Chapter 2

Numerical Exploration

All work and no play does not a computational scientist make. It is essential to be able to *play* with a computational idea before moving on to its formal codification and development. This is very much a comment about the role of intuition. A computational experiment can get our mind moving in a creative direction. In that sense, merely watching what a program does is no different then watching a chemistry experiment unfold: it gets us to think about concepts and relationships. It builds intuition.

The chapter begins with a small example to illustrate this point. The area of a circle is computed as a limit of regular polygon areas. We "discover" π by writing and running a sequence of programs.

Sometimes our understanding of an established result is solidified by experiments that confirm its correctness. In §2.2 we check out a theorem from number theory that says $3^{2k+1}+2^k$ is divisible by 7 for all positive integers k.

To set the stage for more involved "computational trips" into mathematics and science, we explore the landscape of floating point numbers. The terrain is *finite* and *dangerous*. Our aim is simply to build a working intuition for the limits of floating point arithmetic. Formal models are not developed. We're quite happy just to run a few well chosen computational experiments that show the lay of the land and build an appreciation for the inexactitude of real arithmetic.

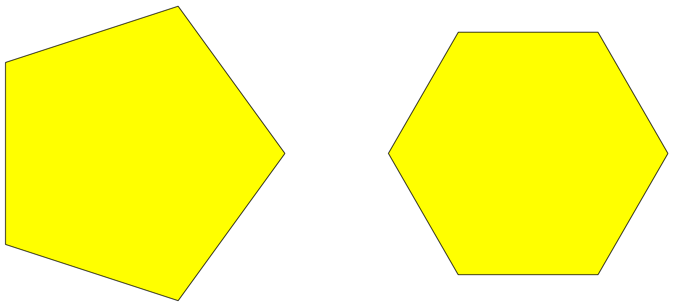


FIGURE 2.1 *Regular n-gons*

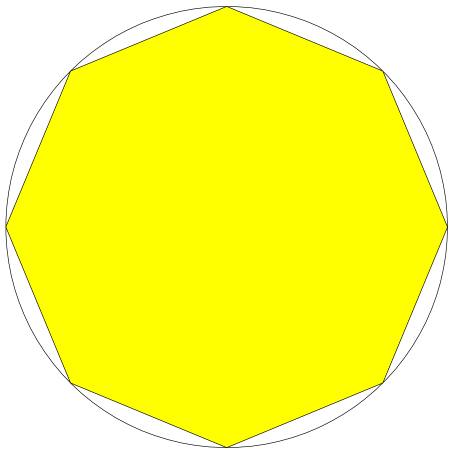


FIGURE 2.2 *Inscribed n-gon*

The design of effective problem-solving *environments* for the computational scientist is a research area of immense importance. The goal is to shorten the path from concept to computer program. We have much to say about this throughout the text, In §2.4 we develop the notion of an interactive framework that fosters the exploration of elementary computational ideas.

2.1 Limits

A polygon with n equal sides is called a *regular n-gon*. FIGURE 2.1 illustrates two cases. Given n equally spaced points around a circle C, there are two ways to construct a regular n-gon. One is simply to connect the points in order. Each point is then a *vertex* of the n-gon which is said to be *inscribed* in C. See FIGURE 2.2. On the other hand, the tangent lines at each point define a regular n-gon that *circumscribes* C. See FIGURE 2.3. If C has radius one, then the areas of these two regular n-gons are given by

$$A_n = (n/2)\sin(2\pi/n) \qquad \text{(Inscribed)}$$

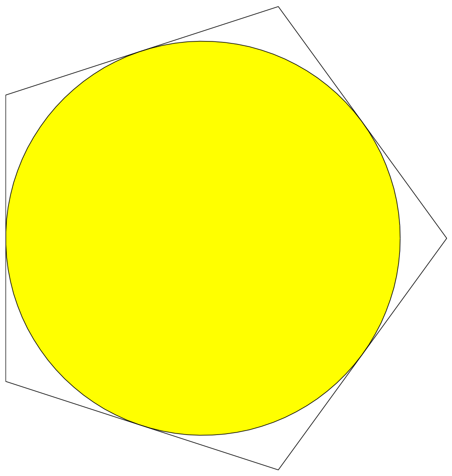


Figure 2.3 *Circumscribed n-gon*

$$B_n \quad = \quad n\tan(\pi/n) \qquad \text{(Circumscribed)}$$

These formulas can be derived by chopping the n-gon into n equal triangles and summing their areas. Now for any value of n that satisfies $n \geq 3$, the fragment

```
% Fragment A
innerA= (n/2)*sin(2*pi/n);
outerA= n*sin(pi/n)/cos(pi/n);
fprintf('%4d %9.6f %9.6f \n', n, innerA, outerA);
```

prints n, A_n, and B_n. It follows that

```
n= 3;
innerA= (n/2)*sin(2*pi/n);
outerA= n*sin(pi/n)/cos(pi/n);
fprintf('%4d %9.6f %9.6f \n', n, innerA, outerA);
n= 4;
innerA= (n/2)*sin(2*pi/n);
outerA= n*sin(pi/n)/cos(pi/n);
fprintf('%4d %9.6f %9.6f \n', n, innerA, outerA);
n= 5;
innerA= (n/2)*sin(2*pi/n);
outerA= n*sin(pi/n)/cos(pi/n);
fprintf('%4d %9.6f %9.6f \n', n, innerA, outerA);
n= 6;
innerA= (n/2)*sin(2*pi/n);
outerA= n*sin(pi/n)/cos(pi/n);
fprintf('%4d %9.6f %9.6f \n', n, innerA, outerA);
```

produces a 4-line table that reports on the areas for n = 3,4,5 and 6:

```
3  1.299038  5.196152
4  2.000000  4.000000
5  2.377641  3.632713
6  2.598076  3.464102
```

This approach to table generation is tedious. It would be much handier if we could specify the repetition as follows:

```
⟨Execute the following fragment for n= 3,4,5,and 6:⟩
   innerA= (n/2)*sin(2*pi/n);
   outerA= n*sin(pi/n)/cos(pi/n);
   fprintf('%4d %9.6f %9.6f \n', n, innerA, outerA);
```

The `for`-loop is designed precisely for this kind of situation:

```
for n= 3:1:6
   innerA= (n/2)*sin(2*pi/n);
   outerA= n*sin(pi/n)/cos(pi/n);
   fprintf('%4d %9.6f %9.6f \n', n, innerA, outerA);
end
```

Here is how it works. The line of code that contains the keyword `for`, sometimes called the *loop header*, specifies all the values that the *index variable* `n` will take on. The expression `3:1:6` reads "3 to 6 with increments of 1," meaning that `n` will take on the values 3, 4, 5, and 6. The loop starts with `n` taking on the first value, 3. Then the *loop body*

```
   innerA= (n/2)*sin(2*pi/n);
   outerA= n*sin(pi/n)/cos(pi/n);
   fprintf('%4d %9.6f %9.6f \n', n, innerA, outerA);
```

is executed. For this *pass* through the loop, the value of `n` is 3 and the inner and outer areas are calculated and printed. After this, `n` takes on the second value, 4, and the loop body executes again. This pattern is repeated where `n` takes on the values 5 and 6, the remaining values as specified by the loop header.

The beauty of this arrangement is that it is just as easy to produce a "3-to-100" table as it is to produce a "3-to-6" table. Indeed, the program `Example2_1` enables us to let n range between any two integers of our choice. Notice the judicious use of comments to assist in the understanding of the loop. The example reveals the general form of the `for`-loop:

```
for ⟨Index Variable⟩ = ⟨Left Bound⟩ : ⟨Increment⟩ : ⟨Right Bound⟩
   ⟨The fragment to be repeated goes here.⟩
end
```

Within the loop body, the index variable can be referenced, but it should never appear to the left of the assignment operator. The expression

```
⟨Left Bound⟩ : ⟨Increment⟩ : ⟨Right Bound⟩
```

```
% Example 2_1:  Compute the areas of regular polygons that are inscribed
% and circumscribed in the unit circle.

% Lower & upper bounds for computation
low= input('Enter least number of sides:  ');
high= input('Enter most number of sides:  ');

% Print the column headings
fprintf('\n n\t A(n)\t B(n)\n');
fprintf('---------------------------\n');

% Compute and print areas of n-gons
for n= low:1:high
   innerA= (n/2)*sin(2*pi/n);       % inscribed area
   outerA= n*sin(pi/n)/cos(pi/n);   % circumscribed area
   fprintf('%4d %9.6f %9.6f \n', n, innerA, outerA);
end

fprintf('---------------------------\n');
```

Sample output:

```
Enter least number of sides:  15
Enter most number of sides:  20

  n     A(n)     B(n)
--------------------------
  15  3.050525  3.188348
  16  3.061467  3.182598
  17  3.070554  3.177851
  18  3.078181  3.173886
  19  3.084645  3.170539
  20  3.090170  3.167689
--------------------------
```

generates the list of values that the index variable will take on, one for each pass of the loop. The ⟨*increment*⟩ may be negative in order to have the index count down instead of up. If ⟨*right bound*⟩ < ⟨*left bound*⟩ and the ⟨*increment*⟩ is positive, or if ⟨*right bound*⟩ > ⟨*left bound*⟩ and the ⟨*increment*⟩ is negative, then the expression does not generate a valid list of values (an *empty set* in mathematical term) for the index variable and the body of the entire loop is skipped and execution continues with the terminating `fprintf` statement.

Instead of soliciting the starting and stopping values of n, we may wish to have the user enter the starting value of n and the *length* of the table that is to be produced. We merely alter the beginning of the program body in `Example2_1` by introducing the variable `lines` for the length of the table, as follows:

```
low= input('Enter least number of sides:  ');
lines= input('Enter length of table:  ');
fprintf('\n n\t A(n)\t B(n)\n');
fprintf('---------------------------\n');

for n= low:1:low+lines-1
    ⋮
end
```

This modification shows how expressions involving variables can “show up” in the `for`-loop statement. To build confidence in the correctness of the loop bounds, it is often useful to check out a few cases “by hand:”

`low`	`lines`	Desired Sequence	`low+lines-1`
3	0	(None)	2
3	1	3	3
3	4	3,4,5,6	6

An increment of one is so commonly used that it is the *default* value of the increment in a `for` loop. For example, the `for` loop header `for k= 3:1:6` is the same as `for k= 3:6`. In both cases, index variable `k` takes on the values 3, 4, 5, 6 one at a time.

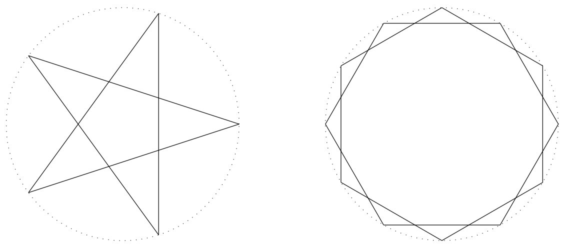
`For`-loops are useful whenever repetition is needed in a regimented fashion, i.e., the loop index variable increases or decreases in *fixed* increments. Suppose we want to produce a table of area values for $n = 5, 10, 15, \ldots, 100$. Repetition is certainly involved because the required table has multiple lines with `n` increasing by 5 in each line. We will then use the following loop:

```
for n= 5:5:100
    ⟨Print the n-th line of the table⟩
end
```

We easily determine that the loop body will execute 20 times. We call such a case where it is easy to determine how many times the loop body will execute *definite iteration.*

Problem 2.1. Write a program that reads in integers a, b, and m where $a \leq b$ and $m > 0$ and prints a table of sine values. The table entries should range from a^o to b^o with spacing $(1/m)^o$. Thus, if $a = 1$, $b = 3$, and $m = 2$, then the table should report the sine of 1^o, 1.5^o, 2^o, 2.5^o, and 3^o.

PROBLEM 2.2. Here are the *n-stars* of size 5 and 12:



The precise definition of an n-star is not important. Suffice it to say that the area of an n-star that is inscribed in the unit circle is given by

$$A(n) = \begin{cases} n\dfrac{\cos(\pi/(2n)) - \cos(3\pi/(2n))}{2\sin(3\pi/(2n))} & \text{if } n \text{ is odd} \\ \\ n\,\dfrac{1-\cos(2\pi/n)}{2\sin(2\pi/n)} & \text{if } n \text{ is even} \end{cases}$$

and its perimeter by

$$E(n) = \begin{cases} \dfrac{\sin(\pi/n)}{\sin(3\pi/(2n))} & \text{if } n \text{ is odd} \\ \\ \dfrac{\sin(\pi/n)}{\sin(2\pi/n)} & \text{if } n \text{ is even} \end{cases} .$$

Write a program that prints a table whose k-th line has the values n, $A(n)$, $E(n)$, $A(n+1)$, $E(n+1)$ where $n = 10k$. The value of k should range from 1 to 20.

As n increases, the regular inscribed and circumscribed n-gons converge to the circle. Since the area of the unit circle is π, we have

$$\lim_{n\to\infty} A_n = \pi \qquad \lim_{n\to\infty} B_n = \pi .$$

Moreover, for all $n >= 3$ we have

$$A_n < \pi < B_n .$$

This limiting behavior allows us to formulate iteration problems where the total number of steps is not known in advance. For example, suppose we want to print a "3-to-n table where n is the smallest integer such that

$$B_n - A_n \leq 0.0001.$$

We know that such an n exists because A_n and B_n each converge to π as n increases.

One solution would be to use a `for`-loop with a large, "safe" upper bound and an `if` inside the loop body to guard against unwanted printing:

```
for n= 3:10000
   innerA= (n/2)*sin(2*pi/n);
   outerA= n*sin(pi/n)/cos(pi/n);
   if  outerA - innerA > 0.0001
         fprintf('%4d %9.6f %9.6f \n', n, innerA, outerA);
   end
end
```

But this is a flawed problem-solving strategy for two reasons:

> The guessing of a "safe" upper bound may be very difficult in practice. In our example, if the chosen upper bound is too small, then the table will be too short.
>
> It is inefficient. In our example, once the last line is printed, all subsequent area computations are superfluous.

What we need is an ability to "jump out" of the iteration as soon as the condition

```
outerA - innerA > 0.001
```

is false. The `while`-loop is designed for this kind of situation and the program `Example2_2` highlights this point. Here is how the program works. Before the loop begins, the variables `n`, `innerA`, and `outerA` are assigned the values 3, A_3, and B_3 respectively. The condition in the `while` statement acts as a guard. If the inner and outer areas differ by more than .001, then the boolean expression is true and the loop body

```
fprintf('%4d %9.6f %9.6f \n', n, innerA, outerA);
n= n+1;
innerA= (n/2)*sin(2*pi/n);
outerA= n*sin(pi/n)/cos(pi/n);
```

is executed. This prints a line in the table and updates the triplet of variables `n`, `innerA`, and `outerA`. Once again the boolean expression in the `while` statement is evaluated. If it is true, then the loop body is again executed. From what we know about the problem, eventually the inner and outer areas will get so close that the condition `outerA - innerA >.0.001` is false. When this happens, the execution of the `while`-loop terminates and control passes to the final `fprintf`.

In general,`while`-loops are structured as follows:

```
⟨Initializations⟩
while ⟨Boolean Expression⟩
   ⟨Fragment to be repeated goes here.⟩
end
```

Note that a `for`-loop can always be written as a `while`-loop. For example

```
for n= 3:6
   innerA= (n/2)*sin(2*pi/n);
   outerA= n*sin(pi/n)/cos(pi/n);
   fprintf('%4d %9.6f %9.6f \n', n, innerA, outerA);
end
```

is equivalent to

```
% Example 2_2:  Convergence of inner and outer areas.

% Print the column headings
fprintf(' n\t A(n)\t B(n)\n');
fprintf('---------------------------\n');

% Initialize n, innerA, and outerA
n= 3;
innerA= 3*sqrt(3)/4;    % inscribed area
outerA= 3*sqrt(3);      % circumscribed area

% Compute and print areas until convergence
while outerA - innerA > .001
   fprintf('%4d %9.6f %9.6f \n', n, innerA, outerA);
   n= n+1;
   innerA= (n/2)*sin(2*pi/n);
   outerA= n*sin(pi/n)/cos(pi/n);
end

fprintf('---------------------------\n');
```

Output:

```
 n      A(n)          B(n)
----------------------------
  3   1.299038      5.196152
  4   2.000000      4.000000
  5   2.377641      3.632713
             ⋮
174   3.140910      3.141934
175   3.140918      3.141930
176   3.140925      3.141926
----------------------------
```

```
n= 3;
while n<=6
   innerA= (n/2)*sin(2*pi/n);
   outerA= n*sin(pi/n)/cos(pi/n);
   fprintf('%4d %9.6f %9.6f \n', n, innerA, outerA);
   n= n+1;
end
```

When confronted with an iterative computation, a `for`-loop is usually appropriate if the number of iterations is known in advance *and* the counting is "regular." Otherwise, the situation calls for a `while` loop.

To illustrate another type of counting, let's modify `Example2_2` so that n is repeatedly doubled instead of incremented:

```
n=4;   outerA= 4;   innerA= 2;
while outerA - innerA > 0.00001
      fprintf('%4d %9.6f %9.6f \n', n, innerA, outerA);
      n= 2*n;
      innerA= (n/2)*sin(2*pi/n);
      outerA= n*sin(pi/n)/cos(pi/n);
end
```

Notice how `n` houses the required powers of two which are obtained through repeated doubling process: `n=2*n`. The value of `n` starts out at 4. The first execution of `n=2*n` replaces this value with 8. When `n=2*n` is carried out during the next pass, the"8" becomes a "16", etc.

The repeated doubling also makes it possible to circumvent the explicit references to `sin`, `cos`, and `pi`. The idea is to use the half-angle formulae

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

to produce the required sines and cosines. The angles that "show up" during the process are all repeated halvings of $\pi/4$:

n	Angle (Radians)
4	$\pi/4$
8	$\pi/8$
16	$\pi/16$
32	$\pi/32$
$\vdots$	$\vdots$

Since the sine and cosine of $\pi/4$ are both $1/\sqrt{2}$ we obtain `Example2_3`. Unlike `Example2_2` that uses π, `Example2_3` computes π.

```
% Example 2_3:  Compute pi.

% Print the column headings
fprintf('\n n\t A(n)\t B(n)\n');
fprintf('---------------------------\n');

% Initialize n, innerA, outerA, and c
n= 4;
innerA= 2;   % inscribed area for n=4
outerA= 4;   % circumscribed area for n=4
c= 1/sqrt(2); % {cos(pi/4)}

% Approximate pi
while outerA - innerA > .0001
   fprintf('%4d %9.6f %9.6f \n', n, innerA, outerA);
   n= 2*n;
   s= sqrt((1-c)/2);   % {s = sin(pi/n)}
   c= sqrt((1+c)/2);   % {c = cos(pi/n)}
   innerA= n*s*c;
   outerA= n*s/c;
end
```

Output:

```
 n      A(n)         B(n)
----------------------------
  4   2.000000       4.000000
  8   2.828427       3.313708
 16   3.061467       3.182598
 32   3.121445       3.151725
 64   3.136548       3.144118
128   3.140331       3.142224
256   3.141277       3.141750
512   3.141514       3.141632
```

Problem 2.3. The volume of a pyramid whose base is a unit regular n-gon of radius r and whose height is one is given by

$$V(n) = \frac{nr^2}{6}\sin(2\pi/n).$$

The volume of a cone whose base is the unit circle and which has height one is given by

$$V_\infty = \frac{\pi}{3}.$$

Write a program that prints the smallest n so that $V_n/V_\infty > .99$.

Problem 2.4. The area of a triangle whose sides have length a, b, and c is given by

$$E = \sqrt{\mu(\mu-a)(\mu-b)(\mu-c)}$$

where $\mu = (a+b+c)/2$. Thus, the area of an equilateral triangle with $a = b = c = 2$ is given by

$$E = \sqrt{3(3-2)(3-2)(3-2)} = \sqrt{3}.$$

Let T_k be the triangle with sides $a = 2$, $b = 2 + 1/2^k$, and $c = 2 - 1/2^k$. As k increases, T_k looks increasingly like an equilateral triangle and its area is increasingly close to $E = \sqrt{3}$. Write a program that prints the smallest value of k such that $|E_k - \sqrt{3}| \le 0.0001$ where E_k is the area of T_k.

Problem 2.5. Write a program that computes the smallest positive k such that $\cot(\pi/2^k) > 1000$. Make repeated use of the half-angle formulae. Here, $\cot(x)$ is the cotangent function: $\cot(x) = \cos(x)/\sin(x)$.

2.2 The Floating Point Terrain

We now turn our attention to the numerical landscape of real numbers. It turns out that floating point arithmetic is inexact. If we think of the computer as a telescope, then this affects its resolution. A good astronomer understands the limits of a telescope that is being used to observe the galaxies. A good computational scientist understands the limits of a computer that is being used to observe the "numerical versions" of those galaxies.

There are just a handful of things to know and the starting point is the representation of a numerical value in scientific notation:

$$x \;=\; +.123\times 10^{+3} \quad y \;=\; -.1000\times 10^{-05} \quad z \;=\; .00000\times 10^{00}\ .$$

The fraction part is called the *mantissa*. It has a sign and length. The mantissas for x, y, and z have length 3, 4, and 5 respectively. The mantissa is always less than one in absolute value and for non-zero numbers, the most significant digit is non-zero. The representations $x \;=\; 1.23\times 10^{+2}$ and $x \;=\; .0123\times 10^4$, are not *normalized*.

The *exponent* part of the notation indicates the power to which the base is raised. In the base-10 examples above, x, y, and z have exponents 3, -5 and 0 respectively. Like the mantissa, the exponent has a sign and a length.

The representation of real numbers in Matlab, type `double`, follows the above style. But there is an added wrinkle: *the amount of memory that is allocated for the mantissa and exponent is fixed and finite.* For example, a computer may permit 3-digit mantissas and 1-digit exponents.[1]

[1]These are unrealistic parameters but they are good enough to communicate the main ideas. Today's "standard" is the IEEE double-precision binary floating point system, which has 53 binary digits in the mantissa and 11 binary digits in the exponent.

Here is a list of some numbers and and their *floating point* representation:

$$a = 12.3 \qquad \texttt{a} = \begin{array}{|c|c|c|c||c|c|}\hline + & 1 & 2 & 3 & + & 2 \\ \hline\end{array}$$

$$b = .000000123 \qquad \texttt{b} = \begin{array}{|c|c|c|c||c|c|}\hline + & 1 & 2 & 3 & - & 6 \\ \hline\end{array}$$

$$c = -12.3 \qquad \texttt{c} = \begin{array}{|c|c|c|c||c|c|}\hline - & 1 & 2 & 3 & + & 2 \\ \hline\end{array}$$

$$d = 0.0 \qquad \texttt{d} = \begin{array}{|c|c|c|c||c|c|}\hline + & 0 & 0 & 0 & + & 0 \\ \hline\end{array}$$

Note that with the limited mantissa length, some numbers can only be stored approximately:

$$a = 12.34 \qquad \texttt{a} = \begin{array}{|c|c|c|c||c|c|}\hline + & 1 & 2 & 3 & + & 2 \\ \hline\end{array}$$

$$b = 12.37 \qquad \texttt{b} = \begin{array}{|c|c|c|c||c|c|}\hline + & 1 & 2 & 4 & + & 2 \\ \hline\end{array}$$

$$c = \pi \qquad \texttt{c} = \begin{array}{|c|c|c|c||c|c|}\hline + & 3 & 1 & 4 & + & 1 \\ \hline\end{array}$$

The reasonable thing to do if there isn't enough "room" to store the exact mantissa is to *round*. Since 12.37 is closer to $.124 \times 10^2$ than $.123 \times 10^2$, the former value is stored. In case of a tie, we assume that the computer rounds up.

To drive home the point that the set of floating point numbers is finite, we display the smallest and the largest positive floating point numbers:

$$\mathit{min} = .0000000001 \qquad \texttt{min} = \begin{array}{|c|c|c|c||c|c|}\hline + & 1 & 0 & 0 & - & 9 \\ \hline\end{array}$$

$$\mathit{max} = 999000000 \qquad \texttt{max} = \begin{array}{|c|c|c|c||c|c|}\hline + & 9 & 9 & 9 & + & 9 \\ \hline\end{array}$$

Some numbers just do not fit at all because the exponent length is too short. For example, if $x = 1234567890 = .1234567890 \times 10^{10}$, then a 2-digit exponent is required. Likewise, $x = .0000000000999 = .999 \times 10^{-10}$ cannot be represented because again, two digits are required to hold the exponent.

A reasonable way to model the addition, subtraction, multiplication or division of two floating point numbers is as follows:

1. Perform the operation exactly.
2. Put the answer in normalized scientific form.
3. Round the mantissa to the allotted number of mantissa digits.

Thus, the addition of x and y where

$$x = 12.3 \qquad \texttt{x} = \begin{array}{|c|c|c|c||c|c|}\hline + & 1 & 2 & 3 & + & 2 \\ \hline\end{array}$$
$$y = 5.27 \qquad \texttt{y} = \begin{array}{|c|c|c|c||c|c|}\hline + & 5 & 2 & 7 & + & 1 \\ \hline\end{array}$$

proceeds as follows:

- $12.3 + 5.27 = 17.57$.
- $17.57 = .1757 \times 10^2$

```
% Example 2_4:  Explore floating point precision.

fprintf('\n k 1+1/2^k \n');
fprintf('---------------------------\n');

k= 0;
small= 1;
onePlusSmall= 2;

% Compute (1 + 1/(2^k)) for k=1..., until rounding error occurs
while  onePlusSmall ~= 1
   k= k + 1;
   small= small/2;   % {small = 1/(2^k)}
   onePlusSmall= 1 + small;
   fprintf('%4d %.16f\n', k, onePlusSmall);
end
```

Output:

```
  k   1 + 1/2^k
---------------------------
  1   1.5000000000000000
  2   1.2500000000000000
  3   1.1250000000000000
  4   1.0625000000000000
              ⋮
 49   1.0000000000000018
 50   1.0000000000000009
 51   1.0000000000000004
 52   1.0000000000000002
 53   1.0000000000000000
```

- `z= x+y`   z =

+	1	7	6	+	2

A consequence of this model of floating point arithmetic is that *rounding errors* usually attend every floating point arithmetic operation. This forces us to depart from the "exact arithmetic" mindset when developing programs that manipulate real numbers. This is a complicated issue and we will only be able to scratch the surface in this introductory text. However, with a few well-chosen examples we can build up an appreciation for the inexactness of floating point arithmetic.

For example, it is possible for the assignment `x= x+y` not to change the value of `x` even if the value in `y` is nonzero. In the model floating point system that we have been using, if $x = 1.00$ and $y = .001$, then the computed sum of `x` and `y` is 1 because 1.001 rounds to 1.00. `Example2_4` illustrates this point. The loop terminates as soon as the floating point addition of one and $1/2^k$ is one. Instead of doing "repeated halving" (repeating the assignment `small= small/2`) to calculate $1/2^k$ for increasing k, one can use the exponentiation operator "^" in `Example2_4`.

The length of the mantissa defines the precision of the floating point arithmetic. Roughly, the relative error in a floating point operation is about b^{-t+1} where b is the base and t is the length of the mantissa. We refer to b^{-t+1} as the *machine precision.* A typical mantissa might be comprised of 53 base-2 digits. This means that machine precision is about $2^{-53} \approx 10^{-16}$. In MATLAB, the value of the machine precision, also called machine *eps*ilon, is stored as the built-in constant `eps`.

PROBLEM 2.6. Compare the output of the following program with what should be produced in exact arithmetic. Explain the difference.

```
% Problem 2_6
b= 2e-17;
c= 1;
sumBC= b + c;
newB= sumBC - c;
fprintf('b:  %.16e %.16e\n', b, newB);
```

PROBLEM 2.7. Modify `Example2_4` so that instead of the table it prints only the value of the largest k that gives $1 + 1/2^k > 1$. Hint: remove the `fprintf` inside the loop, maintain a variable that houses the the "previous" value of `small`, and print the value of that variable after the loop terminates.

So far we have spent most of the time discussing the mantissa and what its finiteness implies. But there are also a couple of things to discuss about the exponent part of the floating point number. When a floating point operation renders a nonzero answer that is too small to represent, then an *underflow* occurs and the result is set to zero. `Example2_5` computes $1/2^k$ for increasingly big k. Eventually, an underflow is produced. At the other end of the scale, a floating point operation can result in an answer that is too big to represent. This is called *floating point overflow.* Overflows produce a special value called `Inf`. In MATLAB, the largest numeric value is stored in the built-in constant `realmax`. Analogous to `Example2_5` that examines the effect of repeated halving, `Example2_6` performs repeated doubling until floating point overflow occurs.

PROBLEM 2.8. Modify `Example2_5` so that only the lines associated with $k = 10, 20, 30$,etc. are printed.

PROBLEM 2.9. Modify `Example2_6` so that instead of printing the table, it just prints the second to last line in the table. Hint: remove the `fprintf` inside the loop, maintain a variable that houses the the "previous" value of `big`, and print the value of that variable after the loop terminates.

PROBLEM 2.10. Write a program that prints the largest integer n so that `x= exp(n)` does *not* assign the value `Inf` to the variable `x`.

```
% Example 2_5:  Explore floating point underflow.

fprintf('\n 1/2^k \n');
fprintf('--------------------------\n');

k= 0;
small= 1;

% Repeatedly half small--k times until 1/(2^k) underflows
while  small ~= 0
   small= small/2;   % {small = 1/(2^k)}
   k= k+1;
   fprintf('%4d %.16e\n', k, small);
end
```

Output:

```
   k   1/2^k
-------------------
   1   5.00e-01
   2   2.50e-01
   3   1.25e-01
   4   6.25e-02
         ⋮
1072   1.98e-323
1073   9.88e-324
1074   4.94e-324
1075   0.00e+00
```

```
% Example 2_6:  Explore floating point overflow.

fprintf('\n k 2^k \n');
fprintf('-------------------\n');

k= 0;
big= 1;

% Repeatedly double big--k times until 2^k overflows
while  big ~= Inf
   big= big*2;   % {big = 2^k}
   k= k+1;
   fprintf('%4d %.2e \n', k, big);
end
```

Output:

```
   k   2^k
------------------
   1   2.00e+00
   2   4.00e+00
   3   8.00e+00
   4   1.60e+01
          ⋮
1021   2.25e+307
1022   4.49e+307
1023   8.99e+307
1024   Inf
```

2.3 Confirming Conjectures

Number theory is a branch of mathematics that deals with the integers and their properties. Many number theoretic results can be couched in elementary terms and can be explored by programs that involve simple iterations. Let us consider the affirmation of the following fact:

If k is a nonnegative integer, then $3^{2k+1} + 2^{k+2}$ is divisible by 7.

This means that there is no remainder when we divide $3^{2k+1} + 2^{k+2}$ by 7. To acquire an understanding of *any* mathematical fact like this, it is best to begin with a few pencil-and-paper verifications:

	$k = 0$	$k = 1$	$k = 2$	$k = 3$
2^{k+2}	4	8	16	32
3^{2k+1}	3	27	243	2187
$2^{k+2} + 3^{2x+1}$	7	35	259	2219

It is easy to check that the sum of the indicated powers is indeed divisible by 7.

`Example2_7` checks the conjecture for $k = 0, \dots, 24$. Note that the last column of output reveals the remainder of $2^{k+2}+3^{2k+1} \div 7$ and that it is identically zero for the values $k = 0, \dots, 24$. We use intermediate variables `p2` and `p3` to store the terms involving the powers of 2 and 3 separately to facilitate our discussion below. Notice the use of a comment at the "top" of the `for` loop for defining the important relationship of the variables in the loop.

If we increase the loop bound to 25, then an error occurs even as the program runs to completion. What goes wrong? At first glance, one may think that an overflow occurs because the sum `s` becomes too big to be stored. However, some simple checks reveal that at $k = 25$, `p3` $\approx 2 \times 10^{24}$ while the largest possible real value is many times the magnitude of `p3` (`realmax` $\approx 10^{308}$). Therefore overflow is *not* the cause of the error—another kind of error occurs before the sum `s` gets large enough to cause an overflow. The real reason for the error is that `p2` becomes negligible relative to `p3` as k becomes "large"! At $k = 25$, `p2+p3` in *floating point arithmetic* gives a result that is *different* from the value of `p2+p3` in *exact* arithmetic. Review §2.2 to see the more detailed explanation of this loss of accuracy in floating point arithmetic using our toy floating point model with a 3-digit mantissa and 1-digit exponent.

`Example2_7` does not *prove* anything. It merely confirms a few special cases cases of a general result. In a research context, a mathematician may choose to run an experiment like this before investing time in a rigorous analytical proof of a general result.

PROBLEM 2.11. Modify `Example 2_7` to verify the conjecture for as many k as possible *without* assuming $k = 24$ to be the upper bound. I.e., your program should find the largest k that can be used. Hint: use a `while` loop and think about what the stopping condition should be given the inaccuracy of floating point arithmetic as discussed above.

```
% Example 2_7:  Confirm that 3^(2n+1)+2^(n+2) is divisible by 7 for n=0..24.

fprintf('\n n 3^(2n+1) + 2^(n+2) Remainder\n');
fprintf('----------------------------------------\n');

% Verify conjecture for n = 0..24
for n= 1:24
   % {s = p2 + p3 = 2^(n+2) + 3^(2*n+1)}
   p2= 2^(n+2);
   p3= 3^(2*n+1);
   s= p2 + p3;
   fprintf('%2d %24.0f %11d \n', n, s, mod(s,7));
end
```

Output:

```
 n        3^(2n+1) + 2^(n+2)   Remainder
----------------------------------------
 0                         7           0
 1                        35           0
 2                       259           0
 3                      2219           0
                     ⋮
21     328256967394545434624           0
22    2954312706550850387968           0
23   26588814358957535526912           0
24  239299329230617593249792           0
```

PROBLEM 2.12. The integer next above $(\sqrt{3}+1)^{2n}$ is divisible by 2^{n+1}. Write a program that confirms this for $n = 1,\ldots,6$.

PROBLEM 2.13. The product of three consecutive whole numbers is exactly divisible by 504 if the middle one is the cube of a whole number. Verify this for the triplets (n^3-1, n^3, n^3+1) with $n = 2,\ldots,10$.

PROBLEM 2.14. Let a_n be the n-th non-perfect square among positive integers. Thus, $a_1 = 2$, $a_2 = 3$, $a_3 = 5$, etc. For $n = 1$ to 10000, confirm that $a_n = n + round(\sqrt{n})$.

PROBLEM 2.15. Let n be a positive integer and let $b(n)$ be the minimum value of $k + (n/k)$ as k is allowed to range through all positive integers. It can be shown that $b(n)$ and $\sqrt{4n+1}$ have the same integer part. Confirm this for all $n \leq 1000$. Hint: You can compute $b(n)$ without a loop.

PROBLEM 2.16. There are at least seven positive integers x that make $x(x+180)$ the square of an integer. Write a program that confirms this conjecture.

PROBLEM 2.17. Write a program that reads in a positive integer q and prints a list of all powers of q that are less then or equal to `realmax`. Organize the computation so that integer overflow does not occur.

2.4 Interactive Frameworks

In the preceding examples and problems, the loops execute without human intervention. Indeed, that is the power of the loop concept, for it makes it possible to specify a very extensive calculation with just a few lines of code. However, loops have an important role to play in the design of *interactive frameworks* that can be used to test computational ideas before they are encapsulated in "serious code."

To illustrate this, let us pretend that `sqrt` is not available and that we want to develop a method for computing $\sqrt{a}$ where a is a positive real number[2]. Here is our idea. Think of the $\sqrt{a}$ problem as the problem of producing a sequence of increasingly square rectangles each of which has area a. If x is an estimate of $\sqrt{a}$, then we associate with x a rectangle with base x and height a/x. Our geometric intuition tells us that $\sqrt{a}$ is in between x and a/x and so we conjecture that

$$x_{new} = \frac{1}{2}\left(x + \frac{a}{x}\right)$$

is a better approximation to $\sqrt{a}$. Here is a fragment that applies this refinement idea two times after prompting for a and an initial guess for its square root:

```
a= input('\nEnter a positive real number:  ');
x= input('Enter an initial guess for the square root:  ');
x= (x+(a/x))/2;
x= (x+(a/x))/2;
absoluteError= abs(x-sqrt(a));
```

[2]What we are about to develop is in fact the root extraction method used by `sqrt`.

```
relativeError= absoluteError/sqrt(a);
fprintf('\nComputed root = %f\n', x);
fprintf('Absolute error = %e\n', absoluteError);
fprintf('Relative error = %e\n', relativeError);
```

It prints the approximate square root x together with its absolute error $|x - \sqrt{a}|$ and its relative error $|x - \sqrt{a}|/\sqrt{a}$.

A program that permits the trial of a single example isn't very handy. Most likely we would want to test our square root idea on a number of different a-values and to explore how the quality of the initial guess affects the accuracy of the computed square root. To that end, we can embed the above fragment in a while loop and obtain an "interactive framework" that supports repeated testing. See `Example2_8`.

In the program, `anotherEg` is a variable whose value is used to determine if another example is to be run. `anotherEg` is assigned a value that is of type `char`. `char` stands for *char*acter and it refers to a character like "y" or "n", or "3" or "+". In short, the value of any keyboard button may be stored in a `char` variable. Each of the special character sequences that we have used in `fprintf` statements, such as "\n" and "\t", is treated as a single special character. Notice from the assignment of `anotherEg` and from the `while` condition that character values are enclosed in single quotes (e.g., `'n'`). Note that the character "2" is not the same as the numeric value 2! If an `input` statement is used to read in a character value, we add a flag `'s'` after the prompt message, separated by a comma, as shown in the last statement in the `while`-loop.

In `Example 2_8`, the `while`-loop continues until the user types in the letter `n`, the "magic character." The while loop could also be controlled by comparing `anotherEg` with `'y'`:

```
while anotherEg = 'y'
```

With this method of checking, the program terminates *unless* the "magic character" (n) is struck. However, this is not such a good thing since keystroke error brings about program termination.

Even with the few examples we see that the quality of our square root "idea" depends strongly on the quality of the initial guess. The natural thing to do after this brief computational experience is to go "back to the drawing boards" and figure a way to improve upon the method. A better initial guess or an increase in the number of refinements might be the recommended course of action.

The interactive framework used above is quite general. To explore some computational idea, you need only "fill in" the following template:

```
% Example 2_8:  Test a method for computing square roots

anotherEg = 'y';   % Continue execution if anotherEg='y'

% Loop until the user quits the program
while  anotherEg  = 'n'
   a= input('\nEnter a positive real number:  ');
   x= input('Enter an initial guess for the square root:  ');
   x= (x+(a/x))/2;
   x= (x+(a/x))/2;
   absoluteError= abs(x-sqrt(a));
   relativeError= absoluteError/sqrt(a);
   fprintf('\nComputed root = %f\n', x);
   fprintf('Absolute error = %e\n', absoluteError);
   fprintf('Relative error = %e\n', relativeError);
   anotherEg = input('\nAnother example (y/n)?  ','s');
end
```

Output:

```
Enter a positive real number:  100
Enter an initial guess for the square root:  5
Computed root = 10.250000
Absolute Error = 2.500000e-01
Relative Error = 2.500000e-02

Another example?  (y/n)?  y

Enter a positive real number:  100
Enter an initial guess for the square root:  9
Computed root = 10.000153
Absolute Error = 1.534684e-04
Relative Error = 1.534684e-05

Another example?  (y/n)?  n
```

```
% Interactive framework
% ⟨Description of the computational idea⟩

anotherEg = 'y';

while anotherEg ~= 'n'

   ⟨A fragment to be tested including fprintf's⟩

   anotherEg= input('Another example?  Enter y (yes) or n (no) ','s');
end
```

Problem 2.18. A sphere with radius 1 has volume equal to $4\pi/3$. How long must the edge of a cube be so that it has the same volume? Use the interactive framework. Do not make a direct calculation of the cube root (i.e., raise a number to the power of 3 or -3).

Problem 2.19. Let a, b, and c be real numbers not all zero. How small can you make the quotient

$$Q = \frac{\sqrt{a^2+b^2+c^2}}{|a|+|b|+|c|}$$

Use the interactive framework. Do not make any direct calculation of the square root or the square.

Problem 2.20. Use the interactive framework to estimate the area of the largest rectangle whose 4 vertices are on the curve defined by $x^4/19 + y^4/17 = 1$? Note: `f= sqrt(sqrt(z))` assigns the value of $z^{1/4}$ to `f`.