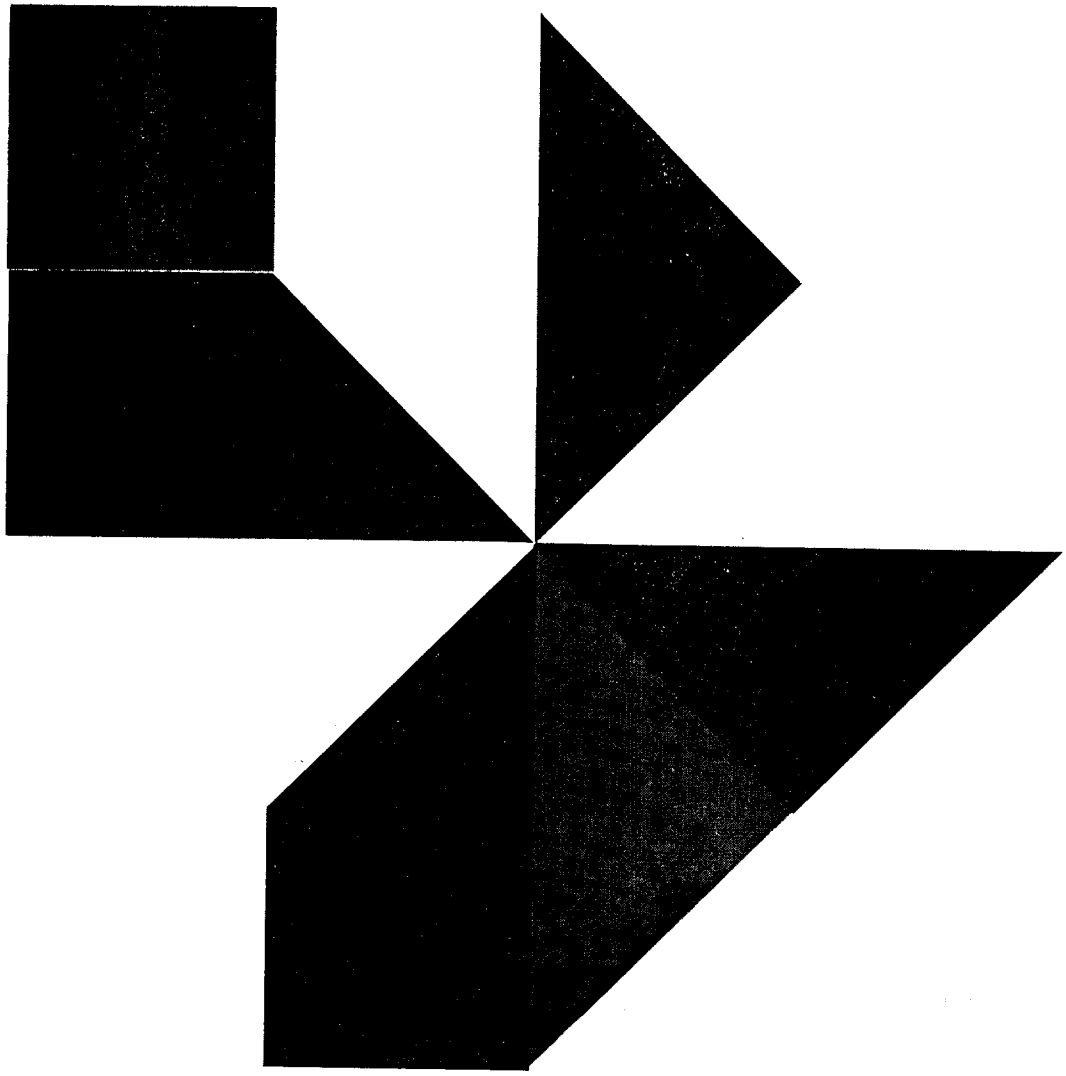
Computer Programming

5



1 INTRODUCTION

Modern high-speed computers have made all forms of computation vastly easier. Calculations that used to take several days to complete can now be carried out in a few seconds. The availability of a modern computer can take a great deal of drudgery out of mathematical computations and makes possible large-scale computations that would otherwise be impossible. This is particularly true in the area of finite mathematics, since each of the branches of mathematics introduced in this book is well suited to computer applications.

It is the purpose of this chapter to give a first introduction to the use of high-speed computers. A computer is an electronic device designed to carry out arithmetical operations and to follow a long list of instructions as to what calculations should be carried out. A computer does no more and no less than a human user instructs it to do; however, it can carry out tasks at tremendous speed and with great accuracy. The key to the use of a computer is learning how to write a set of instructions. Such a set of instructions is called a *program*, and the art of writing such instructions is known as *programming*. This chapter will give a number of examples of programs for high-speed computers designed to carry out calculations in finite mathematics. Although only elementary programming techniques will be illustrated, they will be sufficient to carry out many significant mathematical tasks.

The chapter is written so that it may profitably be studied without having a computer available. However, being able to try out examples on a computer will significantly improve the learning experience. Each section will include many exercises that do not require the use of the computer and also some exercises that must be completed on a computer.

In order to communicate with a human being, it is necessary to understand the language he speaks. Similarly, a user must learn a suitable language for communicating with a computer. Fortunately, there are several easily learned languages that most computers "speak." One language widely used, particularly in educational uses, is BASIC. The present chapter provides a brief introduction to the language BASIC. The reader interested in more sophisticated applications, including many further applications to finite mathematics, is referred to the book *Basic Programming*, which is listed in the suggested readings at the end of the chapter.

Since the language BASIC is almost self-explanatory, it is simplest to learn it by looking at some actual programs. The program EXAMPLE1 is designed to compute several factorials—specifically, 4!, 6!, and 10!. The program consists of five lines, each of which contains one instruction for the computer. It will be noted that each line starts with a line number. These numbers are required in BASIC to make it easy to enter corrections to a given program. For example, if a user wishes to correct a given line, he simply retypes that line with the same line number and the correction is automatically made by the computer. Or, if it is desired to insert a line between, say, lines 20 and 30, one may type the new instruction with any number between 20 and 30 and a correction is automatically made. Thus it is good practice to choose line numbers with gaps between them (for example, multiples of 10) to allow for the insertion of additional instructions. An additional use of line numbers will be explained in the next section.

Most of the variables in this chapter are represented by capital letters, since many computer terminals print only capitals.

Look now at EXAMPLE1. Line 10 in EXAMPLE1 instructs the computer to compute 4!, i.e., $1 \times 2 \times 3 \times 4$. To avoid the ambiguity between the dot as a multiplication sign and as a decimal point, BASIC uses an asterisk (*) to indicate multiplication. Specifically, the LET command in line 10 tells the machine to compute $1 \times 2 \times 3 \times 4$ and to call the answer "X.") Thus after the computer has carried out the instruction in line 10, X will equal 24. This quantity may now be used in the rest of the program. In line 20 the computer is asked to take X and multiply it by 5 and then by 6, and to call the result Y. Thus Y will equal 6! or 720. Similarly, on line 30 the previous result is multiplied by 7, 8, 9, and 10, thus obtaining 10!, and calling the result Z. The instruction LET is designed to carry out a wide variety of computations. The format for the LET command is always to put on the right-hand side of the equals sign the computation that is to be carried out and to indicate the name of the result on the left-hand side of the equals sign.

Computations are useless unless the user can see the result. In a long program there are many partial results that are of no interest to the user and that would take too long to be typed, so that we don't want to print every result. Therefore there is an instruction in BASIC called PRINT which tells the computer to print or type only the desired results. Line 40 instructs the computer to type out X, Y, and Z. It is up to the programmer to

```
EXAMPLE1

10 LET X = 1*2*3*4
20 LET Y = X*5*6
30 LET Z = Y*7*8*9*10
40 PRINT X,Y,Z
50 END
READY

RUN

EXAMPLE1

 24                720               3628800

0.062 SEC.
READY
```

remember that the results stand for 4!, 6!, and 10!, respectively. The final instruction, in line 50, is END. In all programs the last instruction must be an END statement. This line both indicates the physical end of the program and tells the computer to stop.

Immediately after the program we show the results that are printed as the program is executed or "RUN." Three numbers are printed which are the desired results. It is important to note that the computations took only a small fraction of a second.*

Many interesting and useful programs can be written with the minimal vocabulary of LET, PRINT, and END. A more sophisticated example is shown in EXAMPLE2. Line 10 carries out a subtraction and an addition. Line 20 shows one decimal fraction being divided by the sum of two other decimal fractions. Note that parentheses are inserted as usual. Line 30 requires an additional word of explanation. One usually communicates with a computer through a typewriterlike terminal device, and this imposes certain limitations on the way formulas are typed. Specifically, each formula must be contained on a single line. This was already illustrated by the form of division on line 20. Since it is not possible to type an exponent on a higher line, an upward arrow (↑) is used to indicate an exponent. Thus line 30 asks the computer to raise the number 2.15 to the sixth power and to let the answer be Z.

Line 40 illustrates some additional options available for the PRINT instruction. In EXAMPLE1 a comma (,) separated the variables, which is the signal to BASIC to line up the answers in predetermined columns (also called fields), normally up to five columns per line. If one is not interested

*The reader will note that in the computer output zeros appear as 'Ø'. This is done to distinguish the number zero from the letter 'O'. Unfortunately, different conventions for this are used on different computer terminals.

```
EXAMPLE2

10 LET X = 397-128+511
20 LET Y = .57/(.23+.82)
30 LET Z = 2.15↑6
40 PRINT X;Y;Z;2*X;Y*Z
50 END
READY

RUN

EXAMPLE2

 780   0.542857   98.7713   1560   53.6187

0.059 SEC.
READY
```

in a special format but would simply like to have answers printed one after the other, the answers are separated by semicolons (;), as shown in EXAMPLE2. This example also shows that computation instructions may take place within a PRINT statement. In addition to printing X, Y, and Z, we have also asked the computer to PRINT 2*X and Y*Z. Recall that an asterisk (*) is used to denote multiplication to the computer. The results are again shown.

These examples illustrate the fact that once the user masters a simple language for entering requests to the computer, all the hard work can be left to the machine. One of the nice features of modern computers is the fact that one can become quite expert in their use without necessarily having any understanding of how computers work. This is similar to the fact that millions of people use telephones and drive automobiles without having any understanding of the nature of telephone-switching networks or of automobile engines. That is why in this chapter we are concentrating entirely on the art of programming and not on the operation of computers. The following sections will introduce, step by step, some more powerful commands in the language BASIC which will enable the reader to use the computer for more complex calculations.

EXERCISES

Only Exercises 7–11 require the use of a computer.

1. Write a program that will compute and print the sum of the first three positive integers, of the first five positive integers, and of the first ten positive integers.
2. Write a program to compute the cube root of 100. (You will need to recall that a cube root is the same as the $\frac{1}{3}$ power.)

3. Write a two-instruction program to compute $\binom{7}{3}$.
4. The following program contains three *illegal* instructions, that is, instructions not satisfying the rules we have prescribed. Identify them.

```
10 LET X = .12345/.54321
20 LET Y = X↑Z
30 LET U = XY
40 LET X = X-X
50 PRINT X,Y,U+Z
60 PRINT X,Y,Z²
70 END
80 LET X = Y + Z
```

5. Without using a computer, figure out what would be printed when the following program is run.

```
10 LET X = 2
20 LET Y = 7-4
30 LET Z = Y↑X
40 LET Z = Z-2*Y+X
50 PRINT Z/X
60 END
```

[*Ans.* 2.5.]

6. Without using a computer, figure out what would be printed when the following program is run.

```
10 LET X = 20/5
20 LET Y = X↑(1/2)
30 LET Z = 1*2*3
40 LET U = Z-3*Y
50 PRINT U
60 END
```

7. Try the program of Exercise 4 on a computer to see what error messages are printed.
8. Run the program of Exercise 2 on a computer. What is the cube root of 100? [*Ans.* 4.64159.]
9. Use a computer to compute
(.54321/.12345)*(40/37)↑3.
10. Using a computer, employ a trial-and-error method to find the smallest integer whose fifth power is greater than 1,000,000.
11. Use a computer to compute $\binom{20}{7}$ [*Ans.* 77520.]

2 MORE ON THE LANGUAGE BASIC

The programs in Section 1 are not typical in that each instruction is carried out only once by the computer. Such calculations could easily be done with a desk calculator. To make the best use of the great speed of high-speed computers it is desirable to give short programs which result in hundreds,

thousands, or even millions of computer operations. One technique for this is the application of the same instructions to many different sets of data. This will be illustrated in the present section. An even more powerful technique will be shown in next section.

Let us suppose that we wish to carry out a number of divisions. Instead of writing a separate instruction for each operation, we can write a single instruction and use it over and over again, as shown in the program DIVIDE. Line 20 instructs the computer to PRINT the numbers A,B and their quotient. The trick is to specify various pairs A,B. This is accomplished by storing on line 40 a set of data and instructing the computer on line 10 to pick off two of these numbers. The READ statement instructs the computer to pick the next two numbers on the DATA line and call the first number A and the second number B.

Thus the first time line 10 is executed, A will equal 12 and B will equal 4, and thus on line 20 this pair of numbers will be printed as will their quotient A/B = 3. The next time line 10 is executed, A will equal 144 and B will equal 12. This will continue until all the data has been used up.

After reading a pair of numbers and printing the result, we would like the computer to go back and do the same two instructions over again. This is accomplished by a GOTO statement. Line 30 instructs the machine to GOTO 10—that is, to go to line 10, which in this case happens to be the beginning of the program. This is another important use of line numbers.

```
DIVIDE

10 READ A,B
20 PRINT A,B,A/B
30 GOTO 10
40 DATA 12,4,144,12,10,3.45,19782,345
50 END
READY

RUN

DIVIDE

 12              4              3
 144             12             12
 10              3.45           2.89855
 19782           345            57.3391
OUT OF DATA AT 10

STOP
0.079 SEC.
READY
```

```
40 DATA 169,13,2.97,1.23,-200,-50,12345,1289
RUN

DIVIDE

 169              13               13
 2.97             1.23             2.41463
-200             -50               4
 12345            1289             9.57719
OUT OF DATA AT 10

STOP
0.080 SEC.
READY

5 PRINT "FIRST NO.","SECOND NO.","QUOTIENT"
RUN

DIVIDE

FIRST NO.        SECOND NO.       QUOTIENT
 169              13               13
 2.97             1.23             2.41463
-200             -50               4
 12345            1289             9.57719
OUT OF DATA AT 10

STOP
0.090 SEC.
READY
```

We include with a listing of the program the results that are obtained. It will be noted that the four pairs of numbers are printed on separate lines with the quotient in each case printed in the third column. This run also illustrates that there are two different ways of terminating a computer program. One is by reaching an END instruction; the other is for some condition to occur under which the computer can no longer proceed. In this particular case the fifth time it is asked to READ numbers A and B, it finds that there are no numbers left and therefore it prints the OUT OF DATA message. This is a perfectly legitimate way of terminating a program.

The advantage of writing a program with READ and DATA statements is twofold. First, it shortens the program significantly. Second, if the user wishes to reuse the program with different pairs of numbers, he only has to change line 40 and the rest of the program is still valid. If one retypes line 40 with different data and types "RUN" again, the new results will be obtained. This is shown as a second RUN of the program DIVIDE.

We would like to illustrate one more capability of the PRINT instruction.

It is often convenient to label the output of a computer program. A PRINT instruction will PRINT any label contained between quotation marks exactly as you typed it. We can add labels to the program DIVIDE as follows:

```
5 PRINT "FIRST NO.", "SECOND NO.", "QUOTIENT".
```

Making the line number "5" indicates to the computer that the instruction should be inserted at the beginning of the program (i.e., before line 10). The three labels will be printed exactly as indicated. The fact that the labels are separated by commas indicates to the machine that they should be typed in separate columns and they will automatically line up with the three columns of output. A new RUN is shown. Such labels may be inserted anywhere in a PRINT statement, as will be seen in the next program.

A simple computer program will allow us to convert a probability to odds. We recall (see Chapter 3, Section 2) that if the probability that an event will occur is P, and $Q = 1 - P$, then the odds in favor of the event may be expressed as P/Q to 1. This is carried out in the program ODDS.

A set of probabilities is provided on line 90. Line 10 reads one of these probabilities and calls it P. Line 15 computes Q. Line 20 does double duty, both computing the odds and printing the answers. Note that in line 20

```
ODDS

5  PRINT "PROBABILITY","  ODDS "
10 READ P
15 LET Q = 1-P
20 PRINT P,P/Q;"TO 1"
30 GOTO 10
90 DATA .5,.75,.6,.3333333,.1
99 END
READY

RUN

ODDS

PROBABILITY      ODDS
 .5              1 TO 1
 .75             3 TO 1
 .6              1.5 TO 1
 .333333         .5 TO 1
 .1              .111111 TO 1
OUT OF DATA AT 10

STOP
.083 SEC.
READY
```

P is followed by a comma so that the probability will occur in one column and the odds in a separate column. After P/Q we have inserted a semicolon (;) so that the quotient is immediately followed by the phrase "TO 1." The effect of this PRINT format is clearly shown in the RUN. Line 30 simply instructs the program to go back and carry out the computations for the next probability.

As we look at the output we notice that while the first three lines look very clear, the last two are somewhat unnatural. One does not usually say that the odds are 0.5 to 1 in favor of an event; rather one would prefer to say that the odds are 1 to 2 in favor, or 2 to 1 against the event. To achieve this one must have one output format if the odds are in favor of the event (i.e., P greater than Q), and another format if they are not. We must be able to tell the computer that if a certain relationship holds then one thing should happen, and that otherwise something else should happen. This is provided for in BASIC by the IF . . . THEN instruction.

In the program ODDS2 we have inserted a test at line 17. If P is less

```
ODDS2

5  PRINT "PROBABILITY","  ODDS "
10 READ P
15 LET Q = 1-P
17 IF P<Q THEN 40
20 PRINT P,P/Q;"TO 1"
30 GOTO 10
40 PRINT P,"  1 TO";Q/P
50 GOTO 10
90 DATA .5,.75,.6,.3333333,.1
99 END
READY

RUN

ODDS2

PROBABILITY       ODDS
 0.5              1 TO 1
 0.75             3 TO 1
 0.6              1.5 TO 1
 0.333333         1 TO 2.
 0.1              1 TO 9
OUT OF DATA AT 10

STOP
0.090 SEC.
READY
```

than Q then the computer is instructed to skip to line 40 and use the alternate output format. But if P $\geq$ Q the computer goes on to line 20. On line 40 we compute the odds as 1 to Q/P rather than the form used on line 20. A RUN of the modified program is shown and the reader will note that the odds are now in both a simpler and a more natural form.

The significance of the IF . . . THEN statement is that the computer can be instructed to go in one of two different directions. And where it goes depends on the result of previous computations. In those cases where P turns out to be greater than or equal to Q, the computer proceeds with lines 20 and 30. However, if P is less than Q then the computer skips to lines 40 and 50. Thus the same computer program can handle both cases, and uses a simple test to distinguish between them.

The general form of this instructions is:

IF [relationship] THEN [line number].

The line number may be any line number in the program. For the relationship we may use six relational symbols: = (equals), < (is less than), > (is greater than), < = (is less than or equal to), > = (is greater than or equal to), and <> (is not equal to). A more complex example is the following:

IF (X*Y + 3)< =Z THEN 35.

If the current value of X*Y + 3 is less than or equal to the current value of Z, the program takes line 35 as its next instruction. If not, it will proceed in the normal order.

EXERCISES

Only Exercises 10–14 require the use of a computer.

1. Write a program that will READ a list of numbers and compute and print their fifth powers.
2. There are many ways of avoiding the "OUT OF DATA" message. One is to have a dummy number at the end of the DATA (say −99999) and to have the computer terminate when that number is READ. Modify the program of Exercise 1 by adding an IF statement so that it will terminate in this manner.
3. Modify the program ODDS2 so that instead of "1 TO 9" it will print "9 TO 1 AGAINST."
4. Modify the program ODDS2 to avoid the "OUT OF DATA" message.
5. If the DATA in the program ODDS contains an illegal probability (i.e., a negative number of a probability greater than 1), the result will be meaningless. Insert a test to make sure that P is between 0 and 1.
6. Write a program that will read a list of numbers from DATA and find its largest element. You will have to avoid the "OUT OF DATA" termination. (See Exercise 2.)
7. Modify the program of Exercise 6 to find the smallest element.

8. The absolute value Y of a number X may be computed in BASIC by writing LET Y = ABS (X). Design a test to check whether two numbers A and B are within 0.001 of each other.
9. In BASIC, INT(X) is the greatest integer less than or equal to the number X. For example, INT(6.235) = 6, INT(10.999) = 10, INT(15) = 15, and INT(−3.52) = −4. Design a test to check whether an integer X is an even number.
10. By means of the IF . . . THEN statement we can remove the trial-and-error method from Exercise 10, Section 1. Design and RUN a program that will READ a number A (A > 0), and find the smallest integer N whose fifth power is greater than A.
11. Try out the program of Exercise 6 on a computer. Does it work correctly when all the numbers in the DATA are negative?
12. In BASIC, SQR(X) is the square root of X. Use a computer to print a table of square roots for the first ten integers.
13. Modify the program of Exercise 12 to print the square roots of every fifth number between 100 and 200 (i.e., 100, 105, 110, . . . , 200).
14. There is a fast computational technique for finding the square root of a number A without using the "built-in" SQR function. One lets X be a guess at the square root. (For example, X = 1 is all right.) Let Y = A/X. If X is the correct square root, then Y = X. If not, one uses the average of X and Y as the next guess, and repeats the process until X and Y differ by less than a predetermined small error—say 0.000001. (See Exercise 8.) Write and RUN a program which carries out this technique. Check the answers by means of SQR.

3 LOOPS

Let us return to the problem of computing factorials. To compute 10! it is possible to proceed as in EXAMPLE1, or to write a single instruction:

```
LET X = 1*2*3*4*5*6*7*8*9*10.
```

However, this is a nuisance even for 10! and becomes very inconvenient for 25!. It also means that if we wish to compute several different factorials we have to write a different line for each one. We would instead like to write a simple set of instructions which say roughly, "Take the numbers from 1 to 10 and multiply them together." This can be accomplished by the pair of instructions FOR and NEXT.

The heart of the program FCTRL is contained in the "loop" on lines 30–50. The letter K will consecutively stand for the integers 1 through 10. The letter F will contain all the partial results and will eventually equal 10!. To understand line 40 we must remember that in a LET instruction the computer first computes the right-hand side and then lets the letter on the left equal the result. Thus F is multiplied by the current value of K and this becomes the new value of F. Line 50 instructs the computer to go on to the next value of K until all ten numbers have been used up.

```
FCTRL

20 LET F = 1
30 FOR K = 1 TO 10
40 LET F = F*K
50 NEXT K
60 PRINT F
99 END
READY

RUN

FCTRL

 3628800

0.052 SEC.
READY
```

Before starting the loop we must tell the computer what the "initial value" of F should be. In computing a product the initial value must always be 1. If we were computing a sum we would start with 0 (see Exercise 6). After the loop is completed we PRINT the final answer on line 60 and then

Line no.	Result
20	F = 1
30	K = 1
40	F = 1*1 = 1
50	GOES BACK TO 30
30	K = 2
40	F = 1*2 = 2
50	GOES BACK TO 30
30	K = 3
40	F = 2*3 = 6
50	GOES BACK TO 30
	•
	•
	•
	•
	•
	•
30	K = 10
40	F = (362880)*10 = 3628800
50	K EXHAUSTED, DOES NOT GO BACK
60	PRINT VALUE OF 10!
99	STOPS

Figure 1

line 99 instructs the computer to stop. Figure 1 shows what actually happens as each step in the computation is performed.

It is easy to modify this program to compute the factorial of an arbitrary number. In FCTRL2 we first PRINT labels and then READ the number N whose factorial we are trying to compute. In line 30 K now goes from 1 to N. In line 60 we have elected to PRINT both N and its factorial. Line 70 instructs the program to go back and read the next number. Line 90 contains five different values for N. The RUN shows the factorials of these five numbers.

Two comments are in order concerning the output. First, 20! is a number too large for all of the digits to be printed out. Therefore the computer prints it in "scientific notation." The abbreviation E + 18 stands for 10^{18}. In other words, the answer is 2.4329×10^{18}. It is also worth noting that 0! came out to be 1 without any special instruction to the computer. This is one more way of showing that 0! = 1 is the "natural convention."

```
FCTRL2

5 PRINT "NUMBER","FACTORIAL"
10 READ N
20 LET F = 1
30 FOR K = 1 TO N
40 LET F = F*K
50 NEXT K
60 PRINT N,F
70 GOTO 10
90 DATA 4,7,10,20,0
99 END
READY

RUN

FCTRL2

NUMBER          FACTORIAL
 4               24
 7               5040
 10              3628800
 20              2.4329 E+18
 0               1
OUT OF DATA AT 10

STOP
0.082 SEC.
READY
```

As our next illustration of loops we shall write a short program that computes an expected value. We recall from Chapter 3, Section 11, that an expected value is computed for an experiment whose possible outcomes are numbers by multiplying the numerical outcome A with the probability P of the outcome for each possible outcome, and adding up the results. This is carried out in the program EXPECT.

```
EXPECT

10 LET E = 0
20 FOR K = 1 TO 5
30 READ A,P
40 LET E = E + A*P
50 NEXT K
60 PRINT E
90 DATA 1,.3,2,.2,5,.05,-1,.25,-2,.2
99 END
READY

RUN

EXPECT

 0.3

0.059 SEC.
READY
```

Since the expected value E is computed as a sum, its initial value is set to 0. In the loop of lines 20–50, for each of the five possible outcomes we first read the numerical value A and the probability P. We then add to the previous value of E the quantity A*P. This will become the new value of E. After the loop is completed (by going through all five cases) E will be the expected value. This is printed by line 60. Note that the variable K acts as a counter only and does not otherwise enter the computation.

We see that the expected value in this simple illustration is 0.3. Of course in this case the answer could have been obtained more easily by hand computation. However, if the number of cases were significantly larger and the numbers were not as nice, the computer would indeed be useful.

Let us now turn to an application for which a computer is indispensable. We shall write a computer program for the "birthday problem" treated in Chapter 3, Section 4. (This example should be skipped by those who have not read that section.)

The problem was to compute the probability that among R people there are at least two with the same birthday. The trick was to compute first the

probability Q that all the birthdays are different, given by a formula in Section 4 of Chapter 3; then the probability we desire will be $P = 1 - Q$. In the program BIRTHDAY lines 15–45 carry out this computation in five simple instructions. The remaining lines are designed to allow us to compute

```
BIRTHDAY

5  PRINT "PEOPLE","PROBABILITY"
10 READ R
15 LET Q = 1
20 FOR K = 1 TO R
30 LET Q = Q * (366-K)/365
40 NEXT K
45 LET P = 1-Q
50 PRINT R,P
60 GOTO 10
90 DATA 10,20,22,23,30,50
99 END
READY

RUN

BIRTHDAY

PEOPLE          PROBABILITY
 10              .116948
 20              .411438
 22              .475695
 23              .507297
 30              .706316
 50              .970374
OUT OF DATA AT 10

STOP
.100 SEC.
READY
```

the answer for several different values of R and to PRINT the answers. The reader is invited to compare the results with those given in Figure 1 of Chapter 3. That the results agree (except for the fact that these numbers are rounded to three places in Figure 1) is not surprising since they were originally obtained by means of a computer. This is a good example in which a simple computer program and one-tenth of a second of computer time can save hours of laborious hand calculations.

The question is often raised: What is the probability of having more than

one coincidence of birthdays in a given group? That is, what are the chances that there will be three people with the same birthday or two pairs of identical birthdays or even larger coincidences? This probability can be computed in two steps. One first computes (as above) the probability of having some kind of coincidence. Then one computes separately the probability of having precisely one pair of people with the same birthday. The difference of these two quantities will give the probability of a multiple coincidence.

```
BIRTH2

5  PRINT "PEOPLE","PROBABILITY"
10 READ R
15 LET Q = 1
20 FOR K = 1 TO R
30 LET Q = Q * (366-K)/365
40 NEXT K
45 LET P = 1-Q
50 LET E = 1
60 FOR K = 1 TO R-1
65 LET E = E * (366-K)/365
70 NEXT K
75 LET E = E/365
77 LET E = E*R*(R-1)/2
80 PRINT R,P-E
85 GOTO 10
90 DATA 20,25,30,35,36,40,50
99 END
READY

RUN

BIRTH2

PEOPLE          PROBABILITY
 20              8.82398 E-2
 25              0.189257
 30              0.326101
 35              0.480722
 36              0.511803
 40              0.630989
 50              0.855524
OUT OF DATA AT 10

STOP
0.134 SEC.
READY
```

The program BIRTH2 is designed to compute this probability for various numbers of people. The quantities Q and P are computed as before. The quantity E will stand for the probability of exactly one pair with the same birthday. Let us first calculate the probability that the first two people have a specific birthday, say October 26, and that all the other people have different birthdays. This probability is $\frac{1}{365} \times \frac{1}{365} \times \frac{364}{365} \times \frac{363}{365} \times \ldots$. However, the same two people could have had a coincidence of birthdays on any of 365 days, and therefore we must multiply the answer by 365. This will cancel the first factor of $\frac{1}{365}$. This calculation is carried out in BIRTH2, lines 50–75. We must still correct this answer since we have so far assumed that it is the first two people who have a coincidence of birthdays. Such a coincidence may occur for any pair from among the R people, and therefore we must multiply the answer by $\binom{R}{2} = \frac{R \times (R-1)}{2}$, which is carried out in line 77. You should "step through" the program BIRTH2 by hand to see that it is actually carrying out the calculations described above.

The program prints the probability of a multiple coincidence for several different numbers of people. We notice that for 25 people—a number for which we already have a better-than-even chance of having some coincidence—the probability of a greater coincidence is less than .2. The smallest number of people for which a multiple coincidence has better than an even chance is 36. We note that for 50 people the probability of a multiple coincidence is very high.

We have now discussed nine instructions in BASIC. It is significant that these nine instructions are sufficient to write many interesting programs. They are summarized in Figure 2 for the reader's convenience.

Instructions for nine-word BASIC

Instruction	Example	Purpose
LET	LET X = 2+3	Carries out computations
PRINT	PRINT X,Y,X+Y	Prints results
END	END	Terminates computation
READ	READ A,B	Enters numbers from DATA
DATA	DATA 5, −2,3.4	Stores data
GOTO	GOTO 20	Transfers program control
IF . . . THEN	IF X>3 THEN 20	Performs a test
FOR	FOR N = 1 TO 8	Starts a loop
NEXT	NEXT N	Closes a loop

Figure 2

EXERCISES

Only Exercises 9–14 require the use of a computer.

1. Use FOR and NEXT to write a program that will compute the seventh powers of the first ten positive integers.

2. Write a program that will compute the cube roots of the integers from 1 to 20.
3. A loop need not run through all the integers specified in the FOR statement. For example, the instruction

```
FOR N = 1 TO 15 STEP 2
```

will run through the odd numbers from 1 to 15. Write a program to compute the cube roots of the multiples of 10 from 10 to 100.
4. Write a program to print the fifth powers of the even integers up to 30.
5. If we know how many numbers there are on the DATA list, we may avoid the OUT OF DATA message by reading the data within a loop. Modify DIVIDE in this manner. [*Hint:* Remember that a pair of numbers is READ each time.]
6. Write a program to compute the sum of the first 100 integers. What must the initial value of the sum be?
7. Modify the program of Exercise 6 to READ a number N and then to compute the sum of the first N integers.
8. Write a program to compute the sum of an arithmetic series. READ only the numbers A, D, N, and have the computer construct the sum of the series with N terms, starting with A, and increasing by D each time. I.e., the series is

$$A + (A + D) + (A + 2D) + \cdots + (A + (N - 1)D).$$

9. RUN the program of Exercise 7 for several values of N. Check that the answer is always $N(N + 1)/2$.
10. In BASIC, LOG(X) is the natural logarithm of X. Print a table of natural logarithms for the first ten integers.
11. RUN the program of Exercise 3 on a computer.
12. Write a program that computes the sum of the first N odd integers. RUN it for several values of N, and guess what the general formula for the sum is.
13. The technique described in Exercise 5 is not the best one, since when the number of DATA elements is changed, the loop must also be changed. This may be avoided by starting DATA with a single number that tells us how many times we have to go through the loop. Say this is N. Then we start our loop with

```
FOR K = 1 TO N.
```

Modify DIVIDE accordingly, and RUN it.
14. In the Land of Oz the calendar year has 534 days. Modify the programs BIRTHDAY and BIRTH2 accordingly.
    (a) How many people should we have in order to have a better-than-even chance of a coincidence? [*Ans.* 28.]
    (b) How many for a better-than-even chance of a multiple coincidence?

4 LISTS AND TABLES

In many applications we wish to work with an entire array of numbers at the same time. For this BASIC provides "lists" and "tables." A list can be used to store a sequence of numbers, while a table contains a two-dimensional array of numbers. We shall see in the next section that lists can also be used as vectors and tables also as matrices, in the sense of Chapter 4, allowing us to carry out matrix operations.

We have had previous arrays of numbers contained in our DATA statement. However, in each case we READ the numbers one or two at a time, and once we made use of them, we could afford to forget them. A list becomes important when the entire array must be remembered. For example, if we wish merely to read a sequence of numbers, multiply each one by 5, and print the results, there is no need to employ a list. However, an application as simple as reading a sequence of numbers and printing them out in the opposite order requires the use of a list. This is shown in the program BACK.

```
BACK

10 FOR I = 1 TO 8
20 READ L(I)
30 NEXT I
40 FOR I = 8 TO 1 STEP -1
50 PRINT L(I);
60 NEXT I
90 DATA 1,3,6,10,15,21,28,36
99 END
READY

RUN

BACK

 36   28   21   15   10   6   3   1

0.066 SEC.
READY
```

BASIC allows one list or table for each letter of the alphabet. For example, if the letter L is used to designate a list, then L(3) will stand for the third element of the list, while L(7) will stand for the seventh element. It would be more common mathematical notation to write these as L_3 and L_7. However, these cannot be typed on the devices one uses to communicate with computers. Lists and tables are nonetheless often referred to as "sub-

scripted variables" because of the more usual mathematical notations for them.

We have found it convenient in earlier chapters of this book to refer to an arbitrary element of a list of numbers by a notation such as L_i. The analog in BASIC is to write the formula L(I). Then as I runs through the numbers $1, 2, \ldots$, the quantity of L(I) will run through the various elements of the list. We take advantage of this possibility in the program BACK as lines 10–30 read the entire list of eight elements. The first time through the loop I equals 1 and therefore on line 20 we read L(1); thus the data element "1" on line 90 becomes the first element of the list. The second time I = 2 and therefore we read L(2) and thus the data element "3" becomes the second element of the list. Finally, the data element "36" will become L(8).

To print out the list in reverse order we can again employ a three-instruction loop. We want to print each L(I); however, we want $I = 8, 7, \ldots, 1$. Line 40 shows an additional flexibility of the FOR instruction. One can specify any step size by which the program proceeds. (If no step is specified, the computer assumes that the step size is 1.) In this case we specify STEP − 1; thus I = 8 the first time, then 7, then 6, etc.

The semicolon (;) on line 50 will assure that the numbers are printed one after the other without any extra space. If instead of a semicolon we had used a comma (,) the numbers would be printed in columns. If we had used no punctuation at the end of the line, each component would have been printed on a new line.

The program DICE computes the probability of winning in the game of craps (see Chapter 3, Section 11). We shall use the list P to store the probabilities for various possible sums when two dice are rolled. For example. P(5) will be the probability of shooting a 5, which we know to be $\frac{4}{36}$. Whenever a list is used in BASIC, space is automatically allocated in the computer for up to ten elements. Similarly for any table, BASIC will allocate for a table of size up to a 10×10. If larger lists and tables are desired, one must specify this through the use of a DIM or dimension statement. Thus in the program DICE we indicate the list P will have 12 elements. In case several different uses are contemplated for the same list, one must specify a DIM large enough to accommodate the longest list.

Lines 20–40 set up the probabilities for rolling a 2 through a 7. We leave to the reader the verification of these formulas. Lines 50 through 70 take advantage of the symmetry of the problem; e.g. the probability of an 8 is the same as the probability of a 6. At the end of this loop all the various probabilities for different totals on a single roll have been computed and the next step is to compute the probability W for winning in the game of craps.

You will recall that if a 7 or 11 turns up on the first roll, we win immediately. This is reflected in line 100. To this probability we must add the probability of "making our point." That is, if the initial roll is 4, 5, 6, 8, 9, or 10, then we must keep rolling until we either repeat that number (in

```
DICE

10 DIM P(12)
20 FOR K = 2 TO 7
30 LET P(K) = (K-1)/36
40 NEXT K
50 FOR K = 8 TO 12
60 LET P(K) = P(14-K)
70 NEXT K
100 LET W = P(7) + P(11)
110 FOR K = 4 TO 10
112 IF K=7 THEN 130
115 LET C = P(K)/(P(K)+P(7))
120 LET W = W + P(K)*C
130 NEXT K
170 PRINT W
180 PRINT 244/495
199 END
READY

RUN

DICE

 0.492929
 0.492929

0.076 SEC.
READY
```

which case we win) or until a 7 turns up.

This calculation is carried out in the loop on lines 110–130. Since our "point" may be 4, 5, 6, 8, 9, or 10, we allow the loop to run from 4 to 10. However we must exclude 7 as a possibility and this is the reason for line 112: if K is 7, we jump to the end of the loop—in other words, we eliminate this possibility. Line 115 computes the conditional probability that the number K will be repeated given that we shall get either K or a 7. This is the simplest way of computing the probability of getting a K before we get a 7. On line 120 we add to our previous winning probability the probability that we both have K as our initial point and that we win with it. By the time the loop is completed W will equal the probability of winning at craps. This is printed on line 170. We had calculated this probability in Chapter 3 as $\frac{244}{495}$ and we also print this quantity for comparison. We note from the RUN that the two answers are identical.

Let us now consider the use of tables. If T stands for a table, we must

indicate which row and which column in the table we are looking at. Thus T(3,5) will stand for the table entry in row 3 and column 5. As usual the arguments (or subscripts) may be variables. Thus T(I,J) will stand for the entry in row I and column J, which is more usually indicated by T_{ij}. As an illustration we shall recompute one of the tables previously computed in the book. This will be the table of binomial coefficients, usually known as the Pascal triangle. We shall compute the quantities $\binom{N}{J}$ for the values $N = 0, 1, 2, \ldots, 30$ and all possible values of J, namely $J = 0, 1, \ldots, N$. Only two facts are needed to compute the Pascal triangle. One is the fact that $\binom{N}{0} = \binom{N}{N} = 1$. The other is the fact that any entry "inside" the triangle is equal to the sum of the two entries immediately above it.

BINOMC

```
10 DIM B(30,30)
20 FOR N = 0 TO 30
30 LET B(N,0) = 1
40 LET B(N,N) = 1
50 FOR J = 1 TO N-1
60 LET B(N,J) = B(N-1,J-1) + B(N-1,J)
70 NEXT J
80 NEXT N
90
100 PRINT " N"," J","BINOM"
110 FOR K = 1 TO 4
120 READ N,J
130 PRINT N,J,B(N,J)
140 NEXT K
190 DATA 10,5,15,3,25,10,30,15
199 END
READY

RUN

BINOMC

 N             J             BINOM
 10            5              252
 15            3              455
 25            10             3268760
 30            15             1.55118 E+8

0.188 SEC.
READY
```

In the program BINOMC, line 10 saves enough space for a 30×30 table. The entry B(N,J) will stand for $\binom{N}{J}$. The entire calculation of the triangle is carried out on lines 20–80. We let N run from 0 through 30. For each given N we first fill in the $\binom{N}{0}$ and $\binom{N}{N}$ on lines 30 and 40. Then we start the loop on J in which J runs from 1 through $N-1$ to compute the "inside" entries. Line 60 simply states that a given entry of B is the sum of two entries on the previous row. Lines 70 and 80 close the two loops.

This is our first example of a "double loop." Such a double loop is legal as long as one loop is completely contained within the other one. The interpretation is very simple: the computer picks a first value for N and then runs through all the indicated values of J; it then picks the next value of N and repeats the procedure; and so on. Note that in this example the

BINOMPR

```
10 DIM B(30,30)
20 FOR N = 0 TO 30
30 LET B(N,0) = 1
40 LET B(N,N) = 1
50 FOR J = 1 TO N-1
60 LET B(N,J) = B(N-1,J-1) + B(N-1,J)
70 NEXT J
80 NEXT N
90
100 PRINT " N"," J"," P"," PROB."
110 FOR K = 1 TO 3
120 READ N,J,P
130 PRINT N,J,P,B(N,J)*P↑J*(1-P)↑(N-J)
140 NEXT K
190 DATA 10,5,.3,15,7,.4,30,15,.5
199 END
READY

RUN

BINOMPR

 N              J              P              PROB.
 10             5              0.3            0.102919
 15             7              0.4            0.177084
 30             15             0.5            0.144464

0.185 SEC.
READY
```

range of the second loop depends on the value of N in the first loop (see line 50). It is in this way that we fill out a triangle. If we were instead filling out a rectangle or a square, the possible values of J would not depend on the value N.

Just to show that the calculations are correct, we end up by printing four binomial coefficients. It is worth noting that the entire calculation of nearly 500 binomial coefficients—including some very large ones, as can be seen on the last line of the output—took only about two-tenths of a second. The same calculation by paper and pencil is a formidable task.

Some additional comments are in order. Line 90 is blank. This has no effect on the computations, but it separates the two major portions of the program for easier reading. On line 100 we PRINT appropriate labels. It should be noted that since we use commas both here and on line 130, the outputs automatically line up. The loop on lines 110–150 is employed to avoid the "OUT OF DATA" message.

Once we have binomial coefficients computed, they can be used for the solution of many kinds of problems. As an illustration we have included the program BINOMPR which computes binomial probabilities. Lines 10–90 are identical with the previous program since these simply compute the Pascal triangle. In the rest of the program we read the value of N, J, P, and compute the probability of precisely J successes in N trials with probability P for success on each trial. In line 130 we print N, J, and P and then compute and print the binomial probability by the well-known formula (see Chapter 3, Section 8). For example, the second line of the output shows that if we have 15 experiments with probability .4 for success on each experiment, then the probability of precisely 7 successes is about .177. (See previous page.)

EXERCISES

Only Exercises 8–12 require the use of a computer.

1. We wish to read N numbers from a DATA list and perform a task on them. Which of the following tasks require that the numbers be stored in a list?
   (a) Find the largest number.
   (b) Print the even-numbered entries.
   (c) Print first the even-numbered and then the odd-numbered entries.
   (d) Find the sum of the numbers.
   (e) Find the sum of the first and last entries.
   (f) Arrange the numbers in order. [*Ans.* (c) and (f).]
2. To use the same program for tables of different dimensions, one should read first the dimensions of the table, and then read the table. (Otherwise one does not know how many rows and columns to read.) Write such a program.
3. Write a program that will read a table and compute the row sums.
4. Write a program to read a list of four entries and a list of seven entries

and construct a table T so that T(I,J) is the product of the Ith entry of the first list and Jth entry of the second list.

5. Write a program to read a list and arrange the numbers in increasing order.
6. Modify the program of Exercise 5 to arrange the numbers in decreasing order.
7. Write a program to calculate the probability of winning in the dice game of Chapter 3, Section 11, Exercise 18.
8. Use the program DICE to compute the expected value of the game if $1 is bet each time.
9. Use DICE to compute the expected value of the game if we win $2 on 7 or 11, lose $3 on 2, 3, or 12, and win or lose $1 for making or failing to make our point.
10. Compute the row sums of the Pascal triangle for $N = 0, 1, \ldots, 10$. Use the binomial theorem to explain the results.
11. For $N = 0, 1, \ldots, 10$ compute the sum of $\binom{N}{J} 2^J$. Use the binomial theorem to explain the results.
12. This is an exercise in modular arithmetic. For $I,J = 1, 2, \ldots, 6$ let T(I,J) be I*J reduced by 7's. That is, if the product is 7 or greater, keep subtracting 7 until the result is less than 7. Print the table. What pattern do you observe?

5 VECTORS AND MATRICES

A natural use of lists and tables is to use them for vectors and matrices and to carry out matrix operations with them. BASIC recognizes this use by having a special set of instructions that enable one to carry out the matrix operation in a single step. We shall illustrate this by writing two programs for the addition of vectors, one not using the special instructions and one using them.

In the program VECADD the calculations are accomplished in four triples of instructions (loops). The first three instructions read a seven-component vector A, the second triple reads a similar vector B, and the third triple of instructions computes the vector sum letting the vector C stand for the answer. Finally, lines 100–120 print the answer.

VECADD

```
10 FOR I = 1 TO 7
20 READ A(I)
30 NEXT I
40 FOR I = 1 TO 7
50 READ B(I)
60 NEXT I
70 FOR I = 1 TO 7
```

```
80 LET C(I) = A(I) + B(I)
90 NEXT I
100 FOR I = 1 TO 7
110 PRINT C(I);
120 NEXT I
190 DATA 1,2,3,4,5,6,7
191 DATA 5,8,2,0,-1,-3,-7
199 END
READY

RUN

VECADD

 6   10   5   4   4   3   0

0.083 SEC.
READY
```

In the program VECADD2 we have replaced each triple of instructions (each loop) by a single special instruction. One signals to BASIC that an instruction is a special matrix instruction by starting with "MAT." The first two instructions each read a seven-component vector, the third instruction carries out the vector addition, and the fourth instruction prints the vector.

```
VECADD2

10 MAT READ A(7)
40 MAT READ B(7)
70 MAT C = A + B
100 MAT PRINT C;
190 DATA 1,2,3,4,5,6,7
191 DATA 5,8,2,0,-1,-3,-7
199 END
READY

RUN

VECADD2

 6   10   5   4   4   3   0

0.076 SEC.
READY
```

The comparison of the two programs will show exactly what the MAT instructions accomplish. Clearly the second program is simpler and shorter. The saving is even greater when we are dealing with a matrix or if we want to do more complicated matrix operations.

The next program, MATSUB, is similar to VECADD2 except that we are dealing with matrices and we perform a subtraction of two matrices. We have arranged the data for the two 3 × 4 matrices A and B on lines 50–52 and 60–62, respectively. This was done purely for the convenience of the

```
MATSUB

10 MAT READ A(3,4)
20 MAT READ B(3,4)
30 MAT C = A - B
40 MAT PRINT C;
49
50 DATA 1,2,3,4
51 DATA 5,6,7,8
52 DATA 7,6,5,4
59
60 DATA 4,3,2,1
61 DATA 0,-1,-2,-3
62 DATA -2,-3,-2,-1
69
99 END
READY

RUN

MATSUB


-3  -1   1   3
 5   7   9   11
 9   9   7   5

0.084 SEC.
READY
```

reader, so that he may easily check the computed answer. One could have listed the data all on one line, or divided it among several lines in an arbitrary way, as long as the data appeared in the order in which the program calls for it. However, it is usually good practice to arrange the data in a neat format for easier proofreading.

The next program, MATMPY, carries out a matrix multiplication. It reads

a 3×4 matrix A and a 4×2 matrix B and computes the product, a 3×2 matrix C = AB. The program MATMPY should be self-explanatory.

```
MATMPY

10 MAT READ A(3,4)
20 MAT READ B(4,2)
30 MAT C = A*B
40 MAT PRINT C;
49
50 DATA 1,2,3,4
51 DATA 5,6,7,8
52 DATA 7,6,5,4
59
60 DATA 2,1
61 DATA 3,2
62 DATA -1,0
63 DATA -3,-4
69
99 END
READY

RUN

MATMPY


-7 -11
-3 -15
 15   3

0.079 SEC.
READY
```

The single most powerful instruction in BASIC is the one-line command that inverts a matrix. This is illustrated on line 20 of the program MATINV. Line 10 reads a 3×3 matrix A. In line 20 we let $B = A^{-1}$. We then PRINT the inverse. The data in the program is taken from Example 2, Section 6 or Chapter 4. Naturally, we obtain the same result as in that section.

Since matrix inversion is available, it provides a convenient method of solving N equations in N unknowns. We know that we can write equations in the form AX = B. Here A contains the $n \times n$ matrix of coefficients of the left-hand sides of the equations while B is a vector containing the right-hand sides of the equations. The vector X contains the unknowns. If the matrix A has an inverse, then the solution may be written in the form $X = A^{-1}B$. This is carried out in the program EQU. To make the program

```
MATINV

10 MAT READ A(3,3)
20 MAT B = INV(A)
30 MAT PRINT B;
39
40 DATA 1,4,3
41 DATA 2,5,4
42 DATA 1,-3,-2
49
99 END
READY

RUN

MATINV


 2.  -1.   1.
 8.  -5.   2.
-11.   7.  -3.

0.082 SEC.
READY
```

more general, we first read the value of N. This will enable us to solve different numbers of equations in different numbers of unknowns by simply changing the data. We then read the matrix A and the vector B, compute the inverse of A, and compute the solution X on line 50. The answers are printed on line 60. The reader can easily verify that the solution is correct.

```
EQU

10 READ N
20 MAT READ A(N,N)
30 MAT READ B(N)
40 MAT I = INV(A)
50 MAT X = I*B
60 MAT PRINT X;
69
70 DATA 4
79
80 DATA 4,2,6,8
81 DATA 1,2,3,4
82 DATA 4,3,2,1
```

```
83 DATA 8,6,2,4
89
90 DATA -12,-5,5,12
99 END
READY

RUN

EQU


 1   2.  -2.  -1.

Ø.Ø87 SEC.
READY
```

If the reader has ever attempted to solve four equations in four unknowns, he will be happy to see that the same solution may be obtained on a computer in a small fraction of a second. Indeed, the same program will yield the solution of 50 equations in 50 unknowns in roughly 6 seconds! Although we illustrate programming techniques in terms of very simple examples, it is important to remember that the same techniques work on problems much too large to do by hand and often take only a few seconds to do on the computer.

One word of warning is in order for the last two programs. Not all matrices have inverses, and therefore one should really insert in the program a test as to whether the matrix does have an inverse. Such a test exists in BASIC but it is beyond the scope of our present treatment.

EXERCISES

Only Exercises 8–12 require the use of a computer.

1. Write a program that will add two matrices without using MAT instructions.
2. Write a program that will compute the tenth power of a square matrix P.
3. In BASIC, a vector of all 1's, say of five components, may be constructed by means of the instruction

   MAT X = CON(5).

   Write a program to read a 7×5 matrix A and compute AX (where X is the vector of all 1's). Interpret AX.
4. In BASIC a vector or a matrix may be multiplied by a number as follows:

   MAT Y = (5.2)*X.

Then Y = 5.2X. Write a program to read two five-component vectors X and Y and to compute
(a) 3Y.
(b) X + 2Y.
(c) 5.2X − 3.17Y.

5. Write a program that will read two 3×4 matrices A and B and compute
(a) A − 3B.
(b) 5.07A + 7.98B.
6. Write a program that will read a 3×3 matrix A, compute its inverse A^{-1}, and compute the product AA^{-1}.
7. Write a program that will check for two 4×4 matrices A and B whether AB = BA. It should print "YES" if they are equal.
8. Set up DATA for a 3×4 matrix A, a 4×2 matrix B, and a 2×3 matrix C. Compute:
(a) ABC.
(b) BCA.
(c) CAB.
(d) BAC.
9. Set up DATA for a 10×10 matrix all of whose components are zeroes or ones. Attempt to invert it. [*Hint:* Make sure that no row and no column consists entirely of zeroes.]
10. Use the computer to verify that $(AB)^{-1} = B^{-1}A^1$.
11. RUN the program of Exercise 6. How close is the final matrix to an identity matrix? (In general one expects some round-off errors.)
12. Try the program of Exercise 7 for several examples. Can you find examples where AB = BA?

6 APPLICATIONS TO MARKOV CHAINS

Let us illustrate the first theorem of Chapter 4, Section 7. It states that if P is a regular transition matrix, its powers approach a matrix whose rows are identical, each row being a probability vector with positive components. This can be illustrated by taking such a transition matrix and raising it to higher and higher powers. To speed up the process we shall square the matrix each time, so that we shall compute the powers P^2, P^4, P^8, P^{16}, and P^{32}.

In the program OZ we have chosen Exercise 12 in Section 7 of Chapter 4, dealing with the weather in the Land of Oz. We have arranged the transition matrix so that the first row corresponds to "nice," the next to "rain," and the final row to "snow." We first read the 3×3 transition matrix. Then in the loop in lines 15–50 we square this matrix five times. Line 20 carries out the actual squaring and line 30 prints the new matrix. On line 40 we let S take the place of P so that when we go through the loop again it is the new matrix that is squared.

Looking at the output we have a dramatic demonstration of the funda-

```
OZ

10 MAT READ P(3,3)
15 FOR T = 1 TO 5
20 MAT S = P*P
30 MAT PRINT S;
40 MAT P = S
50 NEXT T
90 DATA 0,.5,.5
91 DATA .25,.5,.25
92 DATA .25,.25,.5
99 END
READY

RUN

OZ


 0.25   0.375   0.375
 0.1875   0.4375   0.375
 0.1875   0.375   0.4375

 0.203125   0.398437   0.398437
 0.199219   0.402344   0.398437
 0.199219   0.398437   0.402344

 0.200012   0.399994   0.399994
 0.199997   0.400009   0.399994
 0.199997   0.399994   0.400009

 0.2   0.4   0.4
 0.2   0.4   0.4
 0.2   0.4   0.4

 0.2   0.4   0.4
 0.2   0.4   0.4
 0.2   0.4   0.4

0.139 SEC.
READY
```

mental theorem. By the third squaring—that is, when we look at P^8—the rows of the matrix are almost identical and have nearly assumed their limiting values. In the next two printed matrices, P^{16} and P^{32}, we see that, to the accuracy to which results are printed, we have the limiting probabilities of .2 for nice, .4 for rain, and .4 for snow.

We shall next turn to absorbing Markov chains as treated in Chapter 4, Section 8. We shall show how easy it is to compute the basic quantities N, T, and B on a computer.

In order to specify an absorbing chain, we need to know the number of transient states* (K) and the number of absorbing states (L). We need also to specify the two submatrices R and Q. This is accomplished in lines 10 and 20 of the program TRANS and in the DATA statements. The computation and printing of all the other quantities is accomplished in lines 30–90. Lines 30 and 40 illustrate two additional MAT commands. We can set up an identity matrix of specified size and a constant vector (vector of all 1's) in single instructions. We then let $D = I - Q$ and compute the inverse $N = (I - Q)^{-1}$. Similarly, we compute T and B. We have thus translated the three main theorems on absorbing chains into six instructions in BASIC. Finally, on line 90 we print the matrices N, T, and B. The data used in TRANS is taken from Example 1 of Section 8 in Chapter 4. The output may be compared with that example.

Of course, this is too small an example to make it worth using a computer.

*Nonabsorbing states are commonly called *transient* states.

TRANS

```
10 READ K,L
20 MAT READ R(K,L),Q(K,K)
25
30 MAT I = IDN(K,K)
40 MAT C = CON(K)
50 MAT D = I-Q
60 MAT N = INV(D)
70 MAT T = N*C
80 MAT B = N*R
90 MAT PRINT N,T,B
95
100 DATA 3,2
105
110 DATA .5,0
111 DATA 0,0
112 DATA 0,.5
115
120 DATA 0,.5,0
121 DATA .5,0,.5
122 DATA 0,.5,0
125
199 END
READY

RUN
```

```
TRANS

 1.5          1          0.5
 1            2          1
 0.5          1          1.5

 3            4          3

 0.75         0.25
 0.5          0.5
 0.25         0.75

0.116 SEC.
READY
```

Therefore we change the data to run a more substantial example. Our large example will be a random walk with ten transient states and with probability .7 of taking a step to the right. This chain is symbolically indicated in Figure 3. To find the fundamental matrix for this chain we have to invert a 10×10 matrix, and therefore we have a more substantial challenge for the computer. It is important to note that the main part of the program does not have to be changed at all. The only changes needed are in the DATA statements. (If there were more than ten transient states, one would also have to insert a DIM statement.) We also elected to omit the printing of the matrix N, since it is very large and not particularly interesting.

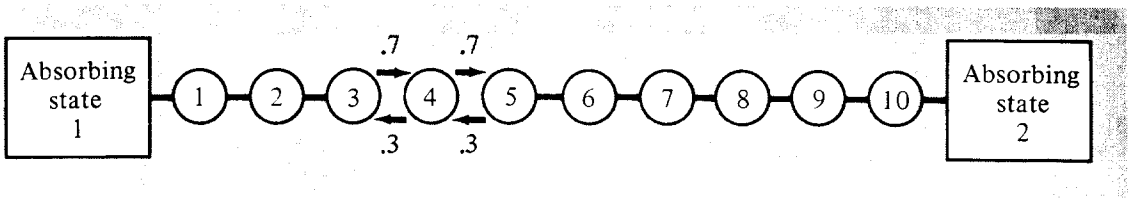


Figure 3

We show the new data statements and the run of the program TRANS2. Looking first at the second half of the output, the matrix B, we note that for most of the states we are almost certain to end up in the second absorbing state (i.e., at the right-hand end). It is surprising that even if one starts in transient state 1, way over on the left, one has a better-than-even chance of ending up on the right. The vector T, containing the expected number of steps before being absorbed, is also quite interesting. If we start near the right-hand endpoint, absorption takes place very fast, as may be expected. However, the result is not obvious on the left-hand side. For example, if one starts in transient state 1, there is probability .3 of being absorbed in a single step. On the other hand, it is more likely that absorption will take place at the right-hand endpoint, which will take a considerable

```
100 DATA 10,2
110 DATA .3,0
111 DATA 0,0
112 DATA 0,0
113 DATA 0,0
114 DATA 0,0
115 DATA 0,0
116 DATA 0,0
117 DATA 0,0
118 DATA 0,0
119 DATA 0,.7
120 DATA 0,.7,0,0,0,0,0,0,0,0
121 DATA .3,0,.7,0,0,0,0,0,0,0
122 DATA 0,.3,0,.7,0,0,0,0,0,0
123 DATA 0,0,.3,0,.7,0,0,0,0,0
124 DATA 0,0,0,.3,0,.7,0,0,0,0
125 DATA 0,0,0,0,.3,0,.7,0,0,0
126 DATA 0,0,0,0,0,.3,0,.7,0,0
127 DATA 0,0,0,0,0,0,.3,0,.7,0
128 DATA 0,0,0,0,0,0,0,.3,0,.7
129 DATA 0,0,0,0,0,0,0,0,.3,0
199 END

READY

RUN

TRANS2

13.22   17.45   17.84   16.57   14.60
12.33    9.93    7.47    4.99    2.50

0.4285    0.5715
0.1836    0.8164
0.0786    0.9214
0.0336    0.9664
0.0144    0.9856
0.0061    0.9939
0.0026    0.9974
0.0010    0.9990
0.0004    0.9996
0.0001    0.9999

0.288 SEC.
READY
```

amount of time. We find that the longest time to absorption, 17.84 steps, is from transient state 3.

For our final application in this section we consider a "Markov chain game." Any absorbing Markov chain can be turned into a game as follows. First assign a "value" to each absorbing state. A positive value may be interpreted as a prize one wins if one reaches this state while a negative value is a penalty to be paid if that state is reached. A player starts at a given transient state and moves from state to state in accordance with the transition probabilities of the Markov chain until an absorbing state is reached, where he receives a payoff or pays a penalty. The interesting question is what the expected payoff is for various different possible starting transient states. If the values are collected into a vector V with as many components as there are absorbing states, then it is very easy to see that the expected payoff for different starting states is given by the vector BV.

As an illustration we shall consider a game based on a two-dimensional random walk as shown in Figure 4. There are ten transient states. From

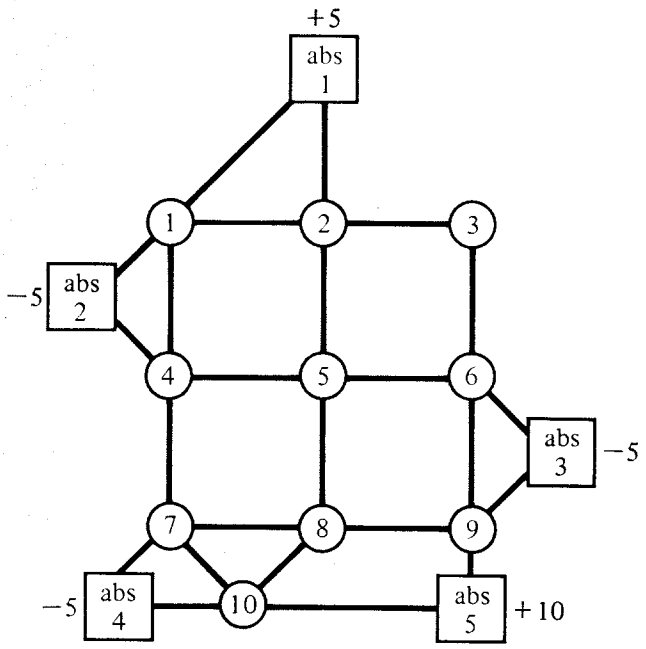


Figure 4

each of these the player moves to any of the states to which it is connected with equal probabilities. Thus from state 4 there is probability $\frac{1}{4}$ of moving to states 1, 5, or 7 or to absorbing state 2. There are five absorbing states, and one receives a prize of \$5 at the first one and \$10 at the last one and loses \$5 at the other three absorbing states. The payoffs balance out, but it is intuitively clear that it is advantageous to start at some states and disadvantageous to start at others. However, if one is offered the chance to play this game with state 5 as the starting state, should one accept?

To solve this problem we have modified the program TRANS by changing

the data and by adding three additional lines. Line 25 will read the vector V from the data in line 130. And line 81 computes the payoff vector P as B*V. Also, we print only the vector P. When we run the resulting program TRANS3 we find that there are some favorable starting states—notably state 2, where the expected payoff is $1.08—and some highly unfavorable states, the worst being state 4, where the expected loss is $1.78. We find

```
25 MAT READ V(L)

81 MAT P = B*V

100 DATA 10,5
110 DATA .25,.25,0,0,0
111 DATA .25,0,0,0,0
112 DATA 0,0,0,0,0
113 DATA 0,.25,0,0,0
114 DATA 0,0,0,0,0
115 DATA 0,0,.25,0,0
116 DATA 0,0,0,.25,0
117 DATA 0,0,0,0,0
118 DATA 0,0,.25,0,.25
119 DATA 0,0,0,.25,.25
120 DATA 0,.25,0,.25,0,0,0,0,0,0
121 DATA .25,0,.25,0,.25,0,0,0,0,0
122 DATA 0,.5,0,0,0,.5,0,0,0,0
123 DATA .25,0,0,0,.25,0,.25,0,0,0
124 DATA 0,.25,0,.25,0,.25,0,.25,0,0
125 DATA 0,0,.25,0,.25,0,0,0,.25,0
126 DATA 0,0,0,.25,0,0,0,.25,0,.25
127 DATA 0,0,0,0,.25,0,.25,0,.25,.25
128 DATA 0,0,0,0,0,.25,0,.25,0,0
129 DATA 0,0,0,0,0,0,.25,.25,0,0
130 DATA +5,-5,-5,-5,+10

READY

RUN

TRANS3

-0.17   +1.08   -0.02   -1.78   -0.46
-1.13   -1.48   -0.03   +0.96   +0.87

0.292 SEC.
READY
```

that in state 5 one almost but not quite breaks even. There is an overall expected loss of 46 cents; thus one should *not* agree to play the game starting at state 5.

Although we gave this application an interpretation as a game, there are many other interpretations of a Markov chain game. There are processes in nature that are described by absorbing Markov chains where one can in a natural way assign a "value" to ending up at a given terminal. Here the expected payoff has a natural interpretation. There is also a method for computing voltages in a simple electric circuit using this technique. (See *Finite Mathematics with Business Applications.*) In recent years Markov chains have acquired considerable importance in applications to many sciences. It is therefore interesting to see how easy it is to compute fundamental quantities for Markov chains by means of a high-speed computer.

EXERCISES

Only Exercises 5–12 require the use of a computer.

1. Vectors in BASIC are column vectors, but a matrix of one row may be used as a row vector. Write a program to read a row vector and a column vector, of four components each, and to compute their product.
2. If A is probability row vector, and if it is repeatedly multiplied on the right by a regular transition matrix P, it will approach the fixed vector. (See Chapter 4, Section 7.) Write a program to carry out this process.
3. For a transient chain, N may be computed as the sum of the infinite series

$$N = I + Q + Q^2 + Q^3 + \ldots .$$

Write a program to compute the first 21 terms of this series.

4. Write a program which, for an absorbing Markov chain, will compute Q and R from N and B. [*Hint:* Compute N^{-1}.]
5. Compute powers of the transition matrix

$$P = \begin{pmatrix} .1 & .2 & .3 & .4 \\ .4 & .3 & .2 & .1 \\ .3 & .1 & .4 & .2 \\ .2 & .4 & .1 & .3 \end{pmatrix}$$

to find the fixed vector.

6. Compute powers of the transition matrix

$$P = \begin{pmatrix} 0 & .3 & 0 & .7 \\ .5 & 0 & .5 & 0 \\ 0 & .6 & 0 & .4 \\ .2 & 0 & .8 & 0 \end{pmatrix}$$

and explain the result.

7. Let h be an arbitrary column vector of three components. Multiply it repeatedly by the OZ transition matrix and observe that it tends to a constant vector. Interpret the constant.
8. Try out the program of Exercise 2 for the Land of Oz.
9. RUN the program of Exercise 4 to verify that it produces Q and R correctly.
10. Apply the program TRANS to Exercise 6 of Chapter 4, Section 8.
11. Modify the program TRANS to verify the identity $QB = B - R$.
12. Design your own Markov chain game and compute the expected values for various starting positions.

7 LINEAR EQUATIONS

The purpose of this section is to translate the flow diagram of Figure 5 in Section 5 of Chapter 4 for the solution of linear equations into a computer program. We shall first do this in a straightforward manner and show that the program reproduces the results of Chapter 4, Section 5. However, we shall then note that the program is inadequate; this will give us an opportunity to consider one of the deeper problems of computer programming—namely, the question of numerical accuracy. We shall assume that there is no variable all of whose coefficients are 0.

The program LINEQU is designed to follow Figure 5 (Chapter 4). Boxes 1–6 correspond to blocks of instructions starting at lines 100, 200, 300, 400, 500, and 600. Box 7 is combined with box 2, and box 8 with box 5. For easy identification each block of instructions starts with a REM (or "remark") statement. Such a REM statement is for the convenience of the programmer and is ignored by the computer. LINEQU is designed to solve M equations in N unknowns. We start by saving space for our list and table, reading M and N, and then reading the tableau T of coefficients. It should be noted that T has N + 1 columns since it contains not only the coefficients of the left-hand side of the equations but also the numbers on the right-hand side.

In the remainder of the program I and J will be the subscripts corresponding to the pivot. The auxiliary variables I1 and J1 are used as running subscripts for rows and columns. The process consists of choosing a pivot in each row and operating with it; this loop starts at line 100 and ends at line 500. Lines 200–220 search for a nonzero element in row I. If such an element is found, we jump to line 300. It should be noted that as we jump out of the loop of lines 200–220, the subscript J is correctly set for the pivot. If we complete the entire loop, then the left-hand side of the equation is zero. In line 230 we check whether the right-hand side is also zero. If it is not, we jump to line 900 and type out "THERE IS NO SOLUTION."

At line 300 we note what the pivot is. We also put into the list P the subscript of the variable we pivoted on. This will make it much easier to identify the solution of the problem. It should be noted that if the equation was identically equal to zero, and hence could be ignored, then at line 240

LINEQU

```
5  DIM T(20,21),P(20)
10 READ M,N
20 MAT READ T(M,N+1)
29
100 REM  START MAIN LOOP
110 FOR I = 1 TO M
120
200 REM  FIND PIVOT
205 FOR J = 1 TO N
210 IF T(I,J) <> 0 THEN 300
220 NEXT J
230 IF T(I,N+1) <> 0 THEN 900
240 LET P(I) = 0
250 GOTO 500
260
300 REM  DIVIDE BY PIVOT
302 LET P = T(I,J)
305 LET P(I) = J
310 FOR J1 = 1 TO N+1
320 LET T(I,J1) = T(I,J1)/P
330 NEXT J1
340
400 REM  SUBTRACT MULTIPLES OF ROW
405 FOR I1 = 1 TO M
410 IF I1 = I THEN 460
420 LET C = T(I1,J)
430 FOR J1 = 1 TO N+1
440 LET T(I1,J1) = T(I1,J1) - C*T(I,J1)
450 NEXT J1
460 NEXT I1
470
500 REM  CLOSE MAIN LOOP
510 NEXT I
520
600 REM  PRINT ANSWERS
605 FOR I = 1 TO M
610 LET P = P(I)
620 IF P = 0 THEN 790
625 LET B = T(I,N+1)
630 PRINT "X";STR$(P);" = ";STR$(B);
640 FOR J = 1 TO N
645 IF J = P THEN 690
650 LET C = T(I,J)
660 IF C = 0 THEN 690
```

```
665 IF C<Ø THEN 68Ø
67Ø PRINT " - ";
675 GOTO 687
68Ø LET C = -C
685 PRINT " + ";
687 PRINT STR$(C);"*";"X";STR$(J);
69Ø NEXT J
7ØØ PRINT
79Ø NEXT I
8ØØ GOTO 999
9ØØ PRINT "THERE IS NO SOLUTION."
9Ø5

91Ø DATA 3,3
92Ø DATA 1,4,3,1
921 DATA 2,5,4,4
922 DATA 1,-3,-2,5
999 END

READY

RUN

LINEQU

X1 = 3.
X2 = -2.
X3 = 2.

Ø.166 SEC.
READY
```

we entered a zero into the list of pivots. Lines 310–330 complete this particular box of the flow diagram by dividing all coefficients in this row of the tableau by the pivot P.

We must now subtract suitable multiples of the pivotal row from the other rows. This is accomplished in lines 400–460. The subscript I1 will run through all the rows; however, on line 410 we make sure that we skip over the pivotal row. C is equated to the appropriate multiplier and the subtraction is carried out in the loop in lines 430–450. When this double loop is completed, on line 500 we go on to the next row. It should be noted that the entire heart of the program is contained in lines 100–500, a total of only 20 instructions in BASIC.

The answers are printed in the loop of lines 600–790. This piece of code could be much simpler except for two complications: First, we want to have a nice format for the answer. Second, we want to handle not only the case

of a unique solution but find all possible solutions in case there are infinitely many of them. To obtain a nice-looking form for the answers, we need to introduce an additional feature of BASIC. (It should be noted that there are many other advanced features of BASIC not covered in this book.) When BASIC prints the numerical value of a variable, it either starts with a minus sign or a blank and places a blank after the number. This is very convenient when we simply want to print a list of numbers one after the other. However, it spoils the output when we want to print, for example, "X5." But writing the string command "STR$(P)" will print a numerical value of P without initial or trailing blanks. (This command is not available in all versions of BASIC.)

If the solution were always unique, the output would be accomplished by the six instructions in lines 600–630 and 790. We look at each equation once, and look up in the list of pivots what the subscript P of the pivot was. If this is 0, then the equation is identically 0 and therefore can be ignored. Otherwise B is the value of the variable and, on line 630, we print out an answer that may look like "X5 = 3.2." Here is where we see the advantage of having remembered the element we pivoted on. Its coefficient ends up being 1, and hence the right-hand side of the equation is its value.

The loop in lines 640–690 handles the case of infinitely many solutions. If the solution is not unique, then variables other than the pivot are left over with nonzero coefficients. These variables may be given arbitrary values. We usually indicate this by "bringing the variable to the right-hand side." Thus we search in the loop to see whether any variable other than the pivot has a nonzero coefficient. If it does, we print it with a suitable coefficient after the value we have already printed on the right-hand side. It should be noted that we had to test on line 665 whether the coefficient was positive or negative and the two cases are treated separately. It is left as an exercise for the reader to step through this part of the program by hand.

```
920 DATA 1,-2,-3,2
921 DATA 1,-4,-13,14
922 DATA -3,5,4,0

READY

RUN

 LINEQU2

X1 = -10 - 7*X3
X2 = -6 - 5*X3

0.153 SEC.
READY
```

The data in LINEQU is taken from Example 1 in Section 5 of Chapter 4. The printed answer agrees with the answer found earlier.

To show that the program also works in the case of infinitely many solutions or no solutions, we change the data to those of Examples 2 and 3 (of Section 5 in Chapter 4) respectively. These are shown in LINEQU2 and LINEQU3. It should be noted that the output format for the case of infinitely many solutions is very easily readable. It shows that X3 may take on any value and it indicates what the corresponding values of X1 and X2 must be.

```
922 DATA -3,5,4,2

READY

RUN

■ LINEQU3

THERE IS NO SOLUTION.

Ø.167 SEC.
READY
```

Next we try out the program with a larger data base. In LINEQU4 we have four equations in five unknowns. The result seems reasonable; indeed,

```
91Ø DATA 4,5
92Ø DATA 3,2,1,2,3,11
921 DATA -1,-1,-2,1,1,-6
922 DATA 1,2,3,4,5,3
923 DATA 2,2,Ø,8,1Ø,2

READY

RUN

■ LINEQU4

X1 = 4.5 + 3.5*X4
X2 = -1. - 7.5*X4
X3 = 1 + 2.5*X4
X5 = -Ø.5

Ø.179 SEC.
READY
```

if we check it all the indicated solutions are correct. However, a more careful check will show that we have failed to find all the solutions! We have run into one of the subtleties of computer programming: a program that to all appearances is correct produces incorrect results. The problem is one of round-off errors. When this happens, one must do troubleshooting on the computer program, or as it is commonly phrased, one must "debug" it. A very useful procedure is to ask the computer to print out not just the final solution but also the intermediate steps. This may be accomplished by replacing line 500 by:

```
500  MAT PRINT T;
```

With this change the tableau will be printed after each iteration. A look at the output would indicate that something went wrong between the third and fourth iterations. The last line in the third iteration would appear as follows:

$$2 \times 10^{-7} \times X5 = -1 \times 10^{-7}.$$

From this the computer concludes that X5 equals -0.5. However, the very small numbers appearing in this equation represent round-off errors and should actually be zero. The reason for round-off errors is the fact that a computer can only carry a fraction to a limited number of decimal places (usually six to nine). Furthermore, it works with a number system to base 2. Whether a rounding is necessary depends on the base. Therefore one must anticipate that in hand calculations where no round-off error appears, one may appear on the computer, or vice versa. In this particular case this very minute round-off error changes the whole nature of the solution. The equation should actually be 0 = 0, and therefore X5 should be available as a variable whose value may be chosen arbitrarily. Therefore, due to a minute error, we lost infinitely many available solutions.

The lesson that we learn is that if a variable's value was computed through complicated calculation, we cannot assume that a 0 will come out to be exactly 0. This forces us to modify lines 210, 230, and 660. We shall assume that any sufficiently small number is produced by a round-off error and should really be a 0. Of course, the question is just what does "sufficiently small" mean? This is a deep and difficult question, and there is no universally satisfactory answer to it. For any proposed solution to this problem one can find a set of equations for which the program will produce the wrong results. We shall, however, show one quite common solution to this dilemma that will handle "normal" cases. Our assumption will be that any coefficient that turns out to be less than 10^{-6} is a round-off error and should be 0.

Thus on line 660 instead of asking whether the coefficient C is equal to 0, we shall ask whether its absolute value is less than 10^{-6}. In BASIC one computes the absolute value of C by writing "ABS(C)." The corresponding corrections must also be made on line 210 and line 230. We show these corrections and the corrected run in LINEQU5. This time we have found all the solutions to the problem.

```
210 IF ABS(T(I,J)) > 1E-6 THEN 300

230 IF ABS(T(I,N+1)) > 1E-6 THEN 900

660 IF ABS(C) < 1E-6 THEN 690

READY

RUN

 LINEQU5

X1 = 6.5 + 3.5*X4 + 4.*X5
X2 = -5.5 - 7.5*X4 - 9.*X5
X3 = 2.5 + 2.5*X4 + 3.*X5

 0.202 SEC.
READY
```

The resulting program is of quite general use in solving linear equations. The exercises will show modifications of this program that may be used for other purposes—e.g., inverting a matrix. We again see that a relatively short BASIC program, and surprisingly short computing times, can solve important practical problems.

This section has also given the reader a first taste of the complex field of finding numerical solutions to mathematical problems. This field is known as *numerical analysis*. It treats the wide variety of difficulties one runs into in finding numerical solutions, and also searches for the most efficient numerical methods of solving a variety of problems.

EXERCISES

Only Exercises 4–10 require the use of a computer.

1. Modify the program LINEQU5 to solve simultaneously two sets of equations with identical left sides but different right sides.
2. If the coefficients of the left side of a set of equations form an $n \times n$ matrix A, and if one successfully pivots on every row (no left side becomes identically zero), then A^{-1} exists. This is true irrespective of the right side of the equation. Modify LINEQU5 to serve as a test of whether a given square matrix has an inverse.
3. Write a program to invert a square matrix using the method of Section 6 in Chapter 4.
4. Use LINEQU5 to solve the equations of Exercise 5 in Section 5 of Chapter 4.

5. Use LINEQU5 to solve the following equations:

$$\begin{aligned} 4X_1 - 3X_2 + 2X_3 - X_4 + 5X_5 &= -10 \\ X_1 - 2X_2 + 3X_3 + X_4 - X_5 &= 12 \\ 2X_1 + X_2 - X_3 + 2X_4 + 5X_5 &= -1 \\ 3X_1 - 2X_2 + X_3 - X_4 + 2X_5 &= -6 \\ 2X_1 - X_2 + 2X_3 - X_4 + X_5 &= 0. \end{aligned}$$

[*Ans.* 1, 2, 3, 4, −2.]

6. Use LINEQU5 on Exercise 12 of Section 5 in Chapter 4 for several values of k. Check the answer there given.
7. RUN the program of Exercise 1 using the DATA of LINEQU2 and LINEQU3.
8. Apply the program of Exercise 2 to the matrices in Exercise 3 of Section 6 in Chapter 4.
9. Apply the program of Exercise 3 to the matrices in Exercise 1 of Section 6 in Chapter 4.
10. Use the program of Exercise 3 to invert the following matrix:

$$A = \begin{pmatrix} .9 & -.1 & -.2 & -.3 & -.1 \\ -.2 & .8 & -.1 & 0 & -.2 \\ -.2 & -.2 & .7 & -.1 & -.1 \\ -.1 & 0 & -.2 & .8 & -.3 \\ -.1 & -.3 & -.2 & -.1 & .9 \end{pmatrix}.$$

SUGGESTED READING

Barrodale, Ian, et al. *Elementary Computer Applications.* New York: John Wiley, 1971.

Gross, Jonathan L., and Brainerd, Walter S. *Fundamental Programming Concepts.* New York: Harper and Row, 1972.

Kemeny, John G., A. S. Schleifer, Jr., J. Laurie Snell, and Gerald L. Thompson. *Finite Mathematics with Business Applications.* 2nd ed. Englewood Cliffs, N.J.: Prentice-Hall, 1972.

Kemeny, John G., and Kurtz, Thomas E. *Basic Programming.* 2nd ed. New York: John Wiley, 1972.