Chapter 1

From Formula to Program

We grow up in mathematics thinking that "the formula" is a ticket to solution space. Want the surface area of a sphere? Use

$$A = 4\pi r^2.$$

Have the cosine of some angle $\theta \in [0, \pi/2]$ and want $\cos(\theta/2)$? Use

$$\cos(\theta/2) = \sqrt{\frac{1+\cos(\theta)}{2}}.$$

Want the minimum value μ of the quadratic function $q(x) = x^2 + bx + c$ on the interval $[L, R]$? Use

$$\mu = \begin{cases} q(-b/2) & \text{if } L \leq -b/2 \leq R \\ \min\{q(L), q(R)\} & \text{otherwise} \end{cases}.$$

Want to know if year y is a leap year? Use the rule that y is a leap year if it is a century year divisible by 400 or a non-century year divisible by 4.

Sometimes the application of a formula involves a simple substitution. Thus, the surface area of a one-inch ball bearing is $4\pi(1/2)^2 = \pi$ square inches. Sometimes we have to check things before choosing the correct "option" within a formula. Since $0 \leq 3 \leq 10$, the minimum value of

$q(x) = x^2 - 6x + 5$ on the interval [0,10] is $q(3) = -4$. Sometimes we must clarify the assumptions upon which the formula rests. Thus, if a century year means something like 1400, 1500, and 2000, then 1900 was not a leap year.

Writing programs that use simple formulas like the above are a good way to begin our introduction to computational science. We'll see that the mere possession of a formula is just the start of the problem-solving process. The real ticket to solution space, if you travel by computer, is the program. And that's what we have to learn to write.

1.1 A Small Example

A *program* is a sequence of instructions that can be carried out by a computer. Let us write a program that solicits the radius of a sphere and then computes and prints its surface area. The surface area of a sphere that has radius r is given by

$$A(r) = 4\pi r^2.$$

There would be little more to say if we were to solve this problem "by hand." Indeed, we would (1) get r, (2) plug into the formula, and (3) write down the result. However, if the computer is to do the same thing, then each of these steps becomes more involved:

```
% Example 1_1:  Compute the surface area of a sphere
% A: surface area of the sphere
% r:  radius of the sphere

r= input('Enter the radius:  ');
A= 4*3.14159*r*r;
fprintf('Surface area is %7.2f.\n', A);
```

This program, like all the others in this text, is written in a programming language called MATLAB. MATLAB is a programming language and tool that is widely used in engineering, science, and mathematics.

As with natural languages such as English and Japanese, programming languages have rules of syntax that must be obeyed. Let us begin the process of learning the grammar of MATLAB by dissecting the above program.

A program is very much like a recipe: it is a step-by-step description of some "cooking" process. A program has a name, just like a recipe has a name. All names in MATLAB including names of programs must satisfy a number of rules:

- A name must begin with a letter.
- Each character in the name must be either a letter, a digit, or an underscore (_).
- Only the first 31 characters of the name are significant; the remaining characters of the name are ignored.

- A name should not be a *reserved word*. A reserved word is one that has a special meaning in MATLAB and should not be used in programs except as intended according to the special meaning.

Thus, `sphereArea`, `area`, and `Example1_1` are valid names whereas `E-1` and `1E` are not. The computer file containing a MATLAB program has a filename consisting of the program name and the extension `.m`. Therefore, a MATLAB program `Example1_1` has the filename `Example1_1.m`. A MATLAB program file is also called an M-file. The list of MATLAB reserved words will be developed as we go along.

The choice of names in your program is extremely important. A recipe has a name, and we expect the recipe name to be descriptive and to represent the item being cooked. Similarly, a program name should be meaningful and descriptive. You wouldn't name a recipe for an apple pie "broccoli bake" or "food," so don't name your programs "programA," or "B" (unless they actually have to do with the letters A or B)!

The simple program above (Example 1_1) has just two parts:

⟨*Comment*⟩
⟨*Action*⟩

The "comment part" is a remark on the program; the "action part" contains the actual instructions for the computer to carry out. Notice our "angle bracket" notation. Words enclosed in the angle brackets ⟨*like this*⟩ are used in the spirit of "fill-in-the-blank." In effect we are saying that for the time being, be content with the word description—details are being hidden or will be filled in later. Words typeset `in this style` are actually part of the program.

A remark that begins with the percent character `%` is called a *comment*. Comments are essential for program readability, although they are not executed by the computer. A lead comment should always be included to describe in a few words what the program does. Thus, upon reading

```
% Example 1_1:  Compute the surface area of a sphere
```

we know that the program `Example1_1` deals with the surface area of a sphere. The writing of good lead comments is important because it is unreasonable to expect the reader to deduce the function of a program by stepping through its instructions.

A comment begins with the `%` character and ends with the end of the line. Long comments can be spread over several lines where each line begins with a `%` character. Thus, we could begin `Example1_1` as follows:

```
% Example 1_1:  Compute the surface area of a sphere.
% See page 326 of ''Calculus'' by G. Strang.
```

The two comments that follow the lead program comment explain some names that are used in the program.

```
% A: surface area of the sphere
% r:  radius of the sphere
```

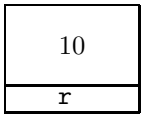

FIGURE 1.1 *Visualizing A Variable*

These comments are sometimes called a *variable dictionary* because they explain to the reader what those names represent in the program. In larger programs, you may want to put variable definitions at judicious points in the program, not at the beginning. This will be illustrated in later programs.

To learn more about *variables*, we now consider the "action part" of the program in detail. In Example 1_1, `r` and `A` are *variables* that hold the values of the radius and the area, respectively.

```
r= input('Enter the radius:  ');
A= 4*3.14159*r*r;
        ⋮
```

If you compute the surface area of a sphere with pencil and paper, then the numbers that arise during the solution process are just written down or perhaps kept in your head. You do not think particularly hard about *where* the intermediate results are recorded. But when the computer calculates the solution, the numbers must have a specific place where they can be stored and retrieved for later use.

For example, the values for the acquired radius and the computed surface area are held in the *variables* `r` and `A` respectively. It is useful to think of variables as "boxes." In everyday life, boxes are often tailored to hold specific items. The same is true of the variables in a program. A variable stores a value of a specific *type*. As you will learn, variables in a program can hold real numbers, integers, true-false values, individual characters, strings of characters, and many other kinds of data. In the MATLAB programming language, a variable is created simply by putting a value into the variable. We will examine the process of putting a value in a variable shortly. The type of a variable is the type of the value that is put into the variable. In `Example1_1`, two variables are used. These are `A` and `r` and they both have type `double` (which stands for double precision floating point number). This means that they can store real numbers like 10.6 and 19×10^3 or integers like -6 and 0. In MATLAB computing, integers are treated as real numbers. You can visualize a variable as a named box that holds a value as depicted in FIGURE 1.1.

We now consider the "action part" of `Example1_1` line by line. It consists of three statements to be executed by the computer:

```
r= input('Enter the radius:  ');
A= 4*3.14159*r*r;
fprintf('Surface area is %7.2f.\n', A);
```

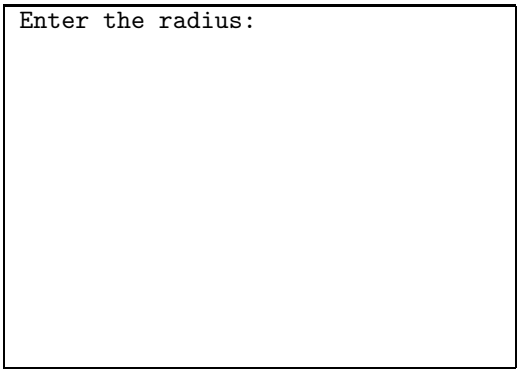

FIGURE 1.2 *Output In Command Window*

The statement

```
r= input('Enter the radius:  ');
```

is an *assignment statement*, which has a lefthand side and a righthand side separated by the *assignment operator* "=" (the equal sign). In an assignment statement, the result of the action on the righthand side is put into, or *assigned to*, the variable stated on the lefthand side. The action on the righthand side of this statement uses the MATLAB reserved word `input` to prompt the user for a value and then read the the value entered from the keyboard. The message "Enter the radius:" enclosed in single quotes is displayed in MATLAB's *Command Window*. See FIGURE 1.2. After displaying the message, called a prompt because it prompts the user for some action, program execution holds until the user enters a value. For example, if we respond to the prompt by typing "10" and then striking the $<$ *enter* $>$ button, then program execution continues and the value of 10 is stored in the newly created variable `r` (the box) as shown earlier in FIGURE 1.1. We will discuss the semicolons that appear at the end of the statements shortly.

Once the data for the problem is in the computer's memory, we are ready to invoke the $A = 4\pi r^2$ formula:

```
A= 4.0*3.14159*r*r;
```

Again, this is an assignment statement. The righthand side is a recipe for a numerical value. The left hand side names the variable where the computed value is to be stored. FIGURE 1.3 shows the status of `r` and `A` after the assignment.

Notice that in an assignment statement, the action on the righthand side is executed *first* before the resulting value is stored in the variable on the lefthand side. Although it is tempting to read the assignment as "`A` equals $4\pi r^2$," it is much more accurate to say that "`A` is *assigned* the value of $4\pi r^2$." This is because the assignment is an action, not a statement about equality. This point is clarified later.

Literal values like "4" and "3.14159" in an arithmetic expression may be entered in different ways. For example, the expressions `4*3.14159` and `4.0*3.14159` are the same. A scientific notation is also available:

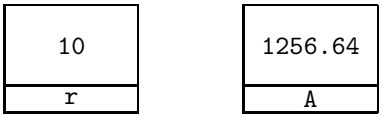

FIGURE 1.3 *Assignment to* `A`

```
A= 0.04E+2*3.14159*r*r;
```

although in this case, it is not a convenience.

The assignment statement under consideration has a particularly simple arithmetic expression: `4*3.14159*r*r`. Asterisks are used to designate multiplication. An equivalent expression using the *power* operator "^" is `4*3.14159*r^2`. What happens if the arithmetic expression is so long that we wish to write it in several lines? Use an ellipsis "..." to indicate that the statement continues on the next line. Therefore, the multi-line statement

```
A= 4*3.14159* ...
   *r*r;
```

and the single-line statement

```
A= 4*3.14159*r*r;
```

will perform the same calculation and assignment.

Because the arithmetic expression is a recipe for a value, it is essential that all the ingredients are present. If we try to calculate the surface area before acquisition of the radius `r`, e.g.,

```
A= 4*3.14159*r*r;
r= input('Enter the radius:  ');
```

then the program would be wrong. *The order of statements in a program body is very important.* Notice the difference between *accessing* a variable and *creating* a variable. In the statement

```
A= 4*3.14159*r*r;
```

the program needs to *access* the value in `r` on the righthand side in order to do the calculation. Therefore variable `r` must exist prior to this statement. On the other hand, variable `A` on the lefthand side is *created* using the value that results from the righthand side calculation—variable `A` needs not exists prior to this statement.

The last action to be performed is the printing of the result. MATLAB provides several ways to print text. The `fprintf` command allows us to print messages that include the values stored in variables. The statement

```
Enter the radius:  10
Surface area is 1256.64.
```

Figure 1.4 *Output In Text Window*

```
fprintf('Surface area is %7.2f.\n', A);
```

prints a line of text that has two components, a message and a number stored in a variable. See Fig.1.4

A message is printed exactly as entered between the single quotation marks in the `fprintf` statement *except* for the character sequences that begin with the percent character "`%`" or with the back slash character "\". The character sequence `%7.2f` in the above `fprintf` statement is a place-holder for the value stored in the variable listed after the quoted text, variable `A`. The place-holder sequence `%7.2f` contains formatting information: print the value using **7** "spaces" in total with **2** decimal places for a **f**loating point number. One may choose to leave out the space formatting information. For example, the character sequence `%f` will substitute in the value (of the variable name given later) as a floating point number using as many spaces as necessary and with the default number of decimal places, four or six in most systems. We will introduce more substitution sequences in later examples. The character sequence `\n` is the "new line" character, sometimes called the carriage return (think about a typewriter). Any text that follows the new line character will be printed in a separate line. For example, if we change the `fprintf` statement to be

```
fprintf('Surface area is %f.\nCool!\n', A);
```

the printed output will be

```
Surface area is 1256.636000.
Cool!
```

Choosing a format so that the output looks nice requires some care. It is typically a detail that should not be addressed until after a preliminary version of the program is working, for then you know the size of the numbers involved and can apply sensible formats. Another consideration is the precision of the computer's arithmetic. This is discussed in the next section.

Before we discuss the semicolon that appears at the end of each executable statement in our program, let's look at the entire program and its output as shown in the box above. Italics are used to indicate user-responses to prompts from the program. We will use such boxes to show programs and their sample output throughout this book.

```
% Example 1_1:  Compute the surface area of a sphere
% A: surface area of the sphere
% r:  radius of the sphere

r= input('Enter the radius:  ');
A= 4*3.14159*r*r;
fprintf('Surface area is %7.2f.\n', A);
```

Sample output:

```
Enter the radius:  10
Surface area is 1256.64.
```

We use a semicolon ";" at the end of a statement to *suppress* MATLAB's automatic value display behavior. If we do *not* use the semicolons in our program, the program executes as we have already discussed, but after completing each statement MATLAB will display the value of the variable created in that statement:

```
Enter the radius:  10
r =
   10
A =
   1.2566e+03
Surface area is 1256.636000.
```

Notice that the `fprintf` and `input` statements are *supposed to* display text, therefore they display text as coded in the statements whether we use semicolons at the end of the statements or not. Since most programs tend to be longer than a few statements, use semicolons to suppress the automatic display in order to keep the output from getting cluttered up. Make a habit of ending each statement with a semicolon.

We give another example program to clarify the statements just learned. `Example1_2` solicits a radius r (assumed to be in miles) and a modification Δr (assumed to be in inches) and prints the increase in spherical surface area (in square miles) when the radius is increased from r to $r + \Delta r$.

Instead of using a variable dictionary (comments) near the top of the program, we define each variable as it is introduced in the program using a comment. The meaning of the variables `r` and `delta` are clearly given by the text prompts of the input statements so additional comments are not necessary for these variables. As you write more complex programs, you will find it convenient to define a variable at the place where it is created. For a large program that has multiple sections, use a variable dictionary for important variables at the top of each section.

A new feature of `Example1_2` is the use of the *built-in constant* `pi`. The number π is so important that it is incorporated in the MATLAB language. When `pi` appears in an arithmetic expression, its value as defined by MATLAB is substituted.

Notice the use of parentheses in the assignment

```
% Example 1_2:  Explore how the surface area of a sphere
% changes with an increase in the radius.

r= input('Enter radius r in miles:  ');
delta= input('Enter delta r in inches:  ');
newr= r + ((delta/5280)/12); % new radius in miles
A= 4*pi*r^2; % original surface area
newA= 4*pi*newr^2; % new surface area
incr= newA - A; % increase in area
fprintf('Increase in area (mile^2) is %f.\n', incr);
```

Sample output:

```
Enter radius r in miles:  4000
Enter delta r in inches:  10
Increase in area (mile^2) is 15.866630.
```

```
newr= r + ((deltar/5280)/12);
```

It turns out that

```
newr= r + deltar/5280/12;
```

renders the same value because of the *rules of precedence*. These are rules that determine the order of operations in an arithmetic expression. Unless overridden by parentheses, multiplicative operations ($*$, /) are performed before all additive operations ($+$,$-$). A succession of multiplicative operations or a succession of additive operations are performed left to right. Thus, `a= 1/2/3/4` is equivalent to `a= ((1/2)/3)/4` and different from `a= 1/(2/(3/4))`. Always use parentheses in ambiguous situations.

Even though the above programs are short and simple, they reveal a number of very important aspects about errors in the computational process. Suppose that the program is used to compute the surface area of the Earth. To begin with, there is a *model error* associated with the assumption that the Earth is a perfect sphere. In fact, the shape of the Earth is better modeled by an oblate spheroid, i.e., an ellipse of revolution. Second, if we use the radius value $r = 3960$ and if this value is determined experimentally, say by a satellite measurement, then there is undoubtedly a *measurement error*. Perhaps the satellite instruments are sensitive to four significant digits and that the "real" Earth has a radius of 3960.2 miles. Third, there is the *mathematical error* associated with the approximation of π. Finally, there is the *roundoff error* associated with the actual computation `A= 4*pi*r*r`. Computers do not do exact real arithmetic. Just as the division of 1 by 3 on your calculator produces something like .333333 and not 1/3 exactly, so may we expect the computer to make a small error every time an arithmetic operation is performed. This is discussed in §2.3.

A great deal of computing experience is required before the interplay between these factors can be fully appreciated. One of our goals is to communicate these subtleties and to build your intuition about them.

PROBLEM 1.1. Assume that the Earth is a sphere with radius 4000 miles. By running `Example1_2` three times, determine the increase in surface area if the Earth is uniformly paved with 1, 5, and 10 inches of cement. (Don't let this happen in real life!) Next modify `Example1_2` so that it also computes the approximate surface area increase via the following formula:

$$\Delta A = A(r + \Delta r) - A(r) \approx 8\pi r \Delta r.$$

This follows the derivative of $A(r)$ can be approximated very well by a divided difference if Δr is very much smaller than r:

$$A'(r) \approx \frac{A(r + \Delta r) - A(r)}{\Delta r}.$$

Compare the two methods using the following choices for r and Δr:

r	Δr
4000	1
4000	5
4000	10
4000	1000

PROBLEM 1.2. The surface area of an oblate spheroid such as the Earth is given by $A = 4\pi r_1 r_2$ where r_1 is the equatorial radius and r_2 is the polar radius. Write a program that reads in these two radii and computes the difference between $4\pi r_1 r_2$ and $4\pi((r_1 + r_2)/2)^2$. Use the Earth data $r_1 = 3963$, $r_2 = 3957$.

1.2 Sines and Cosines

Let us continue the discussion of formulas and programs using as examples some well-known trigonometric identities. Our first calculations involve the *half-angle* formulae for cosine and sine:

$$\begin{aligned} \cos(\theta/2) &= \sqrt{(1 + \cos(\theta))/2} \qquad 0 \le \theta \le \pi/2. \\ \sin(\theta/2) &= \sqrt{(1 - \cos(\theta))/2} \qquad 0 \le \theta \le \pi/2. \end{aligned}$$

These recipes can be used to compute sines and cosines of various angles from a sines and cosines of other angles. For example, since $\cos(\pi/4) = 1/\sqrt{2}$ we can use the half-angle formula for sine to compute $\sin(\pi/8)$:

```
a= 2;
c= 1/sqrt(a);
s= sqrt((1 - c)/2);
fprintf('sin(pi/8) is %f\n', s);
```
(1.2.1)

An excerpt from a program like this is called a *fragment*. We use fragments to communicate new programming ideas whenever the inclusion of the whole program burdens us with unnecessary detail. Right now, we are *not* interested in program comments or user input. The focus is on the assignment statement and the fragment illustrates some new features about this construct.

The fragment makes use of `sqrt`, one of many *built-in* functions that are part of MATLAB. If `a` is a variable that contains a numeric value, then `sqrt(a)` returns the value of its square root. The `sqrt` function can accept literals and so it is legal to replace the first two assignments with

```
c= 1/sqrt(2);
```

More generally, `sqrt` can be applied to any arithmetic expression, e.g.,

```
c= 1/sqrt(2);
s= sqrt((1-c)/2);
```

We can "collapse" the fragment even further:

```
s= sqrt((1-(1/sqrt(2)))/2)
```

But now the role of the half angle formulae is obscured by all the parentheses. Avoid "dense" one-liners like this. It is better to spread the computation of over a few lines thereby highlighting some of the important intermediate results that arise during the course of computation.

The little cosine/sine computation gives us the opportunity to clarify three different types of error. If we type

```
a= 2;
c= 1/sqrt(a);
s= sqrt((1 - c/2);
fprintf('sin(pi/8) is %f\n', s);
```

instead of (1.2.1), then a *syntactic* error results because there are unbalanced parentheses in the third assignment statement. Syntactic errors are grammatical violations in a program. When a MATLAB program is executed, MATLAB checks for syntactic errors in a statement, a step called compilation, before executing that statement. If an error is found in a statement, program execution terminates at that statement with an error message.

Even if a program contains no syntactic errors, it may not run to completion because of a *run-time* error. For example,

```
a= 0;
c= 1/sqrt(a);
s= sqrt((1 - c)/2);
fprintf('sin(pi/8) is %f\n', s);
```

is syntactically correct, but the division in the second statement breaks down because the divisor is zero.

In addition to syntactic and run-time errors, there are *programmer errors*:

```
a= 2;
c= 1/sqrt(a);
s= sqrt(1 - c/2);
fprintf('sin(pi/8) is %f\n', s);
```

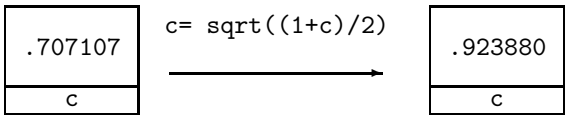

FIGURE 1.5 *Overwriting*

This compiles and runs to completion, but the desired output is not produced. The deleted parentheses in the assignment to `s` means that the program is computing $\sqrt{1-(c/2)}$ instead of $\sqrt{(1-c)/2}$. Programmer errors are often the hardest to detect because the editor and the compiler are not there to point out our mistakes. Moreover, we may be so excited that our program actually runs that we overlook its correctness!

The issue of program correctness is particularly complex, especially as programs get long. One of our goals is to develop problem-solving strategies that are organized in such a way that we can be confident about a program's correctness. [1]

Let us move on to a more complicated example. We can compute $\sin(\pi/16)$ by repeated application of the half-angle formulae:

```
c= 1/sqrt(2);       % {c = cos(pi/4)}
c= sqrt((1+c)/2);   % {c = cos(pi/8)}
s= sqrt((1-c)/2);   % {s = sin(pi/16)}
fprintf('sin(pi/16) is %f\n', s);
```

The inclusion of comments helps us trace what happens as the computation unfolds. These cryptic, mathematical comments are called *assertions*. After the assignment of $1/\sqrt{2}$ to `c` we can *assert* that $c = \cos(\pi/4)$. *By convention*, assertions are enclosed in braces "{}."

The next assignment computes $\cos(\pi/8)$ from $\cos(\pi/4)$ using the cosine half-angle formula. As usual, to the right of the assignment operator "=" is a recipe, or more precisely, an arithmetic expression. To the left of the assignment symbol is the name of the variable where the result is to be stored, in this case, `c`. Thus, even though `c` is involved in the expression, it is the "target" of the operation. First, the righthand side of the assignment statement is evaluated, using the current value of `c`, to a single value. Then this new value is assigned to variable `c`, *overwriting* the previous value. FIGURE 1.5 shows how we should visualize the update of `c`'s value. The current contents of a variable can be overwritten with a new value. The old value is "erased" much as the current contents of a CD are erased during rewriting.

Finally, the sine is computed and the values displayed in FIGURE 1.6 are displayed.

These two values could be computed directly by using the built-in functions `sin` and `cos`. These functions assume input values in radians. Thus, since `pi` houses the value of π,

[1] Just being *confident* in a program's correctness does not mean that we can *guarantee* its correctness. That's a very mathematical exercise and beyond the scope of this introductory text.

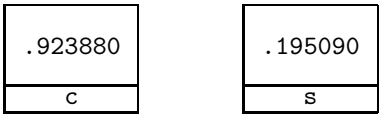

Figure 1.6 *The Final Cosine/Sine Values*

```
s= sin(pi/16);
```

is mathematically equivalent to

```
c= 1/sqrt(2);
s= sqrt(1-c/2)
```

Not surprisingly, there is often more than one way to compute the same thing.

Next we consider the double angle formulae:

$$\begin{aligned} \cos(2\theta) &= \cos^2(\theta) - \sin^2(\theta) \\ \sin(2\theta) &= 2\sin(\theta)\cos(\theta) \end{aligned}$$

The program `Example1_3` solicits an angle and then computes the sine and cosine of the double angle. Note that the input angle is given in degrees and is then converted to radians[2]. This is necessary because `sin` and `cos` assume radian arguments.

After the conversion to radians, the $\cos(\theta)$ and $\sin(\theta)$ are computed and stored in `c` and `s` respectively. Using the double angle formulae, these values are replaced by $\cos(2\theta)$ and $\sin(2\theta)$. Note that the variable `ctemp` is necessary to hold $\cos(\theta)$. The fragment

```
c= cos(theta);
s= sin(theta);
c= c^2 - s^2;   % {cos(2*theta)}
s= 2*c*s;       % {sin(2*theta)}
```

does *not* assign $\sin(2\theta)$ to `s` since the value of `c` used in the last assignment is $\cos(2\theta)$ and not $\cos(\theta)$.

For both cosine and sine, the absolute value function `abs` is used to compute the discrepancy between the double angle method and direct calculation using the built-in `sin` and `cos` functions. The results are then displayed.

We introduce two new character sequences in the `fprintf` statement. In the statement

[2] 180 degrees = π radians

```
% Example 1_3:  Demonstrate the double angle formulae for sine and cosine.

a= input('Enter angle (degrees):  ');
theta= a*pi/180;   % the angle in radians
c= cos(theta);
s= sin(theta);
ctemp= c;
c= c^2 - s^2;   % {cos(2*theta)}
s= 2*s*ctemp;   % {sin(2*theta)}

% Errors in using the double angle formulae
  cosError= abs(c-cos(2*theta));
  sinError= abs(s-sin(2*theta));

fprintf('cos(2*theta) = %.6f\t error = %.1e\n', c, cosError);
fprintf('sin(2*theta) = %.6f\t error = %.1e\n', s, sinError);
```

Sample output:

```
Enter angle (degrees):  30
cos(2*theta) = 0.500000   error = 1.1e-16
sin(2*theta) = 0.866025   error = 0.0e+00
```

```
fprintf('sin(2*theta) = %.6f\t error = %.1e\n', s, sinError);
```

the character sequence `\t` represents a tab, i.e., a sequence of usually two to eight spaces. The substitution sequence `%.1e` specifies that the value of the variable will be printed in "scientific notation" with one decimal place. So far, we have introduced the substitution sequences `%f` for floating point number and `%e` for scientific notation. What if you don't know in advance which format to use? You can use the substitution sequence `%g`, which stands for **g**eneral. This leaves the system to determine which format to use. In the above statement, there are two substitution sequences and two variables are listed. The variable values are substituted into the message to be printed in the order in which they are are listed. Thus, the value in the first variable, `s`, is printed using the first substitution sequence `%.6f` while the value in the second variable, `sinError`, is printed using the sequence `%.1e`.

As a final example, suppose variables `x` and `y` house positive real values x and y and that θ is defined by $\tan(\theta) = y/x$. Our goal is to assign the values $\cos(\theta)$ and $\sin(\theta)$ to variables `c` and `s`. One approach is to use the built-in function `arctan`.[3] The program *fragment*

```
theta= arctan(y/x);
c= cos(theta);
s= sin(theta);
```

makes the required assignments to `c` and `s`. Alternatively, from FIGURE 1.7 we also have

[3]Recall that if $u = \arctan(v)$, then $\tan(u) = v$.

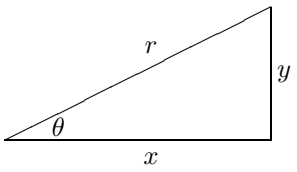

FIGURE 1.7 $\sin(\theta) = y/r$ *and* $\cos(\theta) = x/r$

```
r= sqrt(x*x + y*y);
c= x/r;
s= y/r;
```

Again we see that there is more than one way to compute the same thing. This is especially true in the trigonometric area where there are so many identities to work with.

PROBLEM 1.3. Modify `Example1_3` so that it applies the double angle formulae twice to produce the sine and cosine of 4θ where θ is the input angle. Print the discrepancies from the values `sin(4*theta)` and `cos(4*theta)`.

PROBLEM 1.4. Given that $\cos(60^o) = 1/2$ and $\cos(72^o) = (\sqrt{5}-1)/4$, write fragments that print the following values: **(a)** $\sin(18^o)$, **(b)**$\cos(3^o)$, and **(c)** $\sin(27^o)$. Do not make use of the built-in functions `sin`, `cos`, or `arctan`. Use the half-angle formulae and the identities

$$\begin{aligned}
\cos(\theta_1 + \theta_2) &= \cos(\theta_1)\cos(\theta_2) - \sin(\theta_1)\sin(\theta_2) \\
\sin(\theta_1 + \theta_2) &= \cos(\theta_1)\sin(\theta_2) + \sin(\theta_1)\cos(\theta_2) \\
\cos(-\theta) &= \cos(\theta) \\
\sin(-\theta) &= -\sin(\theta)
\end{aligned}$$

1.3 Max's and Min's

Let us pose some questions about the behavior of the the quadratic function

$$q(x) = x^2 + bx + c$$

on the interval $[L, R]$:

Q_1: Which is smaller, $q(L)$ or $q(R)$?

Q_2: Does the derivative $q'(x)$ have a zero in the interval $[L, R]$?

Q_3: What is the minimum value of $q(x)$ in $[L, R]$?

To answer each question it is necessary to make a comparison of values and then branch to an appropriate course of action. The `if` construct is required to handle this kind of situation.

Consider the first problem where we want to decide whether $q(x)$ is larger at $x = L$ or at $x = R$. Assume that `L`, `R`, `b`, and `c` are initialized real variables and that $L \leq R$. Here is a fragment that prints a message based upon the comparison of $q(L)$ and $q(R)$:

```
% Fragment A
qL= L^2 + b*L + c;
qR= R^2 + b*R + c;
if qL < qR                                                    (1.3.1)
   fprintf('q(L) < q(R)');
else
   fprintf('q(L) >= q(R)');
end
```

The fragment begins by storing the value of $q(L)$ and $q(R)$ in the variables `qL` and `qR` respectively. Then a comparison is made. If the value of `qL` is smaller than the value `qR`, then the message

```
q(L) < q(R)
```

is printed. Otherwise,

```
q(L) >= q(R)
```

is printed.

Alternatives like this are very common in computing and the `if-else` construct is designed to navigate the "fork in the road":

if ⟨*Condition*⟩
    ⟨*Something to do if the condition is true.*⟩
else
    ⟨*Something to do if the condition is false.*⟩
end

Three reserved words are associated with this statement: `if`, `else`, and `end`. The condition is called a *boolean expression* or *logical expression*. Just as arithmetic expressions produce numbers, so do boolean expressions produce true-false values. Once `qL` and `qR` have been assigned values, the boolean expression `qL < qR` is either true or false. The symbol `<` stands for "less than" and is one of the several *relational operators* listed in FIGURE 1.8. Notice that the "equal to" relation has the double equal sign "==" symbol. There are often several ways to organize an `if-else` statement. For example, the fragment

```
% Fragment B
qL= L^2 + b*L + c;
qR= R^2 + b*R + c;
if  qL >= qR
```

Operator	Meaning
>	greater than
>=	greater than or equal to
==	equal to
~=	not equal to
<=	less than or equal to
<	less than

Figure 1.8 *Relational Operators*

```
   fprintf('q(L) >= q(R)');
else
   fprintf('q(L) < q(R)');
end
```

is equivalent to `Fragment A`. If we play with the underlying mathematics, then other possibilities unfold. For example, since

$$q(L) - q(R) = (L^2 + bL + c) - (R^2 + bR + c) = (L^2 - R^2) + b(L - R) = (L - R)(L + R + b)$$

we also have

```
% Fragment C
if  (L-R)*(L+R+b) > 0
   fprintf('q(L) > q(R)');
else
   fprintf('q(L) <= q(R)');
end
```

You may "do more than one thing" as a result of a comparison. For example, to handle the situation

```
if  (L-R)*(L+R+b) > 0
   ⟨Print a message 'q(L) ≥ q(R)' and compute the slope of q at L.⟩
else
   ⟨Print a message 'q(L) < q(R)' and compute the slope of q at R.⟩
```

you will put two statements under each of the `if` and `else` branches:

```
if  (L-R)*(L+R+b) > 0
   fprintf('q(L) > q(R)');
   slope= 2*L + b;
else
   fprintf('q(L) <= q(R)');
   slope= 2*R + b;
end
```

(Recall that the slope of q at x is given by $q'(x) = 2x+b$.) All the statements under the `if` branch will be executed if ⟨*condition*⟩ evaluates to true. All the statements under the `else` branch will be executed if ⟨*condition*⟩ evaluates to false. Notice how we have indented all the statements to be executed under a certain branch. Always *indent sub-structures* so that the alteratives (branches) are visually clear for anyone reading the program.

Sometimes there is "nothing to do" if the comparison in the `if` is false. In this case, just delete the `else` branch. The fragment

```
qL= L^2 + b*L + c;
qR= R^2 + b*R + c;
if  abs(qL-qR) <= 0.001
   fprintf('q(L) is close to q(R)');
   ave= (qL+qR)/2;
end
```

prints the message

```
q(L) is close to q(R)
```

if $|q(L) - q(R)| \leq .001$ and assigns the average value to `ave`. Nothing is done if this boolean expression is false. Here is the general structure of a simple `if-then`:

if ⟨*Condition*⟩
   ⟨*Something to do if the condition is true.*⟩
end

Now consider the second of the three questions posed above: does the derivative of $q(x) = x^2+bx+c$ have a zero in the interval $[L, R]$? Since $q'(x) = 2x+b$, the zero is given by $x_c = -b/2$. Let us write a fragment that prints a message indicating whether or not $L \leq x_c \leq R$ is true. Two comparisons must be made. We must compare x_c with L and x_c with R. Only if the comparison $L \leq x_c$ is true *and* the comparison $x_c \leq R$ is true may we conclude that x_c is in the interval. Here is a fragment that performs these two checks and prints an appropriate message:

```
% Fragment D
xc= -b/2;
if  (L <= xc) && (xc <= R)       % if (L<=xc) AND (xc<=R)
   fprintf('xc is in [L,R]');
else
   fprintf('xc is not in [L,R]');
end
```

The expression

```
(L <= xc) && (xc <= R)
```

is a boolean expression and therefore has a value of true or false. This *and* operation has the form

⟨*Boolean Expression*⟩ `and` ⟨*Boolean Expression*⟩

a	b	a && b
false	false	false
false	true	false
true	false	false
true	true	true

Figure 1.9 *The* and *Operation*

a	b	a || b
false	false	false
false	true	true
true	false	true
true	true	true

Figure 1.10 *The* or *Operation*

It compares two boolean values and returns another boolean value according to the table in Figure 1.9. `Fragment D` is equivalent to:

```
% Fragment E
xc= -b/2;
if  (L > xc) || (xc > R)          % if (L<=xc) OR (xc<=R)
   fprintf('xc is not in [L,R]')
else
   fprintf('xc is in [L,R]');
end
```

This illustrates the *or* operation. This logical operation is defined in Figure 1.10. A third logical operation called the *not* operation *negates* a boolean value. The *not* operator is the tilde "∼"symbol. See Figure 1.11. Here is another rewrite of `Fragment D` that uses the *not* operation:

```
xc= -b/2;
if  ∼((L <= xc) && (xc <= R))      % if NOT((L<=xc)AND(xc<=R))
   fprintf('xc is not in [L,R]');
else
   fprintf('xc is in [L,R]');
end
```

a	∼a
false	true
true	false

Figure 1.11 *The* not *Operation*

As a last example, let us compute the minimum value of the function $q(x) = x^2 + bx + c$ on the interval $[L, R]$. Calling this minimum m, here is the "formula":

$$m = \begin{cases} q(x_c) & \text{if } x_c \in [L, R] \\ \text{the smaller of } q(L) \text{ and } q(R) & \text{if } x_c \text{ is not in } [L, R] \end{cases}$$

This either/or situation calls for an if-else construct:

```
xc= -b/2;
if  (L <= xc) && (xc <= R)
    ⟨The minimum value is q(x_c) = −b^2/4 + c.⟩
else
    ⟨The minimum value is the smaller of q(L) and q(R)⟩
end
```

If the condition is true, then `m = -(b^2)/4+c`. If the condition is false, then the code necessary to identify the smaller of $q(L)$ and $q(R)$ involves another `if-else`:

```
if  (L-R)*(L+R+b)<0
    m= L*L+b*L+c;
else
    m= R*R+b*R+c;
end
```

Putting it all together we obtain the program `Example1_4`. The program shows how `if` statements can be *nested*, i.e., one `if` statement is contained within another `if` statement. But notice that as we developed the program, we never dealt with more than one `if` at a time. The mission of the "outer" `if` is established first without regard to the details of the alternatives. We didn't have to think about the "inner" `if` until the issue of q's value at the endpoints surfaced. This is an example of *top-down* problem solving, a truly essential skill for the computational scientist. Examples of top-down problem solving permeate the rest of the text.

We mention in closing it takes a while to develop a facility with boolean expressions. Arithmetic expressions pose no comparable difficulty, because we have had years of schooling in mathematics. But through a carefully planned sequence of "boolean challenges", you will become as adept with `&&` (and), `||` (or), and $\sim$ (not) as you are with "+", "-", "*", and "/".

PROBLEM 1.5. Modify `Example1_4` so that it handles quadratics with a variable quadratic coefficient, i.e., quadratics of the form $q(x) = ax^2 + bx + c$. Assume $a \neq 0$.

1.4 Quotients and Remainders

According to the rules of the Gregorian calendar, a year is a leap year if it is divisible by four with the exception of century years that are not divisible by 400. Thus, 1992 and 2000 are leap years while 1993 and 2100 are not.

```
% Example 1_4:  Minimum of x*x + bx + c on [L,R]

b= input('Enter coefficient b:  ');
c= input('Enter coefficient c:  ');
L= input('Enter left end of interval:  ');
R= input('Enter right end of interval:  ');
% m is minimum of the quadratic on [L,R]

xc= -b/2;   % the critical value of the quadratic
if  (L<=xc && xc<=R)
   m= c - b*b/4;
else  % {xc is not in [L,R]}
   if   ((L-R)*(L+R+b) < 0)
      m= L*L + b*L + c;
   else
      m= R*R + b*R + c;
   end
end
fprintf('min is %f\n', m);
```

Sample output:

```
Enter coefficient b:  -5
Enter coefficient c:  6
Enter left end of interval:  -10
Enter right end of interval:  10
min is -0.250000
```

```
% Example 1_5:  Leap year calculation

year= input('Enter year:  ');
if  ( mod(year,4) ~= 0 )
   fprintf('%d is an ordinary year\n', year);
else
   if  ( mod(year,100)==0 && mod(year,400) ~= 0 )
      fprintf('%d is an ordinary year\n', year);
   else
      fprintf('%d is a leap year\n', year);
   end
end
```

Sample output:

```
Enter year:  1900
1900 is an ordinary year
```

The leap year "formula" involves integer arithmetic. Integer arithmetic is not quite the same as real arithmetic. For example, the division of one integer by another produces a *quotient* and a *remainder*. Using $\div$ to designate this operation, we see that $23 \div 8 = 2$ with remainder 5. Recall that MATLAB always performs arithmetic calculations using the real (and complex) number system, so we have to do extra work to get the integer quotient and remainder. We will use the MATLAB built-in functions `mod` and `floor` to get integer values in the following "calendar problems."

The program `Example1_5` indicates whether a given year is an ordinary year or a leap year. After a value is read into variable `year`, a question is posed about its divisibility by 4. If the value in `year` is a multiple of 4, then the remainder of `year`$\div 4$ is zero, or we say that "`year` mod 4" is zero. The expression in the outer `if` statement

```
mod(year,4)
```

uses the built-in function `mod` to calculate the *remainder* of the integer division `year`$\div 4$.

The output of `Example1_5` prints within a message an integer value that has no fraction part. We use the substitution sequence "`%d`" to substitute in a variable value as an integer, without a decimal point.

It is instructive to "derive" `Example1_5` taking the "top-down" approach. Assume that `year` houses the value in question. We first examine `year` to see if it is divisible by 4:

```
if  mod(year,4) ~= 0
   fprintf('%d is an ordinary year\n', year);
else
   ⟨The case when year is divisible by 4.⟩
end
```

If `year` is divisible by 4, then we must handle the century years and the solution fragment expands to:

```
if  mod(year,4) ~= 0
   fprintf('%d is an ordinary year\n', year);
else
   if  mod(year,100) == 0
      ⟨The century year is divisible by 100.⟩
   else
      fprintf('%d is a leap year\n', year);
   end
end
```

This handling requires a check if the century year is divisible by 400:

```
if  mod(year,4) ~= 0
   fprintf('%d is an ordinary year\n', year);
else
   ⟨The case when year is divisible by 4.⟩
end
```

If `year` is divisible by 4, then we must handle the century years and the solution fragment expands to:

```
if  mod(year,4) ~= 0
   fprintf('%d is an ordinary year\n', year);
else
   if  mod(year,100) == 0
      if  (year mod 400) <> 0
         fprintf('%d is an ordinary year\n', year);
      else
         fprintf('%d is a leap year\n', year);
      end
   else
      fprintf('%d is a leap year\n', year)
   end
end
```

Often, after a top-down development like this it is possible to simplify the derived code using *and* and *or* operations. With such a manipulation it is possible to remove the innermost `if` and that produces `Example1_5`.

We now show yet another alternate solution to our question. `Example1_6` uses a new "clause" in our `if-else` construct: the `elseif` branch. Notice that `elseif` is *one* reserved word, distinct from the previously introduced reserved words `else` and `if`. Recall that in the simple `if-else` construct, we want to branch our program into *two* alternate paths:

```
if ⟨Condition⟩
   ⟨Things to do if the condition is true.⟩
```

```
% Example 1_6:  Leap year calculation
% Use ELSEIF branch

year= input('Enter year:  ');
if  ( mod(year,4) ~= 0 )
   fprintf('%d is an ordinary year\n', year);
elseif   ( mod(year,100)==0 && mod(year,400) ~= 0 )
   fprintf('%d is an ordinary year\n', year);
else
   fprintf('%d is a leap year\n', year);
end
```

Sample output:

```
Enter year:  1900
1900 is an ordinary year
```

else
⟨*Things to do if the condition is false.*⟩
end

If there are other "conditions" that we need to consider, we can *nest* `if` statements as shown in Examples `1_4` and `1_5` where each individual `if` statement again deals with two alternatives. Think of the `elseif` branch as a quick way to deal with a *third* alternative without nesting:

if ⟨*Condition 1*⟩
⟨*Things to do if the condition 1 is true.*⟩
elseif ⟨*Condition 2*⟩
⟨*Things to do if the condition 2 is true.*⟩
else
⟨*Things to do if all previous conditions are false.*⟩
end

Only *one* of the three branches above will be executed.

How do you decide between using a *nested* `if-else` construct and an `if-elseif-else` construct? The `if-else` construct (with nesting) is more general and is available in almost all programming languages. Furthermore, the `if-else` construct is simple, dealing with just two options at a time, fitting in with the top-down development strategy. However, when a problem at hand requires the selection of one action among three (or more) easily separated alternatives, then the `if-elseif-else` construct is a winner! For now, focus on using top-down development to create your programs instead of worrying about *nested* `if-else` versus `if-elseif-else`. In fact, these options are not mutually exclusive—you can nest an `if-else` inside an `if-elseif-else` or vice versa!

Let us review the general `if` statement:

if ⟨*Condition 1*⟩

```
% Example 1_7:  Compute the number of leap year days between Jan 1, 1900,
% and December 31 of a prescribed year up to the 22nd century.

year= input('Enter a year (1900-2199):  ');
if   (year<2100)
   leapdays= fix((year-1900)/4);
else
   leapdays= fix((year-1900)/4)-1;
end
fprintf('There are %d leap year days from Jan 1, 1900, to ', leapdays);
fprintf('Dec 31, %d.\n', year);
```

Sample output:

```
Enter a year (1900-2199):  2005
There are 26 leap year days from Jan 1, 1900, to Dec 31, 2005.
```

⟨*Things to do if the condition 1 is true.*⟩
elseif ⟨*Condition 2*⟩
⟨*Things to do if the condition 2 is true.*⟩
elseif ⟨*Condition 3*⟩
⟨*Things to do if the condition 3 is true.*⟩
⋮
else
⟨*Things to do if all previous conditions are false.*⟩
end

One `if` statement ends with one reserved word `end`. There can be any number of `elseif` branches but at most one `else` branch in an `if` statement. At most one branch of the `if` statement will execute.

The program `Example1_7` is concerned with the number of leap years that have occurred from 1900 to a specified year and introduces the `fix` function for rounding a real value towards zero. Let `x` be a real value. The value of `fix(x)` is the integer obtained by throwing away the fraction part. Thus, `fix(-4.7)` is the integer -4 while `fix(2.1)` is the integer 2.

Problem 1.6. Modify `Example1_7` so that it changes the "base date" from January 1, 1900 to January 1, 1600.

Problem 1.7. Write a program that solicits a time period T in seconds and then prints its equivalent in units of hours, minutes, and seconds. Thus, if $T = 10000$, then

$$T = 2 \cdot 3600 + 46 \cdot 60 + 40$$

implying that 10000 seconds equals 2 hours, 46 minutes, and 40 seconds.

PROBLEM 1.8. Assume that `L` and `R` are integers whose values satisfy $0 \leq L \leq R$. Define m to be the largest value that the cosine function attains on the set $\{L^o, \ldots, R^o\}$. Thus, if $L = 34$ and $R = 38$, m is the largest of the numbers $\cos(34^o)$, $\cos(35^o)$, $\cos(36^o)$, $\cos(37^o)$, and $\cos(38^o)$. Write a program that solicits `L` and `R` from the user and then determines and prints m.

Many applications require computations that produce integers from reals and vice versa. A nice setting to practice these transitions deals with angles. We need a definition to get started. We say that an angle θ measured in degrees is normalized if $0 \leq \theta < 360$. *Any* angle d can be written in the form

$$d = 360w + \theta$$

where w is an integer called the *winding number* and θ is the normalized angle. Let's consider the problem of computing the winding number and normalization of an angle that is specified in degrees. Assume that `d` is a real variable whose value is non-negative. We need to determine how many integral multiples of 360 are contained in d. Again, we will use the `fix` function. The fragment

```
w= fix(d/360);
theta= d - 360*w;
```

assigns the winding number to `w` and the normalization to `theta`. Why don't we use the `mod` function to calculate `theta`? The `mod` function usually is associated with integer values in most programming languages, therefore we show the calculation above without using `mod`. However, function `mod` in MATLAB works for real values as well, so we could have used the assignment statement `theta= mod(d,360)`.

While `fix` takes a real value and obtains an integer by "removing" the fraction part, `round` takes a real value and returns the value of the nearest integer. Thus, `round(-4.7)` is -5 while `round(2.8)` is 3. In case there are two equally distant nearest integers, then the one further away from 0 is selected. Thus, `round(2.5)` is 3. As an application, here is a statement that converts an angular measure in radians stored in `angle` to the nearest integral degree:

```
degree= round((angle/pi)*180);
```

PROBLEM 1.9. A sign on a taxi reads "5 dollars for the first eighth mile or fraction thereof and 2 dollars for each successive eighth mile or fraction thereof." Here is a small table that clarifies the method of charging:

Distance	Charge
0.10	5
0.20	7
0.99	19
1.0	19

Write a program that solicits the distance traveled and prints the charge for the trip.

PROBLEM 1.10. **(a)** Give a boolean expression that is true if the value in a real variable `x` is closer closer to an integer than to a real number whose fractional part equals one-half. **(b)** Assume that `a` and `b` are real with $a < b$. Write a fragment that prints the number of integers in the interval $[a, b]$. **(c)** Assume that `a`, `b`, and `c` are real

variables with positive values and that a and b are not integral multiples of c. Write a fragment that prints the message "ok" if there is a real number strictly in between a and b that is an integral multiple of c. Indicate the type of any additional variables required by your solution. Assume that $a < b$.

Problem 1.11. Assume that `L` and `R` are real with $L < R$. Write a program that reads L and R and prints the maximum value of $cos(x)$ on $[L, R]$.

Problem 1.12. Suppose we have two rays which make make positive angles a and b with the positive x-axis. Write a program that reads the two angles (in radians), and determines whether or not the two rays make an acute angle. Examples: $a = \pi/6$ and $b = 23\pi/6$ do make an acute angle, while $a = \pi/6$ and $b = 3\pi$ do not.

Problem 1.13. Write a program that reads in two nonnegative real numbers a and b and indicates whether or not the two rays that make positive angles a^o and b^o with the positive x-axis are in the same quadrant. For clarity we assume that if $0^o <= x < 360^o$ then

$$x \text{ is in the } \left\{ \begin{array}{c} \text{First} \\ \text{Second} \\ \text{Third} \\ \text{Fourth} \end{array} \right\} \text{ quadrant if } \left\{ \begin{array}{c} 0 \leq x < 90 \\ 90 \leq x < 180 \\ 180 \leq x < 270 \\ 270 \leq x < 360 \end{array} \right\}.$$