

Programming and Software

In the previous chapter, we used a net force to develop a mathematical model to predict the fall velocity of a parachutist. This model took the form of a differential equation,

$$\frac{dv}{dt} = g - \frac{c}{m}v$$

We also learned that a solution to this equation could be obtained by a simple numerical approach called Euler's method,

$$v_{i+1} = v_i + \frac{dv_i}{dt} \Delta t$$

Given an initial condition, this equation can be implemented repeatedly to compute the velocity as a function of time. However, to obtain good accuracy, many small steps must be taken. This would be extremely laborious and time-consuming to implement by hand. However, with the aid of the computer, such calculations can be performed easily.

So our next task is to figure out how to do this. The present chapter will introduce you to how the computer is used as a tool to obtain such solutions.

2.1 PACKAGES AND PROGRAMMING

Today, there are two types of software users. On one hand, there are those who take what they are given. That is, they limit themselves to the capabilities found in the software's standard mode of operation. For example, it is a straightforward proposition to solve a system of linear equations or to generate a plot of x - y values with either Excel or MATLAB software. Because this usually involves a minimum of effort, most users tend to adopt this "vanilla" mode of operation. In addition, since the designers of these packages anticipate most typical user needs, many meaningful problems can be solved in this way.

But what happens when problems arise that are beyond the standard capability of the tool? Unfortunately, throwing up your hands and saying, "Sorry boss, no can do!" is not acceptable in most engineering circles. In such cases, you have two alternatives.

First, you can look for a different package and see if it is capable of solving the problem. That is one of the reasons we have chosen to cover both Excel and MATLAB in this book. As you will see, neither one is all encompassing and each has different strengths.

By being conversant with both, you will greatly increase the range of problems you can address.

Second, you can grow and become a “power user” by learning to write Excel VBA¹ macros or MATLAB M-files. And what are these? They are nothing more than computer programs that allow you to extend the capabilities of these tools. Because engineers should never be content to be tool limited, they will do whatever is necessary to solve their problems. A powerful way to do this is to learn to write programs in the Excel and MATLAB environments. Furthermore, the programming skills required for macros and M-files are the same as those needed to effectively develop programs in languages like Fortran 90 or C.

The major goal of the present chapter is to show you how this can be done. However, we do assume that you have been exposed to the rudiments of computer programming. Therefore, our emphasis here is on facets of programming that directly affect its use in engineering problem solving.

2.1.1 Computer Programs

Computer programs are merely a set of instructions that direct the computer to perform a certain task. Since many individuals write programs for a broad range of applications, most high-level computer languages, like Fortran 90 and C, have rich capabilities. Although some engineers might need to tap the full range of these capabilities, most merely require the ability to perform engineering-oriented numerical calculations.

Looked at from this perspective, we can narrow down the complexity to a few programming topics. These are:

- Simple information representation (constants, variables, and type declarations).
- Advanced information representation (data structure, arrays, and records).
- Mathematical formulas (assignment, priority rules, and intrinsic functions).
- Input/output.
- Logical representation (sequence, selection, and repetition).
- Modular programming (functions and subroutines).

Because we assume that you have had some prior exposure to programming, we will not spend time on the first four of these areas. At best, we offer them as a checklist that covers what you will need to know to implement the programs that follow.

However, we will devote some time to the last two topics. We emphasize logical representation because it is the single area that most influences an algorithm’s coherence and understandability. We include modular programming because it also contributes greatly to a program’s organization. In addition, modules provide a means to archive useful algorithms in a convenient format for subsequent applications.

2.2 STRUCTURED PROGRAMMING

In the early days of computer, programmers usually did not pay much attention to whether their programs were clear and easy to understand. Today, it is recognized that there are many benefits to writing organized, well-structured code. Aside from the obvious benefit of making software much easier to share, it also helps generate much more efficient

¹VBA is the acronym for Visual Basic for Applications.

program development. That is, well-structured algorithms are invariably easier to debug and test, resulting in programs that take a shorter time to develop, test, and update.

Computer scientists have systematically studied the factors and procedures needed to develop high-quality software of this kind. In essence, *structured programming* is a set of rules that prescribe good style habits for the programmer. Although structured programming is flexible enough to allow considerable creativity and personal expression, its rules impose enough constraints to render the resulting codes far superior to unstructured versions. In particular, the finished product is more elegant and easier to understand.





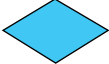



A key idea behind structured programming is that any numerical algorithm can be composed using the three fundamental control structures: sequence, selection, and repetition. By limiting ourselves to these structures, the resulting computer code will be clearer and easier to follow.

In the following paragraphs, we will describe each of these structures. To keep this description generic, we will employ flowcharts and pseudocode. A *flowchart* is a visual or graphical representation of an algorithm. The flowchart employs a series of blocks and arrows, each of which represents a particular operation or step in the algorithm (Fig. 2.1). The arrows represent the sequence in which the operations are implemented.

Not everyone involved with computer programming agrees that flowcharting is a productive endeavor. In fact, some experienced programmers do not advocate flowcharts. However, we feel that there are three good reasons for studying them. First, they are still used for expressing and communicating algorithms. Second, even if they are not employed routinely, there will be times when they will prove useful in planning, unraveling, or communicating the logic of your own or someone else's program. Finally, and most important for our purposes, they are excellent pedagogical tools. From a teaching perspective, they

FIGURE 2.1

Symbols used in flowcharts.

SYMBOL	NAME	FUNCTION
	Terminal	Represents the beginning or end of a program.
	Flowlines	Represents the flow of logic. The humps on the horizontal arrow indicate that it passes over and does not connect with the vertical flowlines.
	Process	Represents calculations or data manipulations.
	Input/output	Represents inputs or outputs of data and information.
	Decision	Represents a comparison, question, or decision that determines alternative paths to be followed.
	Junction	Represents the confluence of flowlines.
	Off-page connector	Represents a break that is continued on another page.
	Count-controlled loop	Used for loops which repeat a prespecified number of iterations.

are ideal vehicles for visualizing some of the fundamental control structures employed in computer programming.

An alternative approach to express an algorithm that bridges the gap between flowcharts and computer code is called *pseudocode*. This technique uses code-like statements in place of the graphical symbols of the flowchart. We have adopted some style conventions for the pseudocode in this book. Keywords such as IF, DO, INPUT, etc., are capitalized, whereas the conditions, processing steps, and tasks are in lowercase. Additionally, the processing steps are indented. Thus the keywords form a “sandwich” around the steps to visually define the extent of each control structure.

One advantage of pseudocode is that it is easier to develop a program with it than with a flowchart. The pseudocode is also easier to modify and share with others. However, because of their graphic form, flowcharts sometimes are better suited for visualizing complex algorithms. In the present text, we will use flowcharts for pedagogical purposes. Pseudocode will be our principal vehicle for communicating algorithms related to numerical methods.

2.2.1 Logical Representation

Sequence. The sequence structure expresses the trivial idea that unless you direct it otherwise, the computer code is to be implemented one instruction at a time. As in Fig. 2.2, the structure can be expressed generically as a flowchart or as pseudocode.

Selection. In contrast to the step-by-step sequence structure, selection provides a means to split the program’s flow into branches based on the outcome of a logical condition. Figure 2.3 shows the two most fundamental ways for doing this.

The single-alternative decision, or *IF/THEN* structure (Fig. 2.3*a*), allows for a detour in the program flow if a logical condition is true. If it is false, nothing happens and the program moves directly to the next statement following the *ENDIF*. The double-alternative decision, or *IF/THEN/ELSE* structure (Fig. 2.3*b*), behaves in the same manner for a true condition. However, if the condition is false, the program implements the code between the *ELSE* and the *ENDIF*.

FIGURE 2.2

(*a*) Flowchart and
(*b*) pseudocode for the
sequence structure.

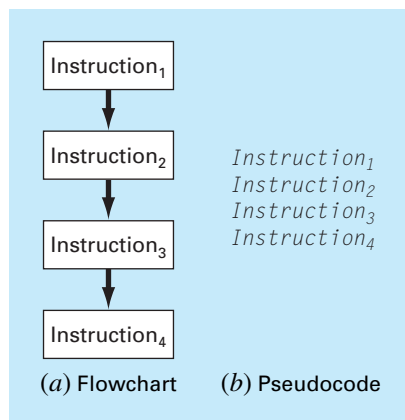
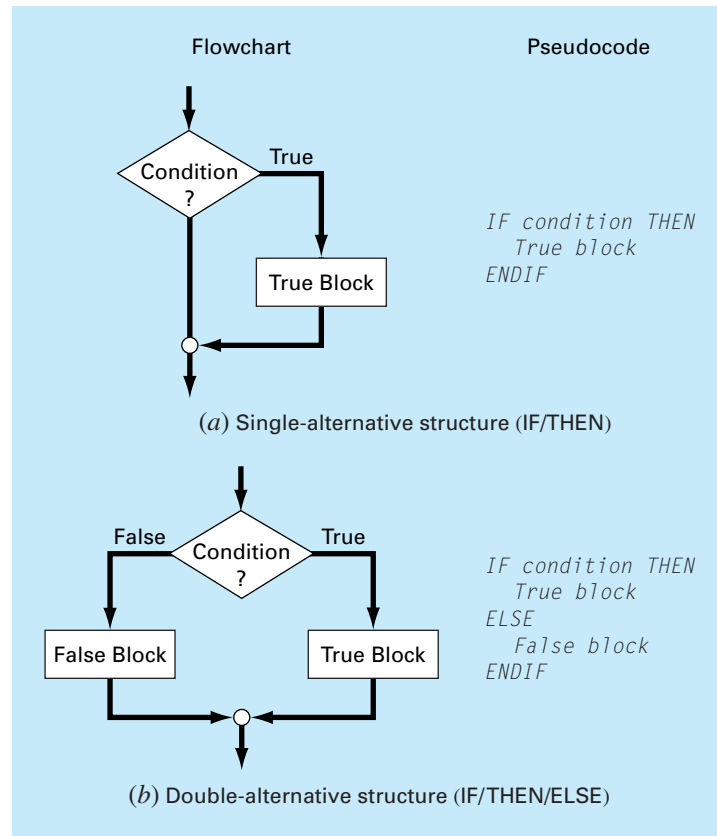


FIGURE 2.3

Flowchart and pseudocode for simple selection constructs. (a) Single-alternative selection (IF/THEN) and (b) double-alternative selection (IF/THEN/ELSE).

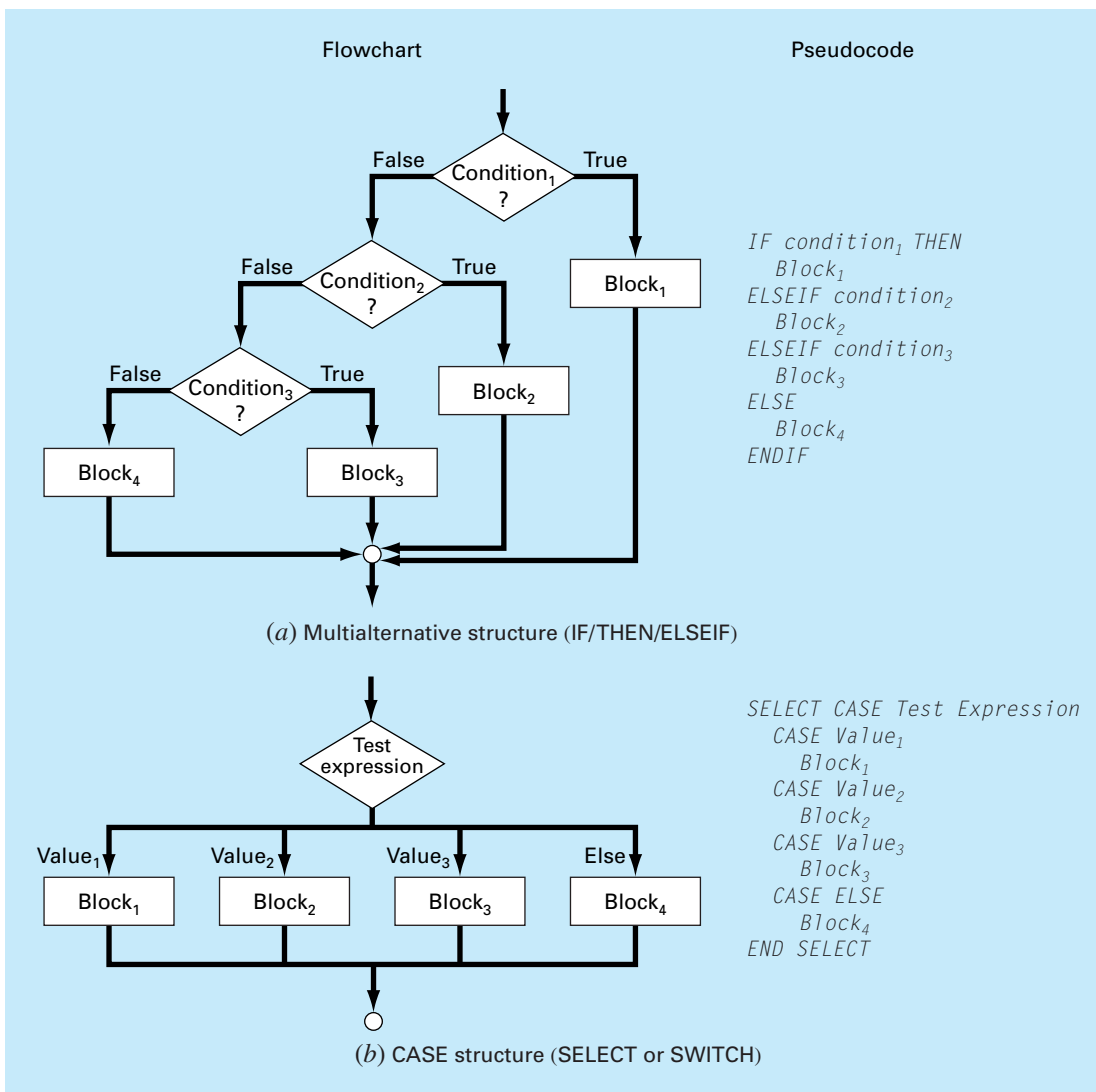


Although the IF/THEN and the IF/THEN/ELSE constructs are sufficient to construct any numerical algorithm, two other variants are commonly used. Suppose that the ELSE clause of an IF/THEN/ELSE contains another IF/THEN. For such cases, the ELSE and the IF can be combined in the *IF/THEN/ELSEIF* structure shown in Fig. 2.4a.

Notice how in Fig. 2.4a there is a chain or “cascade” of decisions. The first one is the IF statement, and each successive decision is an ELSEIF statement. Going down the chain, the first condition encountered that tests true will cause a branch to its corresponding code block followed by an exit of the structure. At the end of the chain of conditions, if all the conditions have tested false, an optional ELSE block can be included.

The *CASE* structure is a variant on this type of decision making (Fig. 2.4b). Rather than testing individual conditions, the branching is based on the value of a single *test expression*. Depending on its value, different blocks of code will be implemented. In addition, an optional block can be implemented if the expression takes on none of the prescribed values (CASE ELSE).

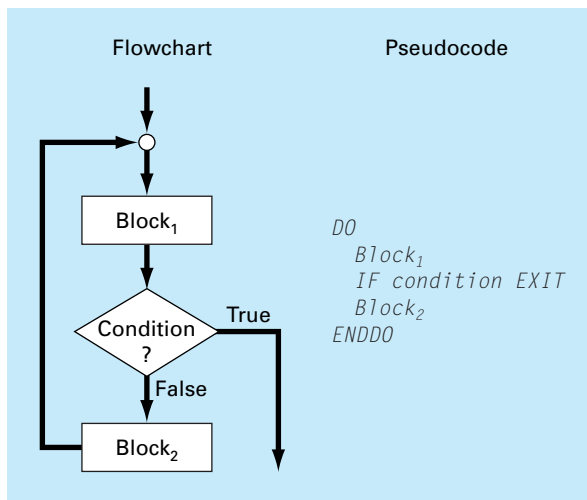
Repetition. Repetition provides a means to implement instructions repeatedly. The resulting constructs, called *loops*, come in two “flavors” distinguished by how they are terminated.

**FIGURE 2.4**

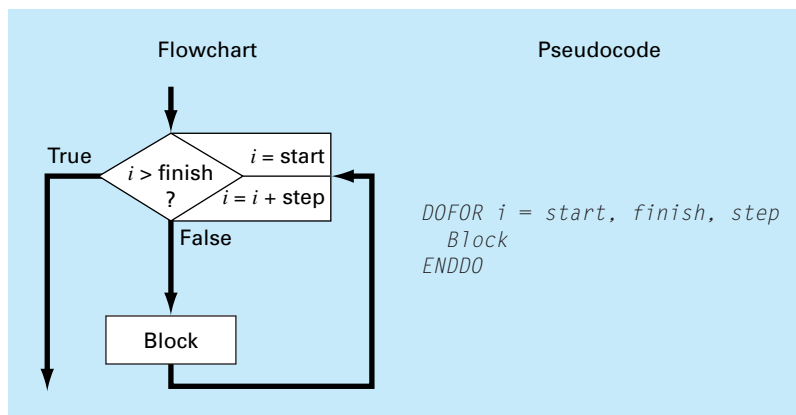
Flowchart and pseudocode for supplementary selection or branching constructs. (a) Multiple-alternative selection (IF/THEN/ELSEIF) and (b) CASE construct.

The first and most fundamental type is called a *decision loop* because it terminates based on the result of a logical condition. Figure 2.5 shows the most generic type of decision loop, the *DOEXIT construct*, also called a *break loop*. This structure repeats until a logical condition is true.

It is not necessary to have two blocks in this structure. If the first block is not included, the structure is sometimes called a *pretest loop* because the logical test is performed before anything occurs. Alternatively, if the second block is omitted, it is called a *posttest loop*.

**FIGURE 2.5**

The DOEXIT or break loop.

**FIGURE 2.6**

The count-controlled or DOFOR construct.

Because both blocks are included, the general case in Fig. 2.5 is sometimes called a *midtest loop*.

It should be noted that the DOEXIT was introduced in Fortran 90 in an effort to simplify decision loops. This control construct is a standard part of the Excel VBA macro language but is not standard in C or MATLAB, which use the so-called WHILE structure. Because we believe that the DOEXIT is superior, we have adopted it as our decision loop structure throughout this book. In order to ensure that our algorithms are directly implemented in both MATLAB and Excel, we will show how the break loop can be simulated with the WHILE structure later in this chapter (see Sec. 2.5).

The break loop in Fig. 2.5 is called a logical loop because it terminates on a logical condition. In contrast, a *count-controlled* or *DOFOR loop* (Fig. 2.6) performs a specified number of repetitions, or iterations.

The count-controlled loop works as follows. The index (represented as i in Fig. 2.6) is a variable that is set at an initial value of *start*. The program then tests whether the *index* is

less than or equal to the final value, *finish*. If so, it executes the body of the loop, and then cycles back to the DO statement. Every time the ENDDO statement is encountered, the *index* is automatically increased by the *step*. Thus the index acts as a counter. Then, when the *index* is greater than the final value (*finish*), the computer automatically exits the loop and transfers control to the line following the ENDDO statement. Note that for nearly all computer languages, including those of Excel and MATLAB, if the *step* is omitted, the computer assumes it is equal to 1.²

The numerical algorithms outlined in the following pages will be developed exclusively from the structures outlined in Figs. 2.2 through 2.6. The following example illustrates the basic approach by developing an algorithm to determine the roots for the quadratic formula.

EXAMPLE 2.1

Algorithm for Roots of a Quadratic

Problem Statement. The roots of a quadratic equation

$$ax^2 + bx + c = 0$$

can be determined with the quadratic formula,

$$\begin{aligned} x_1 &= \frac{-b \pm \sqrt{|b^2 - 4ac|}}{2a} \\ x_2 & \end{aligned} \quad (2.1)$$

Develop an algorithm that does the following:

-
- Step 1: Prompts the user for the coefficients, *a*, *b*, and *c*.
 - Step 2: Implements the quadratic formula, guarding against all eventualities (for example, avoiding division by zero and allowing for complex roots).
 - Step 3: Displays the solution, that is, the values for *x*.
 - Step 4: Allows the user the option to return to step 1 and repeat the process.
-

Solution. We will use a top-down approach to develop our algorithm. That is, we will successively refine the algorithm rather than trying to work out all the details the first time around.

To do this, let us assume for the present that the quadratic formula is foolproof regardless of the values of the coefficients (obviously not true, but good enough for now). A structured algorithm to implement the scheme is

```

DO
  INPUT a, b, c
  r1 = (-b + SQRT(b2 - 4ac))/(2a)
  r2 = (-b - SQRT(b2 - 4ac))/(2a)
  DISPLAY r1, r2
  DISPLAY 'Try again? Answer yes or no'
  INPUT response
  IF response = 'no' EXIT
ENDDO

```

²A negative step can be used. In such cases, the loop terminates when the index is less than the final value.

A DOEXIT construct is used to implement the quadratic formula repeatedly as long as the condition is false. The condition depends on the value of the character variable *response*. If *response* is equal to 'yes' the calculation is implemented. If not, that is, *response* = 'no' the loop terminates. Thus, the user controls termination by inputting a value for *response*.

Now although the above algorithm works for certain cases, it is not foolproof. Depending on the values of the coefficients, the algorithm might not work. Here is what can happen:

- If $a = 0$, an immediate problem arises because of division by zero. In fact, close inspection of Eq. (2.1) indicates that two different cases can arise. That is,
 - If $b \neq 0$, the equation reduces to a linear equation with one real root, $-c/b$.
 - If $b = 0$, then no solution exists. That is, the problem is trivial.
- If $a \neq 0$, two possible cases occur depending on the value of the discriminant, $d = b^2 - 4ac$. That is,
 - If $d \geq 0$, two real roots occur.
 - If $d < 0$, two complex roots occur.

Notice how we have used indentation to highlight the decisional structure that underlies the mathematics. This structure then readily translates to a set of coupled IF/THEN/ELSE structures that can be inserted in place of the shaded statements in the previous code to give the final algorithm:

```

DO
  INPUT a, b, c
  r1 = 0: r2 = 0: i1 = 0: i2 = 0
  IF a = 0 THEN
    IF b ≠ 0 THEN
      r1 = -c/b
    ELSE
      DISPLAY "Trivial solution"
    ENDIF
  ELSE
    discr = b2 - 4 * a * c
    IF discr ≥ 0 THEN
      r1 = (-b + Sqrt(discr))/(2 * a)
      r2 = (-b - Sqrt(discr))/(2 * a)
    ELSE
      r1 = -b/(2 * a)
      r2 = r1
      i1 = Sqrt(Abs(discr))/(2 * a)
      i2 = -i1
    ENDIF
  ENDIF
  DISPLAY r1, r2, i1, i2
  DISPLAY 'Try again? Answer yes or no'
  INPUT response
  IF response = 'no' EXIT
ENDDO

```

The approach in the foregoing example can be employed to develop an algorithm for the parachutist problem. Recall that, given an initial condition for time and velocity, the problem involved iteratively solving the formula

$$v_{i+1} = v_i + \frac{dv_i}{dt} \Delta t \quad (2.2)$$

Now also remember that if we desired to attain good accuracy, we would need to employ small steps. Therefore, we would probably want to apply the formula repeatedly from the initial time to the final time. Consequently, an algorithm to solve the problem would be based on a loop.

For example, suppose that we started the computation at $t = 0$ and wanted to predict the velocity at $t = 4$ s using a time step of $\Delta t = 0.5$ s. We would, therefore, need to apply Eq. (2.2) eight times, that is,

$$n = \frac{4}{0.5} = 8$$

where n = the number of iterations of the loop. Because this result is exact, that is, the ratio is an integer, we can use a count-controlled loop as the basis for the algorithm. Here is an example of the pseudocode:

```

g = 9.8
INPUT cd, m
INPUT ti, vi, tf, dt
t = ti
v = vi
n = (tf - ti) / dt
DOFOR i = 1 TO n
  dvdt = g - (cd / m) * v
  v = v + dvdt * dt
  t = t + dt
ENDDO
DISPLAY v

```

Although this scheme is simple to program, it is not foolproof. In particular, it will work only if the computation interval is evenly divisible by the time step.³ In order to cover such cases, a decision loop can be substituted in place of the shaded area in the previous pseudocode. The final result is

```

g = 9.8
INPUT cd, m
INPUT ti, vi, tf, dt
t = ti
v = vi

```

³This problem is compounded by the fact that computers use base-2 number representation for their internal math. Consequently, some apparently evenly divisible numbers do not yield integers when the division is implemented on a computer. We will cover this in Chap. 3.

```
h = dt
DO
  IF t + dt > tf THEN
    h = tf - t
  ENDF
  dvdt = g - (cd / m) * v
  v = v + dvdt * h
  t = t + h
  IF t ≥ tf EXIT
ENDDO
DISPLAY v
```

As soon as we enter the loop, we use an IF/THEN structure to test whether adding $t + dt$ will take us beyond the end of the interval. If it does not, which would usually be the case at first, we do nothing. If it does, we would need to shorten the interval by setting the variable step h to $tf - t$. By doing this, we guarantee that the next step falls exactly on tf . After we implement this final step, the loop will terminate because the condition $t \geq tf$ will test true.

Notice that before entering the loop, we assign the value of the time step, dt , to another variable, h . We create this dummy variable so that our routine does not change the given value of dt if and when we shorten the time step. We do this in anticipation that we might need to use the original value of dt somewhere else in the event that this code is integrated within a larger program.

It should be noted that the algorithm is still not foolproof. For example, the user could have mistakenly entered a step size greater than the calculation interval, for example, $tf - ti = 5$ and $dt = 20$. Thus, you might want to include error traps in your code to catch such errors and to then allow the user to correct the mistake.

2.3 MODULAR PROGRAMMING

Imagine how difficult it would be to study a textbook that had no chapters, sections, or paragraphs. Breaking complicated tasks or subjects into more manageable parts is one way to make them easier to handle. In the same spirit, computer programs can be divided into small subprograms, or modules, that can be developed and tested separately. This approach is called *modular programming*.

The most important attribute of modules is that they be as independent and self-contained as possible. In addition, they are typically designed to perform a specific, well-defined function and have one entry and one exit point. As such, they are usually short (generally 50 to 100 instructions in length) and highly focused.

In standard high-level languages such as Fortran 90 or C, the primary programming element used to represent each module is the procedure. A procedure is a series of computer instructions that together perform a given task. Two types of procedures are commonly employed: *functions* and *subroutines*. The former usually returns a single result, whereas the latter returns several.

In addition, it should be mentioned that much of the programming related to software packages like Excel and MATLAB involves the development of subprograms. Hence,

Excel macros and MATLAB functions are designed to receive some information, perform a calculation, and return results. Thus, modular thinking is also consistent with how programming is implemented in package environments.

Modular programming has a number of advantages. The use of small, self-contained units makes the underlying logic easier to devise and to understand for both the developer and the user. Development is facilitated because each module can be perfected in isolation. In fact, for large projects, different programmers can work on individual parts. Modular design also increases the ease with which a program can be debugged and tested because errors can be more easily isolated. Finally, program maintenance and modification are facilitated. This is primarily due to the fact that new modules can be developed to perform additional tasks and then easily incorporated into the already coherent and organized scheme.

While all these attributes are reason enough to use modules, the most important reason related to numerical engineering problem solving is that they allow you to maintain your own library of useful modules for later use in other programs. This will be the philosophy of this book: All the algorithms will be presented as modules.

This approach is illustrated in Fig. 2.7 which shows a function developed to implement Euler's method. Notice that this function application and the previous versions differ in how they handle input/output. In the former versions, input and output directly come from (via INPUT statements) and to (via DISPLAY statements) the user. In the function, the inputs are passed into the FUNCTION via its argument list

Function Euler(dt, ti, tf, yi)

and the output is returned via the assignment statement

y = Euler(dt, ti, tf, yi)

In addition, recognize how generic the routine has become. There are no references to the specifics of the parachutist problem. For example, rather than calling the dependent

FIGURE 2.7

Pseudocode for a function that solves a differential equation using Euler's method.

```

FUNCTION Euler(dt, ti, tf, yi)
  t = ti
  y = yi
  h = dt
DO
  IF t + dt > tf THEN
    h = tf - t
  ENDF
  dydt = dy(t, y)
  y = y + dydt * h
  t = t + h
  IF t ≥ tf EXIT
ENDDO
Euler = y
END

```

variable v for velocity, the more generic label, y , is used within the function. Further, notice that the derivative is not computed within the function by an explicit equation. Rather, another function, dy , must be invoked to compute it. This acknowledges the fact that we might want to use this function for many different problems beyond solving for the parachutist's velocity.

2.4 EXCEL

Excel is the spreadsheet produced by Microsoft, Inc. Spreadsheets are a special type of mathematical software that allow the user to enter and perform calculations on rows and columns of data. As such, they are a computerized version of a large accounting worksheet on which large interconnected calculations can be implemented and displayed. Because the entire calculation is updated when any value on the sheet is changed, spreadsheets are ideal for “what if?” sorts of analysis.

Excel has some built-in numerical capabilities including equation solving, curve fitting, and optimization. It also includes VBA as a macro language that can be used to implement numerical calculations. Finally, it has several visualization tools, such as graphs and three-dimensional surface plots, that serve as valuable adjuncts for numerical analysis. In the present section, we will show how these capabilities can be used to solve the parachutist problem.

To do this, let us first set up a simple spreadsheet. As shown below, the first step involves entering labels and numbers into the spreadsheet cells.

	A	B	C	D
1	Parachutist Problem			
2				
3	m	68.1 kg		
4	cd	12.5 kg/s		
5	dt	0.1 s		
6				
7	t	vnum (m/s)	vanal (m/s)	
8	0	0.000		
9	2			

Before we write a macro program to calculate the numerical value, we can make our subsequent work easier by attaching names to the parameter values. To do this, select cells A3:B5 (the easiest way to do this is by moving the mouse to A3, holding down the left mouse button and dragging down to B5). Next, make the menu selection

Insert Name Create Left column OK

To verify that this has worked properly, select cell B3 and check that the label “m” appears in the name box (located on the left side of the sheet just below the menu bars).

Move to cell C8 and enter the analytical solution (Eq. 1.9),

$$=9.8*m/cd*(1-\exp(-cd/m*A8))$$

When this formula is entered, the value 0 should appear in cell C8. Then copy the formula down to cell C9 to give a value of 16.405 m/s.

All the above is typical of the standard use of Excel. For example, at this point you could change parameter values and see how the analytical solution changes.

Now, we will illustrate how VBA macros can be used to extend the standard capabilities. Figure 2.8 lists pseudocode alongside Excel VBA code for all the control structures described in the previous section (Figs. 2.2 through 2.6). Notice how, although the details differ, the structure of the pseudocode and the VBA code are identical.

We can now use some of the constructs from Fig. 2.8 to write a macro function to numerically compute velocity. Open VBA by selecting⁴

```
Tools Macro Visual Basic Editor
```

Once inside the *Visual Basic Editor* (VBE), select

```
Insert Module
```

and a new code window will open up. The following VBA function can be developed directly from the pseudocode in Fig. 2.7. Type it into the code window.

```
Option Explicit
```

```
Function Euler(dt, ti, tf, yi, m, cd)
```

```
Dim h As Single, t As Single, y As Single, dydt As Single
```

```
t = ti
```

```
y = yi
```

```
h = dt
```

```
Do
```

```
    If t + dt > tf Then
```

```
        h = tf - t
```

```
    End If
```

```
    dydt = dy(t, y, m, cd)
```

```
    y = y + dydt * h
```

```
    t = t + h
```

```
    If t >= tf Then Exit Do
```

```
Loop
```

```
Euler = y
```

```
End Function
```

Compare this macro with the pseudocode from Fig. 2.7 and recognize how similar they are. Also, see how we have expanded the function's argument list to include the necessary parameters for the parachutist velocity model. The resulting velocity, v , is then passed back to the spreadsheet via the function name.

Also notice how we have included another function to compute the derivative. This can be entered in the same module by typing it directly below the Euler function,

```
Function dy(t, v, m, cd)
```

```
Const g As Single = 9.8
```

```
dy = g - (cd / m) * v
```

```
End Function
```

⁴The hot key combination Alt-F11 is even quicker!

(a) Pseudocode	(b) Excel VBA
IF/THEN:	
<i>IF condition THEN</i> <i>True block</i> <i>ENDIF</i>	If b <> 0 Then r1 = -c / b End If
IF/THEN/ELSE:	
<i>IF condition THEN</i> <i>True block</i> <i>ELSE</i> <i>False block</i> <i>ENDIF</i>	If a < 0 Then b = Sqr(Abs(a)) Else b = Sqr(a) End If
IF/THEN/ELSEIF:	
<i>IF condition₁ THEN</i> <i>Block₁</i> <i>ELSEIF condition₂</i> <i>Block₂</i> <i>ELSEIF condition₃</i> <i>Block₃</i> <i>ELSE</i> <i>Block₄</i> <i>ENDIF</i>	If class = 1 Then x = x + 8 ElseIf class < 1 Then x = x - 8 ElseIf class < 10 Then x = x - 32 Else x = x - 64 End If
CASE:	
<i>SELECT CASE Test Expression</i> <i>CASE Value₁</i> <i>Block₁</i> <i>CASE Value₂</i> <i>Block₂</i> <i>CASE Value₃</i> <i>Block₃</i> <i>CASE ELSE</i> <i>Block₄</i> <i>END SELECT</i>	Select Case a + b Case Is < -50 x = -5 Case Is < 0 x = -5 - (a + b) / 10 Case Is < 50 x = (a + b) / 10 Case Else x = 5 End Select
DOEXIT:	
<i>DO</i> <i>Block₁</i> <i>IF condition EXIT</i> <i>Block₂</i> <i>ENDIF</i>	Do i = i + 1 If i >= 10 Then Exit Do j = i*x Loop
COUNT-CONTROLLED LOOP:	
<i>DOFOR i = start, finish, step</i> <i>Block</i> <i>ENDDO</i>	For i = 1 To 10 Step 2 x = x + i Next i

FIGURE 2.8

The fundamental control structures in (a) pseudocode and (b) Excel VBA.

The final step is to return to the spreadsheet and invoke the function by entering the following formula in cell B9

```
=Euler(dt,A8,A9,B8,m,cd)
```

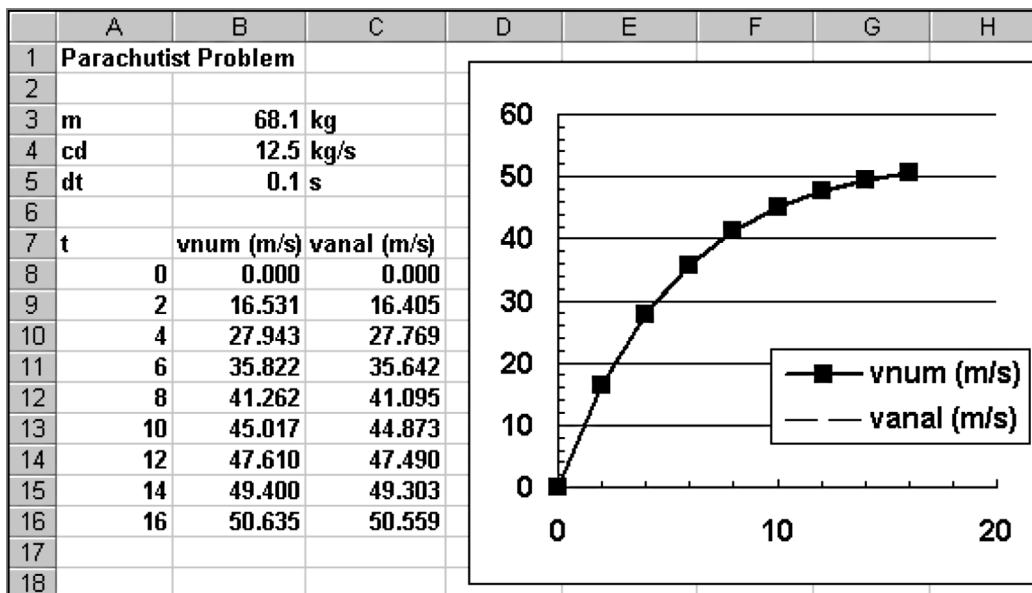
The result of the numerical integration, 16.531, will appear in cell B9.

You should appreciate what has happened here. When you enter the function into the spreadsheet cell, the parameters are passed into the VBA program where the calculation is performed and the result is then passed back and displayed in the cell. In effect, the VBA macro language allows you to use Excel as your input/output mechanism. All sorts of benefits arise from this fact.

For example, now that you have set up the calculation, you can play with it. Suppose that the jumper was much heavier, say, $m = 100$ kg (about 220 pounds). Enter 100 into cell B3 and the spreadsheet will update immediately to show a value of 17.438 in cell B9. Change the mass back to 68.1 kg and the previous result, 16.531, automatically reappears in cell B9.

Now let us take the process one step further by filling in some additional numbers for the time. Enter the numbers 4, 6, . . . 16 in cells A10 through A16. Then copy the formulas from cells B9:C9 down to rows 10 through 16. Notice how the VBA program calculates the numerical result correctly for each new row. (To verify this, change dt to 2 and compare with the results previously computed by hand in Example 1.2.) An additional embellishment would be to develop an x - y plot of the results using the Excel Chart Wizard.

The final spreadsheet is shown below. We now have created a pretty nice problem-solving tool. You can perform sensitivity analyses by changing the values for each of the parameters. As each new value is entered, the computation and the graph would be automatically updated. It is this interactive nature that makes Excel so powerful. However, recognize that the ability to solve this problem hinges on being able to write the macro with VBA.



It is the combination of the Excel environment with the VBA programming language that truly opens up a world of possibilities for engineering problem solving. In the coming chapters, we will illustrate how this is accomplished.

2.5 MATLAB

MATLAB is the flagship software product of The MathWorks, Inc., which was cofounded by the numerical analysts Cleve Moler and John N. Little. As the name implies, MATLAB was originally developed as a *matrix laboratory*. To this day, the major element of MATLAB is still the matrix. Mathematical manipulations of matrices are very conveniently implemented in an easy-to-use, interactive environment. To these matrix manipulations, MATLAB has added a variety of numerical functions, symbolic computations, and visualization tools. As a consequence, the present version represents a fairly comprehensive technical computing environment.

MATLAB has a variety of functions and operators that allow convenient implementation of many of the numerical methods developed in this book. These will be described in detail in the individual chapters that follow. In addition, programs can be written as so-called *M-files* that can be used to implement numerical calculations. Let us explore how this is done.

First, you should recognize that normal MATLAB use is closely related to programming. For example, suppose that we wanted to determine the analytical solution to the parachutist problem. This could be done with the following series of MATLAB commands

```
>> g=9.8;  
>> m=68.1;  
>> cd=12.5;  
>> tf=2;  
>> v=g*m/cd*(1-exp(-cd/m*tf))
```

with the result being displayed as

```
v =  
    16.4050
```

Thus, the sequence of commands is just like the sequence of instructions in a typical programming language.

Now what if you want to deviate from the sequential structure. Although there are some neat ways to inject some nonsequential capabilities in the standard command mode, the inclusion of decisions and loops is best done by creating a MATLAB document called an M-file. To do this, make the menu selection

```
File New Mfile
```

and a new window will open with a heading “MATLAB Editor/Debugger.” In this window, you can type and edit MATLAB programs. Type the following code there:

```
g=9.8;  
m=68.1;  
cd=12.5;  
tf=2;  
v=g*m/cd*(1-exp(-cd/m*tf))
```

Notice how the commands are written in exactly the way as they would be written in the front end of MATLAB. Save the program with the name: `analpara`. MATLAB will automatically attach the extension `.m` to denote it as an M-file: `analpara.m`.

To run the program, you must go back to the command mode. The most direct way to do this is to click on the “MATLAB Command Window” button on the task bar (which is usually at the bottom of the screen).

The program can now be run by typing the name of the M-file, `analpara`, which should look like

```
>> analpara
```

If you have done everything correctly, MATLAB should respond with the correct answer:

```
v =  
    16.4050
```

Now one problem with the foregoing is that it is set up to compute one case only. You can make it more flexible by having the user input some of the variables. For example, suppose that you wanted to assess the impact of mass on the velocity at 2 s. The M-file could be rewritten as the following to accomplish this

```
g=9.8;  
m=input('mass (kg) : ');  
cd=12.5;  
tf=2;  
v=g*m/cd*(1-exp(-cd/m*tf))
```

Save this as `analpara2.m`. If you typed `analpara2` while being in command mode, the prompt would show

```
mass (kg) :
```

The user could then enter a value like 100, and the result will be displayed as

```
v =  
    17.3420
```

Now it should be pretty clear how we can program a numerical solution with an M-file. In order to do this, we must first understand how MATLAB handles logical and looping structures. Figure 2.9 lists pseudocode alongside MATLAB code for all the control structures from the previous section. Although the structures of the pseudocode and the MATLAB code are very similar, there are some slight differences that should be noted.

In particular, look at how we have represented the DOEXIT structure. In place of the DO, we use the statement WHILE(1). Because MATLAB interprets the number 1 as corresponding to “true,” this statement will repeat infinitely in the same manner as the DO statement. The loop is terminated with a *break* command. This command transfers control to the statement following the *end* statement that terminates the loop.

(a) Pseudocode	(b) MATLAB
IF/THEN:	
<i>IF condition THEN</i> <i>True block</i> <i>ENDIF</i>	<code>if b ~= 0 r1 = -c / b; end</code>
IF/THEN/ELSE:	
<i>IF condition THEN</i> <i>True block</i> <i>ELSE</i> <i>False block</i> <i>ENDIF</i>	<code>if a < 0 b = sqrt(abs(a)); else b = sqrt(a); end</code>
IF/THEN/ELSEIF:	
<i>IF condition₁ THEN</i> <i>Block₁</i> <i>ELSEIF condition₂</i> <i>Block₂</i> <i>ELSEIF condition₃</i> <i>Block₃</i> <i>ELSE</i> <i>Block₄</i> <i>ENDIF</i>	<code>if class == 1 x = x + 8; elseif class < 1 x = x - 8; elseif class < 10 x = x - 32; else x = x - 64; end</code>
CASE:	
<i>SELECT CASE Test Expression</i> <i>CASE Value₁</i> <i>Block₁</i> <i>CASE Value₂</i> <i>Block₂</i> <i>CASE Value₃</i> <i>Block₃</i> <i>CASE ELSE</i> <i>Block₄</i> <i>END SELECT</i>	<code>switch a + b case 1 x = -5; case 2 x = -5 - (a + b) / 10; case 3 x = (a + b) / 10; otherwise x = 5; end</code>
DOEXIT:	
<i>DO</i> <i>Block₁</i> <i>IF condition EXIT</i> <i>Block₂</i> <i>ENDIF</i>	<code>while (1) i = i + 1; if i >= 10, break, end j = i*x; end</code>
COUNT-CONTROLLED LOOP:	
<i>DOFOR i = start, finish, step</i> <i>Block</i> <i>ENDDO</i>	<code>for i = 1:2:10 x = x + i; end</code>

FIGURE 2.9

The fundamental control structures in (a) pseudocode and (b) the MATLAB programming language.

Also notice that the parameters of the count-controlled loop are ordered differently. For the pseudocode, the loop parameters are specified as *start*, *finish*, *step*. For MATLAB, the parameters are ordered as *start:step:finish*.

The following MATLAB M-file can now be developed directly from the pseudocode in Fig. 2.7. Type it into the MATLAB Editor/Debugger:

```
g=9.8;
m=input('mass (kg):');
cd=12.5;
ti=0;
tf=2;
vi=0;
dt=0.1;
t = ti;
v = vi;
h = dt;
while (1)
    if t + dt > tf
        h = tf - t;
    end
    dvdt = g - (cd / m) * v;
    v = v + dvdt * h;
    t = t + h;
    if t >= tf, break, end
end
disp('velocity (m/s):')
disp(v)
```

Save this file as `numpara.m` and return to the command mode and run it by entering: `numpara`. The following output should result:

```
mass (kg): 100

velocity (m/s):
    17.4381
```

As a final step in this development, let us take the above M-file and convert it into a proper function. This can be done in the following M-file based on the pseudocode from Fig. 2.7

```
function euler = f(dt,ti,tf,yi,m,cd)
t = ti;
y = yi;
h = dt;
while (1)
    if t + dt > tf
        h = tf - t;
    end
    dydt = dy(t, y, m, cd);
    y = y + dydt * h;
    t = t + h;
    if t >= tf, break, end
end
euler = y;
```

Save this file as euler.m and then create another M-file to compute the derivative,

```
function dy = f(t, v, m, cd)
g = 9.8;
dy = g - (cd / m) * v;
```

Save this file as dy.m and return to the command mode. In order to invoke the function and see the result, you can type in the following commands

```
>> m=68.1;
>> cd=12.5;
>> ti=0;
>> tf=2.;
>> vi=0;
>> dt=0.1;
>> euler(dt,ti,tf,vi,m,cd)
```

When the last command is entered, the answer will be displayed as

```
ans =
16.5309
```

It is the combination of the MATLAB environment with the M-file programming language that truly opens up a world of possibilities for engineering problem solving. In the coming chapters we will illustrate how this is accomplished.

2.6 OTHER LANGUAGES AND LIBRARIES

In the previous sections, we showed how Excel and MATLAB function procedures for Euler's method could be developed from an algorithm expressed as pseudocode. You should recognize that similar functions can be written in high-level languages like Fortran 90 and C++. For example, a Fortran 90 function for Euler's method is

```
Function Euler(dt, ti, tf, yi, m, cd)

REAL dt, ti, tf, yi, m, cd
Real h, t, y, dydt

t = ti
y = yi
h = dt
Do
  If (t + dt > tf) Then
    h = tf - t
  End If
  dydt = dy(t, y, m, cd)
  y = y + dydt * h
  t = t + h
  If (t >= tf) Exit
End Do
Euler = y
End Function
```

For C, the result would look quite similar to the MATLAB function. The point is that once a well-structured algorithm is developed in pseudocode form, it can be readily implemented in a variety of programming environments.

In this book, our approach will be to provide you with well-structured procedures written as pseudocode. This collection of algorithms then constitutes a numerical library that can be accessed to perform specific numerical tasks in a range of software tools and programming languages.

Beyond your own programs, you should be aware that commercial programming libraries contain many useful numerical procedures. For example, the *Numerical Recipe* library includes a large range of algorithms written in Fortran and C.⁵ These procedures are described in both book (for example, Press et al. 1992) and electronic form.

For Fortran, the *IMSL* (*International Mathematical and Statistical Library*) provides over 700 procedures spanning all the numerical areas covered in this text. Because of the widespread use of Fortran in engineering, we include IMSL applications throughout the book.

PROBLEMS

2.1 Write pseudocode to implement the flowchart depicted in Fig. P2.1. Make sure that proper indentation is included to make the structure clear.

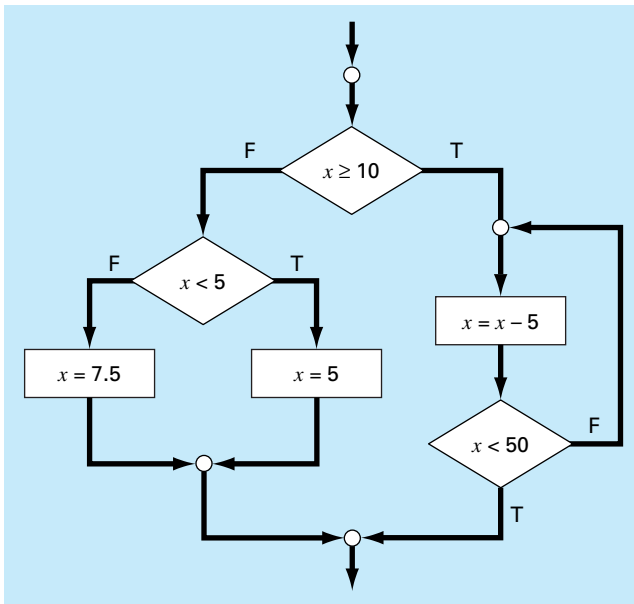


Figure P2.1

2.2 Rewrite the following pseudocode using proper indentation

```

DO
  i = i + 1
  IF z > 50 EXIT
  x = x + 5
  IF x > 5 THEN
    y = x
  ELSE
    y = 0
  ENDIF
  z = x + y
ENDDO

```

2.3 A value for the concentration of a pollutant in a lake is recorded on each card in a set of index cards. A card marked “end of data” is placed at the end of the set. Write an algorithm to determine the sum, the average, and the maximum of these values.

2.4 Write a structured flowchart for Prob. 2.3.

2.5 Develop, debug, and document a program to determine the roots of a quadratic equation, $ax^2 + bx + c$, in either a high-level language or a macro language of your choice. Use a subroutine procedure to compute the roots (either real or complex). Perform test runs for the cases (a) $a = 1, b = 6, c = 2$; (b) $a = 0, b = -4, c = 1.6$; (c) $a = 3, b = 2.5, c = 7$.

2.6 The cosine function can be evaluated by the following infinite series:

$$\cos x = 1 - \frac{x^2}{2!} + \frac{x^4}{4!} - \frac{x^6}{6!} + \dots$$

⁵Numerical Recipe procedures are also available in book and electronic format for Pascal, MS BASIC, and MATLAB. Information on all the Numerical Recipe products can be found at <http://www.nr.com/>.

Write an algorithm to implement this formula so that it computes and prints out the values of $\cos x$ as each term in the series is added. In other words, compute and print in sequence the values for

$$\begin{aligned} \cos x &= 1 \\ \cos x &= 1 - \frac{x^2}{2!} \\ \cos x &= 1 - \frac{x^2}{2!} + \frac{x^4}{4!} \end{aligned}$$

up to the order term n of your choosing. For each of the preceding, compute and display the percent relative error as

$$\% \text{ error} = \frac{\text{true} - \text{series approximation}}{\text{true}} \times 100\%$$

As a test case, employ the program to compute $\cos(1.25)$ for six terms.

2.7 Write the algorithm for Prob. 2.6 as (a) a structured flowchart and (b) pseudocode.

2.8 Develop, debug, and document a program for Prob. 2.6 in either a high-level language or a macro language of your choice. Employ the library function for the cosine in your computer to determine the true value. Have the program print out the series approximation and the error at each step. As a test case, employ the program to compute $\cos(1.25)$ for up to and including the term $x^{10}/10!$. Interpret your results.

2.9 The following algorithm is designed to determine a grade for a course that consists of quizzes, homework, and a final exam:

- Step 1: Input course number and name.
- Step 2: Input weighting factors for quizzes (WQ), homework (WH), and the final exam (WF).
- Step 3: Input quiz grades and determine an average quiz grade (AQ).
- Step 4: Input homework grades and determine an average homework grade (AH).
- Step 5: If this course has a final grade, continue to step 6. If not, go to step 9.
- Step 6: Input final exam grade (FE).
- Step 7: Determine average grade AG according to

$$AG = \frac{WQ \times AQ + WH \times AH + WF \times FE}{WQ + WH + WF} \times 100\%$$

- Step 8: Go to step 10.
- Step 9: Determine average grade AG according to

$$AG = \frac{WQ \times AQ + WH \times AH}{WQ + WH} \times 100\%$$

- Step 10: Print out course number, name, and average grade.
- Step 11: Terminate computation.
- (a) Write well-structured pseudocode to implement this algorithm.
- (b) Write, debug, and document a structured computer program based on this algorithm. Test it using the following data to

calculate a grade without the final exam and a grade with the final exam: WQ = 35; WH = 30; WF = 35; quizzes = 98, 85, 90, 65, 99; homework = 95, 90, 87, 100, 92, 77; and final exam = 92.

2.10 The “divide and average” method, an old-time method for approximating the square root of any positive number a can be formulated as

$$x = \frac{x + a/x}{2}$$

- (a) Write well-structured pseudocode to implement this algorithm as depicted in Fig. P2.10. Use proper indentation so that the structure is clear.
- (b) Develop, debug, and document a program to implement this equation in either a high-level language or a macro language of your choice. Structure your code according to Fig. P2.10.

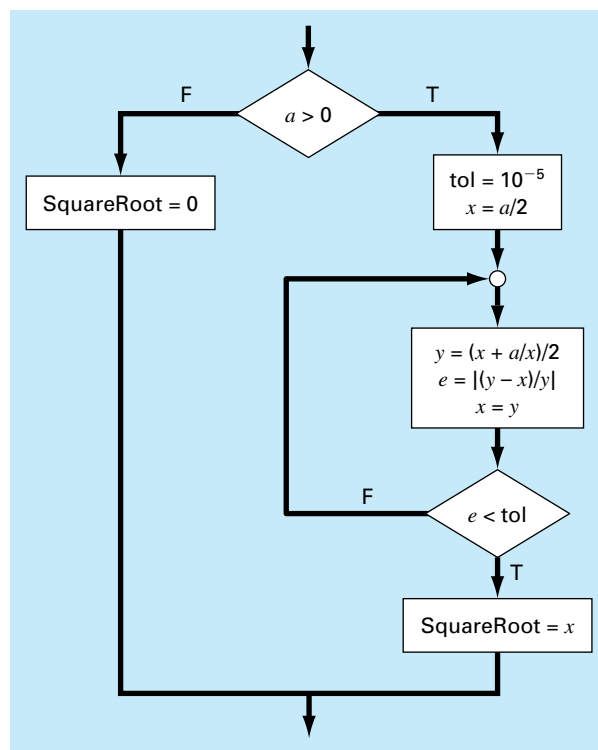


Figure P2.10

2.11 An amount of money P is invested in an account where interest is compounded at the end of the period. The future worth F yielded at an interest rate i after n periods may be determined from the following formula:

$$F = P(1 + i)^n$$

Write a program that will calculate the future worth of an investment for each year from 1 through n . The input to the function should include the initial investment P , the interest rate i (as a decimal), and the number of years n for which the future worth is to be calculated. The output should consist of a table with headings and columns for n and F . Run the program for $P = \$100,000$, $i = 0.06$, and $n = 5$ years.

2.12 Economic formulas are available to compute annual payments for loans. Suppose that you borrow an amount of money P and agree to repay it in n annual payments at an interest rate of i . The formula to compute the annual payment A is

$$A = P \frac{i(1+i)^n}{(1+i)^n - 1}$$

Write a program to compute A . Test it with $P = \$55,000$ and an interest rate of 6.6% ($i = 0.066$). Compute results for $n = 1, 2, 3, 4$, and 5 and display the results as a table with headings and columns for n and A .

2.13 The average daily temperature for an area can be approximated by the following function,

$$T = T_{\text{mean}} + (T_{\text{peak}} - T_{\text{mean}}) \cos(\omega(t - t_{\text{peak}}))$$

where T_{mean} = the average annual temperature, T_{peak} = the peak temperature, $\omega =$ the frequency of the annual variation ($= 2\pi/365$), and $t_{\text{peak}} =$ day of the peak temperature ($\cong 205$ d). Develop a program that computes the average temperature between two days of the year for a particular city. Test it for (a) January–February ($t = 0$ to 59) in Miami, Florida ($T_{\text{mean}} = 22.1^\circ\text{C}$; $T_{\text{peak}} = 28.3^\circ\text{C}$), and (b) July–August ($t = 180$ to 242) in Boston, Massachusetts ($T_{\text{mean}} = 10.7^\circ\text{C}$; $T_{\text{peak}} = 22.9^\circ\text{C}$).

2.14 Develop, debug, and test a program in either a high-level language or a macro language of your choice to compute the velocity of the falling parachutist as outlined in Example 1.2. Design the program so that it allows the user to input values for the drag coefficient and mass. Test the program by duplicating the results from Example 1.2. Repeat the computation but employ step sizes of 1 and 0.5 s. Compare your results with the analytical solution obtained previously in Example 1.1. Does a smaller step size make the results better or worse? Explain your results.

2.15 The bubble sort is an inefficient, but easy-to-program, sorting technique. The idea behind the sort is to move down through an array comparing adjacent pairs and swapping the values if they are out of order. For this method to sort the array completely, it may need to pass through it many times. As the passes proceed for an ascending-order sort, the smaller elements in the array appear to rise toward the top like bubbles. Eventually, there will be a pass through the array where no swaps are required. Then, the array is sorted. After the first pass, the largest value in the array drops

directly to the bottom. Consequently, the second pass only has to proceed to the second-to-last value, and so on. Develop a program to set up an array of 20 random numbers and sort them in ascending order with the bubble sort (Fig. P2.15).

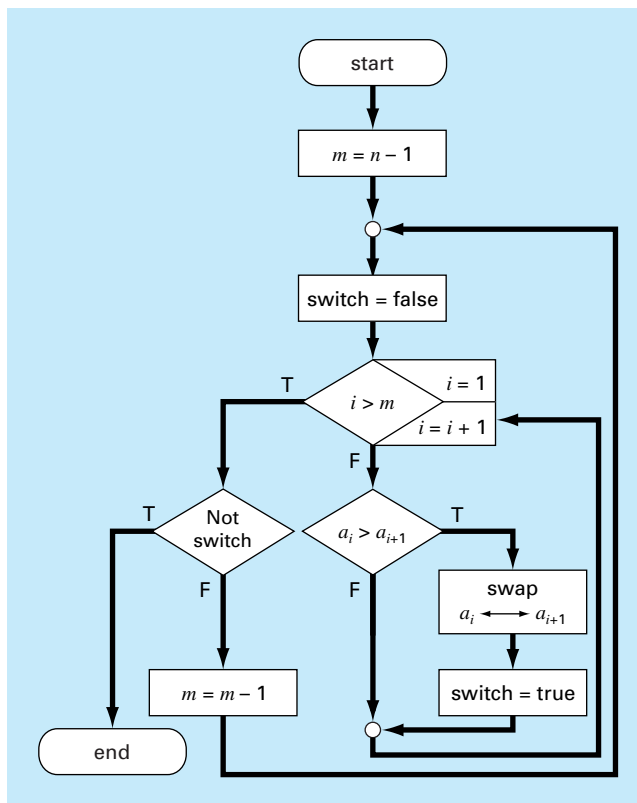


Figure P2.15

2.16 Figure P2.16 shows a cylindrical tank with a conical base. If the liquid level is quite low in the conical part, the volume is simply the conical volume of liquid. If the liquid level is midrange in the cylindrical part, the total volume of liquid includes the filled conical part and the partially filled cylindrical part. Write a well-structured function procedure to compute the tank's volume as a function of given values of R and d . Use decisional control structures (like If/Then, ElseIf, Else, End If). Design the function so that it returns the volume for all cases where the depth is less than $3R$. Return an error message ("Overtop") if you overtop the tank, that is, $d > 3R$. Test it with the following data:

R	1	1	1	1
d	0.5	1.2	3.0	3.1

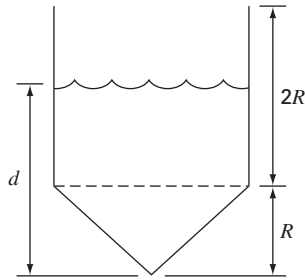


Figure P2.16

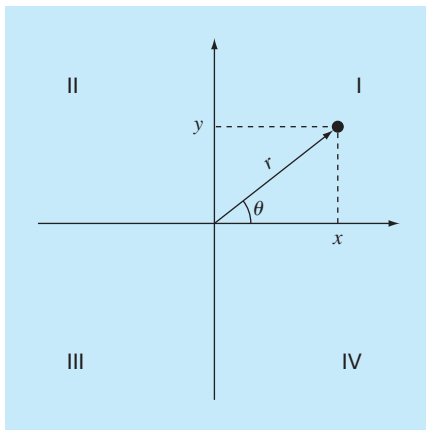


Figure P2.17

2.17 Two distances are required to specify the location of a point relative to an origin in two-dimensional space (Fig. P2.17):

- The horizontal and vertical distances (x, y) in Cartesian coordinates
- The radius and angle (r, θ) in radial coordinates.

It is relatively straightforward to compute Cartesian coordinates (x, y) on the basis of polar coordinates (r, θ). The reverse process is not so simple. The radius can be computed by the following formula:

$$r = \sqrt{x^2 + y^2}$$

If the coordinates lie within the first and fourth quadrants (i.e., $x > 0$), then a simple formula can be used to compute θ

$$\theta = \tan^{-1} \left(\frac{y}{x} \right)$$

The difficulty arises for the other cases. The following table summarizes the possibilities:

x	y	θ
<0	>0	$\tan^{-1}(y/x) + \pi$
<0	<0	$\tan^{-1}(y/x) - \pi$
<0	$=0$	π
$=0$	>0	$\pi/2$
$=0$	<0	$-\pi/2$
$=0$	$=0$	0

- Write a well-structured flowchart for a subroutine procedure to calculate r and θ as a function of x and y . Express the final results for θ in degrees.
- Write a well-structured function procedure based on your flowchart. Test your program by using it to fill out the following table:

x	y	r	θ
1	0		
1	1		
0	1		
-1	1		
-1	0		
-1	-1		
0	-1		
1	-1		
0	0		

2.18 Develop a well-structured function procedure that is passed a numeric grade from 0 to 100 and returns a letter grade according to the scheme:

Letter	Criteria
A	$90 \leq \text{numeric grade} \leq 100$
B	$80 \leq \text{numeric grade} < 90$
C	$70 \leq \text{numeric grade} < 80$
D	$60 \leq \text{numeric grade} < 70$
F	numeric grade < 60

2.19 Develop well-structured function procedures to determine (a) the factorial; (b) the minimum value in a vector; and (c) the average of the values in a vector.

2.20 Develop well-structured programs to (a) determine the square root of the sum of the squares of the elements of a two-dimensional array (i.e., a matrix) and (b) normalize a matrix by dividing each row by the maximum absolute value in the row so that the maximum element in each row is 1.