CHAPTER 5

Loop Statements and Vectorizing Code

KEY TERMS

looping statements
counted loops
conditional loops
action
vectorized code
iterate
loop or iterator variable
echo-printing
running sum
running product
preallocate
nested loop
outer loop
inner loop
infinite loop
factorial
sentinel
counting
error-checking

CONTENTS

Consider the problem of calculating the area of a circle with a radius of 0.3 cm. A MATLAB® program certainly is not needed to do that; you'd use your calculator instead, and punch in $\pi * 0.3^2$. However, if a table of circle areas is desired, for radii ranging from 0.1 to 100 cm in steps of 0.05 (e.g., 0.1, 0.15, 0.2, etc.), it would be very tedious to use a calculator and write it all down. One of the great uses of programming languages and software packages such as MATLAB is the ability to repeat a process such as this.

This chapter will cover statements in MATLAB that allow other statement(s) to be repeated. The statements that do this are called ***looping statements***, or ***loops***. There are two basic kinds of loops in programming: ***counted loops*** and ***conditional loops***. A counted loop is a loop that repeats statements a specified number of times (so, ahead of time, it is known how many times the statements are to be repeated). In a counted loop, for example, you might say "repeat these statements 10 times". A conditional loop also repeats statements, but ahead of time it is not known *how many* times the statements will need to be repeated. With a conditional loop, for example, you might say "repeat these statements until this condition becomes false". The statement(s) that are repeated in any loop are called the ***action*** of the loop.

There are two different loop statements in MATLAB: the **for** statement and the **while** statement. In practice, the **for** statement is used as the counted loop, and

MATLAB®. https://doi.org/10.1016/B978-0-12-815479-3.00005-2

the **while** is usually used as the conditional loop. To keep it simple, that is how they will be presented here.

In many programming languages, looping through the elements in a vector or matrix is a very fundamental concept. In MATLAB, however, as it is written to work with vectors and matrices, looping through elements is usually not necessary. Instead, "vectorized code" is used, which means replacing the loops through arrays with the use of built-in functions and operators. Both methods will be described in this chapter. The earlier sections will focus on "the programming concepts", using loops. These will be contrasted with "the efficient methods", using ***vectorized code***. Loops are still relevant and necessary in MATLAB in other contexts, just not normally when working with vectors or matrices.

5.1 THE FOR LOOP

The **for** statement, or the **for** loop, is used when it is necessary to repeat statement(s) in a script or function, and when it *is* known ahead of time how many times the statements will be repeated. The statements that are repeated are called the action of the loop. For example, it may be known that the action of the loop will be repeated five times. The terminology used is that we ***iterate*** through the action of the loop five times.

The variable that is used to iterate through values is called a ***loop variable*** or an ***iterator variable***. For example, the variable might iterate through the integers 1 through 5 (e.g., 1, 2, 3, 4, and then 5). Although, in general, variable names should be mnemonic, it is common in many languages for an iterator variable to be given the name *i* (and if more than one iterator variable is needed, *i*, *j*, *k*, *l*, etc.) This is historical and is because of the way integer variables were named in Fortran. However, in MATLAB both **i** and **j** are built-in functions that return the value $\sqrt{-1}$, so using either as a loop variable will override that value. If that is not an issue, it is okay to use *i* as a loop variable.

The general form of the **for** loop is:

```
for loopvar = range
    action
end
```

where *loopvar* is the loop variable, "range" is the range of values through which the loop variable is to iterate, and the action of the loop consists of all statements up to the **end**. Just like with **if** statements, the action is indented to make it easier to see. The range can be specified using any vector, but normally the easiest way to specify the range of values is to use the colon operator.

As an example, we will print a column of numbers from 1 to 5.

THE PROGRAMMING CONCEPT

The loop could be entered in the Command Window, although, like **if** and **switch** statements, loops will make more sense in scripts and functions. In the Command Window, the results would appear after the **for** loop:

```
>> for i = 1:5
       fprintf('%d\n',i)
   end
1
2
3
4
5
```

What the **for** statement accomplished was to print the value of *i* and then the newline character for every value of *i*, from 1 through 5 in steps of 1. The first thing that happens is that *i* is initialized to have the value 1. Then, the action of the loop is executed, which is the **fprintf** statement that prints the value of *i* (1), and then the newline character to move the cursor down. Then, *i* is incremented to have the value of 2. Next, the action of the loop is executed, which prints 2 and the newline. Then, *i* is incremented to 3 and that is printed; then, *i* is incremented to 4 and that is printed; and then, finally, *i* is incremented to 5 and that is printed. The final value of *i* is 5; this value can be used once the loop has finished.

THE EFFICIENT METHOD

Of course, **disp** could also be used to print a column vector, to achieve the same result:

```
>> disp([1:5]')
     1
     2
     3
     4
     5
```

QUICK QUESTION!

How could you print this column of integers (using the programming method):

```
  0
 50
100
150
200
```

Answer: In a loop, you could print these values starting at 0, incrementing by 50, and ending at 200. Each is printed using a field width of 3.

```
>> for i = 0:50:200
       fprintf('%3d\n',i)
   end
```

5.1.1 For Loops That Do Not Use the Iterator Variable in the Action

In the previous example, the value of the loop variable was used in the action of the **for** loop: it was printed. It is not always necessary to actually use the value of the loop variable, however. Sometimes the variable is simply used to iterate, or repeat, an action a specified number of times. For example,

```
for i = 1:3
    fprintf('I will not chew gum\n')
end
```

produces the output:

```
I will not chew gum
I will not chew gum
I will not chew gum
```

The variable *i* is necessary to repeat the action three times, even though the value of *i* is not used in the action of the loop.

QUICK QUESTION!

What would be the result of the following **for** loop?

```
for i = 4:2:8
    fprintf('I will not chew gum\n')
end
```

Answer: Exactly the same output as above! It doesn't matter that the loop variable iterates through the values 4, then 6, then 8 instead of 1, 2, 3. As the loop variable is not used in the action, this is just another way of specifying that the action should be repeated three times. Of course, using 1:3 makes more sense!

PRACTICE 5.1

Write a **for** loop that will print a column of five *'s.

5.1.2 Input in a for Loop

The following script repeats the process of prompting the user for a number and ***echo-printing*** the number (which means simply printing it back out). A **for** loop specifies how many times this is to occur. This is another example in which the loop variable is not used in the action, but, instead, just specifies how many times to repeat the action.

forecho.m

```
% This script loops to repeat the action of
% prompting the user for a number and echo-printing it

for iv = 1:3
    inputnum = input('Enter a number: ');
    fprintf('You entered %.1f\n',inputnum)
end
```

```
>> forecho
Enter a number: 33
You entered 33.0
Enter a number: 1.1
You entered 1.1
Enter a number: 55
You entered 55.0
```

In this example, the loop variable *iv* iterates through the values 1 through 3, so the action is repeated three times. The action consists of prompting the user for a number and echo-printing it with one decimal place.

5.1.3 Finding Sums and Products

A very common application of a **for** loop is to calculate sums and products. For example, instead of just echo-printing the numbers that the user enters, we could calculate the sum of the numbers. In order to do this, we need to add each value to a ***running sum***. A running sum keeps changing, as we keep adding to it. First, the sum has to be initialized to 0.

As an example, we will write a script *sumnnums* that will sum the *n* numbers entered by the user; *n* is a random integer that is generated. In a script to calculate the sum, we need a loop or iterator variable *i*, and also a variable to store the running sum. In this case, we will use a variable *runsum* as the running sum. Every time through the loop, the next value that the user enters is added to the value of *runsum*. This script will print the end result, which is the sum of all of the numbers, stored in the variable *runsum*.

sumnnums.m

```
% sumnnums calculates the sum of the n numbers
% entered by the user

n = randi([3 10]);
runsum = 0;
for i = 1:n
    inputnum = input('Enter a number: ');
    runsum = runsum + inputnum;
end
fprintf('The sum is %.2f\n', runsum)
```

Here is an example in which 3 is generated to be the value of the variable *n*; the script calculates and prints the sum of the numbers the user enters, 4 + 3.2 + 1.1, or 8.3:

```
>> sumnnums
Enter a number: 4
Enter a number: 3.2
Enter a number: 1.1
The sum is 8.30
```

Another very common application of a **for** loop is to find a ***running product***. With a product, the running product must be initialized to 1 (as opposed to a running sum, which is initialized to 0).

PRACTICE 5.2

Write a script *prodnnums* that is similar to the *sumnnums* script, but will calculate and print the product of the numbers entered by the user.

5.1.4 Preallocating Vectors

When numbers are entered by the user, it is often necessary to store them in a vector. There are two basic methods that could be used to accomplish this. One method is to start with an empty vector and extend the vector by adding each number to it as the numbers are entered by the user. Extending a vector, however, is very inefficient. What happens is that every time a vector is extended, a new "chunk" of memory must be found that is large enough for the new vector, and all of the values must be copied from the original location in memory to the new one. This can take a long time to execute.

A better method is to ***preallocate*** the vector to the correct size and then change the value of each element to be the numbers that the user enters. This method involves referring to each index in the result vector and placing each number into the next element in the result vector. This method is far superior if it is known ahead of time how many elements the vector will have. One common method is to use the **zeros** function to preallocate the vector to the correct length.

The following is a script that accomplishes this and prints the resulting vector. The script generates a random integer *n* and repeats the process *n* times. As it is known that the resulting vector will have *n* elements, the vector can be preallocated.

forgenvec.m

```
% forgenvec creates a vector of length n
% It prompts the user and puts n numbers into a vector

n = randi([2 5]);
numvec = zeros(1,n);
for iv = 1:n
    inputnum = input('Enter a number: ');
    numvec(iv) = inputnum;
end
fprintf('The vector is: \n')
disp(numvec)
```

Next is an example of executing this script.

```
>> forgenvec
Enter a number: 44
Enter a number: 2.3
Enter a number: 11
The vector is:
   44.0000    2.3000    11.0000
```

It is very important to notice that the loop variable iv is used as the index into the vector.

QUICK QUESTION!

If you need to just print the sum or average of the numbers that the user enters, would you need to store them in a vector variable?

Answer: No. You could just add each to a running sum as you read them in a loop.

QUICK QUESTION!

What if you wanted to calculate how many of the numbers that the user entered were greater than the average?

Answer: Yes, then you would need to store them in a vector because you would have to go back through them to count how many were greater than the average (or, alternatively, you could go back and ask the user to enter them again!!).

5.1.5 For Loop Example: subplot

A function that is very useful with all types of plots is **subplot**, which creates a matrix of plots in the current Figure Window. Three arguments are passed to it in the form **subplot(r,c,n)**, where *r* and *c* are the dimensions of the matrix and

n is the number of the particular plot within this matrix. The plots are numbered rowwise starting in the upper left corner. In many cases, it is useful to create a **subplot** in a **for** loop so the loop variable can iterate through the integers 1 through *n*.

For example, if it is desired to have three plots next to each other in one Figure Window, the function would be called **subplot(1,3,n).** The matrix dimensions in the Figure Window would be 1×3 in this case, and from left to right the individual plots would be numbered 1, 2, and then 3 (these would be the values of *n*). The first two arguments would always be 1 and 3, as they specify the dimensions of the matrix within the Figure Window.

When the **subplot** function is called in a loop, the first two arguments will always be the same as they give the dimensions of the matrix. The third argument will iterate through the numbers assigned to the elements of the matrix. When the **subplot** function is called, it makes the specified element the "active" plot; then, any plot function can be used, complete with formatting such as axis labeling and titles within that element. Note that the **subplot** function just specifies the dimensions of the matrix in the Figure Window, which is the "active" element; **subplot** itself does not plot anything.

For example, the following **subplot** shows the difference, in one Figure Window, between using 20 points and 40 points to plot **sin(x)** between 0 and 2 * π. The **subplot** function creates a 1×2 row vector of plots in the Figure Window, so that the two plots are shown side by side. The loop variable *i* iterates through the values 1 and then 2.

The first time through the loop, when *i* has the value 1, 20 * 1 or 20 points are used, and the value of the third argument to the **subplot** function is 1. The second time through the loop, 40 points are used and the third argument to **subplot** is 2. The resulting Figure Window with both plots is shown in Figure 5.1.

subplotex.m

```
% Demonstrates subplot using a for loop
for i = 1:2
    x = linspace(0,2*pi,20*i);
    y = sin(x);
    subplot(1,2,i)
    plot(x,y,'ko')
    xlabel('x')
    ylabel('sin(x)')
    title('sin plot')
end
```

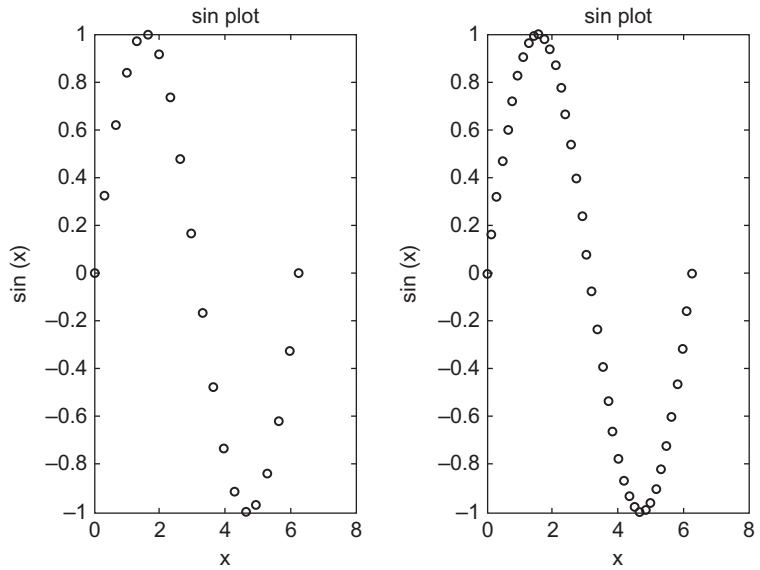

FIGURE 5.1
Subplot to demonstrate a plot using 20 points and 40 points.

Note that once string manipulating functions have been covered in Chapter 7, it will be possible to have customized titles (e.g., showing the number of points).

5.2 NESTED FOR LOOPS

The action of a loop can be any valid statement(s). When the action of a loop is another loop, this is called a ***nested loop***.

The general form of a nested **for** loop is as follows:

```
for loopvarone = rangeone        ← outer loop
    % actionone includes the inner loop
    for loopvartwo = rangetwo    ← inner loop
        actiontwo
    end
end
```

The first **for** loop is called the ***outer loop***; the second **for** loop is called the ***inner loop***. The action of the outer loop consists (in part; there could be other statements) of the entire inner loop.

As an example, a nested **for** loop will be demonstrated in a script that will print a box of stars (*). Variables in the script will specify how many rows and

columns to print. For example, if *rows* has the value 3 and *columns* has the value 5, a 3×5 box would be printed. As lines of output are controlled by printing the newline character, the basic algorithm is as follows.

- For every row of output:
  - Print the required number of stars
  - Move the cursor down to the next line (print '\n')

printstars.m

```
% Prints a box of stars
% How many will be specified by two variables
% for the number of rows and columns

rows = 3;
columns = 5;
% loop over the rows
for i=1:rows
    % for every row loop to print *'s and then one \n
    for j=1:columns
        fprintf('*')
    end
    fprintf('\n')
end
```

Executing the script displays the output:

```
>> printstars
*****
*****
*****
```

The variable *rows* specifies the number of rows to print, and the variable *columns* specifies how many stars to print in each row. There are two loop variables: *i* is the loop variable over the rows and *j* is the loop variable over the columns. As the number of rows is known and the number of columns is known (given by the variables *rows* and *columns*), **for** loops are used. There is one **for** loop to loop over the rows, and another to print the required number of stars for every row.

The values of the loop variables are not used within the loops, but are used simply to iterate the correct number of times. The first **for** loop specifies that the action will be repeated "rows" times. The action of this loop is to print stars and then the newline character. Specifically, the action is to loop to print *columns* stars (e.g., five stars) across on one line. Then, the newline character is printed after all five stars to move the cursor down to the next line.

In this case, the outer loop is over the rows, and the inner loop is over the columns. The outer loop must be over the rows because the script is printing a certain number of rows of output. For each row, a loop is necessary to print the required number of stars; this is the inner **for** loop.

When this script is executed, first the outer loop variable *i* is initialized to 1. Then, the action is executed. The action consists of the inner loop and then printing the newline character. So, while the outer loop variable has the value 1, the inner loop variable *j* iterates through all of its values. As the value of *columns* is 5, the inner loop will print a single star five times. Then, the newline character is printed and then the outer loop variable *i* is incremented to 2. The action of the outer loop is then executed again, meaning the inner loop will print five stars, and then the newline character will be printed. This continues, and, in all, the action of the outer loop will be executed *rows* times.

Notice that the action of the outer loop consists of two statements (the **for** loop and an **fprintf** statement). The action of the inner loop, however, is only a single **fprintf** statement.

The **fprintf** statement to print the newline character must be separate from the other **fprintf** statement that prints the star character. If we simply had

```
fprintf('*\n')
```

as the action of the inner loop (without the separate **fprintf**), this would print a long column of 15 stars, not a 3×5 box.

QUICK QUESTION!

How could this script be modified to print a triangle of stars instead of a box such as the following:

```
*
**
***
```

Answer: In this case, the number of stars to print in each row is the same as the row number (e.g., one star is printed in row 1, two stars in row 2, and so on). The inner **for** loop does not loop to *columns*, but to the value of the row loop variable (so we do not need the variable *columns*):

printtristars.m

```
% Prints a triangle of stars
% How many will be specified by a variable
% for the number of rows
rows = 3;
for i=1:rows
    % inner loop just iterates to the value of i
    for j=1:i
        fprintf('*')
    end
    fprintf('\n')
end
```

In the previous examples, the loop variables were just used to specify the number of times the action is to be repeated. In the next example, the actual values of the loop variables will be printed.

printloopvars.m

```
% Displays the loop variables
for i = 1:3
    for j = 1:2
        fprintf('i=%d, j=%d\n',i,j)
    end
    fprintf('\n')
end
```

Executing this script would print the values of both *i* and *j* on one line every time the action of the inner loop is executed. The action of the outer loop consists of the inner loop and printing a newline character, so there is a separation between the actions of the outer loop:

```
>> printloopvars
i=1, j=1
i=1, j=2

i=2, j=1
i=2, j=2

i=3, j=1
i=3, j=2
```

Now, instead of just printing the loop variables, we can use them to produce a multiplication table, by multiplying the values of the loop variables.

The following function *multtable* calculates and returns a matrix which is a multiplication table. Two arguments are passed to the function, which are the number of rows and columns for this matrix.

multtable.m

```
function outmat = multtable(rows, columns)
% multtable returns a matrix which is a
% multiplication table
% Format: multtable(nRows, nColumns)

% Preallocate the matrix
outmat = zeros(rows,columns);
for i = 1:rows
    for j = 1:columns
        outmat(i,j) = i * j;
    end
end
end
```

In the following example of calling this function, the resulting matrix has three rows and five columns:

```
>> multtable(3,5)
ans =
     1     2     3     4     5
     2     4     6     8    10
     3     6     9    12    15
```

Note that this is a function that returns a matrix. It preallocates the matrix to zeros, and then replaces each element. As the number of rows and columns are known, **for** loops are used. The outer loop loops over the rows, and the inner loop loops over the columns. The action of the nested loop calculates i * j for all values of *i* and *j*. ***Just like with vectors, it is again important to notice that the loop variables are used as the indices into the matrix.***

First, when *i* has the value 1, *j* iterates through the values 1 through 5, so first we are calculating 1*1, then 1*2, then 1*3, then 1*4, and, finally, 1*5. These are the values in the first row (first in element outmat(1,1), then outmat(1,2), then outmat(1,3), then outmat(1,4), and finally outmat(1,5)). Then, when *i* has the value 2, the elements in the second row of the output matrix are calculated, as *j* again iterates through the values from 1 through 5. Finally, when *i* has the value 3, the values in the third row are calculated (3*1, 3*2, 3*3, 3*4, and 3*5).

This function could be used in a script that prompts the user for the number of rows and columns, calls this function to return a multiplication table, and writes the resulting matrix to a file:

createmulttab.m

```
% Prompt the user for rows and columns and
% create a multiplication table to store in
% a file "mymulttable.dat"

num_rows = input('Enter the number of rows: ');
num_cols = input('Enter the number of columns: ');
multmatrix = multtable(num_rows, num_cols);
save mymulttable.dat multmatrix -ascii
```

The following is an example of running this script, and then loading from the file into a matrix in order to verify that the file was created:

```
>> createmulttab
Enter the number of rows: 6
Enter the number of columns: 4

>> load mymulttable.dat
```

```
>> mymulttable
mymulttable =
     1     2     3     4
     2     4     6     8
     3     6     9    12
     4     8    12    16
     5    10    15    20
     6    12    18    24
```

PRACTICE 5.3

For each of the following (they are separate), determine what would be printed. Then, check your answers by trying them in MATLAB.

```
mat = [7 11 3; 3:5];
[r, c] = size(mat);
for i = 1:r
    fprintf('The sum is %d\n', sum(mat(i,:)))
end
----------------------------------------
for i = 1:2
    fprintf('%d: ', i)
    for j = 1:4
        fprintf('%d ', j)
    end
    fprintf('\n')
end
```

5.2.1 Combining Nested for Loops and if Statements

The statements inside of a nested loop can be any valid statements, including any selection statement. For example, there could be an **if** or **if-else** statement as the action, or part of the action, in a loop.

As an example, assume there is a file called "datavals.dat" containing results recorded from an experiment. However, some were recorded erroneously. The numbers are all supposed to be positive. The following script reads from this file into a matrix. It prints the sum from each row of only the positive numbers. We will assume that the file contains integers, but will not assume how many lines are in the file nor how many numbers per line (although we will assume that there are the same number of integers on every line).

sumonlypos.m

```
% Sums only positive numbers from file
% Reads from the file into a matrix and then
%   calculates and prints the sum of only the
%   positive numbers from each row

load datavals.dat
[r c] = size(datavals);

for row = 1:r
    runsum = 0;
    for col = 1:c
        if datavals(row,col) >= 0
            runsum = runsum + datavals(row,col);
        end
    end
    fprintf('The sum for row %d is %d\n',row,runsum)
end
```

For example, *if* the file contains:

```
33  -11    2
 4    5    9
22    5   -7
 2   11    3
```

the output from the program would look like this:

```
>> sumonlypos
The sum for row 1 is 35
The sum for row 2 is 18
The sum for row 3 is 27
The sum for row 4 is 16
```

The file is loaded and the data are stored in a matrix variable. The script finds the dimensions of the matrix and then loops through all of the elements in the matrix by using a nested loop; the outer loop iterates through the rows and the inner loop iterates through the columns. This is important; as an action is desired for every row, the outer loop has to be over the rows. For each element, an **if** statement determines whether the element is positive or not. It only adds the positive values to the row sum. As the sum is found for each row, the *runsum* variable is initialized to 0 for every row, meaning inside of the outer loop.

QUICK QUESTION!

Would it matter if the order of the loops was reversed in this example, so that the outer loop iterates over the columns and the inner loop over the rows?

Answer: Yes, as we want a sum for every row the outer loop must be over the rows.

QUICK QUESTION!

What would you have to change in order to calculate and print the sum of only the positive numbers from each column instead of each row?

Answer: You would reverse the two loops, and change the sentence to say "The sum of column...". That is all that would change. The elements in the matrix would still be referenced as datavals(row,col). The row index is always given first, then the column index—regardless of the order of the loops.

PRACTICE 5.4

Write a function *mymatmin* that finds the minimum value in each column of a matrix argument and returns a vector of the column minimums. Use the programming method. An example of calling the function follows:

```
>> mat = randi(20,3,4)
mat =
    15    19    17     5
     6    14    13    13
     9     5     3    13

>> mymatmin(mat)
ans =
     6     5     3     5
```

QUICK QUESTION!

Would the function *mymatmin* in Practice 5.4 also work for a vector argument?

Answer: Yes, it should, as a vector is just a subset of a matrix. In this case, one of the loop actions would be executed only one time (for the rows if it is a row vector or for the columns if it is a column vector).

5.3 WHILE LOOPS

The **while** statement is used as the conditional loop in MATLAB; it is used to repeat an action when ahead of time it is *not known how many* times the action will be repeated. The general form of the **while** statement is:

```
while condition
    action
end
```

The action, which consists of any number of statement(s), is executed as long as the condition is **true**.

The way it works is that first the condition is evaluated. If it is logically **true**, the action is executed. So, to begin with, the **while** statement is just like an **if** statement. However, at that point the condition is evaluated again. If it is still **true**, the action is executed again. Then, the condition is evaluated again. If it is still **true**, the action is executed again. Then, the condition is....eventually, this has to stop! Eventually, something in the action has to change something in the condition, so it becomes **false**. The condition must eventually become **false** to avoid an ***infinite loop***. (If this happens, Ctrl-C will exit the loop.)

As an example of a conditional loop, we will write a function that will find the first ***factorial*** that is greater than the input argument *high*. For an integer n, the factorial of n, written as n!, is defined as n! = 1 * 2 * 3 * 4 * … * n. To calculate a factorial, a **for** loop would be used. However, in this case we do not know the value of n, so we have to keep calculating the next factorial until a level is reached, which means using a **while** loop.

The basic algorithm is to have two variables: one that iterates through the values 1,2, 3, and so on, and one that stores the factorial of the iterator at each step. We start with 1 and 1 factorial, which is 1. Then, we check the factorial. If it is not greater than *high*, the iterator variable will then increment to 2 and find its factorial (2). If this is not greater than *high*, the iterator will then increment to 3 and the function will find its factorial (6). This continues until we get to the first factorial that is greater than *high*.

So, the process of incrementing a variable and finding its factorial is repeated until we get to the first value greater than *high*. This is implemented using a **while** loop:

factgthigh.m

```
function facgt = factgthigh(high)
% factgthigh returns the first factorial > input
% Format: factgthigh(inputInteger)

i=0;
fac=1;
while fac <= high
    i=i+1;
    fac = fac * i;
end
facgt = fac;
end
```

An example of calling the function, passing 5000 for the value of the input argument *high*, follows:

```
>> factgthigh(5000)
ans =
        5040
```

The iterator variable *i* is initialized to 0, and the running product variable *fac*, which will store the factorial of each value of *i*, is initialized to 1. The first time the **while** loop is executed, the condition is **true**: 1 is less than or equal to 5000. So, the action of the loop is executed, which is to increment *i* to 1 and *fac* becomes 1 (1 * 1).

After the execution of the action of the loop, the condition is evaluated again. As it will still be **true**, the action is executed: *i* is incremented to 2 and *fac* will get the value 2 (1*2). The value 2 is still <= 5000, so the action will be executed again: *i* will be incremented to 3 and *fac* will get the value 6 (2 * 3). This continues, until the first value of *fac* is found that is greater than 5000. As soon as *fac* gets to this value, the condition will be **false** and the **while** loop will end. At that point, the factorial is assigned to the output argument, which returns the value.

The reason that *i* is initialized to 0 rather than 1 is that the first time the loop action is executed, *i* becomes 1 and *fac* becomes 1, so we have 1 and 1!, which is 1.

5.3.1 Multiple Conditions in a while Loop

In the *factgthigh* function, the condition in the **while** loop consisted of one expression, which tested whether or not the variable *fac* was less than or equal to the variable *high*. In many cases, however, the condition will be more complicated than that and could use either the **or** operator || or the **and** operator &&. For example, it may be that it is desired to stay in a **while** loop as long as a variable *x* is in a particular range:

```
while x >= 0 && x <= 100
```

As another example, continuing the action of a loop may be desired as long as at least one of two variables is in a specified range:

```
while x < 50 || y < 100
```

5.3.2 Reading From a File Using a while Loop

The following example illustrates reading from a data file using a **while** loop. Data from an experiment has been recorded in a file called "experd.dat".

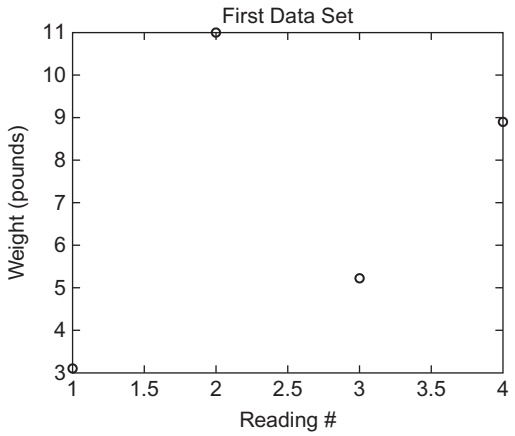

FIGURE 5.2
Plot of some (but not all) data from a file.

The file has some weights followed by a −99 and then more weights, all on the same line. The only data values that we are interested in, however, are those before −99. The −99 is an example of a ***sentinel***, which is a marker in between data sets.

The algorithm for the script is as follows:

- Read the data from the file into a vector.
- Create a new vector variable *newvec* that only has the data values up to but not including the −99.
- Plot the new vector values, using black circles.

For example, *if* the file contains the following:

```
3.1   11   5.2   8.9   -99   4.4   62
```

the plot produced would look like Figure 5.2.

For simplicity, we will assume that the file format is as specified. Using **load** will create a vector with the name *experd*, which contains the values from the file.

THE PROGRAMMING CONCEPT

Using the programming method, we would loop through the vector until the −99 is found, creating the new vector by storing each element from *experd* in the vector *newvec*.

Continued

THE PROGRAMMING CONCEPT—CONT'D

findvalwhile.m

```
% Reads data from a file, but only plots the numbers
% up to a flag of -99. Uses a while loop.

load experd.dat

i = 1;
while experd(i) ~= -99
     newvec(i) = experd(i);
     i = i + 1;
end

plot(newvec,'ko')
xlabel('Reading #')
ylabel('Weight(pounds)')
title('First Data Set')
```

Note that this extends the vector *newvec* every time the action of the loop is executed.

THE EFFICIENT METHOD

Using the **find** function, we can locate the index of the element that stores the −99. Then, the new vector comprises all of the original vector from the first element to the index *before* the index of the element that stores the −99.

findval.m

```
% Reads data from a file, but only plots the numbers
% up to a flag of -99.Uses find and the colon operator

load experd.dat

where = find(experd == -99);
newvec = experd(1:where-1);

plot(newvec,'ko')
xlabel('Reading #')
ylabel('Weight(pounds)')
title('First Data Set')
```

5.3.3 Input in a while Loop

Sometimes a **while** loop is used to process input from the user as long as the user is entering data in a correct format. The following script repeats the process of prompting the user, reading in a positive number, and echo-printing it, as

long as the user correctly enters positive numbers when prompted. As soon as the user types in a negative number, the script will print "OK" and end.

whileposnum.m

```
% Prompts the user and echo prints the numbers entered
% until the user enters a negative number

inputnum=input('Enter a positive number: ');
while inputnum >= 0
    fprintf('You entered a %d.\n\n',inputnum)
    inputnum = input('Enter a positive number: ');
end
fprintf('OK!\n')
```

When the script is executed, the input/output might look like this:

```
>> whileposnum
Enter a positive number: 6
You entered a 6.

Enter a positive number: -2
OK!
```

Note that the prompt is repeated in the script: once before the loop, and then again at the end of the action. This is done so that every time the condition is evaluated, there is a new value of *inputnum* to check. If the user enters a negative number the first time, no values would be echo-printed:

```
>> whileposnum
Enter a positive number: -33
OK!
```

This example illustrates a very important feature of ***while*** *loops: it is possible that the action will not be executed at all, if the value of the condition is false the first time it is evaluated.*

As we have seen previously, MATLAB will give an error message if a character is entered rather than a number.

```
>> whileposnum
Enter a positive number: a
Error using input
Unrecognized function or variable 'a'.

Error in whileposnum (line 4)
inputnum=input('Enter a positive number: ');

Enter a positive number: -4
OK!
```

However, if the character is actually the name of a variable, it will use the value of that variable as the input. For example:

```
>> a=5;
>> whileposnum
Enter a positive number: a
You entered a 5.

Enter a positive number: -4
OK!
```

5.3.3.1 Extending a Vector

If it is desired to store all of the positive numbers that the user enters, we would store them one at a time in a vector. However, as we do not know ahead of time how many elements we will need, we cannot preallocate to the correct size. The two methods of extending a vector one element at a time are shown here. We can start with an empty vector and concatenate each value to the vector, or we can increment an index.

```
numvec = [];
inputnum=input('Enter a positive number: ');
while inputnum >= 0
    numvec = [numvec inputnum];
    inputnum = input('Enter a positive number: ');
end
% OR:

i = 0;
inputnum=input('Enter a positive number: ');
while inputnum >= 0
    i = i + 1;
    numvec(i) = inputnum;
    inputnum = input('Enter a positive number: ');
end
```

Keep in mind that both of these are inefficient and should be avoided if the vector can be preallocated.

5.3.4 Counting in a while Loop

Although **while** loops are used when the number of times the action will be repeated is not known ahead of time, it is often useful to know how many times the action was, in fact, repeated. In that case, it is necessary to ***count*** the number of times the action is executed. The following variation on the previous script counts the number of positive numbers that the user successfully enters.

countposnum.m

```
% Prompts the user for positive numbers and echo prints as
% long as the user enters positive numbers

% Counts the positive numbers entered by the user
counter=0;
inputnum=input('Enter a positive number: ');
while inputnum >= 0
    fprintf('You entered a %d.\n\n',inputnum)
    counter = counter + 1;
    inputnum = input('Enter a positive number: ');
end
fprintf('Thanks, you entered %d positive numbers.\n',counter)
```

The script initializes a variable *counter* to 0. Then, in the **while** loop action, every time the user successfully enters a number, the script increments the counter variable. At the end of the script, it prints the number of positive numbers that were entered.

```
>> countposnum
Enter a positive number: 4
You entered a 4.

Enter a positive number: 11
You entered a 11.

Enter a positive number: -4
Thanks, you entered 2 positive numbers.
```

PRACTICE 5.5

Write a script *avenegnum* that will repeat the process of prompting the user for negative numbers, until the user enters a zero or positive number, as just shown. Instead of echo-printing them, however, the script will print the average (of just the negative numbers). If no negative numbers are entered, the script will print an error message instead of the average. Use the programming method. Examples of executing this script follow:

```
>> avenegnum
Enter a negative number: 5
No negative numbers to average.

>> avenegnum
Enter a positive number: -8
Enter a positive number: -3
Enter a positive number: -4
Enter a positive number: 6
The average was -5.00
```

5.3.5 Error-Checking User Input in a while Loop

In most applications, when the user is prompted to enter something, there is a valid range of values. If the user enters an incorrect value, rather than having the program carry on with an incorrect value, or just printing an error message, the program should repeat the prompt. The program should keep prompting the user, reading the value, and checking it until the user enters a value that is in the correct range. This is a very common application of a conditional loop: looping until the user correctly enters a value in a program. This is called ***error-checking***.

For example, the following script prompts the user to enter a positive number, and loops to print an error message and repeat the prompt until the user finally enters a positive number.

readonenum.m

```
% Loop until the user enters a positive number

inputnum=input('Enter a positive number: ');
while inputnum < 0
    inputnum = input('Invalid! Enter a positive number: ');
end
fprintf('Thanks, you entered a %.1f \n',inputnum)
```

An example of running this script follows:

```
>> readonenum
Enter a positive number: -5
Invalid! Enter a positive number: -2.2
Invalid! Enter a positive number: c
Error using input
Unrecognized function or variable 'c'.
Error in readonenum (line 5)
inputnum = input('Invalid! Enter a positive number: ');
Invalid! Enter a positive number: 44
Thanks, you entered a 44.0
```

Note that MATLAB itself catches the character input and prints an error message, and repeats the prompt when the *c* was entered.

QUICK QUESTION!

How could we vary the previous example so that the script asks the user to enter positive numbers *n* times, where *n* is an integer defined to be 3?

Answer: Every time the user enters a value, the script checks and in a **while** loop keeps telling the user that it's invalid until a valid positive number is entered. By putting the error-check in a **for** loop that repeats n times, the user is forced eventually to enter three positive numbers, as shown in the following.

QUICK QUESTION!—CONT'D

readnnums.m

```
% Loop until the user enters n positive numbers
n=3;
fprintf('Please enter %d positive numbers\n\n',n)
for i=1:n
    inputnum=input('Enter a positive number: ');
    while inputnum < 0
        inputnum = input('Invalid! Enter a positive number: ');
    end
    fprintf('Thanks, you entered a %.1f \n',inputnum)
end
```

```
>> readnnums
Please enter 3 positive numbers

Enter a positive number: 5.2
Thanks, you entered a 5.2
Enter a positive number: 6
Thanks, you entered a 6.0
Enter a positive number: -7.7
Invalid! Enter a positive number: 5
Thanks, you entered a 5.0
```

5.3.5.1 Error-Checking for Integers

As MATLAB uses the type **double** by default for all values, to check to make sure that the user has entered an integer, the program has to convert the input value to an integer type (e.g., **int32**) and then check to see whether that is equal to the original input. The following examples illustrate the concept.

If the value of the variable *num* is a real number, converting it to the type **int32** will round it, so the result is not the same as the original value.

```
>> num = 3.3;
>> inum = int32(num)
inum =
           3
>> num == inum
ans =
      0
```

If, however, the value of the variable *num* is an integer, converting it to an integer type will not change the value.

```
>> num = 4;
>> inum = int32(num)
inum =
           4
>> num == inum
ans =
     1
```

The following script uses this idea to error-check for integer data; it loops until the user correctly enters an integer.

readoneint.m

```
% Error-check until the user enters an integer
inputnum = input('Enter an integer: ');
num2 = int32(inputnum);
while num2 ~= inputnum
    inputnum = input('Invalid! Enter an integer: ');
    num2 = int32(inputnum);
end
fprintf('Thanks, you entered a %d \n',inputnum)
```

Examples of running this script are:

```
>> readoneint
Enter an integer: 9.5
Invalid! Enter an integer: 3.6
Invalid! Enter an integer: -11
Thanks, you entered a -11

>> readoneint
Enter an integer: 5
Thanks, you entered a 5
```

Note: this assumes that the user enters something. Use the **isempty** function to be sure.

Putting these ideas together, the following script loops until the user correctly enters a positive integer. There are two parts to the condition, as the value must be positive and must be an integer.

readoneposint.m

```
% Error checks until the user enters a positive integer
inputnum = input('Enter a positive integer: ');
num2 = int32(inputnum);
while num2 ~= inputnum || num2 < 0
    inputnum = input('Invalid! Enter a positive integer: ');
    num2 = int32(inputnum);
end
fprintf('Thanks, you entered a %d \n',inputnum)
```

```
>> readoneposint
Enter a positive integer: 5.5
Invalid! Enter a positive integer: -4
Invalid! Enter a positive integer: 11
Thanks, you entered a 11
```

PRACTICE 5.6

Modify the script *readoneposint* to read *n* positive integers, instead of just one.

5.4 LOOPS WITH VECTORS AND MATRICES; VECTORIZING

In most programming languages when performing an operation on a vector, a **for** loop is used to loop through the entire vector, using the loop variable as the index into the vector. In general, in MATLAB, assuming there is a vector variable *vec*, the indices range from 1 to the length of the vector, and the **for** statement loops through all of the elements performing the same operation on each one:

```
for i = 1:length(vec)
    % do something with vec(i)
end
```

In fact, this is one reason to store values in a vector. Typically, values in a vector represent "the same thing", so, typically, in a program the same operation would be performed on every element.

Similarly, for an operation on a matrix, a nested loop would be required, and the loop variables over the rows and columns are used as the subscripts into the matrix. In general, assuming a matrix variable *mat*, we use **size** to return separately the number of rows and columns, and use these variables in the **for** loops. If an action is desired for every row in the matrix, the nested for loop would look like this:

```
[r, c] = size(mat);
for row = 1:r
   for col = 1:c
      % do something with mat(row,col)
   end
end
```

If, instead, an action is desired for every column in the matrix, the outer loop would be over the columns. (Note, however, that the reference to a matrix element always refers to the row index first and then the column index.)

```
[r, c] = size(mat);
for col = 1:c
   for row = 1:r
      % do something with mat(row,col)
   end
end
```

Typically, this is not necessary in MATLAB! Although **for** loops are very useful for many other applications in MATLAB, they are not typically used for operations on vectors or matrices; instead, the efficient method is to use built-in functions and/or operators. This is called vectorized code. The use of loops and selection statements with vectors and matrices is a basic programming concept with many other languages, and so both "the programming concept" and "the efficient method" are highlighted in this section and, to some extent, throughout the rest of this book.

5.4.1 Vectorizing Sums and Products

For example, let's say that we want to perform a scalar multiplication; in this case, multiply every element of a vector v by 3, and store the result back in v, where v is initialized as follows:

```
>> v= [3   7   2   1];
```

THE PROGRAMMING CONCEPT

To accomplish this, we can loop through all of the elements in the vector and multiply each element by 3. In the following, the output is suppressed in the loop, and then the resulting vector is shown:

```
>> for i = 1:length(v)
      v(i) = v(i) * 3;
   end
>> v
v =
   9    21    6    3
```

THE EFFICIENT METHOD

```
>> v = v * 3
```

How could we calculate the factorial of n, n! = 1 * 2 * 3 * 4 *… * n?

THE PROGRAMMING CONCEPT

The basic algorithm is to initialize a running product to 1 and multiply the running product by every integer from 1 to n. This is implemented in a function:

myfact.m

```
function runprod = myfact(n)
% myfact returns n!
% Format of call: myfact(n)

runprod = 1;
for i = 1:n
    runprod = runprod * i;
end
end
```

Any positive integer argument could be passed to this function, and it will calculate the factorial of that number. For example, if 5 is passed, the function will calculate and return 1*2*3*4*5, or 120:

```
>> myfact(5)
ans =
   120
```

THE EFFICIENT METHOD

MATLAB has a built-in function, **factorial**, that will find the factorial of an integer *n*. The **prod** function could also be used to find the product of the vector 1:5.

```
>> factorial(5)
ans =
   120
>> prod(1:5)
ans =
   120
```

QUICK QUESTION!

MATLAB has a **cumsum** function that will return a vector of all of the running sums of an input vector. However, many other languages do not, so how could we write our own?

Answer: Essentially, there are two programming methods that could be used to simulate the **cumsum** function. One method is to start with an empty vector and extend the vector by adding each running sum to it as the running sums are calculated. A better method is to preallocate the vector to the correct size and then change the value of each element to be successive running sums.

myveccumsum.m

```
function outvec = myveccumsum(vec)
% myveccumsum imitates cumsum for a vector
% It preallocates the output vector
% Format: myveccumsum(vector)

outvec = zeros(size(vec));
runsum = 0;
for i = 1:length(vec)
    runsum = runsum + vec(i);
    outvec(i) = runsum;
end
end
```

An example of calling the function follows:

```
>> myveccumsum([5 9 4])
ans =
     5   14   18
```

PRACTICE 5.7

Write a function that imitates the **cumprod** function. Use the method of preallocating the output vector.

QUICK QUESTION!

How would we sum each individual column of a matrix?

Answer: The programming method would require a nested loop in which the outer loop is over the columns. The function will sum each column and return a row vector containing the results.

matcolsum.m

```
function outsum = matcolsum(mat)
% matcolsum finds the sum of every column in a matrix
% Returns a vector of the column sums
% Format: matcolsum(matrix)

[row, col] = size(mat);

% Preallocate the vector to the number of columns
outsum = zeros(1,col);

% Every column is being summed so the outer loop
% has to be over the columns
for i = 1:col
    % Initialize the running sum to 0 for every column
    runsum = 0;
    for j = 1:row
        runsum = runsum + mat(j,i);
    end
    outsum(i) = runsum;
end
end
```

Note that the output argument will be a row vector containing the same number of columns as the input argument matrix. Also, as the function is calculating a sum for each column, the *runsum* variable must be initialized to 0 for every column, so it is initialized inside of the outer loop.

```
>> mat = [3:5; 2 5 7]
mat =
     3     4     5
     2     5     7
>> matcolsum(mat)
ans =
     5     9     12
```

Of course, the built-in **sum** function in MATLAB would accomplish the same thing, as we have already seen.

PRACTICE 5.8

Modify the function *matcolsum*. Create a function *matrowsum* to calculate and return a vector of all of the row sums instead of column sums. For example, calling it and passing the *mat* variable above would result in the following:

```
>> matrowsum(mat)
ans =
     12     14
```

5.4.2 Vectorizing Loops with Selection Statements

In many applications, it is useful to determine whether numbers in a matrix are positive, zero, or negative.

THE PROGRAMMING CONCEPT

A function *signum* follows that will accomplish this:

signum.m

```
function outmat = signum(mat)
% signum imitates the sign function
% Format: signum(matrix)

[r, c] = size(mat);
for i = 1:r
    for j = 1:c
        if mat(i,j) > 0
            outmat(i,j) = 1;
        elseif mat(i,j) == 0
            outmat(i,j) = 0;
        else
            outmat(i,j) = -1;
        end
    end
end
end
```

Here is an example of using this function:

```
>> mat = [0  4  -3;  -1  0  2]
mat =
     0     4    -3
    -1     0     2
>> signum(mat)
ans =
     0     1    -1
    -1     0     1
```

THE EFFICIENT METHOD

Close inspection reveals that the function accomplishes the same task as the built-in **sign** function!

```
>> sign(mat)
ans =
     0     1    -1
    -1     0     1
```

Another example of a common application on a vector is to find the minimum and/or maximum value in the vector.

THE PROGRAMMING CONCEPT

For instance, the algorithm to find the minimum value in a vector is as follows:

- The working minimum (the minimum that has been found so far) is the first element in the vector to begin with.
- Loop through the rest of the vector (from the second element to the end).
  - If any element is less than the working minimum, then that element is the new working minimum.

The following function implements this algorithm and returns the minimum value found in the vector.

myminvec.m

```
function outmin = myminvec(vec)
% myminvec returns the minimum value in a vector
% Format: myminvec(vector)

outmin = vec(1);
for i = 2:length(vec)
    if vec(i) < outmin
       outmin = vec(i);
    end
end
end
```

```
>> vec = [3 8 99 -1];
>> myminvec(vec)
ans =
    -1
>> vec = [3 8 99 11];
>> myminvec(vec)
ans =
    3
```

Note that an **if** statement is used in the loop rather than an **if-else** statement. If the value of the next element in the vector is less than *outmin*, then the value of *outmin* is changed; otherwise, no action is necessary.

THE EFFICIENT METHOD

Use the **min** function:

```
>> vec = [5  9  4];
>> min(vec)
ans =
     4
```

QUICK QUESTION!

Determine what the following function accomplishes:

xxx.m

```
function logresult = xxx(vec)
% QQ for you - what does this do?

logresult = false;
i = 1;
while i <= length(vec) && logresult == false
    if vec(i) ~= 0
       logresult = true;
    end
    i = i + 1;
end
end
```

Answer: The output produced by this function is the same as the **any** function for a vector. It initializes the output argument to **false**. It then loops through the vector and, if any element is nonzero, changes the output argument to **true**. It loops until either a nonzero value is found or it has gone through all elements.

QUICK QUESTION!

Determine what the following function accomplishes.

yyy.m

```
function logresult = yyy(mat)
% QQ for you - what does this do?

count = 0;
[r, c] = size(mat);
for i = 1:r
    for j = 1:c
        if mat(i,j) ~= 0
           count = count + 1;
        end
    end
end

logresult = count == numel(mat);
end
```

Answer: The output produced by this function is the same as the **all** function.

As another example, we will write a function that will receive a vector and an integer as input arguments and will return a logical vector that stores **logical true** only for elements of the vector that are greater than the integer and **false** for the other elements.

THE PROGRAMMING CONCEPT

The function receives two input arguments: the vector, and an integer *n* with which to compare. It loops through every element in the input vector and stores in the result vector either **true** or **false** depending on whether vec(i) > n is **true** or **false**.

testvecgtn.m

```
function outvec = testvecgtn(vec,n)
% testvecgtn tests whether elements in vector
% are greater than n or not
% Format: testvecgtn(vector, n)

% Preallocate the vector to logical false
outvec = false(size(vec));
for i = 1:length(vec)
    % If an element is > n, change to true
    if vec(i) > n
       outvec(i) = true;
    end
end
end
```

```
>> ov = testvecgtn([44 2 11 -3 5 8], 6)
ov =
     1     0     1     0     0     1
>> class(ov)
ans =
logical
```

Note that as the vector was preallocated to **false**, the **else** clause is not necessary.

THE EFFICIENT METHOD

As we have seen, the relational operator > will automatically create a **logical** vector.

testvecgtnii.m

```
function outvec = testvecgtnii(vec,n)
% testvecgtnii tests whether elements in vector
     % are greater than n or not with no loop
% Format: testvecgtnii(vector, n)

outvec = vec > n;
end
```

PRACTICE 5.9

Call the function *testvecgtnii*, passing a vector and a value for n. Use MATLAB code to count how many values in the vector were greater than n.

5.4.3 Tips for Writing Efficient Code

To be able to write efficient code in MATLAB, including vectorizing, there are several important features to keep in mind:

- Scalar and array operations
- Logical vectors
- Built-in functions
- Preallocation of vectors

There are many functions in MATLAB that can be utilized instead of code that uses loops and selection statements. These functions have been demonstrated already, but it is worth repeating them to emphasize their utility:

- **sum** and **prod**: find the sum or product of every element in a vector or column in a matrix
- **cumsum** and **cumprod**: return a vector or matrix of the cumulative (running) sums or products
- **min** and **max**: find the minimum value in a vector or in every column of a matrix
- **any**, **all**, **find**: work with logical expressions
- "is" functions, such as **isletter** and **isequal**: return **logical** values

In almost all cases, code that is faster to write by the programmer is also faster for MATLAB to execute. So, "efficient code" means that it is both efficient for the programmer and for MATLAB.

PRACTICE 5.10

Vectorize the following (rewrite the code efficiently):

```
i = 0;
for inc = 0: 0.5: 3
    i = i + 1;
    myvec(i) = sqrt(inc);
end
-------------------------------------
[r c] = size(mat);
newmat = zeros(r,c);
for i = 1:r
    for j = 1:c
        newmat(i,j) = sign(mat(i,j));
    end
end
```

MATLAB has a built-in function **checkcode** that can detect potential problems within scripts and functions. Consider, for example, the following script that extends a vector within a loop:

badcode.m

```
for j = 1:4
    vec(j) = j
end
```

The function **checkcode** will flag this, as well as the good programming practice of suppressing output within scripts:

```
>> checkcode('badcode')
L 2 (C 5-7): The variable 'vec' appears to change size on every loop iteration
(within a script). Consider preallocating for speed.
L 2 (C 12): Terminate statement with semicolon to suppress output (within a
script).
```

The same information is shown in Code Analyzer Reports, which can be produced within MATLAB for one file (script or function) or for all code files within a folder. Clicking on the down arrow for the Current Folder and then choosing Reports and then Code Analyzer Report will check the code for all files within the Current Folder. When viewing a file within the Editor, click on the down arrow and then Show Code Analyzer Report for a report on just that one file.

5.5 TIMING

MATLAB has built-in functions that determine how long it takes code to execute. One set of related functions is **tic**/**toc**. These functions are placed around code and will print the time it took for the code to execute. Essentially, the function **tic** turns a timer on, and then **toc** evaluates the timer and prints the result. Here is a script that illustrates these functions.

fortictoc.m

```
tic
mysum = 0;
for i = 1:20000000
    mysum = mysum + i;
end
toc
```

```
>> fortictoc
Elapsed time is 0.087294 seconds.
```

Note that when using timing functions such as **tic**/**toc**, be aware that other processes running in the background (e.g., any web browser) will affect the speed of your code.

Here is an example of a script that demonstrates how much preallocating a vector speeds up the code.

tictocprealloc.m

```
% This shows the timing difference between
% preallocating a vector vs. not

clear
disp('No preallocation')
tic
for i = 1:10000
    x(i) = sqrt(i);
end
toc

disp('Preallocation')
tic
y = zeros(1,10000);
for i = 1:10000
    y(i) = sqrt(i);
end
toc
```

```
>> tictocprealloc
No preallocation
Elapsed time is 0.005070 seconds.
Preallocation
Elapsed time is 0.000273 seconds.
```

QUICK QUESTION!

Preallocation can speed up code, but to preallocate, it is necessary to know the desired size. What if you do not know the eventual size of a vector (or matrix)? Does that mean that you have to extend it rather than preallocating?

Answer: If you know the maximum size that it could possibly be, you can preallocate to a size that is larger than necessary and then delete the "unused" elements. To do that, you would have to count the number of elements that are actually used. For example, if you have a vector *vec* that has been preallocated and a variable *count* that stores the number of elements that were actually used, this will trim the unnecessary elements:

```
vec = vec(1:count)
```

MATLAB also has a Profiler that will generate detailed reports on execution time of codes. In newer versions of MATLAB, from the Editor click on Run and Time; this will bring up a report in the Profile Viewer. Choose the function name to see a very detailed report, including a Code Analyzer Report. From the Command Window, this can be accessed using **profile on** and **profile off**, and **profile viewer**.

```
>> profile on
>> tictocprealloc
No preallocation
Elapsed time is 0.047721 seconds.
Preallocation
Elapsed time is 0.040621 seconds.
>> profile viewer
>> profile off
```

Explore Other Interesting Features

Explore what happens when you use a matrix rather than a vector to specify the range in a **for** loop. For example,

```
for i = mat
  disp(i)
end
```

Take a guess before you investigate!

Try the **pause** function in loops.

Investigate the **vectorize** function.

The **tic** and **toc** functions are in the **timefun** help topic. Type **help timefun** to investigate some of the other timing functions. ■

SUMMARY

COMMON PITFALLS

- Forgetting to initialize a running sum or count variable to 0
- Forgetting to initialize a running product variable to 1
- In cases where loops are necessary, not realizing that if an action is required for every row in a matrix, the outer loop must be over the rows (and if an action is required for every column, the outer loop must be over the columns)
- Not realizing that it is possible that the action of a **while** loop will never be executed

- Not error-checking input into a program
- Not vectorizing code whenever possible. If it is not necessary to use loops in MATLAB, don't!
- Forgetting that **subplot** numbers the plots rowwise rather than columnwise.
- Not realizing that the **subplot** function just creates a matrix within the Figure Window. Each part of this matrix must then be filled with a plot, using any type of plot function.

PROGRAMMING STYLE GUIDELINES

- Use loops for repetition only when necessary
  - **for** statements as counted loops
  - **while** statements as conditional loops
- Do not use *i* or *j* for iterator variable names if the use of the built-in constants **i** and **j** is desired.
- Indent the action of loops.
- If the loop variable is just being used to specify how many times the action of the loop is to be executed, use the colon operator 1:n where *n* is the number of times the action is to be executed.
- Preallocate vectors and matrices whenever possible (when the size is known ahead of time).
- When data are read in a loop, only store them in an array if it will be necessary to access the individual data values again.
- Vectorize whenever possible.

MATLAB Reserved Words
for
while
end

MATLAB Functions and Commands	
subplot	profile
factorial	
checkcode	
tic / toc	

Exercises

1. Write a **for** loop that will print the column of real numbers from 1.5 to 2.7 in steps of 0.2.
2. In the Command Window, write a **for** loop that will iterate through the integers from 32 to 255. For each, show the corresponding character from the character encoding. Play with this! Try printing characters beyond the standard ASCII, in small groups. For example, print the characters that correspond to integers from 300 to 340.
3. Prompt the user for an integer *n* and print "MATLAB rocks!" n times.
4. When would it matter if a **for** loop contained `for i = 1:3` vs. `for i = [3 5 6]`, and when would it not matter?
5. Create a matrix variable *mat*. Fill in the rest of the **fprintf** statement below so that it will print the product of all of the numbers in every row of the matrix, in the format:

   ```
   The product of row 1 is 44
   ```

   Note that the value of *col* is not needed.

   ```
   [row col] = size(mat);
   for i = 1:row
      fprintf('The product of row %d is %d\n',
   end
   ```

6. Write a function *sumsteps2* that calculates and returns the sum of 1 to *n* in steps of 2, where *n* is an argument passed to the function. For example, if 11 is passed, it will return 1 + 3 + 5 + 7 + 9 + 11. Do this using a **for** loop. Calling the function will look like this:

   ```
   >> sumsteps2(11)
   ans =
       36
   ```

7. Write a function *prodby2* that will receive a value of a positive integer *n* and will calculate and return the product of the odd integers from 1 to *n* (or from 1 to *n* – 1 if *n* is even). Use a **for** loop.
8. Write a script that will:
   - generate a random integer in the inclusive range from 2 to 5
   - loop that many times to
     - prompt the user for a number
     - print the sum of the numbers entered so far with one decimal place
9. Write a script that will load data from a file into a matrix. Create the data file first and make sure that there is the same number of values on every line in the file so that it can be loaded into a matrix. Using a **for** loop, it will then create a subplot for every row in the matrix and will plot the numbers from each row element in the Figure Window.

10. Write code that will prompt the user for 5 numbers and store them in a vector. Make sure that you preallocate the vector!
11. Execute this script and be amazed by the results! You can try more points to get a clearer picture, but it may take a while to run.

```
clear
clf
x = rand;
y = rand;
plot(x,y)

hold on
for it = 1:10000
    choic = round(rand*2);
    if choic == 0
       x = x/2;
       y = y/2;
    elseif choic == 1
       x = (x+1)/2;
       y = y/2;
    else
       x = (x+0.5)/2;
       y = (y+1)/2;
    end
    plot(x,y)

    hold on
end
```

12. Come up with "trigger" words in a problem statement that would tell you when it's appropriate to use **for** loops and/or nested **for** loops.
13. With a matrix, when would:
    - your outer loop be over the rows
    - your outer loop be over the columns
    - it not matter which is the outer and which is the inner loop?
14. Write a function *myones* that will receive two input arguments *n* and *m* and will return an $n \times m$ matrix of all ones. Do NOT use any built-in functions (so, yes, the code will be inefficient).
15. Write a script that will print the following multiplication table:

```
1
2 4
3 6 9
4 8 12 16
5 10 15 20 25
```

16. Write a function that will receive a matrix as an input argument and will calculate and return the overall average of all numbers in the matrix. Use loops, not built-in functions, to calculate the average.
17. Trace this to determine the output, and then enter it to verify your answer.

```
for i = 2:2:6
   fprintf('%d: ', i)
   for j = i:-1:1
      fprintf('*')
   end
   fprintf('\n')
end
```

18. Sales (in millions) from two different divisions of a company for the four quarters of 2017 are stored in two vector variables, e.g.,:

```
div1 = [4.3 4.4 3.9 5.2];
div2 = [2.4 2.6 2.8 2.9];
```

Using **subplot**, show side by side the sales figures for the two divisions, using **bar** charts.

19. Write an algorithm for an ATM program. Think about where there would be selection statements, menus, loops (counted vs. conditional), etc., but don't write MATLAB code, just an algorithm (pseudo-code).
20. Trace this to figure out what the result will be, and then type it into MATLAB to verify the results.

```
count = 0;
number = 8;
while number > 3
      fprintf('number is %d\n', number)
      number = number - 2;
      count = count + 1;
end
fprintf('count is %d\n', count)
```

21. It is good programming style to always use a **for** loop when the number of times to repeat the action is known. The following code uses a **while** loop that mimics what a **for** loop should do. Rewrite it using a **for** loop that accomplishes exactly the same thing.

```
myprod = 1;
i = 1;
while i <= 4
   num = input('Enter a number: ');
   myprod = myprod * num;
   i = i + 1;
end
```

22. Write a script that will generate random integers in the range from 0 to 50, and print them, until one is finally generated that is greater than 25. The script should print how many attempts it took.
23. The inverse of the mathematical constant e can be approximated as follows:

$$\frac{1}{e} \approx \left(1 - \frac{1}{n}\right)^n$$

Write a script that will loop through values of n until the difference between the approximation and the actual value is less than 0.0001. The script should then print out the built-in value of e^{-1} and the approximation to 4 decimal places and also print the value of n required for such accuracy.

24. Write a script that will prompt the user for a keyword in MATLAB, error-checking until a keyword is entered.
25. A blizzard is a massive snowstorm. Definitions vary, but for our purposes, we will assume that a blizzard is characterized by both winds of 30 mph or higher and blowing snow that leads to visibility of 0.5 miles or less, sustained for at least 4 hours. Data from a storm one day has been stored in a file *stormtrack.dat*. There are 24 lines in the file, one for each hour of the day. Each line in the file has the wind speed and visibility at a location. Create a sample data file. Read this data from the file and determine whether blizzard conditions were met during this day or not.
26. Given the following loop:

```
while x < 10
     action
end
```

- For what values of the variable x would the action of the loop be skipped entirely?
- If the variable x is initialized to have the value of 5 before the loop, what would the action have to include in order for this to not be an infinite loop?

27. Write a script called *prtemps* that will prompt the user for a maximum Celsius value in the range from −16 to 20; error-check to make sure it's in that range. Then, print a table showing degrees Fahrenheit and degrees Celsius until this maximum is reached. The first value that exceeds the maximum should not be printed. The table should start at 0 degrees Fahrenheit and increase by 5 degrees Fahrenheit until the max (in Celsius) is reached. Both temperatures should be printed with a field width of 6 and one decimal place. The formula is C = 5/9 (F-32).
28. Unix is a command line-driven operating system. The Unix command line behaves in a similar manner to the MATLAB interpreter: users repeatedly enter commands at a prompt (denoted by the character '>'). You are to write a "command line" that repeatedly prompts the user for a command and repeats it back to them. The user can exit the command line when only the character 'q' is passed (meaning "quit"). For example:

```
>> command_line
> ls ../folder
You entered: ls ../folder
> echo hello world!
You entered: echo hello world!
> q
Goodbye
```

29. Assume that a vector variable vec has been initialized. Vectorize the following; write ONE assignment statement that will accomplish exactly the same thing.

```
for i = 1:length(vec)
   vec(i) = vec(i) * 3;
end
vec % just for display
```

30. Assume that a vector variable vec has been initialized. Vectorize the following; write ONE assignment statement that will accomplish exactly the same thing.

```
for i = 1:length(vec)
   if vec(i) < 0
      res(i) = -1
   elseif vec(i) == 0
      res(i) = 0;
   else
      res(i) = 1;
   end
end
res % just for display
```

31. The following code was written by somebody who does not know how to use MATLAB efficiently. Rewrite this as a single statement that will accomplish exactly the same thing for a matrix variable *mat* (e.g., vectorize this code):

```
[r c] = size(mat);
for i = 1:r
   for j = 1:c
       mat(i,j) = mat(i,j) * 2;
   end
end
```

32. Vectorize the following code. Write one assignment statement that would accomplish the same thing. Assume that *mat* is a matrix variable that has been initialized.

```
myvar = 0;
[r,c] = size(mat);
for i = 1:r
```

```
        for j = 1:c
            myvar = myvar + mat(i,j);
            end
        end
    end
    myvar % just for display
```

33. Vectorize the following code. Write statement(s) that accomplish the same thing, eliminating the loop. Assume that there is a vector v that has a negative number in it, e.g.,:

```
    >> v = [4 11 22 5 33 -8 3 99 52];
newv = [];
i = 1;
while v(i) >= 0
    newv(i) = v(i);
    i = i + 1;
end
newv % Note: just to display
```

34. Assume that a vector variable vec has been initialized. Unvectorize the following: write loops and/or if statements to accomplish exactly the same thing.

```
newvec = diff(vec)
```

35. Give some examples of when you would need to use a counted loop in MATLAB, and when you would not.
36. For each of the following, decide whether you would use a **for** loop, a **while** loop, a nested loop (and if so what kind, e.g., a **for** loop inside of another **for** loop, a **while** loop inside of a **for** loop, etc.), or no loop at all.
    - sum the integers 1 through 50
    - add 3 to all numbers in a vector
    - prompt the user for a string, and keep doing this until the string that the user enters is a keyword in MATLAB
    - find the minimum in every column of a matrix
    - prompt the user for 5 numbers and find their sum
    - prompt the user for 10 numbers, find the average, and also find how many of the numbers were greater than the average
    - generate a random integer n in the range from 10 to 20. Prompt the user for n positive numbers, error-checking to make sure you get n positive numbers (and just echo print each one)
    - prompt the user for positive numbers until the user enters a negative number. Calculate and print the average of the positive numbers, or an error message if none are entered

37. Write a script that will prompt the user for a quiz grade and error-check until the user enters a valid quiz grade. The script will then echo print the grade. For this case, valid grades are in the range from 0 to 10 in steps of 0.5. Do this by creating a vector of valid grades and then use **any** or **all** in the condition in the **while** loop.
38. Which is faster: using **false** or using **logical**(0) to preallocate a matrix to all **logical** zeros? Write a script to test this.
39. Which is faster: using a **switch** statement or using a nested **if-else**? Write a script to test this.
40. Write a script *beautyofmath* that produces the following output. The script should iterate from 1 to 9 to produce the expressions on the left, perform the specified operation to get the results shown on the right, and print exactly in the format shown here.

```
>> beautyofmath
1 x 8 + 1 = 9
12 x 8 + 2 = 98
123 x 8 + 3 = 987
1234 x 8 + 4 = 9876
12345 x 8 + 5 = 98765
123456 x 8 + 6 = 987654
1234567 x 8 + 7 = 9876543
12345678 x 8 + 8 = 98765432
123456789 x 8 + 9 = 987654321
```

41. The Wind Chill Factor (WCF) measures how cold it feels with a given air temperature T (in degrees Fahrenheit) and wind speed V (in miles per hour). One formula for WCF is

$$WCF = 35.7 + 0.6\,T - 35.7\left(V^{0.16}\right) + 0.43\,T\left(V^{0.16}\right)$$

Write a function to receive the temperature and wind speed as input arguments and return the WCF. Using loops, print a table showing wind chill factors for temperatures ranging from −20 to 55 in steps of 5, and wind speeds ranging from 0 to 55 in steps of 5. Call the function to calculate each wind chill factor.
42. Instead of printing the WCFs in the previous problem, create a matrix of WCFs and write them to a file. Use the programming method, using nested loops.
43. Write a script to add two 30-digit numbers and print the result. This is not as easy as it might sound at first, because integer types may not be able to store a value this large. One way to handle large integers is to store them in vectors, where each element in the vector stores a digit of the integer. Your script should initialize two 30-digit integers, storing each in a vector, and then add these integers, also storing the result in a vector. Create the original numbers using the **randi** function. Hint: add 2 numbers on paper first, and pay attention to what you do!

44. Write a "Guess My Number Game" program. The program generates a random integer in a specified range, and the user (the player) has to guess the number. The program allows the use to play as many times as he/she would like; at the conclusion of each game, the program asks whether the player wants to play again.

    The basic algorithm is:

    1. The program starts by printing instructions on the screen.
    2. For every game:
       - the program generates a new random integer in the range from MIN to MAX. Treat MIN and MAX like constants; start by initializing them to 1 and 100
       - loop to prompt the player for a guess until the player correctly guesses the integer
         - for each guess, the program prints whether the player's guess was too low, too high, or correct
       - at the conclusion (when the integer has been guessed):
         - print the total number of guesses for that game
         - print a message regarding how well the player did in that game (e.g., the player took way too long to guess the number, the player was awesome, etc.). To do this, you will have to decide on ranges for your messages and give a rationale for your decision in a comment in the program.
    3. After all games have been played, print a summary showing the average number of guesses.

45. Write your own code to perform matrix multiplication. Recall that to multiply two matrices, the inner dimensions must be the same.

    $$[A]_{mxn}[B]_{nxp} = [C]_{mxp}$$

    Every element in the resulting matrix C is obtained by:

    $$c_{ij} = \sum_{k=1}^{n} a_{ik} b_{kj}.$$

    So, three nested loops are required.