$\Omega := \text{sqrt}\left(\frac{G * Msat}{(r*r*r)}\right);$$

   An angular velocity variable is to be used in the equation:

   $angle := angle + \Omega * dt;$

   that is, analogously to linear velocity. Once the moon is moving in this circular orbit, the forces on the ring particle can be computed just as in program Lunacy.
3. The normal mode frequencies are:

   $$f1 = 2 * \pi * \text{sqrt}\left(\frac{K}{m}\right)$$
   $$f2 = 2 * \pi * \text{sqrt}\left(\frac{K + KK}{m}\right)$$
4. The steady-state value of N is just the carrying capacity K.
5. The program Pulse may be used directly for this problem. All you have to alter is the procedure setup, where the soliton formula should be typed in with appropriate overall scaling so the wave fits onto the screen.
6. The diffusion solution always looks like a pulse of some kind that flattens out in time. The problem with reflecting walls is relatively difficult. The *method*

of images applies here. You can take the solution at a point x between the walls (at −a and +a) to be the sum of solutions for identical sources at ± 2a, ± 3a, ± 4a, . . . This will ensure a zero derivative at each wall.

7. The program Surface is easily altered to plot the functions f(r,theta). All you need is these additional facts:

```
x: = r * cos(theta);
y: = r * sin(theta);
the z coordinate is just f itself
```

Program 7.1

```
program pulse;
(* DRAWS SPACE-TIME PROPAGATION OF WAVE PULSE *)

 const
  max = 20;
  dt = 0.06;
  c = 0.85
  b = 0.76;
  a = -1;
  psiscale = 3;

 type
  wave = array[-max..max] of real;

 var
  psi : wave;
  t : real;

 procedure CLEAR;
  var
   windowrect : rect;
 begin
  hideall;
  setrect(windowrect, 2, 35, 512, 342);
  setdrawingrect(windowrect);
  showdrawing;
 end;

 procedure MAP (x, y : real;
         var hor, ver : integer);
 begin
  if abs(x) > 1.9 then
   x := x / abs(x) * 1.9;
  hor := 255 + trunc(x * 130);
  if abs(y) > 1 then
   y := y / abs(y);
  ver := 138 - trunc(y * 130);
 end;

 procedure DRAW (x, y : real);
  var
   h1, v1 : integer;
 begin
  map(x, y, h1, v1);
  lineto(h1, v1);
 end;

 procedure SHIFT (x, y : real);
 var
  h1, v1 : integer;
begin
 MAP(x, y, h1, v1);
 moveto(h1, v1);
end;

procedure ROTATE (var x, y, z : real;
        a, b, c : real);
 var
  sx, sy, sz, tx, ty, tz : real;
begin
 sx := x * cos(c) - y * sin(c);
 sy := x * sin(c) + y * cos(c);
 sz := z;
 tx := sx;
 ty := sy * cos(b) - sz * sin(b);
 tz := sy * sin(b) + sz * cos(b);
  x := tx * cos(a) - ty * sin(a);
  y := tx * sin(a) + ty * cos(a);
  z := tz;
 end;

 procedure SDRAW (x, y, z, a, b, c : real);
  var
   u, v, w : real;
 begin
  u := x;
  v := y;
  w := z;
  ROTATE(u, v, w, a, b, c);
  DRAW(u, v);
 end;

 procedure SMOVE (x, y, z, a, b, c : real);
  var
   u, v, w : real;
 begin
  u :=
  v := y;
  w := z;
  ROTATE(u, v, w, a, b, c);
  SHIFT(u, v);
 end;

 procedure setup (var psi : wave;
         t : real);
  var
   ii : integer;
   x : real;
 begin
```

(continued)

```
  for ii := -max to max do
   begin
    x := ii / max;
    psi[ii] := exp(-sqr(6 * x - 4 * t + 1)) / psiscale;
   end
 end;

 procedure ax;
 begin
  SMOVE(0.8, -0.3, -1, a, b, c);
  SDRAW(-0.8, -0.3, -1, a, b, c);
  SDRAW(-0.8, 0.3, -1, a, b, c);
  SMOVE(-0.8, -0.3, -1, a, b, c);
  SDRAW(-0.8, -0.3, 1, a, b, c);
 end;

 procedure showwave (psi : wave;
         t : real);
  var
   ii : integer;
 begin
  for ii := -max to max do
   begin
    if ii = -max then
     SMOVE(-0.8, psi[ii] - 0.3, -1 + t, a, b, c)
    else
     SDRAW(0.8 * ii / max, psi[ii] - 0.3, -1 + t, a, b, c);
   end;
 end;

begin
 CLEAR;
 ax;
 t := 0;
 setup(psi, t);
 showwave(psi, 0);
 repeat
  t := t + dt;
  setup(psi, t);
  showwave(psi, t);
 until t > 1.6;
end.
```

Program 7.2

```
program StrangeAttractor;
(* GRAPHS 'Lorenz Attractor' IN THREE SPACE *)

 const
  sigma = 10;
 bb = 2.7;
 r = 28;
 a = 0;
 b = 1;
 c = -1;
 dt = 0.01;

var
 x, y, z, t, vx, vy, vz : real;

procedure CLEAR;
 var
  windowrect : rect;
begin
 hideall;
 setrect(windowrect, 2, 35, 512, 342);
 setdrawingrect(windowrect);
 showdrawing;
end;

procedure MAP (x, y : real;
        var hor, ver : integer);
begin
 if abs(x) > 1.9 then
  x := x / abs(x) * 1.9;
 hor := 255 + trunc(x * 130);
 if abs(y) > 1 then
  y := y / abs(y);
 ver := 138 - trunc(y * 130);
end;

procedure DRAW (x, y : real);
 var
  h1, v1 : integer;
begin
 MAP(x, y, h1, v1);
 lineto(h1, v1);
end;

procedure SHIFT (x, y : real);
 var
  h1, v1 : integer;
begin
 MAP(x, y, h1, v1);
 moveto(h1, v1);
end;

procedure ROTATE (var x, y, z : real;
        a, b, c : real);
 var
  sx, sy, sz, tx, ty, tz : real;
```

(continued)

```
 begin
  sx := x * cos(c) - y * sin(c);
  sy := x * sin(c) + y * cos(c);
  sz := z;
  tx := sx;
  ty := sy * cos(b) - sz * sin(b);
  tz := sy * sin(b) + sz * cos(b);
  x := tx * cos(a) - ty * sin(a);
  y := tx * sin(a) + ty * cos(a);
  z := tz;
 end;

 procedure SDRAW (x, y, z, a, b, c : real);
  var
   u, v, w : real;
 begin
  u := x;
  v := y;
  w := z;
  ROTATE(u, v, w, a, b, c);
  DRAW(u, v);
 end;

 procedure SMOVE (x, y, z, a, b, c : real);
  var
   u, v, w : real;
 begin
  u := x;
  v := y;
  w := z;
  ROTATE(u, v, w, a, b, c);
  SHIFT(u, v);
 end;

 procedure AXES (a, b, c : real);
 begin
  SMOVE(-1, 0, 0, a, b, c);
  SDRAW(1, 0, 0, a, b, c);
  SMOVE(0, -1, 0, a, b, c);
  SDRAW(0, 1, 0, a, b, c);
  SMOVE(0, 0, -1, a, b, c);
  SDRAW(0, 0, 1, a, b, c);
 end;

begin
 CLEAR;
 x := 0.01;
 y := 0;
 z := 0;
 t := 0;
 AXES(a, b, c);
 SMOVE(x, y, z, a, b, c);
 repeat
  vx := -sigma * (x - y);
  vy := -x * (z - r) - y;
  vz := x * y - bb * z;
  x := x + vx * dt;
  y := y + vy * dt;
  z := z + vz * dt;
  SDRAW(x / 70, y / 70, z / 70, a, b, c);
 until false;
end.
```

Program 7.3

```
program lunacy;
(* USER LAUNCHES PROJECTILE WHICH ORBITS WITHIN EARTH-MOON SYSTEM *)

 const
  K = 6.67;
  L = 1;
  dt = 0.01;

 var
  rcubed, rr, ss, speed, an : real;
  crash : boolean;
  vel, accel, pos, oldpos : record
    x, y : real;
   end;

 procedure CLEAR;
  var
   windowrect : rect;
 begin
  hideall;
  setrect(windowrect, 2, 35, 512, 342);
  setdrawingrect(windowrect);
  showdrawing;
 end;

 procedure MAP (x, y : real;
         var hor, ver : integer);
 begin
  if abs(x) > 1.9 then
   x := x / abs(x) * 1.9;
  hor := 255 + round(x * 130);
  if abs(y) > 1 then
   y := y / abs(y);
  ver := 138 - round(y * 130);
 end;
```

(continued)

```
procedure UNMAP (h, v : integer;
        var x, y : real);
begin
 x := (h - 255) / 130;
 y := (138 - v) / 130;
end;

procedure DRAW (x, y : real);
 var
  h1, v1 : integer;
begin
 MAP(x, y, h1, v1);
 lineto(h1, v1);
end;

procedure SHIFT (x, y : real);
 var
  h1, v1 : integer;
begin
 MAP(x, y, h1, v1);
 moveto(h1, v1);
end;

procedure PLOT (x, y : real);
begin
 SHIFT(x, y);
 DRAW(x, y);
end;

procedure disk (x, y, r : real);
 var
  h2, v2, h1, v1 : integer;
begin
 MAP(x - r, y + r, h2, v2);
 MAP(x + r, y - r, h1, v1);
 paintoval(v2, h2, v1, h1);
end;

procedure explode (x, y, r : real);
 var
  h2, v2, h1, v1, k : integer;
begin
 MAP(x - r, y + r, h2, v2);
 MAP(x + r, y - r, h1, v1);
 for k := 1 to 20 do
  begin
   case (k mod 3) of
    2 :
     penpat(gray);
    1 :
      penpat(white);
     0 :
      penpat(black);
    end;
    paintoval(v2, h2, v1, h1);
   end;
  eraseoval(v2, h2, v1, h1);
 end;

 function angle (x, y, xold, yold : real) : real;
{returns the angle of the vector whose tail is at 'old'}
  const
   ppi = 3.14159265359;
  var
   an : real;
 begin
  an := arctan((y - yold) / (abs(x - xold) + 0.000001));
  if x < xold then
   if y < yold then
    an := -ppi - an
   else
    an := ppi - an;
  angle := an;
 end;

 procedure stroke (var x, y, xold, yold : real);
{animates a mouse-stroke, returns endpoint (x,y), original}
{point (xold,yold) as reals}
  var
   h1, h2, v1, v2 : integer;
 begin
  while not button do
   begin
   end;
  getmouse(h1, v1);
  h2 := h1;
  v2 := v1;
  while button do
   begin
    penpat(white);
    moveto(h1, v1);
    lineto(h2, v2);
    getmouse(h2, v2);
    moveto(h1, v1);
    penpat(black);
    lineto(h2, v2);
   end;
    penpat(white);
    moveto(h1, v1);
    lineto(h2, v2);
```

(continued)

```
  penpat(black);
  UNMAP(h1, v1, xold, yold);
  UNMAP(h2, v2, x, y);
 end;

begin
 CLEAR;
 penpat(gray);
 disk(-0.9, 0, 0.2);
 disk(+0.9, 0, 0.05);
 penpat(black);
 stroke(oldpos.x, oldpos.y, pos.x, pos.y);
 an := angle(pos.x, pos.y, oldpos.x, oldpos.y);
 speed := 5 * sqrt(sqr(pos.x - oldpos.x) + sqr(pos.y - oldpos.y));
 vel.x := cos(an) * speed;
 vel.y := sin(an) * speed;
 repeat
  rr := sqrt(sqr(pos.x + 0.9) + sqr(pos.y));
  rcubed := rr * sqr(rr);
  accel.x := -K * (pos.x + 0.9) / rcubed;
  accel.y := -K * pos.y / rcubed;
  ss := sqrt(sqr(pos.x - 0.9) + sqr(pos.y));
  if (rr < 0.18) or (ss < 0.048) then
   crash := true;
  if not crash then
   begin
    rcubed := ss * sqr(ss);
    accel.x := accel.x - L * (pos.x - 0.9) / rcubed;
    accel.y := accel.y - L * (pos.y) / rcubed;
    vel.x := vel.x + accel.x * dt;
    vel.y := vel.y + accel.y * dt;
    pos.x := pos.x + vel.x * dt;
    pos.y := pos.y + vel.y * dt;
    PLOT(pos.x, pos.y);
   end;
 until crash;
 if rr < 0.18 then
  explode(-0.9, 0, 0.2)
 else
  explode(+0.9, 0, 0.05);
end.
```

8 Serial I/O and External Devices

THEME: This chapter consists of optional, advanced material involving system calls from Macintosh Pascal. It is intended for those readers interested in serial communication to external devices. Covered are the subjects of terminal emulation and laboratory Input/Output.

GOALS: To be able to emulate various types of serial terminals with programs.

LIBRARIES USED: Graphics.lib (for the laboratory I/O methods).

REFERENCE MATERIALS: Texts and references on RS-232 serial communication, *Inside Macintosh* technical manual EMM-2048a *User Manual* (see end of chapter).

Basic Considerations for Serial I/O

Pascal language standards leave little room for handling serial communication. The basic problem is that the common input statement:

```
read(file,c);
```

where file is the serial port and c is a character to be read, will dwell until a character comes in from the outside world. There are "trick" ways around this in most Pascal systems. One of these is euphemistically referred to as 'lazy I/O,' and involves file pointers. We do not cover lazy I/O here; instead, we use methods of system calls from Macintosh Pascal, using a Generic(Status: integer; Regs: record) procedure, and write new procedures which do not dwell in the absence of a character.

To see why such machinations and special procedures are required to do terminal emulation from Pascal, follow the essential requirements for simple terminal design:

Step 1 If a keyboard key has been pressed, send the character out the serial port.

Step 2 If a character comes into the serial port from outside, put it on the screen.

The system being described is that of *full duplex*, meaning that either the host (remote) computer or the user (at the terminal) can send at any time. Unfortunately, both steps 1 and 2 must be able to happen simultaneously, or at least almost at the same time. The simplest approach is to scan through the possibilities 1 and 2 perpetually. This is called *polling,* in that the serial port and keyboard are alternately examined. Now you can see why:

```
read(f,c);
```

will not work. That would represent step 2, but would prevent going back quickly to step 1 if there is no character. A program written with read statements doing the work is therefore in danger of hanging up at some point. In the next section we describe how this problem can be solved at the expense of a fair amount of programming at system level.

There are ways to use simple read statements if specific hardware is to be connected to the serial port. At the end of this chapter we give some examples of such specific applications, but these examples will not work with arbitrary hosts connected at the serial port.

Another consideration for terminals is that you need to decode various keys for the popular meanings. Control keys, the ASCII <Delete> character, and so on, should be decoded in a proper terminal emulation program. This also requires some special Macintosh Pascal procedures.

9600 Baud Terminal Emulator

Macintosh Pascal, presenting the user as it does with an interpretive environment, is much too slow to keep up in real time with incoming characters at 9600 baud. But there are ways to ensure that no characters are lost. You can use the so-called 'Xon/Xoff' protocol, by which <ctrl-S> and <ctrl-Q> are sent to the host to 'hold off further transmission' and 'continue transmission from where you left off,' respectively. Most hosts in the computer world recognize this protocol. The Macintosh will support this protocol, and this is necessary for programs written even with compilers—the screen itself cannot generally keep up with 9600 baud. Even sophisticated commercial terminal emulators for the Macintosh go only to a few thousand baud rate, dominated, as implied, by the scrolling speed of the screen display.

Program 8.1, Terminal, uses protocols correctly to allow 9600 baud connection to serial RS-232 hosts having the Xon/Xoff protocol. The effective baud rate is on the order of 600 baud, but keep in mind that characters will not be lost; the program will simply be 'behind in time,' and long listings on the terminal screen will eventually be complete.

In the program, a necessarily long declaration block to set up the serial port parameters is evident. The *Inside Macintosh* documentation describes the meaning of these technical parameters, of which the baud rate is just one. There are several important constructs in program Terminal, for example:

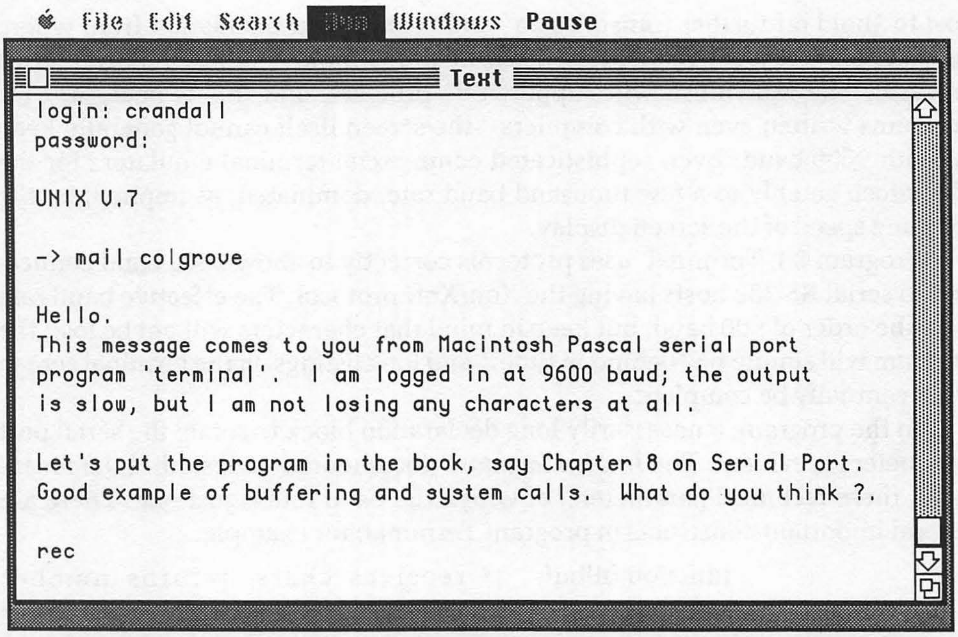
function fillbuf;	(* receives chars, returns number received *)
procedure handshake( );	(* handles Xon/Xoff protocol to prevent overflow *)
function getkey( );	(* handles keyboard mapping and returns character *)
procedure cleanbuf( );	(* reduces to 7-bit ASCII and handles line-feeds *)

These and other constructs allow the main loop at the end of the program Terminal to function in a manner consistent with the considerations of the last section: when the receive buffer is empty, the keyboard is still scanned for typed characters.

Figure 8.1 shows a typical log-in session in which the program Terminal was used to gain access to a UNIX time-sharing system at 9600 baud.

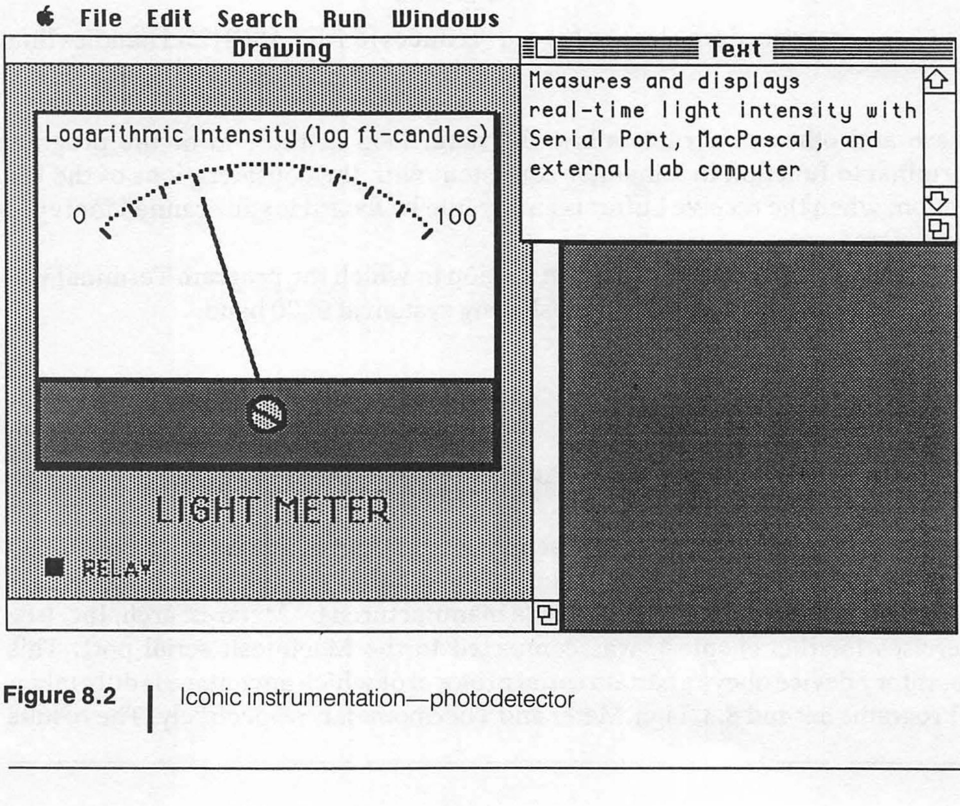
Laboratory Serial Applications

The serial methods available to Macintosh Pascal programmers can be applied more simply when the nature of the external hardware is precisely known. A laboratory computer, type EMM-2048a manufactured by Metaresearch, Inc. (see exercises for this chapter), was connected to the Macintosh serial port. This laboratory device obeys a certain serial protocol of which advantage is duly taken in Programs 8.2 and 8.3, LightMeter and Thermometer, respectively. The results



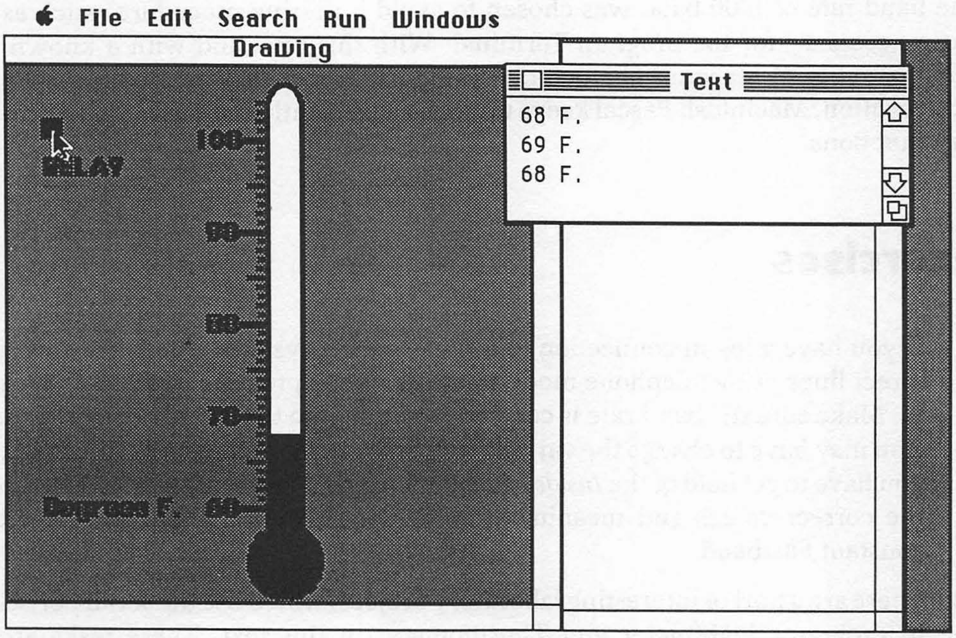
```
login: crandall
password:

UNIX V.7

-> mail colgrove

Hello.
This message comes to you from Macintosh Pascal serial port
program 'terminal'.  I am logged in at 9600 baud; the output
is slow, but I am not losing any characters at all.

Let's put the program in the book, say Chapter 8 on Serial Port.
Good example of buffering and system calls.  What do you think ?.

rec
```

Figure 8.1 | Log-in session with Terminal program


File Edit Search Run Windows
Drawing
Logarithmic Intensity (log ft-candles)
0
100
LIGHT METER
RELAY
Text

```
Measures and displays
real-time light intensity with
Serial Port, MacPascal, and
external lab computer
```

Figure 8.2 | Iconic instrumentation—photodetector



```
68 F.
69 F.
68 F.
```

Figure 8.3 | Iconic instrumentation—temperature sensor

of these programs appear as 'iconic instrumentation' on the Macintosh screen.

The appearances of the iconic instrumentation screens are shown in Figures 8.2 and 8.3. The displays show actual laboratory data—these are not simulations. Rather, a physical photodetector (for program Lightmeter) and a physical Kelvin temperature sensor (for program Thermometer the Kelvin temperature data are converted to Fahrenheit) were connected to the EMM-2048a.

The essential steps performed by these programs are as follows:

Step 1	Initialize serial port in the manner of program Terminal.
Step 2	Initialize the iconic panel display.
Step 3	'Wake up' the remote computer by using its known protocol (EMM-2048a User Manual).
Step 4	Interrogate, by sending characters to the serial port, and expect a voltage or frequency response (the EMM-2048a has analog/digital and voltage/frequency conversion capability).
Step 5	Receive the return characters from the remote computer and decode these into numerical data.
Step 6	Graph the needle position (for Lightmeter) or the mercury level (for Thermometer) according to the calculated data.
Step 7	Loop back to step 4.

The baud rate of 1200 baud was chosen to avoid buffering procedures such as those necessary for the program Terminal. With this rate, and with a known, small quantity of response characters coming back from the remote after each interrogation, Macintosh Pascal keeps up speed sufficiently for the stated laboratory functions.

Exercises

1. If you have a log-in connection to a time-sharing system, whether through direct lines or by telephone modem, try using the program Terminal to log in. Make sure the baud rate is correctly written into the initial declarations. You may have to change the various stop bits, parity, and so on. In that case, you have to get hold of the *Inside Macintosh* Apple documentation, to look up the correct values and meanings for the serial control bits in the 16-bit constant Fastbaud.
2. There are a host of interesting laboratory projects which use the serial port as in programs Lightmeter and Thermometer in this text. These tasks are surprisingly easy to get going, and iconic instrumentation is a wonderful thing, but you need commercial hardware for these tasks. This hardware has been developed at Reed College, under the Apple University Consortium plan, and is available from: Metaresearch, Inc.; 1100 SE Woodward, Portland, OR 97202.

Program 8.1

```
program terminal;
(* SERIAL TERMINAL EMULATOR, FULLY BUFFERED TO 9600 BAUD *)
(* Control characters keyboard-mapped with <Command> key *)
(* <Enter> key is ASCII 127 <Delete> (<Rubout>) *)
(* Effective throughput is  ~600 baud *)

 label
  1;

 const
  nBytes = 1024;          { #bytes in input buffers, determines
                            safety margin at high baud rate }
  maxchars = 255;         { max line length }
  nlines = 25;            { #lines on screen }
  AIn = -6;               { refnum of serial input port A }
  AOut = -7;              { refnum of serial output port A }
  PBRead = $A002;         { Trap Number of _Read }
  Control = $A004;        { Trap Number of _Control }
  Status = $A005;         { Trap Number of _Status }
  FastBaud = $CC0A;       { 9600 baud, 8 data bits, 2 stop, no parity}
 { CCBD is 600 baud, CC5E is 1200 baud, C17C is 300 baud }

 type

  byte = 0..255;
  ptr = ^byte;

  hardbuf = packed array[0..nBytes] of char;
  phardbuf = ^hardbuf;

  hd_pblk = record            { parameter block header }
    IOLink : LongInt;         { queue link in header }
    IOType : Integer;         { type byte for safety check }
    IOTrap : Integer;         { FS: the Trap }
    IOCmdAddr : LongInt;      { FS: Address to dispatch }
    IOCompletion : LongInt;   { pointer to IOCompletion routine }
    IOResult : Integer;       { IO result code }
    IOFileName : LongInt;     { file name pointer }
    IOVRefNum : Integer;      { volume refnum }
    IORefNum : Integer;       { reference number for I/O operation }
    csCode : integer;         { type of call }
   end;

  ctl_pblk = record           { parameter block for control calls }
    header : hd_pblk;
    csParam : Integer;        { baud rate etc. }
   end;
```

(continued)

```
  stat_pblk = record       { status calls }
    header : hd_pblk;
    csParam : LongInt;     { status info }
   end;

  buf_pblk = record        { set buffer size }
    header : hd_pblk;
    pBuf : phardbuf;       { pointer to hardware input buffer }
    buflen : Integer;      { buffer length }
   end;

  read_pblk = record       { read from hardware buffer }
    header : hd_pblk;
    dummy : longint;
    pBuf : ptr;            { buffer to read into }
    reqcount : longint;    { #bytes requested }
    actcount : longint     { #bytes actually delivered }
    posmode : integer;     { read from start, middle, or end of file? }
    posoffset : longin     { offset from start, ... }
   end;

 var

{ dummies }
  i, j, t : integer;
  l : longint;
  ch : char;
  tptr : ptr;

{ screen draw vars }
  finfo : fontinfo;
  linelen, lgn : integer         { line height }
  scrollbox : rect;              { area of text window }

{ high level io stuff }
  ibuf : packed array[0..maxchars] of char  { working input buffer }
  hbuf : phardbuf;               { hardware input buffer }
  ct : integer;                  { index into ibuf }
  hipt, lopt : integer;          { high, low points for xon, xoff }
  xoffsent : boolean;
  serialPort : text;
{ low level io stuff }
  ctl_block : ctl_pblk;
  stat_block : stat_pblk;
  buf_block : buf_pblk;
  read_block : read_pblk;
  regs : record
    a0 : ^ctl_pblk;
    a1, a2, a3, a4 : LongInt;
    d0, d1, d2, d3, d4, d5, d6, d7 : LongInt;
   end;
  function getbufsize : longint;
  begin
   Regs.A0 := @stat_block;
   Generic(Status, Regs);
   if Regs.D0 <> 0 then
    writeln('status error: ', Regs.D0);
   getbufsize := stat_block.csParam;
  end;

  function fillbuf : longint;
   var
    L : longint;
  begin
   L := getbufsize;
   if L > maxchars then
    L := maxchars;
   if L > 0 then
    begin
     read_block.reqcount := L;
     Regs.A0 := @read_block;
     Generic(PBRead, Regs);
     if Regs.D0 <> 0 then
      writeln('read error: ', Regs.D0);
    end;
   fillbuf := L;
  end;

  procedure handshake (force : boolean);
   var
    l : longint;
  begin
  l := getbufsize;
  if ((l > hipt) and not xoffsent) or force then
   begin
    write(serialPort, chr(19));
    xoffsent := true;
   end
  else if (l < lopt) and xoffsent then
   begin
    write(serialPort, chr(17));
    xoffsent := false;
   end;
 end;

 procedure cleanbuf (var L : longint);
  var
   i, j : integer;
 begin
  for i := 0 to L - 1 do
   begin
    j := ord(ibuf[i]) mod 128;
```

(continued)

```
   if j = 10 then
    j := 32;
   ibuf[i] := chr(j);
  end;
end;

function getkey : integer;
 var
  t : integer;
  ok : boolean;
  theEvent : EventRecord;
begin
 t := 0;
 ok := GetNextEvent(keydownmask + autokeymask, theEvent);
 if ok then
  begin
   t := (theEvent.message div 256) mod 256;       { keycode }
   if t = 52 then    { Enter }
    t := 127            { DEL }
   else
    t := theEvent.message mod 256;

{ command key }
   if bitand(theEvent.modifiers, cmdKey) <> 0 then
    begin
     if t in [97..122] then
      t := t - 96
     else if t in [91..93] then
      t := t - 64;
     if bitand(theEvent.modifiers, shiftKey) <> 0 then
      if t in [1..15] then
       t := t + 16;
    end;

  end;      {  if ok  }

 getkey := t;
end;     {  getkey  }

procedure putchar (t : integer);
begin
 if t > 0 then
  write(serialPort, chr(t));
end;

  { main program }
  begin
 hideall;
{ init serial parameters }
 open(serialPort, 'modem:');
 lopt := nbytes div 4;
 hipt := lopt * 3;
 xoffsent := false;

{ Reset SCC Channel with new baud, parity, etc. }
 with ctl_block, header do
  begin
   IOCompletion := 0;
   IORefNum := AOut;
   csCode := 8;
   csParam := FastBaud;
  end;
 Regs.A0 := @ctl_block;          { A0 pointer to Parameter Block }
 Generic(Control, Regs);
 if Regs.D0 <> 0 then
  writeln('Serial Port A error: ', Regs.D0); {Error codes returned in D0 }

{ set new buffer }
 new(hbuf);
 with buf_block, header do
  begin
   IOCompletion := 0;
   IORefNum := AOut;
   csCode := 9;
   pbuf := hbuf;
   buflen := nBytes;
  end;
 Regs.A0 := @buf_block;          { A0 pointer to Parameter Block }
 Generic(Control, Regs);
 if Regs.D0 <> 0 then
  writeln('set buffer error: ', Regs.D0); { Error codes returned in D0 }

{ init status block for getbufsize call }
 with stat_block.header do
  begin
   IOCompletion := 0;
   IORefNum := AOut;
   csCode := 2;
  end;

{ init block for read calls }
 with read_block, header do
  begin
   IOCompletion := 0;
```

(continued)

```
   IORefNum := AIn;
   pBuf := @ibuf;
   posmode := 0;
   posoffset := 0;
  end;

{ init terminal screen }
 textfont(4);
 textsize(9);
 getfontinfo(finfo);
 with finfo do
  linelen := ascent + descent + leading;
 setrect(scrollbox, 0, 0, 500, nlines * linelen);
 offsetrect(scrollbox, 5, 50);
 settextrect(scrollbox);
 showtext;

{ main loop }

 repeat

{ halt? }
  if button then
   goto 1;

{ get char and send to serial port }
  handshake(false);
  putchar(getkey);

{ get and print a line of text }
  l := fillbuf;
  if l > 0 then
   begin
    handshake(true);
    cleanbuf(l);
    if l > 0 then
     write(ibuf : l);
    handshake(false);
   end;

 until false;
 close(serialPort);

1 :
end.
```

Program 8.2

```
program LightMeter;
(* CREATES WORKING LIGHTMETER BY COMMUNICATING WITH SPECIAL*)
(*  EXTERNAL SERIAL HARDWARE (EMM-2048a Laboratory Computer) *)
```

```
 const
  AOut = -7;{RefNum of serial output port A}
  Control = $A004;{Trap Number of _Control}
  SetSCC = 8;{Reset SCC Channel}
  FastBaud = $CC5E;{BD is 600 baud, 8 data bits, 2 stop and no parity bit}
{5E is 1200 baud, 17C is 300 baud}
  xc = -0.655;
  yc = -0.28;
  pi = 3.1416;

 type
  ParamBlk = record
    IOLink : LongInt;{queue link in header}
    IOType : Integer;{type byte for safety check}
    IOTrap : Integer;{FS: the Trap}
    IOCmdAddr : LongInt;{FS: Address to dispatch}
    IOCompletion : LongInt;{pointer to IOCompletion routine}
    IOResult : Integer;{IO result code}
    IOFileName : LongInt;{file name pointer}
    IOVRefNum : Integer;{refnum}
    IORefNum : Integer;{reference number for I/O operation}
    IOFileType : Integer;{Type and permission access}
    IONewType : Integer;{baud rate etc.}
   end;

 var
  windowrect, textrect, click : rect;
  first, veryfirst, bleck : boolean;
  arr : array[1..256] of char;
  ff, xold, thetaold, x, y, theta, per, mul, freq, freq0, lightlevel : real;
  i, m : integer;
  ch, h, k : char;
  serialPort : text;
  regs : record
    a0 : ^ParamBlk;
    a1, a2, a3, a4 : LongInt;
    d0, d1, d2, d3, d4, d5, d6, d7 : LongInt;
   end;
  ParamBlock : ParamBlk;

 function change (var bleck : boolean) : boolean;
  function mousein : boolean;
   var
    t1, t2 : boolean;
    h, v : integer;
  begin
   getmouse(h, v);
   t1 := (h <= 270) and (h >= 20);
   t2 := (v <= 280) and (v >= 30);
   mousein := (t1) and (t2);
```

(continued)

```
  end;
 begin
  if (button) and (mousein) then
   begin
    change := true;
    if bleck then
     begin
      eraserect(click);
      pensize(1, 1);
      framerect(click);
      pensize(2, 2);
      bleck := false;
     end
    else
     begin
      paintrect(click);
      bleck := true;
     end;
   end
  else
   change := false;
 end; {change}

{GRAPHICS ROUTINES}
 function hor (x : real) : integer;
 begin
  hor := 230 + trunc(x * 140);
 end; {hor}

 function vert (y : real) : integer;
 begin
  vert := 137 - trunc(y * 137);
 end;  {vert}

 procedure draw (x, y : real);
 begin
  lineto(hor(x), vert(y));
 end;  {draw}

 procedure shift (x, y : real);
 begin
  moveto(hor(x), vert(y));
 end;  {shift}

 function dec (h, k : char) : integer;
 begin
  dec := 16 * (ord(h) - 48 - 7 * (ord(h) div 64)) + ord(k) - 48 - 7 * (ord(k)
  div 64);
 end;
 procedure select (mux : integer);
 begin
  write(serialPort, 'x0103', chr(13));
  repeat
   read(serialPort, ch);
  until ch = '-';
  write(serialPort, '0', chr(48 + mux), chr(13));
  repeat
   read(serialPort, ch);
  until ch = '-';
  write(serialPort, 'q');
  repeat
   read(serialPort, ch);
  until ch = '>';
 end;

begin
 bleck := true;
 hideall;
 setrect(windowrect, 2, 35, 300, 342);
 setdrawingrect(windowrect);
 showdrawing;
 penpat(ltgray);
 paintrect(0, 0, 342, 512);      {background}
 eraserect(25, 15, 170, 265);  {meter face}
 penpat(gray);
 paintrect(170, 15, 220, 265); {meter casing}
 penpat(ltgray);
 paintoval(180, 130, 200, 150);{pin}
 penpat(black);
 pensize(2, 2);
 framerect(25, 15, 220, 265);
 frameoval(180, 130, 200, 150);
 moveto(130, 184);
 lineto(148, 194);
 moveto(15, 170);
 lineto(263, 170);
 pensize(2, 2);
 moveto(15, 220);
 lineto(265, 220);
 lineto(265, 26);
 pensize(2, 2);
 moveto(21, 42);
 drawstring('Logarithmic Intensity (log ft-candles)');
 moveto(35, 87);
 drawstring('0');
 moveto(230, 87);
 drawstring('100');
 moveto(40, 280);
```

(continued)

```
 drawstring('RELAY');
 moveto(80, 250);
 textface([bold]);
 textsize(18);
 drawstring('LIGHT METER');
 setrect(click, 20, 270, 30, 280);
 paintrect(click);
 for m := -25 to 25 do
  begin
   if m mod 5 = 0 then
    ff := 0.86
   else
    ff := 0.9;
   theta := pi * (1 / 2 - m / 100);
   shift(xc + ff * cos(theta), yc + ff * sin(theta));
   draw(xc + 0.9 * cos(theta), yc + 0.9 * sin(theta));
  end;
 open(serialPort, 'modem:');
 ParamBlock.IOCompletion := 0;
 ParamBlock.IORefNum := AOut;
 ParamBlock.IOFileType := SetSCC;
 ParamBlock.IONewType := FastBaud;
 Regs.A0 := @ParamBlock;{A0 pointer to Parameter Block}
 Generic(Control, Regs);{Change the baud rate}
 if Regs.D0 <> 0 then
  writeln('Serial Port B error:', Regs.D0);{Error codes returned in D0}
 setrect(textrect, 280, 35, 512, 130);
 settextrect(textrect);
 showtext;
 textface([]);
 write('Measures and displays real-time');
 write(' light intensity with Serial Port,');
 write(' MacPascal, and external');
 writeln(' lab computer');
 veryfirst := true;
 close(serialPort);
end.
```

Program 8.3

```
program Thermometer;
 (* CREATES WORKING THERMOMETER BY COMMUNICATING WITH SPECIAL *)
 (* EXTERNAL SERIAL HARDWARE (EMM-2048a Laboratory Computer)  *)

  const
   AOut = -7;{RefNum of serial output port A}
   Control = $A004;{Trap Number of _Control}
   SetSCC = 8;{Reset SCC Channel}
   FastBaud = $CC5E;{BD is 600 baud, 8 data bits, 2 stop and no parity bit}
                          {5E is 1200 baud, 17C is 300 baud}
```

```
type

 ParamBlk = record
   IOLink : LongInt;{queue link in header}
   IOType : Integer;{type byte for safety check}
   IOTrap : Integer;{FS: the Trap}
   IOCmdAddr : LongInt;{FS: Address to dispatch}
   IOCompletion : LongInt;{pointer to IOCompletion routine}
   IOResult : Integer;{IO result code}
   IOFileName : LongInt;{file name pointer}
   IOVRefNum : Integer;{refnum}
   IORefNum : Integer;{reference number for I/O operation}
   IOFileType : Integer;{Type and permission access}
   IONewType : Integer;{baud rate etc.}
  end;
var
 arr : array[1..256] of char;
 per, mul, freq, mean, temp, next, last : real;
 i, m, lasttop, oldtemp, g : integer;
 ch, h, k : char;
 serialPort : text;
 click : rect;
 bleck, raise : boolean;
 regs : record
   a0 : ^ParamBlk;
   a1, a2, a3, a4 : LongInt;
   d0, d1, d2, d3, d4, d5, d6, d7 : LongInt;
  end;
 ParamBlock : ParamBlk;

function change (var bleck : boolean) : boolean;
 function mousein : boolean;
  var
   t1, t2 : boolean;
   h, v : integer;
 begin
  getmouse(h, v);
  t1 := (h <= 30) and (h >= 20);
  t2 := (v <= 40) and (v >= 30);
  mousein := (t1) and (t2);
 end; {mousein}
begin
 if (button) and (mousein) then
  begin
   change := true;
   if bleck then
    begin
     eraserect(click);
     framerect(click);
     bleck := false;
    end
```

(continued)

```
else
 begin
  paintrect(click);
  bleck := true;
 end;
  end
 else
  change := false;
end; {change}

function dec (h, k : char) : integer;
begin
 dec := 16 * (ord(h) - 48 - 7 * (ord(h) div 64)) + ord(k) - 48 - 7 * (ord(k)
 div 64);
end; {dec}

procedure thermometer;
 const
  mx = 140;
  lx = 125;
  ax = 130;
  bx = 135;
 var
  windowrect, therm : rect;
  y : integer;
begin
 setrect(windowrect, 2, 35, 300, 342);
 setdrawingrect(windowrect);
 showdrawing;
 penpat(gray);
 paintrect(0, 0, 342, 512);
 penpat(black);
 showtext;
 setrect(therm, 140, 10, 160, 270);
 pensize(2, 2);
 eraseroundrect(therm, 30, 30);
 frameroundrect(therm, 30, 30);
 frameoval(250, 130, 290, 170);
 eraserect(250, 142, 270, 158);
 eraseoval(252, 132, 288, 168);
 y := 290;
 repeat
  y := y - 50;
  moveto(mx, y);
  lineto(lx, y);
 until y = 40;
 TextFace([bold]);
 moveto(107, 245);
 drawstring('60');
 moveto(107, 195);
```

```
   drawstring('70');
   moveto(107, 145);
   drawstring('80');
   moveto(107, 95);
   drawstring('90');
   moveto(100, 45);
   drawstring('100');
   y := 240;
   pensize(1, 1);
   repeat
    y := y - 25;
    moveto(ax, y);
    lineto(mx, y);
   until y = 40;
   y := 240;
   repeat
    y := y - 5;
    moveto(bx, y);
    lineto(mx, y);
   until y = 20;
   moveto(20, 245);
   drawstring('Degrees F.');
   moveto(20, 60);
   drawstring('RELAY');
   setrect(click, 20, 30, 30, 40);
   paintrect(click);
   paintoval(250, 130, 290, 170);
   paintrect(250, 142, 251, 158);
   oldtemp := 54;
   lasttop := 270;
   next := 70;
   last := 70;
  end; {thermometer}

{Changes the temperature in the thermometer by}
{either painting or erasing a rectangle}
  procedure settemp;
   const
    left = 142;
    right = 158;
   var
    top, bottom : integer;
  begin
   if round(mean) > 104 then
    mean := 104;
   if round(mean) > oldtemp then
    begin
     top := 240 - ((round(mean) - 60) * 5);
     bottom := lasttop;
     paintrect(top, left, bottom, right);
```

(continued)

```
     lasttop := top;
    end;
   if round(mean) < 54 then
    mean := 54;
   if round(mean) < oldtemp then
    begin
     bottom := 240 - ((round(mean) - 60) * 5);
     top := lasttop;
     eraserect(top, left, bottom, right);
     lasttop := bottom;
    end;
   oldtemp := round(mean);
  end; {settemp}

  procedure changetemp (raise : boolean);
 {changes temp by turning light on and off}
   var
    m : integer;
  begin
   if raise then
    begin
     eraserect(click);
     framerect(click);
     bleck := false;
     write(serialPort, 's0');
    end
   else
    begin
     paintrect(click);
     bleck := true;
     write(serialPort, 'r0');
    end;
   for m := 1 to 2 do
    read(serialPort, ch);
   write(serialPort, chr(13));
   repeat
    read(serialPort, ch);
   until ch = '>';
  end; {changetemp}

  procedure select (mux : integer);
  begin
   write(serialPort, 'x0103', chr(13));
   repeat
    read(serialPort, ch);
   until ch = '-';
   write(serialPort, '0', chr(48 + mux), chr(13));
   repeat
    read(serialPort, ch);
   until ch = '-';
   write(serialPort, 'q');
   repeat
    read(serialPort, ch);
   until ch = '>';
  end;

begin
 open(serialPort, 'modem:');
 ParamBlock.IOCompletion := 0;
 ParamBlock.IORefNum := AOut;
 ParamBlock.IOFileType := SetSCC;
 ParamBlock.IONewType := FastBaud;
 Regs.A0 := @ParamBlock;{A0 pointer to Parameter Block}
 Generic(Control, Regs);{Change the baud rate}
 if Regs.D0 <> 0 then
  writeln('Serial Port B error: ', Regs.D0);{Error codes returned in D0}
 thermometer;
 write(serialPort, chr(4), chr(23), '00', chr(13));
 repeat
  read(serialPort, ch);
 until ch = '>';
 select(4);
 bleck := true;
 raise := false;
 repeat
  write(serialPort, 'f0800');
  for m := 1 to 5 do
   read(serialPort, ch);
  write(serialPort, chr(13));
  read(serialPort, ch);
  read(serialPort, ch);
  m := 0;
  repeat
   m := m + 1;
   read(serialPort, ch);
   arr[m] := ch;
  until (ch = '>') or (ch = '-');
  h := arr[1];
  k := arr[2];
  per := 256 * dec(h, k);
  h := arr[3];
  k := arr[4];
  per := per + dec(h, k);
  h := arr[6];
  k := arr[7];
  mul := 256 * dec(h, k);
  h := arr[8];
  k := arr[9];
  mul := mul + dec(h, k);
  freq := mul / per * 16573;
  last := next;
  next := temp;
```

(continued)

```
  temp := (0.5 * (freq - 9210) + 85);
  mean := (temp * 0.6) + (next * 0.3) + (last * 0.1);
  writeln(mean : 1 : 1, ' Deg. F.');
  if (mean >= 70) and (mean < 90) then
   begin
    raise := true;
   end
  else
   begin
    raise := false;
   end;
  changetemp(raise);
  settemp;
  if change(bleck) then
   begin
    if bleck then
     begin
      raise := false;
     end
    else
     begin
      raise := true;
     end;
    changetemp(raise);
   end;
 until false;
 close(serialPort);
end.
```

9 Selected Scientific Applications

THEME: This chapter contains selected applications which represent the kinds of modeling problems encountered by the practicing scientist.

GOALS: To attain a familiarity with the notion of real-world, practical, applied problems to which Macintosh Pascal may apply.

LIBRARIES USED: DiffEqu.lib and graphics libraries

REFERENCE MATERIALS: Appropriate texts and references from the fields of science, science education, engineering, and signal processing.

Fourier Analysis

We start this chapter of selected applications by investigating some of the programming possibilities associated with Fourier Analysis. This kind of analysis is very important in engineering and applied physics, and it manages to find its way into many other fields. Since the discovery of the Fast Fourier Transform in the 1960's, Fourier Analysis has grown into an important aspect of computer science studies also. The *Fast Fourier Transform* is actually an efficient algorithm for computation of the standard Discrete Transform. This, in turn, is defined as follows. Let a signal be given by:

```
x[0], x[1], x[2], . . .x[N - 1]
```

so that the total number of signal data is N. Then the transform is defined also to have N elements, denoted:

$\hat{x}[0], \hat{x}[1], \ldots \hat{x}[N-1]$

If the x[j] represent real-world data, each taken after a sampling time increment of t, then the frequency (in Hertz) corresponding to the component $\hat{x}[j]$ is given by:

freq[j] = 2 * π * j/(Nt)

The Fourier formula which relates the transform elements x[j] to the signal elements x[k] is as follows:

$$\hat{x}[j] = \frac{1}{\text{sqrt}(N)} \sum_{k=0}^{N-1} x[k] * \exp\left(\frac{2\pi ijk}{N}\right)$$

The inverse Discrete Fourier Transform is given by a simple conjugation of the complex exponential:

$$x[k] = \frac{1}{\text{sqrt}(N)} \sum_{j=0}^{N-1} \hat{x}[j] * \exp\left(\frac{-2\pi\, ijk}{N}\right)$$

These transformations would appear to require on the order of N*N complex multiply operations, since N are required for each $\hat{x}[j]$ in the first summation, and in that same formula k ranges through N frequency values.

But the discovery of the Fast Fourier Transform (FFT) was based on the key observation that many of the factors exp( ) are redundant in a certain sense. In fact, for fixed j, the numbers:

exp(2πijk/N)

are just a particular permutation of the complex N-th roots of unity as k ranges from 0 to N-1. When a different j is chosen and fixed, the permutation as k runs over its range is knowable, and this knowledge can be used to recast the Fourier sum as a sum over the complex roots instead of over, for example, k.

The FFT algorithm is not easy to understand, but once it is programmed, the transformation performed by an FFT program will apply to any signal. Program 9.1, FFT, does the algorithm for the case that N is a power of two.

The program listing shows that 8 data are expected. One happy byproduct of the FFT approach is that the computations can be performed 'in place.' This means that the array of signal data, which for this program is an array of type 'complex,' is continually re-calculated until the transform desired resides in the very places originally taken by the signal. Thus the transform will be the set of N complex pairs:

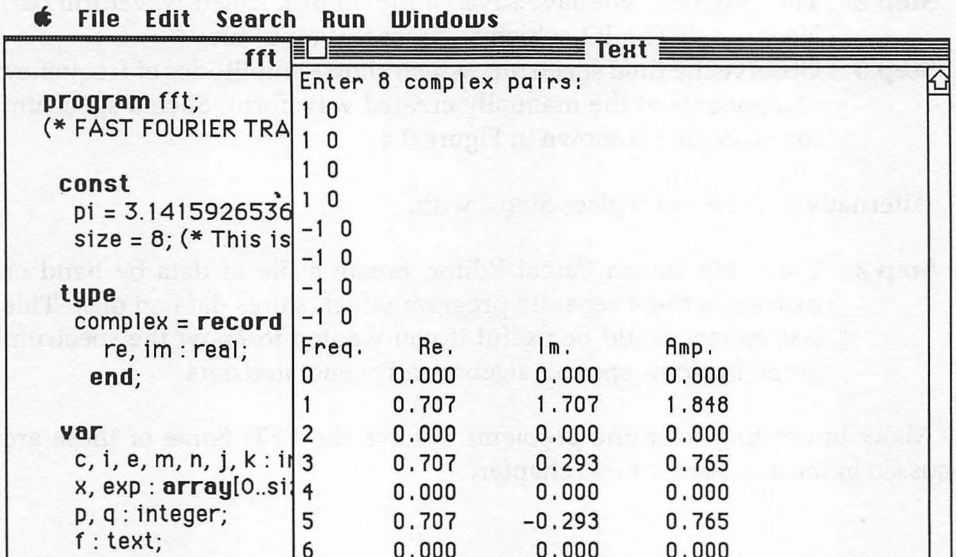
x[j].re, x[j].im

If desired, the amplitude of the j-th Fourier component is the quadrature sum of these: namely, the square root of the sum of their squares.

In Figure 9.1, the user has input the squarewave data:

```
1 0
1 0
1 0
1 0
-1 0
-1 0
-1 0
-1 0
```

Note that the zeros correspond to the notion that the input data is pure real—with vanishing imaginary parts. The output of the FFT program shows the familiar



```
program fft;
(* FAST FOURIER TRA

  const
    pi = 3.1415926536
    size = 8; (* This is

  type
    complex = record
      re, im : real;
    end;

  var
    c, i, e, m, n, j, k : i
    x, exp : array[0..si
    p, q : integer;
    f : text;
```

```
Enter 8 complex pairs:
1 0
1 0
1 0
1 0
-1 0
-1 0
-1 0
-1 0
Freq.   Re.      Im.      Amp.
0       0.000    0.000    0.000
1       0.707    1.707    1.848
2       0.000    0.000    0.000
3       0.707    0.293    0.765
4       0.000    0.000    0.000
5       0.707   -0.293    0.765
6       0.000    0.000    0.000
7       0.707   -1.707    1.848
```

Figure 9.1 | FFT output—squarewave data input decay of squarewave harmonics that we saw in Chapter 4, program SquareWave: the j indices 1, 3 have decreasing spectral amplitudes, in that order.

The inverse transform is obtained by simply entering conjugates, meaning you reverse the sign of the imaginary part. Figure 9.2 shows what happens when you input the complex conjugates of the output from the original run; namely, you recover the original squarewave data.

Once you have a working FFT, there are options for enhancement. One option is to extend the algorithm to handle other data sizes N than powers of two. This can be done either by changing the so-called 'decimation' algorithm (which is the loop containing the call to procedure inplace) or by extending the data size to the next power of two by appending zeros to the signal. Both methods are discussed in most signal processing texts.

Another enhancement option is embodied in Program 9.2, FFTfileIO. This program allows input of data not from keyboard but from disk. The filename assumed is waveform.dat. Program 9.3, Wavemaker, lets the user input a waveform graphically.

The mouse is to be pressed and slid across the data graph, as shown in Figure 9.3.

Figure 9.4 shows the result of the following steps:

Step 1 Run program WaveMaker to create a waveform on a finite interval.
Step 2 This program will have saved a file on disk called waveform.dat. Now run FFTfileIO, which assumes this input file.
Step 3 Observe the final spectrum, which shows amplitudes of frequency components of the manually created waveform. Such a spectrum for Figure 9.3 is shown in Figure 9.4.

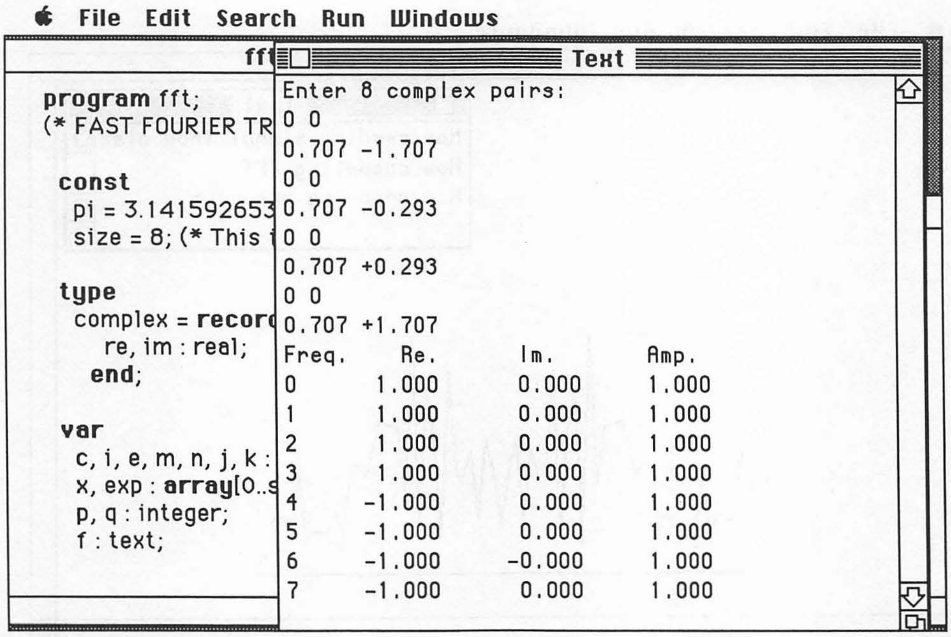
Alternatively, you can replace Step 2 with:

Step 2 Using Macintosh Pascal Editor, create a file of data by hand or instead, write a separate program which stores data on disk. This last option would be useful if you wanted to know the spectrum graph for some special, algebraically generated data.

Many interesting scientific problems involve the FFT. Some of these are discussed in the exercises for this chapter.

Advanced Mathematical Methods

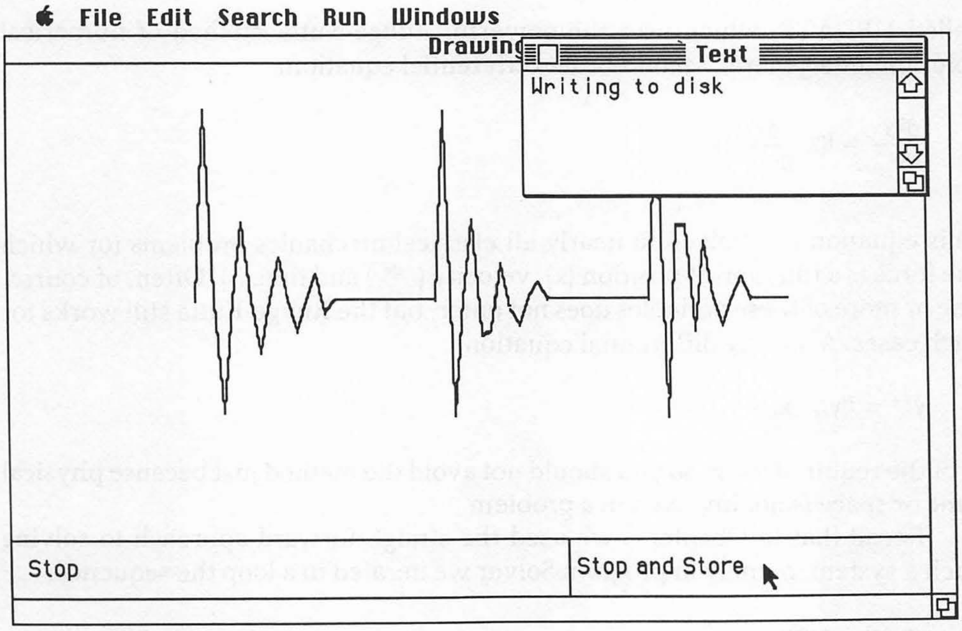
Nearly all of the concepts and programs discussed in previous chapters can be approached in more sophisticated ways. One example is that of advanced techniques in solving differential equations. The DiffEqu.lib contains a procedure



```
program fft;
(* FAST FOURIER TR

const
 pi = 3.141592653
 size = 8; (* This i

type
 complex = record
    re, im : real;
  end;

var
 c, i, e, m, n, j, k :
 x, exp : array[0..s
 p, q : integer;
 f : text;
```

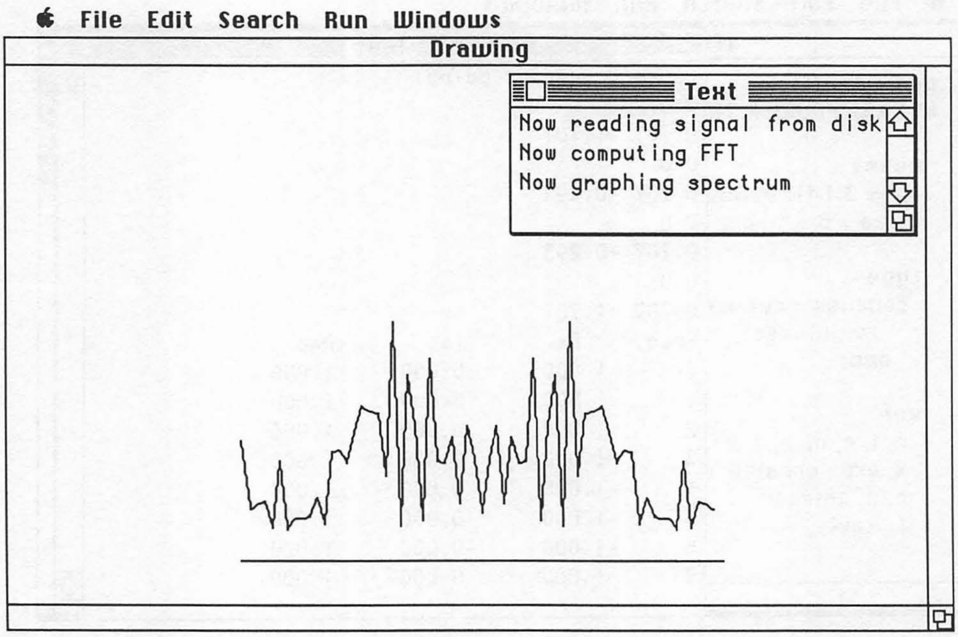
```
Enter 8 complex pairs:
0 0
0.707 -1.707
0 0
0.707 -0.293
0 0
0.707 +0.293
0 0
0.707 +1.707
Freq.       Re.        Im.       Amp.
0          1.000      0.000     1.000
1          1.000      0.000     1.000
2          1.000      0.000     1.000
3          1.000      0.000     1.000
4         -1.000      0.000     1.000
5         -1.000      0.000     1.000
6         -1.000     -0.000     1.000
7         -1.000      0.000     1.000
```

Figure 9.2 | FFT output—inverse transform



```
Writing to disk
```

Figure 9.3 | WaveMaker Output



```
Now reading signal from disk
Now computing FFT
Now graphing spectrum
```

Figure 9.4 | FFF FileIO Output

called UPDATE, which uses the powerful Runge-Kutta method of numerical solution for a general, second-order differential equation:

$$\frac{d^2x}{dt^2} = f(x, \frac{dx}{dt}, t)$$

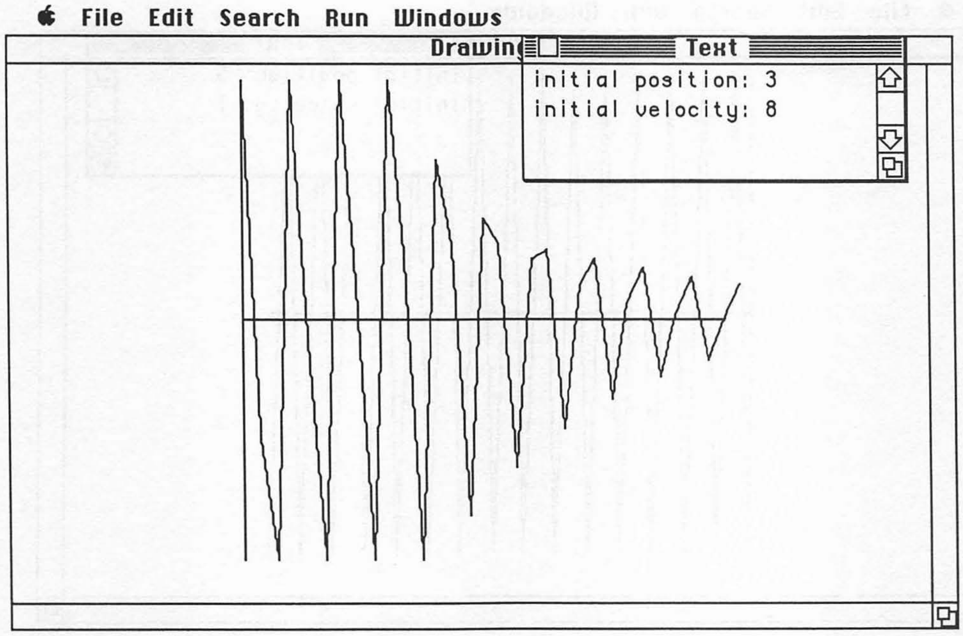
This equation is involved in nearly all classical mechanics problems for which the force is a function of position (x), velocity ($\frac{dx}{dt}$) and time (t). Often, of course, one or more of these variables does not enter, but the Runge-Kutta still works for such cases. Also, any differential equation:

$$y'' = f(y, y', x)$$

is of the required form, so you should not avoid the method just because physical time or space is not involved in a problem.

Recall that in Chapter 3 we used the straightforward approach to solving such a system, namely in program Solver we iterated in a loop the sequence:

```
v:= v + a*dt;
x:= x + v*dt;
t:= t+dt;
```



```
initial position: 3
initial velocity: 8
```

Figure 9.5 | RKoscillator output—note damping

The Runge-Kutta method uses more knowledge of the trajectory to predict, in a sharper way, the future evolution. Specifically, the Runge-Kutta algorithm used in procedure UPDATE involves several different time-steps, not just dt. The temporary variables kl, k2, k3, k4 represent fine adjustments to the calculus approximation, and while the UPDATE procedure is slower than the straightforward approach, you may use a larger dt for the same accuracy; many problems turn out to go faster overall when this more complicated iteration is used.

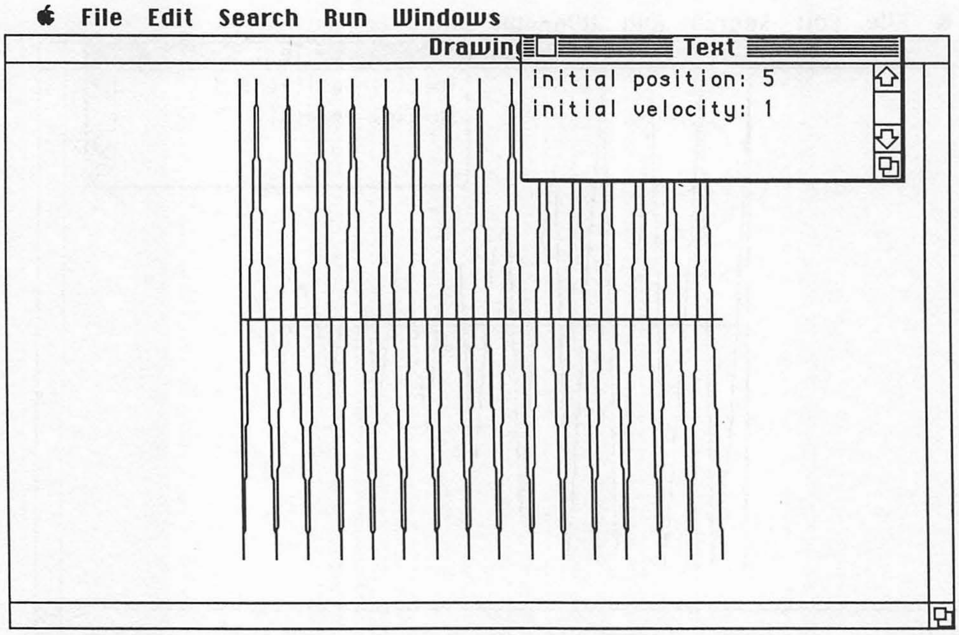
The UPDATE procedure is to be used as shown in Program 9.4, RKoscillator. The idea is that a call such as:

```
UPDATE(pos,vel,t,dt);
```

will update the first three variables (thought of for this problem as position, velocity, and time, respectively).

Figure 9.5 shows the damped oscillator output which is remarkably accurate considering the large value (0.2) ascribed to dt. Recall that in program Oscillator, Chapter 3, we used dt = 0.03.

Program 9.5, Comparison, shows the original oscillator program modified to be on equal footing (i.e., have the same dt = 0.2). Output Figure 9.6 shows clearly that this dt fails completely for the simple approach: the mass is not at all



```
initial position: 5
initial velocity: 1
```

Figure 9.6 | Comparison output—note absence of damping

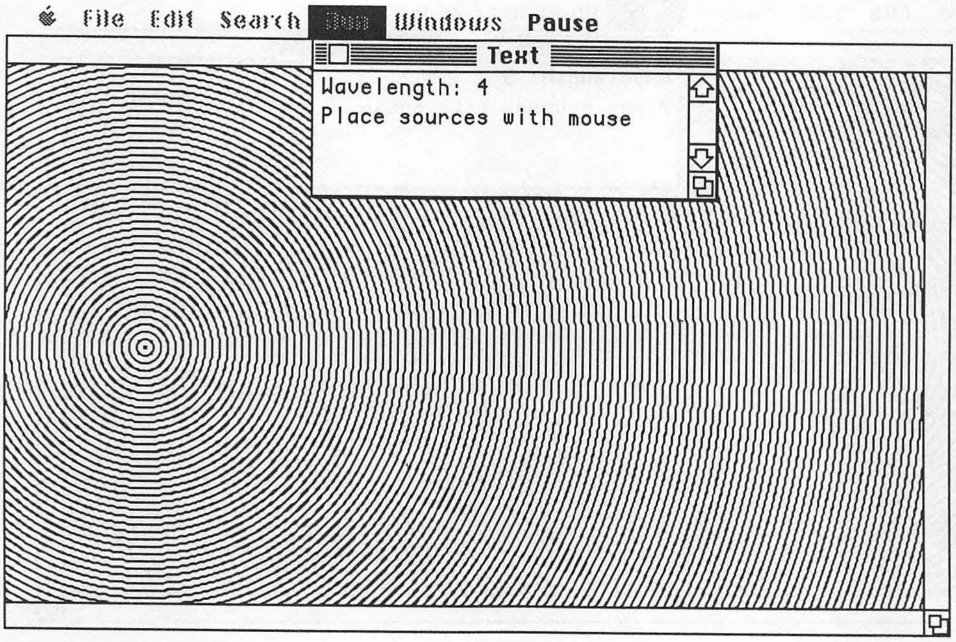
damped. Typical initial data in program Comparison will even cause a floating point blow-up.

Runge-Kutta methods are particularly good for orbital problems and problems for which Macintosh Pascal appears to be too slow for the required accuracy or stability. But the most important feature of the Runge-Kutta method for the present treatment is that it exemplifies a case in which simplicity in programming technique is sacrificed in favor of higher overall performance.

Classroom Programs

By classroom programs we mean projects intended to convey physical concepts—projects from which you may learn the important phenomena of the physical world in new ways. The example selected for this chapter involves diffraction: the interference of waves. The program is very simple, but this fact should not detract from the educational potential of such a problem. As is mentioned in the exercises, there are many ways to make the problem more complex and richer if you wish to do so.

You may be familiar with the standard classroom apparatus known as a *ripple tank* in which small probes (usually two) disturb a water surface, causing circular waves to emanate from the two source points. You can see bands emanating from the general region of the sources (usually close together). A



```
Wavelength: 4
Place sources with mouse
```

Figure 9.7 | Diffraction output

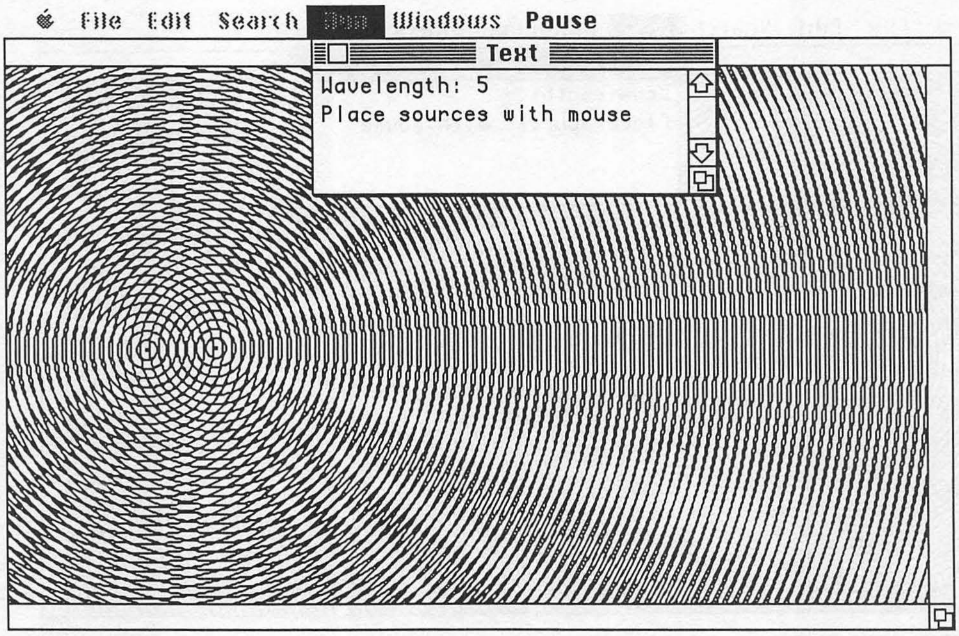
Pascal program, taking advantage of the fine resolution of the Macintosh screen, can be written to model this water phenomenon, which also applies in its basics to light waves.

Program 9.6, Diffraction, lets the user click the mouse at the position of a new source.

Figures 9.7 and 9.8 show typical output from two wave sources. This program is a good example of a programming task which, in its simplest rendition, conveys one basic scientific fact. Observe the famous diffraction phenomenon evident from these figures: the separation of the bands varies inversely with the wavelength; the larger wavelength has smaller separation of the radial bands.

Testing of Theories

Computers can be used to test theories, whether they be from the realm of pure mathematics or from applied science. Witness the recent computer assault on the Four-Color Theorem, which had lain unresolved in many aspects until a large program proved some especially difficult cases. On the applied side of the spectrum, computers have been brought to bear on the formidable computa-



```
Wavelength: 5
Place sources with mouse
```

Figure 9.8 | Diffraction output

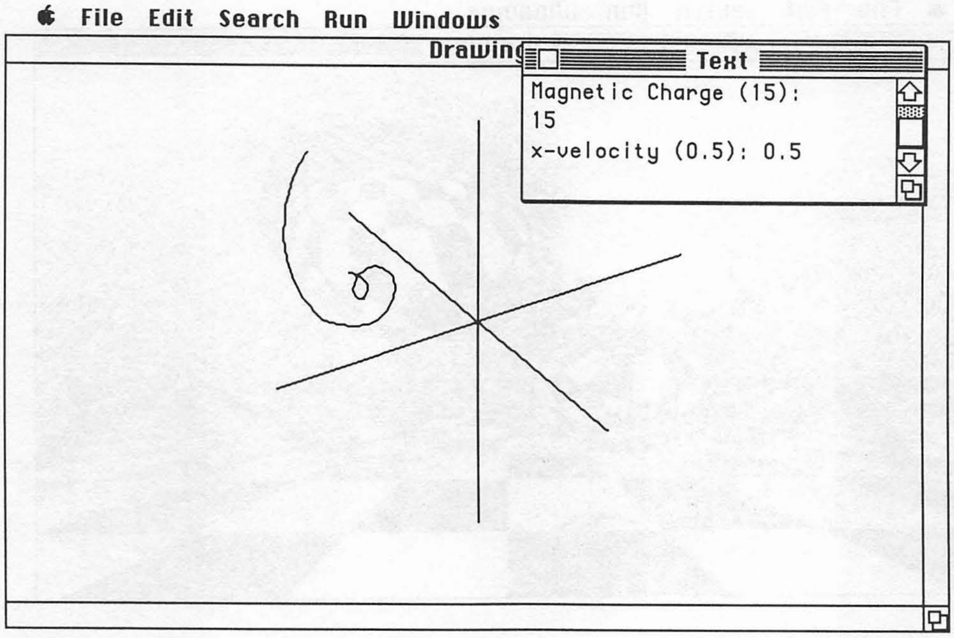
tional problems of physics and chemistry, for example, plasma dynamics and atomic physics.

We have selected a relatively simple but instructive example of a program which tests a theory. The example involves a *magnetic monople.* Such particles may exist but have not yet been found by experimenters. They would have a unit magnetic 'bare' charge—without the usual second pole locatable in nature. The field of a magnetic monopole would be radial, given by:

$$\vec{B} = \frac{g\vec{r}}{r^3}$$

that is, point in the r-direction and have inverse-square magnitude. Now there are several interesting hypotheses which, although they have been proved in many separate ways, are nevertheless difficult to visualize. The proofs themselves are somewhat difficult to grasp without considerable education in theoretical methods of physics. But a program with appropriate graphics verifies (does not prove, but supports) the statements. Consider a charged particle, say an electron, moving near a magnetic monopole. What is claimed is that:

1. Certain orbits will spiral in and then 'reflect back' from the monopole.
2. The actual speed of the orbiting electron never changes, even though the motion is complicated, especially so near the monopole.



```
Magnetic Charge (15):
15
x-velocity (0.5): 0.5
```

Figure 9.9 | Monopole output

3. The actual path taken lies completely on the surface of some cone, wrapped just like a string which is drawn tight around a conical surface.

Program 9.7, Monopole, and associated graphical output Figure 9.9, shows typical motion. Point one is easily verified, and point three is not hard to visualize. This visualization and a verification of point 2 are discussed in the exercises at the end of this chapter.

Deep Calculation

We close this chapter with an example mentioned in Chapter 1. There are times when you simply live with the slowness of execution of a language, turning to other matters, for example, as a graphics screen is filling up slowly. But slowness is relative, and the example discussed here has counterparts in the field of high-speed, large computers.

There are by now famous and spectacular synthetic images which have come out of advanced computer and video technology. You can find, for example, 'bowls of ornaments' or 'collections of wooden implements' displayed on color video monitors which, it is reported, took such-and-such a computer (for example a stand-alone Cray or VAX 11/780) several hours to complete. The following is a program whose output is several orders of magnitude less complex

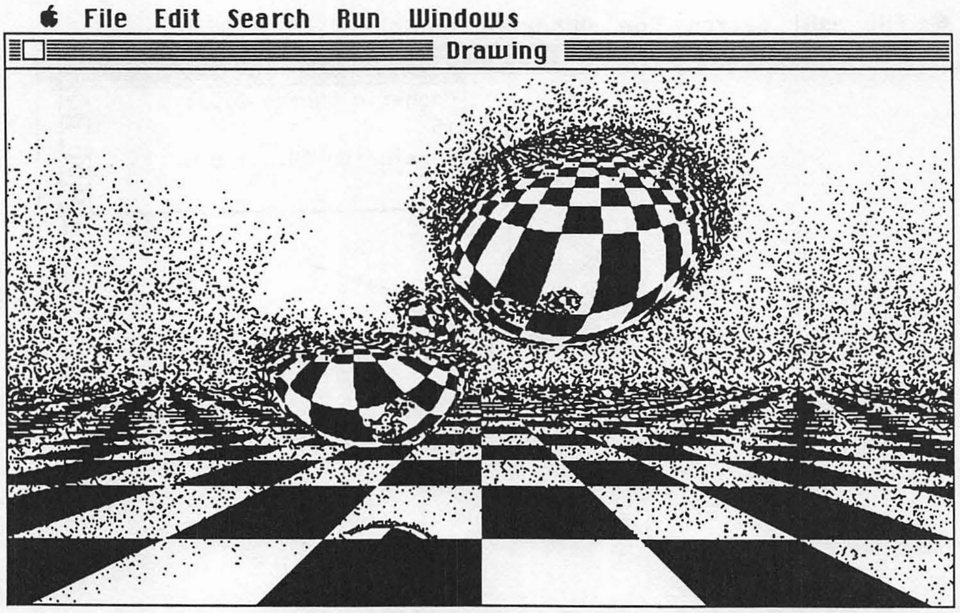


Figure 9.10 | Scene output

than those, and which takes Macintosh Pascal a matter of hours to complete, but which is nevertheless compelling.

Program 9.8, Scene, computes by tedious ray-tracing (again, not really tedious to the user who, having run this program, only need return at a later time on the clock) the visual scene of a reflecting object floating above a checkered floor.

The output, Figure 9.10, shows the beauty and accuracy of the ray-tracing approach, and serves as a fine example of the limits to which Macintosh Pascal can be put.

Exercises

1. Make alterations to the program FFT so that the amplitude (last "writeln-ed" quantity in the program listing) is graphed against an horizontal axis (frequency) which should be given small tick marks because the frequency values go from 0 through size – 1 in discrete jumps.

   Try out your graphic method on a signal that is a forced sum of sine waves. For example, choose size = 128 (the FFT will take a little while for this many terms) and create a signal for n: = 0 to size – 1 as:

   ```
   x[n].re: = sin(n/3) + sin(n/2); x[n].im: =0;
   ```

The graphed spectrum should show the two peaks corresponding to the frequencies $\frac{size}{(4\pi)}$ and $\frac{size}{(6\pi)}$, but are there any more peaks? Remember that there is a theorem in Fourier Analysis which says that if the input data is pure real, then the spectrum is symmetrical about the center frequency.

2. Show with graphics techniques that the Fourier transform of a noise signal is also noisy. Specifically, start with a signal:

   ```
   x[n].re := GAUSS(0,1);      x[n].im:= 0;
   ```

   for n = 0 to size-1, size = power of two, which will be what is called 'white noise'—each datum is a Gaussian distributed random selection, here with mean 0 and standard deviation 1—and plot it and its Fourier amplitude spectrum. Then show that 'pink noise' has a spectrum peaked at low frequencies. You can make pink noise by adding to each datum a piece of the last few. This is a good way to force some 'correlation' between one noise datum and the last. After you have filled an array with white noise, you can track through n and do:

   ```
   x[n]:= x[n] + x[n-1]/2 + x[n-2]/4;
   ```

   or some such correlation. The FFT spectrum will then show the 'pinkness' that you expect when the noise data are correlated in this way.

3. Find some recorded data which expresses a natural phenomenon, such as the height of tide vs. time of day for several days, or the temperature in a city vs. time of day, or a medical parameter of an organism vs. time of day. Enter this data into a text file on your disk and run a program such as FFTFileIO to get the frequency spectrum. The spectrum often reveals something interesting. For the given example, the things you should be able to find with Fourier technique are, respectively: the approximate period of rotation of the moon around the Earth, the fact that 24 hours is the length of a day, the circadian rhythm (24-hour periodicity) of most (larger) organisms.

4. There is a general result that the derivative (with respect to time) of a time series has a spectrum related to the original spectrum as follows: if x[n] is the n-th transform component of the original signal, and y[n] is the n-th transform component of the differentiated signal, then the amplitude of y[n] is approximately a constant times n times the amplitude of x[n]. Test this hypothesis out on some sample signals.

5. Try to 'manufacture' some digital filters, using the following basic principles. If you have a signal x, then the new signal z created by the recursion:

   ```
   z[0]:= x[0];
   z[n]:= x[n] + z[n-1]/2;     (* for n:= 1 to size-1 *)
   ```

will be, amazingly enough, a low-passed version of the x signal. This means that although z looks something like x, low frequency components of x survive preferentially in z. This is similar to the effect of a tone control on an audio system which performs treble cut (removes high frequencies). What is intriguing about digital filters is that they are so easy to program—usually a simple loop will transform the orginal signal as required.

Test the low-pass phenomenon out on some sinusoidal signal data: graph the original sine wave and the low-passed version, for various frequencies, to see that for higher frequencies (up to a point—you cannot go too near to the frequency $\frac{size}{(2 \cdot 2 \cdot \pi)}$) there is attenuation when z is graphed.

You can try FFT techniques to see if the spectrum of more complex x, z signals are indeed digitally filtered.

There are many kinds of digital filters. Usually the recurrence relation is one of a general class given by the form:

```
z[n]:= x[n]/a0 + x[n-1]/a1 + . . .
   + z[n-1]/b1 + z[n-2]/b2 + . . .
```

That is, the new datum z[n] is correlated with various data from both x and z in the past. With some work you can discover a bandpass filter which allows one frequency through preferentially, and so on.

6. With some differential equation techniques, solve the SAM problem of N. Tufillaro. SAM stands for 'Swinging Atwood Machine': one mass m1, which will always move straight up and down, hangs from a pulley; but the pulley cord extends horizontally over to a second pulley from which a mass m2 hangs but is allowed to 'swing' (combining up and down with left and right motion). Graph the trajectories of the left-hand mass. What do you think (intuitively) happens if you start m2 at some initial angle $\neq 0$ (with respect to the vertical) and you let it go, but m2 < m1? It would seem like m2 would crash into its (point-sized) pulley. But this is not so! Any ratio $\frac{m1}{m2} > 1$ gives rise to bounded, 'forever' motion of the swinging mass.

7. With a program find the escape velocity of the Earth. You start with an initial velocity V at the Earth's surface and solve the differential equation:

$$\frac{d^2r}{dt^2} = - \frac{GM}{sqr(r)}$$

finding out, the best way you can within the program, when the initial velocity is just enough to eventually break away from the Earth.

8. For the magnetic monopole project of Chapter 9, show numerically the theoretical prediction that the speed (absolute value of velocity) does not change during the motion. If it does, there must be an error in the differential

equation solver. As an extra option, try to invoke a Runge-Kutta solver for the special equations of motion for the monopole problem.

9. Work out a way to verify numerically the prediction that the electron in the monopole field does, in fact, move on the surface of some cone. Hint: what is true of all the vectors which lie on a conical surface?

10. Using Macintosh Quickdraw it is possible to create some really satisfying visual effects. Try writing a program which enables you, by some mouse means of your invention, to 'sail through space' in different directions, watching globes fly by as you do so. You can use the principles of 3-D graphics and perspective of Chapter 6, and the facts about screen and mouse from Chapter 2, to draw, from your vantage point (x,y,z) in 3-space, the appearance of globes at any time. You can animate either by full screen blanking between overall re-drawings, or by blanking then re-drawing each globe in succession. The former is easier to program, since the latter requires you do the closest globes last; but the latter looks better to some 'navigators'.

Answers

1. It is a good idea to compute, prior to graphing, the maximum amplitude over the 0. . . size – 1 Fourier components. Call this *ampmax*. Then use shifts and draws to the points:

   $$x, \frac{\text{amplitude[j]}}{\text{ampmax}}$$

   for a spectral plot which will now fit nicely on the screen. There should be four major peaks in your graph.

2. The noisy spectrum, according to theory, is also Gaussian. There is no easy way to tell this, except for the qualitative observation that the spectrum looks 'something like' the original signal. One exception is that for real signal data, the spectrum is symmetrical about the center frequency. For 'pink' noise, the spectrum will be again very noisy in the qualitative sense, but should have noticeably greater amplitude for lower frequencies.

3. This is an exploratory problem whose results depend on the nature of the input data. The observations with respect to peaks in the spectrum must usually be qualitative.

4. The canonical example of this is the squarewave-triangle wave system, for which the former is the derivative of the latter.

5. The digital filter systems of the recursive type indicated are easy to solve theoretically if the input data x[n] is periodic. In fact, if:

   ```
   x[n] = exp(iWn)
   ```

   then the trial solution:

   ```
   z[n] = A(n) exp(iWn)
   ```

   results, when substitued into the recursion relation for the filter, in an algebraic equation for A(n). This factor A(n) will, of course, be the filter function for the filter in question. On balance, this theoretical calculation is the best way to approach the problem, especially if a particular filter is desired. You can try different coefficients and graph the function A(n) until it has the desired shape. Then the actual digital filtering will have the correct behavior.

6. The 'magic ratios' of $\frac{m1}{m2}$ are interesting. A special case is $\frac{m1}{m2} = 3$, for which every trajectory of mass m2 which starts right at its pulley, and for any initial angle and velocity $\neq 0$, forms a perfect 'teardrop' shape. For other ratios, you can also get 'smiles' and 'figure eights.'

7. Start with a trial v and r = radius of Earth, then iterate:

   ```
   v:= v - GM/sqr(r) * dt;
   r:= r + v * dt;
   t:= t + dt;
   ```

   Good values for dt should be in the 100-second region if you use book values (in M.K.S. system of units) for M, r, and G.

   The problem of finding the initial v which barely allows escape is not easy, but with a little work, you can show that the escape velocity is of the order of 11000 meters per second.

8. The reason why speed is constant over the whole trajectory is that the magnetic force q v X B is always perpendicular to the velocity vector v. If you define a dot product function, then:

   ```
   speed = sqrt(dot(v,v))
   ```

   It is interesting to try to get more precise trajectories by continually normalizing the velocity vector so that the speed is forced to always be equal to the initial speed. This expedient—of using a known physical theorem to normalize a solution to a differential equation—often enhances accuracy significantly.

9. The condition that a vector v lie on a certain cone is that for an appropriate vector u:

   $$\frac{\mathrm{dot(u,v)}}{\mathrm{sqrt(dot(v,v))}}$$

   is a constant.

10. The remarks relating to Exercise 5, Chapter 6 are relevant for this problem. It is probably best always to keep track of all globes at all times. This can be done in three arrays of x[i], y[i], z[i]; i = 1. .N, or in a matrix of dimensions 3 by N.

Program 9.1

```
program fft;
(* FAST FOURIER TRANSFORM PROGRAM *)

 const
  pi = 3.1415926536;
  size = 8; (* This is data sample size - must be power of two *)

 type
  complex = record
    re, im : real;
   end;

 var
  c, i, e, m, n, j, k : integer;
  x, exp : array[0..size] of complex;
  p, q : integer;
  f : text;

 procedure inplace (var g, h : complex;
         f : complex);
(* Performs in-place computation for data pair (g,h) and exponential*)
(* multiplier f *)
  var
   tmp : real;
 begin
  g.re := g.re + (f.re * h.re - f.im * h.im);
  g.im := g.im + (f.re * h.im + f.im * h.re);
  tmp := g.re - 2 * (f.re * h.re - f.im * h.im);
  h.im := g.im - 2 * (f.re * h.im + f.im * h.re);
  h.re := tmp;
 end;

begin
 showtext;
 writeln('Enter ', size : 1, ' complex pairs: ');
 c := 1;
 n := round(ln(size) / ln(2)); (* i.e., size = 2^n *)
 for j := 0 to size - 1 do
  begin
(* Next, fix sin and cos array elements for maximum speed later *)
   exp[j].re := cos(2 * pi * j / size);
   exp[j].im := sin(2 * pi * j / size);
(* Next, scramble the input order with reverse-complement-binary *)
   e := size div 2;
   i := j;
   k := 0;
   m := 1;
```

```
    p := i div e;
    i := i mod e;
    e := e div 2;
    k := k + m * p;
    m := m + m;
   until e = 0;
(* Next, get the actual signal data as re, im pair *)
   readln(x[k].re, x[k].im);
  end;
 e := size;
(* Next, use decimation-in-frequency FFT algorithm *)
(* j will be the count of inplace full-vector iterations *)
 for j := 0 to n - 1 do
  begin
   e := e div 2;
   for k := 0 to c - 1 do
    begin
     for i := 0 to e - 1 do
      begin
       p := k + c * (2 * i);
       q := p + c;
       m := (p * e) mod size;
   (* Next, process the (p-th,q-th) 'butterfly' *)
       inplace(x[p], x[q], exp[m]);
      end;
    end;
   c := c + c;
  end;
 writeln('Freq.    Re.       Im.        Amp.');
(* Next, output the re,im parts and amplitudes of Fourier Transform *)
 for j := 0 to size - 1 do
  begin
   x[j].re := x[j].re / sqrt(size);
   x[j].im := x[j].im / sqrt(size);
   write(j : 1, ' ', x[j].re : 10 : 3, ' ', x[j].im : 10 : 3);
   writeln(sqrt(sqr(x[j].re) + sqr(x[j].im)) : 10 : 3);
  end;
end.
```

Program 9.2

```
 program fftfileIO;
 (* FFT PROGRAM WITH FILE INPUT AND SPECTRUM GRAPH OUTPUT *)

  cons
   pi = 3.1415926536;
   size = 64; (* This is data sample size - must be a power of two *)
```

(continued)

```
type
 complex = record
   re, im : real;
  end;

var
 c, i, e, m, n, j, k : integer;
 x, exp : array[0..size] of complex;
 p, q : integer;
 f : text;
 maxamp : real;

procedure inplace (var g, h : complex;
        f : complex);
(* Performs in-place computation for data pair (g,h) and exponential*)
(* multiplier f *)
 var
  tmp : real;
begin
 g.re := g.re + (f.re * h.re - f.im * h.im);
 g.im := g.im + (f.re * h.im + f.im * h.re);
 tmp := g.re - 2 * (f.re * h.re - f.im * h.im);
 h.im := g.im - 2 * (f.re * h.im + f.im * h.re);
 h.re := tmp;
end;

procedure CLEAR;
 var
  windowrect : rect;
begin
 hideall;
 setrect(windowrect, 2, 35, 512, 342);
 setdrawingrect(windowrect);
 showdrawing;
end;

procedure MAP (x, y : real;
        var hor, ver : integer);
begin
 if abs(x) > 1.9 then
  x := x / abs(x) * 1.9;
 hor := 255 + trunc(x * 130);
 if abs(y) > 1 then
  y := y / abs(y);
 ver := 138 - trunc(y * 130);
end;

procedure DRAW (x, y : real);
 var
  h1, v1 : integer;
begin
```

```
 MAP(x, y, h1, v1);
 lineto(h1, v1);
end; {draw}

procedure SHIFT (x, y : real);
 var
  h1, v1 : integer;
begin
  MAP(x, y, h1, v1);
  moveto(h1, v1);
 end; {shift}


begin
 CLEAR;
 reset(f, 'waveform.dat');
 showtext;
 writeln('Now reading signal from disk');
 c := 1;
 n := round(ln(size) / ln(2)); (* i.e., size = 2^n *)
 for j := 0 to size - 1 do
  begin
(* Next, fix sin and cos array elements for maximum speed later *)
   exp[j].re := cos(2 * pi * j / size);
   exp[j].im := sin(2 * pi * j / size);
(* Next, scramble the input order with reverse-complement-binary *)
   e := size div 2;
   i := j;
   k := 0;
   m := 1;
   repeat
    p := i div e;
    i := i mod e;
    e := e div 2;
    k := k + m * p;
    m := m + m;
   until e = 0;
(* Next, get the actual signal data as re, im pair *)
   readln(f, x[k].re, x[k].im);
  end;
 close(f);
 writeln('Now computing FFT');
 e := size;
(* Next, use decimation-in-frequency FFT algorithm *)
(* j will be the count of inplace full-vector iterations *)
 for j := 0 to n - 1 do
  begin
   e := e div 2;
   for k := 0 to c - 1 do
    begin
     for i := 0 to e - 1 do
```

(continued)

```
      begin
       p := k + c * (2 * i);
       q := p + c;
       m := (p * e) mod size;
  (* Next, process the (p-th,q-th) 'butterfly' *)
       inplace(x[p], x[q], exp[m]);
      end;
    end;
   c := c + c;
  end;
 maxamp := 0;
(* Next, find the maximum unnormalized amplitude *)
 for j := 0 to size - 1 do
  begin
   x[j].re := sqrt(sqr(x[j].re) + sqr(x[j].im));
   if x[j].re > maxamp then
    maxamp := x[j].re;
  end;
 SHIFT(-1, -1);
 DRAW(1, -1);
 SHIFT(-1, x[0].re / maxamp - 1);
 writeln('Now graphing spectrum');
 for j := 0 to size - 1 do
  begin
   DRAW(-1 + 2 * j / size, x[j].re / maxamp - 1);
  end;
end.
```

Program 9.3

```
program wavemaker;
(* USED TO DRAW WAVEFORMS FOR ANALYSIS BY OTHER PROGRAMS *)

 const
  bins = 64;     (* Make compatible with FFT data size *)
  pixels = 5;

 var
  y : array[0..bins] of integer;
  x, xx, yy, n : integer;
  f : text;

 procedure CLEAR;
  var
   windowrect : rect;
 begin
  hideall;
```

```
    setrect(windowrect, 2, 35, 512, 342);
    setdrawingrect(windowrect);
    showdrawing;
  end;

begin
  CLEAR;
  for n := 0 to bins - 1 do
   y[n] := 128;
  moveto(100, 128);
  lineto(100 + pixels * 64, 128);
  moveto(0, 260);
  lineto(500, 260);
  moveto(8, 280);
  writedraw('Stop');
  moveto(300, 350);
  lineto(300, 260);
  moveto(304, 280);
  writedraw('Stop and Store');
  pensize(1, 2);
  repeat
   getmouse(x, yy);
   if (button) and (yy < 258) and (yy > 2) then
    begin
     n := round((x - 100) / pixels);
     if (n > 0) and (n < bins - 1) then
      begin
       penpat(white);
       xx := 100 + pixels * (n - 1);
       moveto(xx, y[n - 1]);
       lineto(xx + pixels, y[n]);
       lineto(xx + 2 * pixels, y[n + 1]);
       penpat(black);
       y[n] := yy;
       lineto(xx + pixels, y[n]);
       lineto(xx, y[n - 1]);
      end;
    end;
  until (yy > 260) and (button);
  if (x > 300) then
   begin
    showtext;
    rewrite(f, 'waveform.dat');
    writeln('Writing to disk');
    for n := 0 to bins - 1 do
     writeln(f, y[n] - 128 : 1, '    0');
    close(f);
   end;
end.
```

Program 9.4

```
program RKoscillator;
(* RUNGE-KUTTA METHOD FOR DAMPED OSCILLATOR*)
(* POOR dt = 0.2 INTENTIONALLY CHOSEN *)
(* TO DEMONSTRATE STABILITY *)

 const
  m = 1;
  k = 100;
  b = 1;

 var
  pos, vel : real;
  t : real;

 procedure UPDATE (var x, dxdt, t : real;
         dt : real);
  var
   k1, k2, k3, k4 : real;

  function f (x, dxdt, t : real) : real;
  begin
   f := -k * x / m - 2 * b * dxdt / m;
  end;

 begin
  k1 := dt * f(x, dxdt, t);
  k2 := dt * f(x + dt * (dxdt + k1 / 4) / 2, dxdt + k1 / 2, t + dt / 2);
  k3 := dt * f(x + dt * (dxdt + k2 / 4) / 2, dxdt + k2 / 2, t + dt / 2);
  k4 := dt * f(x + dt * (dxdt + k3 / 2), dxdt + k3, t + dt);
  x := x + dt * (dxdt + (k1 + k2 + k3) / 6);
  dxdt := dxdt + (k1 + 2 * k2 + 2 * k3 + k4) / 6;
  t := t + dt;
 end;

 procedure CLEAR;
(* Activates and expands Drawing Window to fill screen *)
  var
   windowrect : rect;
 begin
  hideall;
  setrect(windowrect, 2, 35, 512, 342);
  setdrawingrect(windowrect);
  showdrawing;
 end;

 procedure MAP (x, y : real;
         var hor, ver : integer);
```

```
(* Returns hor,ver as pixel integers for given real coords (x,y) *)
 begin
  if abs(x) > 1.9 then
   x := x / abs(x) * 1.9;
  hor := 255 + trunc(x * 130);
  if abs(y) > 1 then
   y := y / abs(y);
  ver := 138 - trunc(y * 130);
 end;

 procedure DRAW (x, y : real);
(* Draws to (x,y) *)
  var
   h1, v1 : integer;
 begin
  MAP(x, y, h1, v1);
  lineto(h1, v1);
 end;
 procedure SHIFT (x, y : real);
(* moves invisible to (x,y) *)
  var
   h1, v1 : integer;
 begin
  MAP(x, y, h1, v1);
  moveto(h1, v1);
 end;


begin
 CLEAR;
 SHIFT(-1, 1);
 DRAW(-1, -1);
 SHIFT(-1, 0);
 DRAW(1, 0);
 showtext;
 write('initial position: ');
 readln(pos);
 write('initial velocity: ');
 readln(vel);
 t := 0;
 SHIFT(-1, pos);
 repeat
  UPDATE(pos, vel, t, 0.2);
  DRAW(t / 3 - 1, pos);
 until t > 6;  (* Or some appropriate condition *)
end.
```

Program 9.5

```
program comparison;
(* MODIFED DAMPED OSCILLATOR, FOR COMPARISON WITH*)
(* RUNGE-KUTTA METHOD HAVING SAME dt = 0.2 *)

 const
  dt = 0.2; (* this is your small time increment *)
  m = 1;
  k = 100;
  b = 1;
 var
  pos, vel, acc : real;
  t : real;

 procedure CLEAR;
(* Activates and expands Drawing Window to fill screen *)
  var
   windowrect : rect;
 begin
  hideall;
  setrect(windowrect, 2, 35, 512, 342);
  setdrawingrect(windowrect);
  showdrawing;
 end;

 procedure MAP (x, y : real;
         var hor, ver : integer);
(* Returns hor,ver as pixel integers for given real coords (x,y)
 begin
  if abs(x) > 1.9 then
   x := x / abs(x) * 1.9;
  hor := 255 + trunc(x * 130);
  if abs(y) > 1 then
   y := y / abs(y);
  ver := 138 - trunc(y * 130);
 end;

 procedure DRAW (x, y : real);
(* Draws to (x,y) *)
  var
   h1, v1 : integer;
 begin
  MAP(x, y, h1, v1);
  lineto(h1, v1);
 end;

 procedure SHIFT (x, y : real);
(* moves invisible to (x,y) *)
  var
```

```
   h1, v1 : integer;
 begin
  MAP(x, y, h1, v1);
  moveto(h1, v1);
 end;


begin
 CLEAR;
 SHIFT(-1, 1);
 DRAW(-1, -1);
 SHIFT(-1, 0);
 DRAW(1, 0);
 showtext;
 write('initial position: ');
 readln(pos);
 write('initial velocity: ');
 readln(vel);
 t := 0;
 repeat
  acc := -k * pos / m - 2 * b * vel / m;
  vel := vel + acc * dt;
  pos := pos + vel * dt;
  if t = 0 then
   SHIFT(t / 3 - 1, pos)
  else
   DRAW(t / 3 - 1, pos);
  t := t + dt;
 until t > 6;  (* Or some appropriate condition *)
end.
```

Program 9.6

```
program diffraction;
(* WAVE DIFFRACTION MODEL - USER CREATES SPHERICAL*
(* COHERENT SOURCES *)

 var
  x, y, lambda : integer;

 procedure CLEAR;
  var
   windowrect : rect;
 begin
  hideall;
  setrect(windowrect, 2, 35, 512, 342);
  setdrawingrect(windowrect);
  showdrawing;
 end;
```

(continued)

```
procedure dosource (x, y, lambda : integer);
 var
  r : integer;
  a1, a2, a3, a4, a : real;
begin
 r := 1;
 a1 := sqrt(sqr(x) + sqr(y));
 a2 := sqrt(sqr(x) + sqr(300 - y));
 a3 := sqrt(sqr(510 - x) + sqr(y));
 a4 := sqrt(sqr(510 - x) + sqr(300 - y));
 if a1 > a2 then
  a := a1
 else
  a := a2;
 if a3 > a then
  a := a3;
 if a4 > a then
  a := a4;
 repeat
  frameoval(y - r, x - r, y + r, x + r);
  r := r + lambda;
 until r > a;
end;

begin
 CLEAR;
 showtext;
 write('Wavelength: ');
 readln(lambda);
 write('Place sources with mouse');
 repeat
  if button then
   begin
    getmouse(x, y);
    if (y > 2) then
     dosource(x, y, lambda);
   end;
 until false;
end.
```

Program 9.7

```
program monopole;
(* MODELS ORBITAL MOTION OF ELECTRON NEAR *)
(* A MAGNETIC MONOPOLE *)

 const
  dt = 0.02;
  aa = 0;
 bb = 1;
 cc = -1;

type
 vector = array[1..3] of real;

var
 r, v, B, F : vector;
 g : real;

procedure cross (var c : vector;
        a, b : vector);
begin
 c[1] := a[2] * b[3] - a[3] * b[2];
 c[2] := a[3] * b[1] - a[1] * b[3];
 c[3] := a[1] * b[2] - a[2] * b[1];
end;

function norm (v : vector) : real;
begin
 norm := sqrt(sqr(v[1]) + sqr(v[2]) + sqr(v[3]));
end;

procedure compute (var B : vector;
        r : vector);
 var
  n : integer;
  nor : real;
begin
 nor := norm(r);
 nor := nor * sqr(nor);
 for n := 1 to 3 do
  B[n] := g * r[n] / nor;
end;

procedure UPDATE (var x : vector;
        dxdt : vector;
        dt : real);
 var
  n : integer;
begin
 for n := 1 to 3 do
  x[n] := x[n] + dxdt[n] * dt;
end;

procedure CLEAR;
 var
  windowrect : rect;
begin
 hideall;
```

(continued)

```
  setrect(windowrect, 2, 35, 512, 342);
  setdrawingrect(windowrect);
  showdrawing;
 end;

 procedure MAP (x, y : real;
         var hor, ver : integer);
 begin
  if abs(x) > 1.9 then
   x := x / abs(x) * 1.9;
  hor := 255 + trunc(x * 130);
  if abs(y) > 1 then
   y := y / abs(y);
  ver := 138 - trunc(y * 130);
 end;

 procedure DRAW (x, y : real);
  var
   h1, v1 : integer;
 begin
  MAP(x, y, h1, v1);
  lineto(h1, v1);
 end;

 procedure SHIFT (x, y : real);
  var
   h1, v1 : integer;
 begin
  MAP(x, y, h1, v1);
  moveto(h1, v1);
 end;

 procedure ROTATE (var x, y, z : real;
         a, b, c : real);
  var
   sx, sy, sz, tx, ty, tz : real;
 begin
  sx := x * cos(c) - y * sin(c);
  sy := x * sin(c) + y * cos(c);
  sz := z;
  tx := sx;
  ty := sy * cos(b) - sz * sin(b);
  tz := sy * sin(b) + sz * cos(b);
  x := tx * cos(a) - ty * sin(a);
  y := tx * sin(a) + ty * cos(a);
  z := tz;
 end;

 procedure SDRAW (x, y, z, a, b, c : real);
  var
  u, v, w : real;

  begin
   u := x;
   v := y;
   w := z;
   ROTATE(u, v, w, a, b, c);
   DRAW(u, v);
  end;

  procedure SMOVE (x, y, z, a, b, c : real);
   var
    u, v, w : real;
  begin
   u := x;
   v := y;
   w := z;
   ROTATE(u, v, w, a, b, c);
   SHIFT(u, v);
  end;

  procedure AXES (a, b, c : real);
  begin
   SMOVE(-1, 0, 0, a, b, c);
   SDRAW(1, 0, 0, a, b, c);
   SMOVE(0, -1, 0, a, b, c);
   SDRAW(0, 1, 0, a, b, c);
   SMOVE(0, 0, -1, a, b, c);
   SDRAW(0, 0, 1, a, b, c);
  end;

begin
 CLEAR;
 AXES(aa, bb, cc);
 showtext;
 repeat
  write('Magnetic Charge (15): ');
  readln(g);
  write('x-velocity (0.5): ');
  readln(v[1]);
  r[1] := -1;
  r[2] := 0;
  r[3] := 0.3;
  v[2] := 0.5;
  v[3] := -0.2;
  SMOVE(r[1], r[2], r[3], aa, bb, cc);
  repeat
   compute(B, r);
   cross(F, v, B);      (* Lorentz Force is F = q v X B *)
   UPDATE(v, F, dt);
```

(continued)

```
  UPDATE(r, v, dt);
  SDRAW(r[1], r[2], r[3], aa, bb, cc);
 until norm(r) > 1.7;
 until false;
end.
```

Program 9.8

```
program scene;
(* CREATES VISUAL SCENE OF FLOATING MIRRORED SPHERES *)
(* ALLOW 24 HOURS FOR THE RUN *)
 type
  vector = array[1..3] of real;

 var
  v, o, n, r, center1, center2 : vector;
  i, xpixel, ypixel : integer;
  screenx, screeny, shade : real;
  intersection, blackhit : boolean;


 procedure CLEAR;
  var
   windowrect : rect;
 begin
  hideall;
  setrect(windowrect, 2, 35, 512, 342);
  setdrawingrect(windowrect);
  showdrawing;
 end;

 function dot (o, v : vector) : real;
  { dot returns the dot product of o and v }
 begin
  dot := o[1] * v[1] + o[2] * v[2] + o[3] * v[3];
 end; { dot }

 function magnitude (v : vector) : real;
  { returns the length of vector v }
 begin
  magnitude := sqrt(sqr(v[1]) + sqr(v[2]) + sqr(v[3]));
 end; { magnitude }

 procedure reflect (n : vector;
         var v : vector);
  { calculates r, the reflection of v relative to n }
  var
   i : integer;
   length : real;
 begin
  length := dot(v, n);
  for i := 1 to 3 do
   v[i] := v[i] - 2 * n[i] * length;
 end; { reflect }

 function intersectsphere1 (o, v : vector) : boolean;
{ tests whether the ray o+tv intersects sphere with radius 1 centered at
  center1 }
 begin
  o[1] := o[1] - center1[1];
  o[2] := o[2] - center1[2];
  o[3] := o[3] - center1[3];
  if (sqr(magnitude(o)) - sqr(dot(o, v))) < 1 then
   intersectsphere1 := true
  else
   intersectsphere1 := false;
 end;{ intersectsphere1 }

 function intersectsphere2 (o, v : vector) : boolean;
{ tests whether the ray o+tv intersects sphere with radius 1 centered at
  radius2 }
 begin
  o[1] := o[1] - center2[1];
  o[2] := o[2] - center2[2];
  o[3] := o[3] - center2[3];
  if (sqr(magnitude(o)) - sqr(dot(o, v))) < 1 then
   intersectsphere2 := true
  else
   intersectsphere2 := false;
 end;{ intersectsphere2 }

 function intersectplane (v : vector) : boolean;
  { checks for intersection of v with plane }
 begin
  if v[3] < 0 then
   intersectplane := true
  else
   intersectplane := false;
 end; { intersectplane }

 function shadepixel (v : vector) : real;
  { shadepixel chooses a shade between 0 and 1 }
 begin
  if blackhit = true then
   shadepixel := 0
  else
   shadepixel := 1.4 * (v[3] + 1.0) / 2.0;
```

(continued)

```
  { background gets lighter as it gets higher}
  {change this to change how background looks}
 end; { shadepixel }

procedure drawpixel (xpixel, ypixel : integer;
        shade : real);
{ drawpixel colors the pixel white or black }
begin
 if shade < ((random + 32767) / (2 * 32767)) then
  begin
   moveto(xpixel, ypixel);
   lineto(xpixel, ypixel);
  end
end; { drawpixel }

procedure coordinatechange (xpixel, ypixel : integer;
        var screenx, screeny : real);
 { changes screen coordinates to "window" coordinates }
begin
 screenx := (1.0 / 171.0) * xpixel - 512.0 / 342.0;
 screeny := 1.0 - (1.0 / 171.0) * ypixel;
end; { coordinatechange }

procedure initialize (var o, v : vector;
        var blackhit : boolean);
 var
  i : integer;
  length : real;
begin
 coordinatechange(xpixel, ypixel, screenx, screeny);
 o[1] := -1.0;
 o[2] := 0.0;
 o[3] := 0.0;
  { o is the "eye", where the initial sight vector v originates }
 v[1] := 1;
 v[2] := screenx;
 v[3] := screeny;
 length := magnitude(v);
 for i := 1 to 3 do
  v[i] := v[i] / length;
 blackhit := false;
 end; { initialize }
procedure newray (var o, v : vector;
        var intersection, isblack : boolean);
 { traces ray v and reflects it off any object it hits }
 var
  i : integer;
  tsphere1, tsphere2, tplane, a, b, c, t : real;
  n, otemp, sphere1hit, sphere2hit, planehit : vector;
begin
 isblack := false;
 intersection := false;
 if intersectspherel(o, v) then
  begin
   otemp[1] := o[1] - centerl[1];
   otemp[2] := o[2] - centerl[2];
   otemp[3] := o[3] - centerl[3];
   a := sqr(magnitude(v));
   b := 2 * dot(otemp, v);
   c := sqr(magnitude(otemp)) - 1.0;
   if (sqr(b) - 4.0 * a * c) > 0.0 then
    t := (-b - sqrt(sqr(b) - 4.0 * a * c)) / (2.0 * a)
   else
    t := 100.0;
   if t > 0.0 then
    begin
     intersection := true;
     tspherel := t;
    end
   else
    tspherel := 100.0;
  end
 else
  tspherel := 100.0;
 if intersectsphere2(o, v) then
  begin
   otemp[1] := o[1] - center2[1];
   otemp[2] := o[2] - center2[2];
   otemp[3] := o[3] - center2[3];
   a := sqr(magnitude(v));
   b := 2 * dot(otemp, v);
   c := sqr(magnitude(otemp)) - 1.0;
   if (sqr(b) - 4.0 * a * c) > 0.0 then
    t := (-b - sqrt(sqr(b) - 4.0 * a * c)) / (2.0 * a)
   else
    t := 100.0;
   if t > 0.0 then
    begin
     intersection := true;
     tsphere2 := t;
    end
   else
    tsphere2 := 100.0;
  end
 else
  tsphere2 := 100.0;
 if intersectplane(v) then
  begin
   intersection := true;
   tplane := (-1.0 - o[3]) / v[3];
   for i := 1 to 3 do
```

(continued)

```
   planehit[i] := o[i] + tplane * v[i];
  end
 else
  tplane := 100.0;
 if intersection then
  begin
   if (((tsphere1 < tsphere2) and (tsphere1 < tplane)) and (ts
   then
  begin
   for i := 1 to 3 do
    sphere1hit[i] := o[i] + tsphere1 * v[i];
   n[1] := sphere1hit[1] - center1[1];
   n[2] := sphere1hit[2] - center1[2];
   n[3] := sphere1hit[3] - center1[3];
   o := sphere1hit;
   reflect(n, v);
  end
 else if (((tsphere2 < tsphere1) and (tsphere2 < tplane)) and (tsphere2 < ,
   90.0)) then
  begin
   for i := 1 to 3 do
    sphere2hit[i] := o[i] + tsphere2 * v[i];
   n[1] := sphere2hit[1] - center2[1];
   n[2] := sphere2hit[2] - center2[2];
   n[3] := sphere2hit[3] - center2[3];
   o := sphere2hit;
   reflect(n, v)
  end
 else if (((tplane < tsphere1) and (tplane < tsphere2)) and (tplane <
   90.0)) then
  begin
   begin
    if (((trunc(planehit[1])+trunc(planehit[2]+300.0)) mod 2) = 1)then
     begin
      isblack := true;
      intersection := false;
     end
    else
     begin
      n[1] := 0.0;
      n[2] := 0.0;
      n[3] := 1.0;
      o := planehit;
      reflect(n, v);
      intersection := true;
     end;
   end;
  end
 else
```

```
    intersection := false
  end;
end; { newray }

begin
 CLEAR;
 center1[1] := 1.0;
 center1[2] := 0.5;
 center1[3] := 1.0;
 center2[1] := 2.0;
 center2[2] := -1.0;
 center2[3] := 0.0;
 { describes centers of spheres }
 for ypixel := 307 downto 1 do
  begin
   for xpixel := 500 downto 1 do
    begin
     initialize(o, v, blackhit);
     intersection := true;
     repeat
      if intersection = true then
       newray(o, v, intersection, blackhit);
     until intersection = false;
     shade := shadepixel(v);
     drawpixel(xpixel, ypixel, shade)
    end
  end
end.
```

Appendix A

Graphics Library

```
(* GRAPHICS.lib *)
(* MACINTOSH LIBRARY FOR GRAPHICS OUTPUT *)

(* The region of the drawing plane defined by the *)
(* inequalities: -1 < x < +1, -1 <y < +1  will be a convenient square*)
(* centered in the graphics window. *)

procedure CLEAR;
(* Activates and expands Drawing Window to fill screen *)
 var
  windowrect : rect;
begin
 hideall;
 setrect(windowrect, 2, 35, 512, 342);
 setdrawingrect(windowrect);
 showdrawing;
end;

procedure MAP (x, y : real;
        var hor, ver : integer);
(* Returns hor,ver as pixel integers for given real coords (x,y) *)
begin
 if abs(x) > 1.9 then
  x := x / abs(x) * 1.9;
 hor := 255 + trunc(x * 130);
 if abs(y) > 1 then
  y := y / abs(y);
 ver := 138 - trunc(y * 130);
end;

procedure DRAW (x, y : real);
(* Draws to (x,y) *)
 var
  h1, v1 : integer;
begin
 MAP(x, y, h1, v1);
 lineto(h1, v1);
end;
```

(continued)

```
procedure SHIFT (x, y : real);
(* moves invisible to (x,y) *)
 var
  h1, v1 : integer;
begin
 MAP(x, y, h1, v1);
 moveto(h1, v1);
end;

procedure PLOT (x, y : real);
(* Plots a point to (x,y) *)
begin
 SHIFT(x, y);
 DRAW(x, y);
end;

procedure CIR (x, y, r : real);
(* Draws circle of radius r centered at (x,y) *)
 var
  h1, v1, h2, v2 : integer;
begin
 MAP(x - r, y + r, h1, v1);
 MAP(x + r, y - r, h2, v2);
 frameoval(v1, h1, v2, h2);
end;

procedure UNMAP (h, v : integer;
        var x, y : real);
(* Forces real coordinates (x,y) to correspond to given pixel(h,v) *)
begin
 x := (h - 255) / 130;
 y := (138 - v) / 130;
end;

procedure REALMOUSE (var x, y : real);
(* The real-coordinates equivalent to 'getmouse(h,v)' *)
 var
  m, n : integer;
begin
 getmouse(m, n);
 UNMAP(m, n, x, y);
end;
```

Statistics Library

```
(* STAT.LIB STATISTICS LIBRARY*)
(* requires globals "size: integer" and type sample = array[..] of real*)
(* arguments of type sample are declared var for efficiency reasons *)
```

```
function RAND (x : real) : real;
(* Returns a random real in (0,x) exclusive *)
begin
 RAND := x * (random + 32768) / 65536;
end;

function SUMM (var v : sample) : real;
 var
  ctr : integer;
  sum : real;
begin
 sum := 0;
 for ctr := 1 to size do
  sum := sum + v[ctr];
 SUMM := sum;
end;

function MEAN (var v : sample) : real;
(* Returns mean value of sample *)
begin
 MEAN := SUMM(v) / size;
end;

function PROD (var u, v : sample) : real;
 var
  ctr : integer;
  sum : real;
begin
 sum := 0;
 for ctr := 1 to size do
  sum := sum + u[ctr] * v[ctr];
 PROD := sum;
end;

function ERROR (var u : sample) : real;
(* Returns standard deviation of sample *)
 var
  m : real;
  ctr : integer;
  z : sample;
begin
 m := MEAN(u);
 for ctr := 1 to size do
  z[ctr] := u[ctr] - m;
 ERROR := sqrt(PROD(z, z) / (size - 1));
end;

function DETR (var u : sample) : real;
begin
 DETR := sqr(SUMM(u)) - size * PROD(u, u)
end;
```

(continued)

```
function BESTM (var x, y : sample) : real;
(* Returns best-fit slope *)
begin
 BESTM := (SUMM(y) * SUMM(x) - size * PROD(x, y)) / DETR(x);
end;

function BESTB (var x, y : sample) : real;
(* Returns best-fit intercept *)
begin
 BESTB := (SUMM(x) * PROD(x, y) - SUMM(y) * PROD(x, x)) / DETR(x);
end;

procedure GETDATA (var x : sample;
        var size : integer);
(* Reads a sample and sets 'size' *)
begin
 size := 0;
 repeat
  size := size + 1;
  readln(x[size]);
 until eof;  (* Hit ENTER for keyboard eof *)
end;

procedure GETPAIRS (var x, y : sample;
        var size : integer);
(* Reads two sample columns of data and sets 'size' *)
begin
 size := 0;
 repeat
  size := size + 1;
  readln(x[size], y[size]);
 until eof;  (* Hit ENTER for keyboard eof *)
end;

function MAXPOINT (var x : sample) : integer;
(* Returns index of sample maximum *)
 var
  pt, ctr : integer;
  m : real;
begin
 m := x[1];
 pt := 1;
 for ctr := 2 to size do
  if x[ctr] > m then
   begin
    pt := ctr;
    m := x[ctr]
   end;
 MAXPOINT := pt;
end;
```

```
function MINPOINT (var x : sample) : integer;
(* Returns index of sample minimum *)
 var
  ctr, pt : integer;
  m : real;
begin
 m := x[1];
 pt := 1;
 for ctr := 2 to size do
  if x[ctr] < m then
   begin
    pt := ctr;
    m := x[ctr]
   end;
 MINPOINT := pt;
end;

function POISS (mean : real) : integer;
(* produces random integers in Poisson distribution *)
 var
  sum : real;
  ctr : integer;
begin
 sum := 0;
 ctr := -1;
 repeat
  ctr := ctr + 1;
  sum := sum - ln(RAND(1))
 until sum > mean;
 POISS := ctr
end; { poiss }

function GAUSS (mean, error : real) : real;
(* Produces random real in Gaussian distribution *)
 var
  u, v, x : real;
begin
 repeat
  u := RAND(1);
  v := RAND(1);
  x := 2.0 * (v - 0.5) / u
 until sqr(x) <= -(4.0 * ln(u));
 GAUSS := x * error + mean;
end;
```

Matrix Algebra Library

```
   (* MATRIX.lib - MATRIX ALGEBRA LIBRARY *)
   (* requires globals type matrix = array[1..dim,1..dim] of real; *)
   (* and vector = array [1..dim] of real *)
```

(continued)

```
procedure READVECT (n : integer;
        var x : vector);
(* Read a vector of dimension n *)
 var
  j : integer;
begin
 for j := 1 to n do
  read(x[j]);
 readln;
end;

procedure READMAT (n : integer;
        var m : matrix);
(* Read an nXn matrix *)
 var
  i, j : integer;
begin
 for i := 1 to n do
  begin
   for j := 1 to n do
    read(m[i, j]);
  end;
end;

procedure WRITEMAT (n : integer;
        m : matrix);
(* Outputs an nXn matrix *)
 var
  i, j : integer;
begin
 for i := 1 to n do
  begin
   for j := 1 to n do
    write(m[i, j] : 6 : 3);
   writeln;
  end;
end;

procedure WRITEVEC (n : integer;
        x : vector);
(* Outputs a vector x of dimension n *)
 var
  i, j : integer;
begin
 for i := 1 to n do
  write(x[i] : 6 : 3);
 writeln;
end;

function DET (n : integer;
        a : matrix) : real;
  (* Return the determinant of nXn matrix a *)
  (* NOTE !! Dimension of matrix var must be at least n+1  *)
   var
    ii, jj, kk, ll, ff, nxt : integer;
    piv, cn, big, temp, term : real;
  begin
   ff := 1;
   for ii := 1 to n - 1 do
    begin
     big := 0;
     for kk := ii to n do
      begin
       term := abs(a[kk, ii]);
       if term - big > 0 then
        begin
         big := term;
         ll := kk;
        end;
      end;
     if ii - ll <> 0 then
      ff := -ff;
     for jj := 1 to n + 1 do
      begin
       temp := a[ii, jj];
       a[ii, jj] := a[ll, jj];
       a[ll, jj] := temp;
      end;
     piv := a[ii, ii];
     nxt := ii + 1;
     for jj := nxt to n do
      begin
       cn := a[jj, ii] / piv;
       for kk := ii to n + 1 do
        a[jj, kk] := a[jj, kk] - cn * a[ii, kk];
      end;
    end;
   temp := 1;
   for ii := 1 to n do
    temp := temp * a[ii, ii];
   det := temp * ff;
  end;

procedure SOLVE (n : integer;
        a : matrix;
        c : vector;
        var x : vector);
 var
  k : integer;
  d : real;
```

(continued)

```
procedure swap (n, k : integer;
         var a : matrix;
         var c : vector);
  var
   e : real;
   j : integer;
 begin
  for j := 1 to n do
   begin
    e := c[j];
    c[j] := a[j, k];
    a[j, k] := e;
   end;
 end;

begin
 d := DET(n, a);
 for k := 1 to n do
  begin
   swap(n, k, a, c);
   x[k] := det(n, a) / d;
   swap(n, k, a, c);
  end;
end; { solve }

procedure CHANGECOL (rows, columns, target : integer;
        var m : matrix;
        newcol : vector);
 var
  i : integer;
begin
 for i := 1 to rows do
  m[i, target] := newcol[i];
end; { changecol }

procedure INVERT (n : integer;
        a : matrix;
        var b : matrix);
(*invert the n-by-n matrix in a into b.*)
 var
  I, tmp : vector;
  j : integer;
begin
 if det(n, a) = 0 then
  writeln('invert : singular matrix')
 else
  begin
   for j := 1 to n do
```

```
    begin
     I[j] := 1;
     solve(n, a, I, tmp);
     changecol(n, n, j, b, tmp);
     I[j] := 0;
    end;
  end;
end;
```

Three Dimensional Graphics Library

```
(* 3D.lib - THREE DIMENSIONAL GRAPHICS LIBRARY *)

procedure CLEAR;
(* Size and clear the Drawing Window *)
 var
  windowrect : rect;
begin
 hideall;
 setrect(windowrect, 2, 35, 512, 342);
 setdrawingrect(windowrect);
 showdrawing;
end;

procedure MAP (x, y : real;
        var hor, ver : integer);
(* Force hor and ver to be integer pixel coordinates *)
begin
 if abs(x) > 1.9 then
  x := x / abs(x) * 1.9;
 hor := 255 + trunc(x * 130);
 if abs(y) > 1 then
  y := y / abs(y);
 ver := 138 - trunc(y * 130);
end;

procedure DRAW (x, y : real);
(* Draw to the point (x,y) *)
 var
  h1, v1 : integer;
begin
 MAP(x, y, h1, v1);
 lineto(h1, v1);
end;

procedure SHIFT (x, y : real);
(* Move invisible to the point (x,y) *)
 var
  h1, v1 : integer;
begin
```

(continued)

```
 MAP(x, y, h1, v1);
 moveto(h1, v1);
end;

procedure PLOT (x, y : real);
(* Place a dot at (x,y) *)
begin
 SHIFT(x, y);
 DRAW(x, y);
end;

procedure ROTATE (var x, y, z : real;
        a, b, c : real);
(* Rotate (x,y,z) by the three Euler angles (a,b,c) *)
 var
  sx, sy, sz, tx, ty, tz : real;
begin
 sx := x * cos(c) - y * sin(c);
 sy := x * sin(c) + y * cos(c);
 sz := z;
 tx := sx;
 ty := sy * cos(b) - sz * sin(b);
 tz := sy * sin(b) + sz * cos(b);
 x := tx * cos(a) - ty * sin(a);
 y := tx * sin(a) + ty * cos(a);
 z := tz;
end;
procedure SPLOT (x, y, z, a, b, c : real);
(* Plot a point at the(a,b,c)-rotated (x,y,z) but no change in(x,y,z) *)
 var
  u, v, w : real;
begin
 u := x;
 v := y;
 w := z;
 ROTATE(u, v, w, a, b, c);
 PLOT(u, v);
end;

procedure SDRAW (x, y, z, a, b, c : real);
(* Draw to the (a,b,c)-rotated point(x,y,z);but no change in(x,y,z)*)
 var
  u, v, w : real;
begin
 u := x;
 v := y;
 w := z;
 ROTATE(u, v, w, a, b, c);
 DRAW(u, v);
end;
```

```
procedure SMOVE (x, y, z, a, b, c : real);
(* Move invisible to the (a,b,c)-rotated but unchanged point (x,y,z)*)
 var
  u, v, w : real;
begin
 u := x;
 v := y;
 w := z;
 ROTATE(u, v, w, a, b, c);
 SHIFT(u, v);
end;

procedure AXES (a, b, c : real);
(* Draw the coordinate axes for new view (a,b,c) *)
begin
 SMOVE(-1, 0, 0, a, b, c);
 SDRAW(1, 0, 0, a, b, c);
 SMOVE(0, -1, 0, a, b, c),
 SDRAW(0, 1, 0, a, b, c);
 SMOVE(0, 0, -1, a, b, c);
 SDRAW(0, 0, 1, a, b, c);
end;
```

Special Mathematical Functions Library

```
(* MATH.lib - Special Mathematical Functions Library *)


function GAMMA (z : real) : real;
(* gamma function of z > 0 *)
 const
  pi = 3.14159265358979323846 3;
 var
  zz : real;
begin
 if (z > 1) then
  GAMMA := (z - 1) * GAMMA(z - 1)
 else if (z = 1) then
  GAMMA := 1
 else if (z > 0.5) then
  GAMMA := pi / (sin(pi * z) * GAMMA(1 - z))
 else
  begin
   zz := 1 / z - 0.5748646 + 0.9512363 * z - 0.6998588 * z * z;
   GAMMA := zz + 0.4245549 * z * z * z - 0.1010678 * z * z * z * z;
  end;
end;
```

(continued)

```
function J (nu, z : real) : real;
(* Bessel function of order nu *)
 var
  n, d, f, sum : real;
  ctr : integer;
begin
 if z = 0 then
  begin
   if nu = 0 then
    j := 1
   else
    j := 0;
  end
 else
  begin
   f := 1 / GAMMA(nu + 1);
   n := -(sqr(z) / 4);
   d := nu + 1;
   sum := f;
   ctr := 1;
   repeat
    f := f * n / (ctr * d);
    d := d + 1;
    ctr := ctr + 1;
    sum := sum + f
   until abs(f) < 0.0000000001;
   if z > 0 then
    J := sum * exp(nu * ln(z / 2))
   else
    J := (1 - 2 * (trunc(nu) mod 2)) * sum * exp(nu * ln(abs(z / 2)))
  end;
end;

function ERF (x : real) : real;
(* Error function of statistics *)
 var
  temp, mult, t, a1, a2, a3, a4, a5, p : real;
begin
 if x < 0 then
  mult := -1
 else
  mult := 1;
 x := abs(x);
 p := 0.3275911;
 a1 := 0.254829592;
 a2 := -0.284496736;
 a3 := 1.421413741;
 a4 := -1.453152027;
 a5 := 1.061405429;
 t := 1 / (1 + p * x);
 temp := a1 * t + a2 * t * t + a3 * t * t * t + a4 * t * t * t * t
 + a5 * t * t * t * t * t;
 ERF := mult * (1 - exp(-sqr(x)) * temp);
end;

function F (a, b, c, z : real) : real;
(* The Gauss Hypergeometric function *)
 var
  g, sum : real;
  n : integer;
begin
 g := 1;
 n := 1;
 sum := 0;
 repeat
  sum := sum + g;
  g := g * a * b * z / (n * c);
  n := n + 1;
  a := a + 1;
  b := b + 1;
  c := c + 1
 until (n > 50) or (abs(g) < 1e-13);
 f := sum;
end;

function M (a, b, z : real) : real;
(* The Kummer (confluent) Hypergeometric function *)
 var
  g, sum : real;
  n : integer;
begin
 sum := 0;
 g := 1;
 n := 1;
 repeat
  sum := sum + g;
  g := g * a * z / (b * n);
  n := n + 1;
  a := a + 1;
  b := b + 1
 until (n > 50) or (abs(g) < 1e-13);
 M := sum;
end;

function U (a, b, z : real) : real;
(* Associated confluent Hypergeometric function *)
 const
  pi = 3.1415926535897932;
```

(continued)

```
begin
 if abs(z) < 1e-10 then
  U := 0
 else
  U := pi / sin(pi * b) * (m(a, b, z) / GAMMA(1 + a - b) * GAMMA(b)) -exp
  ((1 - b) * ln(z)) * M(1 + a - b, 2 - b, z) / (GAMMA(a) * GAMMA(2 - b)));
end;

function HER (n : integer;
        x : real) : real;
(* Hermite polynomial of order n *)
begin
 if n = 1 then
  HER := 2 * x
 else
  HER := x * exp(n * ln(2)) * u(0.5 - n / 2, 3 / 2, sqr(x));
end;

function LEG (n : integer;
        x : real) : real;
(* Legendre polynomial of order n *)
begin
 LEG := F(-n, n + 1, 1, (1 - x) / 2);
end;

function LAG (n : integer;
        a, x : real) : real;
(* Laguerre function of order n, superscript a *)
 var
  fact : real;
begin
 if n mod 2 = 0 then
  fact := 1
 else
  fact := -1;
 LAG := u(-n, a + 1, x) / GAMMA(n + 1) * fact;
end;

function I (nu, x : real) : real;
(* Modified Bessel function of order nu *)
begin
I := exp(-x) * exp(nu ln(x / 2)) * m(nu + 0.5, 2 * nu + 1, 2 * x) /
 GAMMA(nu + 1);
end;

function K (nu, x : real) : real;
(* Modified Bessel function of order nu *)
(* positive args only *)
 const
  pi = 3.1415926535897932;
```

```
begin
 K: =exp(nu * ln(2 * x)) * exp(-x) * sqrt (pi) * U(nu + 0.5, 2 * nu
 +1, 2 * x);
end;

function COMB (n : real;
        m : integer) : real;
(* Combinatorial bracket n-over-m, m an integer *)
begin
 if m = 1 then
  COMB := n
 else
  COMB := COMB(n - 1, m - 1) * n / m
end;

function JAC (n : integer;
        a, b, x : real) : real;
(* Jacobi polynomial of order n, superscripts (a,b) *)
begin
 JAC := COMB(n + a, n) * F(-n, n + a + b + 1, a + 1, (1 - x) / 2);
end;

function TCHEB (n : integer;
        x : real) : real;
(* Tchebyshev polynomial of order n *)
begin
 TCHEB := F(-n, n, 0.5, (1 - x) / 2);
end;
```

Number Theoretic Functions Library

```
(* NUM.lib - Number Theoretic Functions library *)

function PMOD (x, y, z : longint) : longint;
(* return x ^ y (mod z) *)
 var
  e : longint;
begin
 e := 1;
 while y <> 0 do
  begin
   if (y mod 2) <> 0 then
    begin
     y := y - 1;
     e := (e * x) mod z;
    end;
   y := y div 2;
   x := (x * x) mod z;
  end;
```

(continued)

```
 PMOD := e;
end;
function GCD (a, b : longint) : longint;
 var
  r : longint;
begin
 if b < 0 then
  b := -b;
 if a < 0 then
  a := -a;
 if a > 0 then
  begin
   b := b mod a;
   r := 1;
   if b = 0 then
    r := 0;
   while r > 0 do
    begin
     q := a div b;
     r := a - q * b;
     a := b;
     b := r;
    end;
  end;
 GCD := a;
end;
```

Differential Equations Library

```
(* DiffEqu.lib - DIFFERENTIAL EQUATIONS LIBRARY *)
(* Procedure 'Update' is self-contained Runge-Kutta Solver *)
(* for general second-order equation:*)
(*      (dd/dtt)x  =  f(x,dx/dt,t)   *)


procedure UPDATE (var x, dxdt, t : real;
        dt : real);
 var
  k1, k2, k3, k4 : real;

 function f (x, dxdt, t : real) : real;
 begin
  f := -k * x / m - 2 * b * dxdt / m;
 end;

begin
 k1 := dt * f(x, dxdt, t);
 k2 := dt * f(x + dt * (dxdt + k1 / 4) / 2, dxdt + k1 / 2, t + dt / 2);
 k3 := dt  *  f(x + dt * (dxdt + k2 / 4) / 2, dxdt + k2 / 2, t + dt / 2);
 k4 := dt * f(x + dt * (dxdt + k3 / 2), dxdt + k3, t + dt);
 x := x + dt * (dxdt + (k1 + k2 + k3) / 6);
 dxdt := dxdt + (k1 + 2 * k2 + 2 * k3 + k4) / 6;
 t := t + dt;
end;
```

Index

Computers **$18.95**

Master problem-solving with your mouse.

SCIENTIFIC PROGRAMMING WITH MACINTOSH PASCAL

Richard Crandall and Marianne Colgrove show you how to use the Macintosh's mouse and window technology to solve real-world problems in biology, chemistry, mathematics and physics. Using the unique teaching approach developed in courses at Reed College (a member of the Apple University Consortium), Crandall and Colgrove explain the basics of programming in Macintosh Pascal. They take you step by step through keyboard, text, drawing, mouse and windowing routines and show you how to use the sound generator, printer and disk drive. Once you know how the Macintosh works, you'll start solving problems—everything from integral calculus, linear equations, complex numbers, vectors and matrices to probability and statistics.

Scientific Programming with Macintosh Pascal also shows how to take advantage of the Macintosh's superior graphics and animation capabilities. You'll learn techniques for graphing functions and real-valued coordinates as well as how to create 3-dimensional graphs using Kupin space-time models and parametric space curves.

The guide's self-teaching format makes it easy for those who've already learned Pascal to convert to Macintosh Pascal. Examples and exercises from various scientific disciplines are designed to strengthen problem-solving and programming skills. The book features a wealth of pre-tested, powerful routines, covering such areas as statistics, mathematical physics and signal processing. Extensive libraries of scientific routines in Macintosh Pascal are also included.

RICHARD E. CRANDALL is founder of Metasearch, Inc. and a member of the physics faculty at Reed College.

MARIANNE M. COLGROVE is the Education and Documentation Coordinator in the Software Development Laboratory at Reed College.

WILEY PRESS
JOHN WILEY & SONS, Inc.
Business/Law/General Books Division
605 Third Avenue, New York, N.Y. 10158-0012
New York • Chichester • Brisbane • Toronto • Singapore

ISBN 0 471-82176-4