CHAPTER 2

Procedural Programming

This chapter will show you how to create programs. The title, Procedural Programming, refers to the most common programming paradigm, the one that MATLAB supports also. It means that the program is organized into procedures (also called functions) that solve parts of the problem. Functions can be invoked from the Command Window or each other to create complex programs.

Functions

Objectives

Functions let us break up complex problems into smaller, more manageable parts.

(1) We will learn how functions let you create reusable software components that can be applied in many different programs.

(2) We will learn how the environment inside a function is separated from the outside via a well defined interface through which the outside world communicates with it.

(3) We will learn how to define a function to allow input to it when it initiates its execution and output from it when it is done.

(4) We will learn how scripts serve as a collection of commands that execute as if they were typed into the Command Window.

Wouldn't it be great if you could build a set of useful programs that you could use and reuse whenever you needed them? This is exactly what functions let you do. With functions, you can create your own toolbox!

So far, we have worked primarily in the MATLAB Command Window. We have performed simple computations, created matrices, applied various operators and used built-in functions, such as `size` or `plot`. You may have asked yourself a couple of questions. What if I want to issue more than a single command at a time? How can I create my own functions? This section answers both of these questions.

MATLAB, just like other programming languages, comes with hundreds of built-in functions. Still it cannot possibly contain all functions that would be useful to carry out any given programming task. It is, therefore, a very important characteristic of programming languages to support the creation of user-defined functions. In fact, a procedural computer program is nothing more than a collection of functions that call each other.

Without further ado, let's create our first MATLAB function. The built-in MATLAB function **rand**, generates a set of pseudorandom numbers that are greater than 0 and smaller than 1.0. When we call **rand**, we can provide parameters that specify how many numbers **rand** will generate. For example, **rand(n)** will create an **n**-by-**n** matrix of numbers, while **rand(n,m)** generates an **n**-by-**m** matrix of numbers. For more information, type **help rand** in the Command Window. Let's say that we often want to create a 3-by-4 matrix of random numbers that are between 1 and 10. The code would look something like this:

```
>> a = 9 * rand(3,4) + 1

a =
    6.1586    3.6185    4.6575    5.2354
    5.5691    5.5103    6.3379    7.0671
    2.3099    2.2768    8.2844    7.8146
```

We call **rand** with **3** and **4** specifying that we need a **3**-by-**4** matrix. The function returns numbers between 0 and 1, so multiplying by 9 and adding 1 creates exactly what we want: numbers between 1 and 10.

While not terribly complicated, it is still error-prone to type this line in the Command Window every time we need it. Let's create a function instead, so that we need only to remember the function's name when we need the functionality. Type **edit myRand** in the Command Window. This opens a text editor with which we can enter our code:

```
function myRand
a = 9* rand(3,4) + 1
```

According to the syntax rules, a function always starts with the word **function** followed by the name of the function. The word "function" is a keyword. As we learned in the section Matrices and Operations of Chapter 1, a keyword is a word that is defined by the language to have special meaning. In most cases, a keyword is also a **reserved word**, meaning that it cannot be used as a variable name or a function name. A list of all reserved keywords can be gotten by typing the command **iskeyword** in the Command Window. By default, MATLAB highlights all keywords in blue. We will introduce other keywords in later sections.

The same rules apply to selection of function names that apply to the selection of variable names. It is very important to come up with meaningful names, so that just from the name you have a pretty good idea of what the function does. We chose **myRand**, but **rand1to10** or **rand3by4_1to10** might have been more descriptive if a bit cumbersome. Following the first line of the function, we can simply enter our code line-by-line. MATLAB's syntax rules allow an optional "end" statement at the end of the function, as for example:

```
function myRand
a = 9* rand(3,4) + 1
end
```

which, as far as MATLAB is concerned is exactly the same function as our first version. As can be seen by its color, `end` is a keyword. (When it is used as an index, it is not blue.) This option is convenient for programmers who frequently program in languages in which each function must be terminated by `end`, who will tend to type it by habit, and to whom it just looks wrong to leave it off. MATLAB allows an optional semicolon after `end`. MATLAB provides additional latitude in spacing also. Blank lines can be put anywhere, and leading spaces or tabs on individual lines are ignored. Here is a third, completely equivalent version of the function with all three options:

```
function myRand

    a = 9* rand(3,4) + 1

end
```

The **end** is required to terminate a function only in special situations (the nesting of functions inside other functions, which is not covered in this book). In the end, **end** is rarely used by MATLAB programmers to end functions. And

that ends that! Indentation of all commands in a function and blank lines before and after them are also uncommon, but they are sometimes seen, so this book shows examples both with these options and without them.

We execute the function by simply typing its name in the Command Window:

```
>> myRand

a =
    2.1782    5.6458    1.6019    9.0056
    4.0917    8.1700    3.2363    9.9083
    6.9675    6.1894    9.1781    3.9129

>> myRand

a =
    9.8868    6.4121    4.3951    2.3712
    6.5248    8.1801    2.5718    4.4508
    6.8687    6.4940    1.5392    1.1178
```

The function seems to be working well. It creates a 3-by-4 array of random numbers between 1 and 10. For more complex functions, much more testing would be needed, but for this one, we can be satisfied. Or can we?

Let's try it again:

```
>> clear
>> myRand

a =
    9.6884    1.2606    4.2567    1.5158
    3.8637    4.5746    8.0241    1.4319
    9.6004    3.5003    5.0110    5.7773

>> a
Undefined function or variable 'a'.
```

What is going on? We cleared all variables from the memory with the **clear** command (a description of **clear** was given previously), but after that it was redefined while **myRand** was running. So why is **a** undefined?

The reason is that a function is like Las Vegas. What happens in a function stays in the function. The code inside the function is separated from the outside world. It has its own workspace and cannot access the workspace of the Command Window (recall the MATLAB Workspace Window that shows all the variables that are defined and accessible). Furthermore, the variables inside the function are not visible from the outside, so they do not appear in the Command Window workspace.

This is a very important and useful concept. The code you write in a function lives in its own sandbox. It cannot mess up the workspace of the Command Window and, if it was called from another function, it cannot do anything to that function's workspace either. For example, if you have a variable **x** defined in the Command Window and you have a function that uses a variable **x** also, those are two different variables. So, calling the function will not change the value of the **x** in the workspace of the Command Window. Imagine for a second that it could change **x** or any other Command-Window variable for that matter. After a function call, you would have no idea which variables in the Command Window were changed by it. It would be a disaster.

So, why was variable **a** undefined in the workspace? Because **a** was defined only in the function **myRand** and consequently, it had no effect on the Command-Window workspace. But then how did we see the value of **a** in the Command Window when we called **myRand**? We tricked you! There is no semicolon at the end of the line inside the function, so we did not suppress printing. The function simply printed out the value of **a** inside the function! Although **a** is hidden inside **myRand**, printout is allowed.

Variables inside functions are called "local variables". A **local variable** is accessible only by statements inside the function, and they exist only during the function call. Once the function finishes its execution (or *returns*, using computer science terminology), all local variables cease to exist.

Function Output

The question then is: how can we get anything out of a function and into a variable in the Command Window? Functions would not be very useful if they were not able to propagate their results back to the caller. The answer is that functions can provide results through output arguments. An **output argument** is a local variable that you designate to hold a value that is passed to the caller by the function. Let's see how it's done:

```
function a = myRand
a = 9* rand(3,4) + 1
```

Do you see the difference? We changed the first line slightly. After the **function** keyword we inserted a variable name (**a** in this case) and an equal sign before the function name. What this means is that **a** is the output of the **myRand** function and when the function returns, its value will be the value of **a**. Let's try it:

```
>> myRand

a =
    1.7733    5.3673    7.9170    4.5759
    4.7141    6.5068    7.6740    3.3631
    9.8676    1.1010    2.9561    1.7077

ans =
    1.7733    5.3673    7.9170    4.5759
    4.7141    6.5068    7.6740    3.3631
    9.8676    1.1010    2.9561    1.7077

>> a
Undefined function or variable 'a'.
```

Notice how we still print the value of **a** inside the function due to the lack of a semicolon, but now we also have a value returned by the function that MATLAB stores in the **ans** variable by default. Of course, **a** is still undefined in the workspace of the Command Window, but now we know why.

Let's add the semicolon inside the function:

```
function a = myRand
a = 9* rand(3,4) + 1;
```

and try again:

```
>> b = myRand

b =
    2.9811    8.5511    9.5742    7.2589
    5.3649    4.7184    4.0159    9.3050
    5.6377    4.6196    2.2038    5.3877

>> b

b =
    2.9811    8.5511    9.5742    7.2589
    5.3649    4.7184    4.0159    9.3050
    5.6377    4.6196    2.2038    5.3877

>> a
Undefined function or variable 'a'.
```

Notice that we do not have **a** printed anymore due to the semicolon. Also, we assigned the output of the function to the variable **b** which is created in the Command-Window workspace. The **a** in the Command Window is still undefined. If we want a variable **a** to contain the output of the function, we have to do this:

```
>> a = myRand

a =
    6.1722    6.5330    7.3249    9.8977
    1.1449    7.0172    5.2363    5.3673
    9.4135    4.7960    4.6800    2.0554

>> a

a =
    6.1722    6.5330    7.3249    9.8977
    1.1449    7.0172    5.2363    5.3673
    9.4135    4.7960    4.6800    2.0554
```

A very important point is, however, that this variable **a** is a brand new variable that has been just created. After its creation, it was assigned the return value of the function **myRand**. The local variable **a** inside the **myRand** function supplied the return value of the function, but it is a completely different variable, and it does not exist anymore. It disappeared when the function returned.

Function Input

Our little **myRand** function works fine, but what if you wanted a 3-by-4 array of random numbers between 2 and 22 instead of 1 and 10? How about 10 and 100? Or −10 and 10? Writing a new function for every possible combination is not really a good solution. Wouldn't it be great, if we could tell our **myRand** function the lower and upper limit somehow and let it supply the correct answer?

Fortunately, it is quite easy to do just that. We can supply parameters to the function when we call it. These parameters are called "input arguments". An **input argument** is a local variable that you designate to receive a value that is input into the function. Let's modify **myRand**, so that it works with user supplied lower and upper limits:

```
function a = myRand(low, high)
a = (high-low) * rand(3,4) + low;
```

The syntax to specify input arguments is to put them in a comma-separated list in parenthesis after the name of the function. The input arguments, **low** and **high** in this case, become local variables inside **myRand**. The only difference between them and regular local variables is that input arguments are assigned the values that the caller of the function specifies in the actual function call.

For example, when we call **myRand** like this

```
>> myRand(10,100)
ans =
   29.3030   43.5362   33.7959   82.2776
   54.5405   42.1009   77.0064   37.3963
   12.9701   34.7236   21.8515   40.8562
```

the variable **low** will be assigned 10, while variable **high** will get the value 100. Inside **myRand**, we call the **rand** function, multiply the resulting array by 90, add 10, and assign the result to the local variable **a** which happens to be the output argument too. The value of **a** is returned to the caller as the result of the function call and all local variables—**a**, **low**, **high**—disappear.

The caller can also use variables as input arguments. Consider this:

```
>> x = 0.5
x =
    0.5000

>> y = 0.6
y =
    0.6000

>> z = myRand(x,y)
z =
    0.5541    0.5043    0.5777    0.5834
    0.5099    0.5142    0.5358    0.5363
    0.5857    0.5217    0.5025    0.5789

>> x
x =
    0.5000

>> low
Undefined function or variable 'low'.
```

We defined variables **x** and **y** and passed them as input arguments to function **myRand**. What happened was that the value of variable **x** was assigned to the input argument (and local variable) of **myRand** called **low**. Similarly, the value of **y** was assigned to **high**. The function **myRand** computed the result in output argument (and local variable) **a**, and the value of **a** was re-

turned as the result of the function and assigned to variable **z** in the Command-Window workspace. Inside the function, the variable **x**, **y** and **z** are not visible or accessible. Again, once the function returned, all its local variables ceased to exist.

Multiple Outputs

We have seen how easy it is to provide multiple input arguments to a function. But can we return multiple results? Fortunately, MATLAB, unlike many other programming languages (e.g., C, C++, Java), allows you to do that. Let's modify our **myRand** function so that it also returns the sum of the elements of the generated random array.

There is a built-in MATLAB function called **sum**. It computes a sum, as its name implies, but there is one important point to remember about **sum** (and many other frequently used built-in MATLAB functions that work on arrays such as **min**, **max**, **mean**, etc.). As expected, **sum** returns the sum of the elements of a vector:

```
>> sum(1:100)
ans =
        5050

>> sum((1:100)')
ans =
        5050
```

and it does not matter whether the input argument is a column or row vector. However, if we pass a matrix to **sum**, it returns a vector: the sum of the elements in each column of the matrix. Check this out:

```
>> z = [1:5;5:-1:1]
z =
     1     2     3     4     5
     5     4     3     2     1

>> sum(z)
ans =
     6     6     6     6     6
```

To figure out how **sum** works with higher dimensional arrays, try it or check out MATLAB's help (type, **help sum**). In order to get the sum of all the elements of an array, say **A**, then we can sum all the elements this way: **sum(A(:))**, since **A(:)** is a vector of all the elements of **A**. But it only works with vectors. To get the sum of all the elements of two-dimensional array returned by a function, say **rand**, we can call **sum** twice, like this:

```
>> sum(sum(rand(1e3,1e3)))
ans =
   4.9973e+05
```

Here, we first created a 1000-by-1000 matrix of random numbers. The first call of **sum** produces as output a 1000-element vector that contains that sums of each column of the matrix. Finally, the second call to **sum** adds up the elements of that vector, giving us the sum of the elements of the original matrix, that is, the sum of one million random numbers.

It is interesting to see that the sum is close to 500,000. The reason is that **rand** generates random numbers of uniform distribution. What this means is that any number between 0 and 1 has the same probability of being returned by **rand**. Hence, if we call **rand** a large number of times, the average of the numbers returned should be close to 0.5. Hence, their sum should be close to the half of the number of random numbers we added up. Run this code a few times. You will see that the sum will be pretty close to **5.000e+05** every time.

Now we are ready to modify **myRand** to provide not just the random matrix but also the sum of its elements:

```
function [a, s] = myRand(low, high)
a = (high-low) * rand(3,4) + low;
s = sum(a(:));
```

As you can see, the syntax rules call for a comma-separated list of output arguments enclosed inside square brackets. Inside the function, the new output argument, **s**, is just another local variable, except that its value will be returned by the function. Or will it?

```
>> myRand(1,2)
ans =
    1.7298    1.2376    1.7623    1.6364
    1.1760    1.2047    1.3377    1.1386
    1.1159    1.0868    1.8096    1.2190
```

Seemingly nothing different happened. The reason is that we did not supply a variable to catch the second output, the sum. In fact, we did not supply anything to catch even the random matrix. However, MATLAB always assigns the *first* returned output to the **ans** variable when the output is not assigned to a variable explicitly by the user. Hence, we got to see the matrix, but not the sum. Try this instead:

```
>> x = myRand(-1,1)
x =
   -0.7877   -0.2177    0.3730    0.7976
    0.2124    0.3709    0.4674    0.1676
    0.0071   -0.2833   -0.9711    0.5832
```

It happened again. Still we don't get the sum! The reason is that we supplied only one variable to receive the output of the function. If we want to get both results, we have to do it the right way:

```
>> [x y] = myRand(0,1)
x =
    0.7677    0.6132    0.5389    0.8967
    0.5453    0.6289    0.1370    0.4115
    0.1004    0.9295    0.6304    0.4258
y =
    6.6252
```

The required syntax is to put the variables that we want the results assigned to in squared brackets separated by spaces and/or commas. Now we get both results, the matrix and the sum of its elements.

You may be wondering about what happens to output arguments that are not assigned to variables when the function returns. These values are simply discarded. Remember that input and output arguments, that is, the variables inside the function, vanish when the function returns in any case. Only the values of the output arguments are returned—not the variables themselves. Those returned values are either assigned to variables at the place where the function is being called, or they are not. The choice is up to the caller of the function. If they are not assigned, these values are lost forever.

What happens if we define an output argument, but forget to assign a value to it inside the function? Let's try it:

```
function [a, s, w] = myRand(low, high)
a = (high-low) * rand(3,4) + low;
s = sum(a(:));
```

Here we declared three output arguments, but never did anything with **w**. If we use the function without trying to get the third output, everything works fine:

```
>> [x y] = myRand(0,1)
x =
    0.4731    0.0118    0.0690    0.3181
    0.7879    0.9779    0.7480    0.4995
    0.3759    0.0077    0.7134    0.8289
y =
    5.8113
```

No problem. (The reason the result is different from before is that each time we use **rand**, we get a different set of numbers.) But if we tried this:

```
>> [x y z] = myRand(0,1)
Error in myRand (line 3)
    a = (high-low) * rand(3,4) + lower;
Output argument "w" (and maybe others) not assigned dur-
ing call to
"/Textbook/code/functions/myRand.m>myRand".
```

You have to be careful when using multiple outputs. Make sure that you indeed provide a value for each of them. (You may be wondering about the error message above. Why line 3? When an error occurs because an output argu-ment has not been assigned a value, MATLAB picks the closest executable line of the function to the output argument.)

What if a function returns multiple output arguments, but you are interested in only the last one? For example, let's say that you need only the sum of the random numbers generated by the function **myRand** and not the actual array. If you call the function with one argument, MATLAB will return the first one in the output list, that is, the array (**a**) and not the sum (**s**). It seems that you need to call the function with two arguments (as we did above with **x** and **y**), even though you do not need **x** at all. In this case, it is not a big deal, but if another function returns a huge array that you do not need, it takes more time and wastes memory to return the value and assign it to a variable. A few years ago (in release 2009b), MATLAB introduced new syntax to handle this situation. When calling a function with multiple output arguments, you can simply provide a **~** symbol (tilde) in place of each argument that you do not need. For example, you can call **myRand** this way:

```
>> [~, y] = myRand(10,20)
y =
  173.8006
```

Note, however, that you must supply the comma in this case, otherwise MATLAB will report a syntax error. Actual output arguments and the **~** symbol can be used in any combination. The following is another example of a function call:

```
>> [a, ~, ~, c, ~, d] = myFunc(x,y);
```

The last argument should be a real variable and not a **~** symbol, because the **~** can be simply omitted in that case.

Finally, remember that it is an error to call a function with fewer or more than the required number of input arguments. For example:

```
>> myRand(2)
Error using myRand (line 3)
Not enough input arguments.

>> myRand(2,3,4)
Error using myRand
Too many input arguments.
```

Later we will learn how to write functions that handle missing input and/or output arguments.

Formal Definition

Now that we have played with functions a little bit, we can treat the subject somewhat more formally. A function declaration looks like this:

```
function [out_arg1, out_arg2, ...] = function_name (in_arg1, in_arg2, ...)
```

where **function** is a keyword. The form of **function_name** must satisfy the same restrictions that are placed on variable names. Those restrictions are given in subsection Variable names of Introduction to MATLAB in Chapter 2. Note that, like variable names, function names are case sensitive! If there is only one output argument, the square brackets are optional. If there is no output argument, the output argument list including the square brackets and the assignment operator (**=**) can all be omitted. Similarly, if there are no input arguments, the parentheses can be omitted.

Each of the following is a valid function declaration:

```
function func
function func(in1)
function func(in1, in2)
function out1 = func
function out1 = func(in1)
function [out1, out2] = func
function [out1, out2] = func(in1, in2)
```

There are, of course, many more valid combinations.

Except for lines consisting entirely of comments, the function declaration must appear on the first line of the file. Surprisingly perhaps, the function name (i.e., **`func`** in the examples above) need not be the same as the name of the M-file that contains it. This is an interesting feature because, when a call is made to the function from the Command Window, MATLAB depends entirely on the name of the M-file that contains the function to determine which function to call. It *pays no attention to the function name in the function definition!* And that holds for calls from other functions in other M-files. You can put any name you want there without affecting these functions calls. You could in fact use **`func`**, for every function declaration. It makes no difference to MATLAB. It is dangerous, however, to use a name in the declaration that is different form name of the M-file that contains it because it puts you in danger of debugging the wrong function and puts others in danger of calling the wrong function.

If you name them differently, MATLAB, in an attempt to set you straight, will highlight the function name in the edit window in a dark yellow (a rather ugly yellow, probably not by accident). If you allow your mouse pointer to hover over that name, MATLAB produces a "tip" window, reminding you that MATLAB knows the function by its file name, AND it gives you a chance to solve this problem easily by simply clicking on "Fix" in that window. If you click it, then the name in the declaration will automatically change to the name of the M-file (without the **`.m`** extension), and all will be well. It's always a good idea click "Fix" or just to make those names the same in the first place. We recommend it strongly. In fact it would be a good idea to take the pledge right now never to use different names for the declaration and the M-file and never to text while driving. We'll all be safer.

Function names

Henceforth, since MATLAB depends only on the name of the M-file (without the **`.m`** extension) to identify a function, when we say "function name", we will always mean the filename (we took the pledge). The naming rules allow you a great deal of flexibility in naming your functions' M-files, but not as much flexibility as your operating system does. For example, the operating system will allow you to use a name that starts with a number. If you do that for an M-file, you will not be able to call that function in MATLAB. The operating system will allow you to use characters other than letters, numbers, and underscores, and it may allow you to include spaces in the name as well. Do any of these things, and you will have a useless M-file, because MATLAB will not call it.

In addition to the hard-and-fast rules above, there are a few rules of thumb that you should follow when choosing the name for a function. As we mentioned before, you should come up with meaningful names. The name of the function should give you a good idea about what the function does. Stay away from names that are typically used for variables. As you progress through this book, you'll see many examples of frequently used variable names. For example, **`x`** would be a bad choice for a function name, as would **`A`**, which is commonly used to mean "array". In fact, a single-letter is rarely a good choice for a function name. Finally, you should never use names of built-in MATLAB functions. While is it not illegal, it can create significant confusion.

For example, if you defined a function called **`sum`** that does something different from the built-in MATLAB function we have just seen, here is what can happen. If your file, **`sum.m`** is in the current folder when you call **`sum`**, it is your version that will be called. While that is probably what you wanted, if somebody else is looking at your code, they may not realize that you have

your own function called **sum** and would assume that your program uses the MATLAB **sum** function. This would be confusing indeed!

On the other hand, if your **sum.M-file** is not in the current folder and you try to call it, in all likelihood, MATLAB will find its own version and not yours. Your code will not work correctly, and you will have a hard time figuring out why. If you name your function **my_sum** instead, MATLAB will not find it if you are in the wrong folder and will give you a meaningful error message. You will figure it out quickly what is wrong. The prefix "**my_**" is a good choice, because no built-in MATLAB function name begins with "**my_**".

Since MATLAB has hundreds of built-in functions and since we may write large programs with many functions and many variables, how can we figure out whether a name is already in use? Fortunately, the MathWorks engineers thought of this problem. There is a built-in function called **exist** that tells you whether a name is available. If it returns 0, the name is not in use.

```
>> exist my_sum
ans =
     0

>> exist sum
ans =
     5
```

A non-zero values means that it exists. The number 5 means "built-in function". Other values have other meanings. To find out more type **help exist** in the Command Window.

Another naming error is to define a variable using the name of a MATLAB function. Probably the most common such error and one that has caused many students to yell at MATLAB is the use of sum as a variable name. Suppose you want to know the sum of the first five positive integers. You might do this:

```
>> sum = 1 + 2 + 3 + 4 + 5
sum =
    15
```

The answer is correct. So far everything is rosy. But suppose that later in your programming session you decide to use the built-in function **sum**, perhaps even to find the same summation that you did before:

```
>> sum_to_5 = sum([1 2 3 4 5])
Index exceeds matrix dimensions.
```

The problem here is that when MATLAB sees **sum**, the first place it looks for it is its list of variables (i.e., its workspace). It finds the variable sum there, and it sees that it is a scalar. We have asked for the first 5 elements of a scalar, but a scalar has only one element. That's an error, and things have gone from rosy to code red. And what is infuriating is that MATLAB's error message seems to make no sense. It's complaining about the dimensions of a matrix and there is no matrix anywhere to be found in the command. Yelling may follow: "MATRIX! I didn't say anything about any stinking matrix!!" (Note for the future: MATLAB is deaf.)

Now what do we do? Well we can check to see whether the function **sum** still exists:

```
>> exist sum
ans =
     1
```

Using **help exist** tell us that the answer 1 means that **sum** is a variable. Oh yeah! We used **sum** as the name of a variable a while ago (feeling a bit sheepish now). OK. We admit we made a mistake, but what can do we do about it now? How do we undo a variable definition? Simple. We use the command **clear**, which MATLAB provides for just such contingencies:

```
>> clear sum
>> sum_to_5 = sum([1 2 3 4 5])
sum_to_5 =
    15
```

Rosy again.

Function calls

The formal definition of a function call looks like this:

```
[out_arg1, out_arg2, ...] = function_name (in_arg1, in_arg2, ...)
```

This looks pretty similar to the function declaration itself except for the missing **function** keyword. Other similarities include that the square brackets can be omitted if only one output argument is needed, that you do not need to specify any output arguments if you do not need them, and that the parentheses can be omitted if no input arguments are needed.

Let's take another look at **myRand**, shall we?

```
function [a, s] = myRand(low, high)
    a = (high-low) * rand(3,4) + low;
    s = sum(a(:));
```

and call it this way:

```
>> s = 2
s =
     2

>> mini = -5
mini =
    -5

>> [randomMatrix s] = myRand(mini, 5)
randomMatrix =
    4.2011   -1.1247    0.6044    3.4558
    1.0711   -3.7297    0.7047    0.1309
    0.1346    0.7987    1.0683   -1.3066
s =
    6.0085
```

```
>> mini
mini =
    -5
```

Let's look at what happened step-by-step. Variable **s** was defined in the workspace and was assigned the value **2**. Variable **mini** was defined in the workspace and was assigned **-5**. Then we called **myRand** with **mini** and the number **5** as input arguments and with **randomMatrix** and **s** as output arguments.

Inside the function, before the first line of our code was executed, the input argument (and local variable) **low** was created and assigned the value of **mini**, that is, −5. Input argument (and local variable) **high** was created and assigned the number **5**. The function was then executed, and output arguments (and local variables), **a** and **s**, were created and assigned the random 3-by-4 matrix and the sum of its elements, respectively. The function then returned to the place from which it was called (in this case, the Command Window). The variable **randomMatrix** was created in the Command-Window and got assigned the value of **a**. The variable **s**, which already existed there, was assigned the value that **s** had in the function. All local variables of the function **myRand** were then deleted. The variable **mini** in the workspace did not change at all.

Strings and Commands

We noted in Chapter 1 in the subsection Issuing Commands of the section Introduction to MATLAB that the word "compact" appeared in a mauve color when we used it with the format command:

```
>> format compact
```

This same color appeared in other two-word commands in that section, and it most recently occurred above in the subsection Function names when we issued the command

```
>> exist sum
```

We have also seen it in this function call

```
>> west_earth = imread('globe_west_540.jpg');
```

near the end of Introduction to MATLAB. It's time to learn what this mauveness is all about.

MATLAB is highlighting these words because it is treating them as strings. The term **string** has a special meaning in computer science. It means "sequence of characters". Of course sequences of characters are everywhere in MATLAB and in every other computer language, but most of the time they are used to stand for a value stored in a variable or for an operation defined in a function. When the interpreter encountered **mini** in the previous subsection, for example, it looked into its list of defined variables (i.e., the workspace), found **mini**, looked up its value, found that it equal replaced it with the value −5, and when it encountered **MyRand**, it carried out the operation defined by a function that we had defined. However, when a sequence of characters is punctuated on each end with a single quote (**'**), it is not replaced with anything. Those quote marks indicate to the MATLAB interpreter that the sequence of characters is not to be interpreted as the name of a variable or function. Instead it is treated literally as a string of characters. So, for example, the string `globe_west_540.jpg` is passed directly to **imread** as an input argument with nothing being looked up first. When **imread** receives that string, it treats it as the name of an image file. We will learn later in the section Data Types that MATLAB stores a string as a special type of row vector and that it provides many other built-in functions that take strings as input arguments, and we will learn how to write our own functions that take strings as input.

Was **imread** our first example of a function that takes a string as input? No. Was `globe_west_540.jpg` our first example of a string used as input to a function? No. Those distinctions belongs respectively to **format** and `compact`. The "command" **format** is actually a function that requires a string as an argument. Here is another example of its use:

```
>> format('compact')
```

This version looks like a normal function call with a normal string complete with single quote delimiters, and that is exactly what it is. The other form,

```
>> format compact
```

is exactly equivalent semantically (i.e., with regard to the meaning). The only difference is syntax. The first version uses parentheses and quote marks; the second version uses neither. Here is another example:

```
>> help sqrt
>> help('sqrt') % same meaning as previous command
```

When we call a function that takes only strings as arguments and do not assign its output to anything, we can always use either syntax, whether it is a built-in function or is a function that we have written ourselves. When we use the parentheses-free/quote-free method we tend to call the function a "command", but underneath it all every MATLAB command is just a plain old function.

So now you know why MATLAB has been highlighting things here and there in mauve. Don't like mauve? You can change it. Start by clicking File/Preferences... on the toolbar in version R2012a and earlier or by clicking Preferences on the ribbon in versions R2012b and later. Then click Colors, and you will see that you can change mauve to any color of the rainbow. (You can change the color of comments too and any other color that MATLAB uses for highlighting.)

Subfunctions

We have already used function calls inside functions. Recall how **myRand** relies on **rand** and **sum**. The question then arises: can we call our own functions from within each other? The answer is a resounding yes.

There are two ways of accomplishing this. If your function is quite complex and long, you may want to break it up to smaller pieces. If those pieces are not expected to be useful by themselves other than in the context of this complicated function, then you should create them as "subfunctions". An M-file may contain more than one function. The first one is called the **main function**; each of the others is called a **subfunction**. A file can contain an unlimited number of subfunctions. Let's revisit **myRand** and move the summation part to a subfunction. Of course, **myRand** is neither complex nor long, so you would usually not go to the trouble of making two one-line functions from one two-line function. But we will do it here as an illustrative example. We change the contents of the file **myRand.m** to look like this:

```
function [a, s] = myRand(low, high)
a = (high-low) * rand(3,4) + low;
s = sumAllElements(a);

function summa = sumAllElements(M)
summa = sum(M(:));
```

As pointed out above, the first function in the file is always the main function. That is an important distinction because the main function is the one that is executed when the command **myRand** is given. Since **sumAllElements** is not the first function in the file, it is a subfunction and cannot be called outside the file. It is said to be "invisible" from the outside. It is, however, visible from *inside* the file, and so can be called from within other functions in the file—in this case from within the main function **myRand**. Each function in a file has its own set of input and output arguments and its own set of local variables, independent from any of the other functions. Here again, each function plays in its own sandbox. When we are inside the function **sumAllElements**, none of the variables of **myRand** are visible and vice versa. That's why we have to pass the matrix **a** from **myRand** to **sumAllElements** as an input argument and then pass the results, the value of **summa**, back to **myRand** as an output argument. Instead of arguments **summa** and **M**, we could have used the names **s** and **a**. It would have made no difference at all. Just as variables in a function are different from variables with the same names in the Command Window, if variables named **s** and **a** appeared inside **sumAllElements** they would be different from the variables **s** and **a** in **myRand** even though they would have exactly the same names.

When we call **myRand** from the Command Window, we get exactly the same behavior as before. Again, what happens in a function, stays in the function.

One final note on subfunctions. The **end** keyword that concludes a function is optional. However, you have to use the same convention throughout a given file; you cannot mix conventions in the same file. You either put an **end** at the end of the main function and every subfunction below it in that file, or you do not put any **end**s at all in that file.

This is, therefore, legal:

```
function [a, s] = myRand(low, high)
    a = (high-low) * rand(3,4) + low;
    s = sumAllElements(a);
function summa = sumAllElements(M)
    summa = sum(sum(M));
```

But this is not:

```
function [a, s] = myRand(low, high)
    a = (high-low) * rand(3,4) + low;
    s = sumAllElements(a);
function summa = sumAllElements(M)
    summa = sum(sum(M));
end
```

Here is an example of what what happens when we use the first version:

```
>> myRand(-5,5)
ans =
    3.1472    4.1338   -2.2150    4.6489
    4.0579    1.3236    0.4688   -3.4239
   -3.7301   -4.0246    4.5751    4.7059
```

but here is what happens when we try the second version:

```
>> myRand(-5,5)
Error: File: myRand.m Line: 9 Column: 4
The function "sumAllElements" was closed
 with an 'end', but at least one other function
 definition was not.
 To avoid confusion when using nested functions,
 it is illegal to use both conventions in the same
 file."
```

Whoa! Guess we shouldn't do that! (Hey MATLAB, isn't this an awful lot of red for one tiny little extra **end** statement? Could we maybe lighten up a little bit? And, hey, Mr. Perfect, while we are pointing out every itsy-bitsy mistake, what about that orphaned double-quote mark at the end of your error message? What about that? Huh?)

The other way of calling your own function from one of your other functions is even simpler. If you think that a particular functionality that you need in your function may be useful in other parts of your code, you can simply create a separate function in a separate M-file. As long as you put the M-files in the same folder or put them in folders that are on your path, you are good to go.

In our particular case, you would keep the **myRand** function in **myRand.m** and put the **sumAllElements** function in its own separate file called **sumAllElements.m**. Nothing else needs to be changed and the code will work exactly the same as before.

Scope

We have discussed how the input and output arguments and all other variables defined in a function are all local variables. Another way of saying this is that these variables have local **scope**. The scope of a variable is the set of statements that can access that variable. In other words, it is the set of statements to which the variable is "visible". So far, the scope of a variable is either the statements within a single function, or the statements within the Command Window. When a function is called, it gets a dedicated part of the computer's memory where its arguments and other variables are stored. Nothing else, not the Command Window, nor any other function has access to this memory. The program execution environment, in our case MATLAB, makes sure that the memory is large enough so that the function has enough space to store all of its variables.

When the function returns, the values of the output arguments are passed back to the caller, and then this special local memory area is taken away and all the local variables disappear and the memory that had been reserved for them is now available to be allocated to variables in the Command Window or to other functions.

Global Variables

In rare cases, some functions may need to share a variable due to some special circumstances. Like most languages, MATLAB provides a way to do that. It does it, just as other languages do, by providing the user the option of declaring a variable to be "global". A global variable can be made visible in more than one place. You can access it from the Command Window and from within every function you want to. A global variable has so-called **global scope**. Global scope is visibility in more than one function or in both the Command Window and one or more functions. The counter term is **local scope**, which means accessibility in only one function or only in the Command Window and is the default scope. You can declare a variable to have global scope by using the keyword **global** like this:

```
global x;
```

Unlike C++, Java, and some other languages, MATLAB does not allow you to include an assignment in the same line, as shown in the following attempt to declare **x** to be global while simultaneously assigning it the value 4:

```
>> global x = 4
global x = 4
        |
Error: The expression to the left of the equals sign is
not a valid target for an assignment.
```

You have to make this declaration everywhere you wish to use the variable. That is, you need to include the declaration, `global x`, in every function and/or the Command Window, if you want to use the variable there. If you try to use a variable inside a function and you have not yet declared it to be global, MATLAB will either complain that it is an undefined variable (if you tried to use its value before assigning it one) or create a new local variable with the same name (if you assigned a value to it first).

You may be tempted to use global variables to take advantage of the fact that with them you can avoid using arguments to pass values into and out of functions. This may be convenient and (rarely) may even be justified, if you have a variable that many functions need to modify. However, the use of global variables is hardly ever a good idea, and it is strongly discouraged. Once a function relies on a global variable, its correct operation depends on something that is outside of the function, and that dependence makes the function less reusable. Once global variables are allowed, it is no longer enough to know what the interface of the function is, that is, its list of input and output arguments. You also need to remember that it uses one or more global variables.

Furthermore, relying on global variables is error-prone. You can accidentally overwrite the value of a global variable in one part of the program that may cause an error in another part. These kinds of programming errors are hard to find.

Therefore, for the rest of this book, we shall not use global variables at all, and we recommend that you avoid them as well, until you become an experienced programmer. When that happens, we predict that you will rarely use them in any case.

Advantages Of Functions

Functions allow you to break down your problem into smaller, more manageable and easier to solve sub-problems. When you try to assemble your shiny new furniture from IKEA, the instructions are broken down into separate steps. One step might tell you in detail how to assemble a leg. Then, instead of repeating those detailed instructions three more times, it just says, "now do this for the other three legs." You can consider the leg assembly as a function that is defined once and called four times.

All but the most trivial problems are easier to solve when they are divided into smaller chunks. Of course, how you decide to break down a problem into parts can have a big impact. In general, you should group together closely related tasks into a function and put unrelated issues into separate functions. The process of dividing up a program into smaller functions is called **functional decomposition**.

Another advantage of having functions is that you can work independently on one function without having to remember the details of all your other functions. Furthermore, a team of software developers can decide what functions they need, and then everybody can work on separate ones without needing to know how their teammates are implementing theirs. They can each develop and test their code independently. As a result, multiple parts of the software can be developed in parallel, greatly speeding up the development proc-

ess. Once everybody is ready, they can put their functions together to form the program. Since each function represents a simpler sub-problem and since they were tested separately, in all likelihood, there will be fewer programming errors in the overall program, and those errors should be easier to find. In this parallel development approach, it is crucial that each function uses separate variables that are isolated from the other functions via local scope, with no global variables or perhaps a very few whose names are mutually agreed upon. In this way, the different parts of the program are well isolated from each other preventing unwanted interactions that would happen if one person's function could overwrite somebody else's variable by mistake.

Finally, functions support reusability. You should always strive to write functions that are as flexible and general as possible. Then the same functions may be applicable to different programs, and you will not have to write them again, saving you time and effort. The built-in MATLAB functions serve as good examples. MathWorks has noted the problems that come up most frequently in various engineering and scientific domains and has provided a large number of reusable functions to solve them so that we do not have to solve the same problems that other people have already solved. Why reinvent the wheel?

In this spirit, let's take a final look at our favorite **`myRand`** function. You did not think we were done with it, did you? As you recall, it creates a 3-by-4 matrix of random numbers between the limits specified by the input arguments **`low`** and **`high`**. In our current particular problem, the numbers 3 and 4 may be fixed, but in the future we may need a random matrix of different dimensions. Therefore, we should write a more general version of the function that gets the dimensions as input arguments also. Here is the final version (we promise!):

```
function [a, s] = myRand(low, high, rows, cols)
    a = (high-low) * rand(rows, cols) + low;
    s = sum(a(:));
end
```

Let's run it:

```
>> myRand(0,5,2,4)
ans =
    4.7241    0.8242    1.8092    1.4560
    2.0699    4.6375    0.1405    3.0297

>> myRand(10,20,1,3)
ans =
   18.4718   16.2541   12.4702

>> [R ss] = myRand(0.01,0.02,10,1)
R =
    0.0199
    0.0124
    0.0175
    0.0183
    0.0104
    0.0194
    0.0149
    0.0169
    0.0172
    0.0188
ss =
    0.1657
```

Scripts

In addition to functions, MATLAB allows the creation of another kind of M-file, called the **script**. As we mentioned in Introduction to MATLAB, the first section of Chapter 1, the first two M-files that we created were scripts. A script is a collection of MATLAB commands in an M-file that does not contain a function. The script can be executed by typing the name of the file without the **.m** extension in the Command Window just like a function. But a script is very different from a function. When MATLAB executes a script, it is as if it is executing the script's commands in the Command Window itself. A script does not have its own local variables. If a new variable is created inside a script or an existing variable is modified, the workspace of the Command

Window is affected directly. While a function creates its own sandbox, a script offers no such isolation.

Consider this example script:

```
x = 12;
y = 23
z = rand(2,3);
x + y * z
```

Let's save it in the current folder in a file called **MyScript.m**. Notice that the name does not appear inside the script. Also notice that at the end of some lines we used semicolons to suppress printing, but in some others we did not. Suppose that before running the script, we have already defined two variables in the workspace named **x** and **z**:

```
>> whos
  Name     Size         Bytes  Class    Attributes
  x        1x1              8  double
  z        1x1              8  double

>> x
x =
    45

>> z
z =
     0
```

Now, let's run **MyScript** and see what happens to the various variables:

```
>> MyScript
y =
    23
ans =
   22.9391   13.4166   34.6482
   24.8927   14.0696   33.6026

>> whos
  Name     Size         Bytes  Class    Attributes

  ans      2x3             48  double
  x        1x1              8  double
  y        1x1              8  double
  z        2x3             48  double

>> x
x =
    12

>> y
y =
    23

>> z
z =
    0.4756    0.0616    0.9847
    0.5606    0.0900    0.9392
```

As you can see, the variable **x** and **z** changed, and a new variable **y** was created. Again, everything happened as if we had typed in the lines of the script in the Command Window one-by-one. And that is exactly the point. If you have a set of commands that you frequently need to use in the Command Window, you can save yourself a lot of typing by putting these frequently used commands in a script.

The most important point though is that scripts are very different from functions. You need to write a function when you need a reusable piece of code that needs to work on some input parameters and provide results as outputs without directly modifying the variables in the Command-Window workspace.

Additional Online Resources

- Video lectures by the authors:

  Lesson 3.1 Introductions to Functions (5:39)

  Lesson 3.2 Function I/O (22:15)

  Lesson 3.3 Formal Definition of Functions (2:52)

  Lesson 3.4 Subfunctions (6:17)

  Lesson 3.5 Scope (5:24)

  Lesson 3.6 Advantages of Functions (2:39)

  Lesson 3.7 Scripts (4:27)

Concepts From This Section

Computer Science and Mathematics:

function
function call
argument
input argument
output argument
local variable
global variable
variable scope

MATLAB:

M-file
`rand`
`sum`
subfunction
script

Practice Problems

Problem 1. Write a function named **`blocks`** that takes two positive integers, **`n`** and **`m`**, as input arguments (the function does not have to check the format of the input) and returns one matrix as an output argument. The function needs to return a 2**`n`**-by-2**`m`** matrix where the upper right and lower left **`n`**-by-**`m`** sub matrices are all zeros and the rest of the matrix are all ones. For example, here is an example run:

```
>> blocks(2,3)
ans =
     0     0     0     1     1     1
     0     0     0     1     1     1
     1     1     1     0     0     0
     1     1     1     0     0     0
```

The easiest solution utilizes the built-in function **`zeros`** and **`ones`** (use **`help zeros`** and **`help ones`** to see how to use them). However, there is a way to solve the problem with clever indexing. Do it both ways!

?

Problem 2. Write a function named **`custom_blocks`** that takes an **`n`**-by-**`m`** matrix as an input argument (the function does not have to check the format of the input) and returns a 2**`n`**-by-2**`m`** matrix as an output argument. The upper left **`n`**-by-**`m`** sub matrix of the output matrix is the same as the input matrix. The elements of the upper right **`n`**-by-**`m`** sub matrix are twice as large as the corresponding elements of the input matrix. Similarly, the lower left submatrix is the input matrix multiplied by three and the lower right **`n`**-by-**`m`** submatrix is four times the input matrix. For example, here is an example run:

```
>> custom_blocks([1:3;3:-1:1])

ans =
     1     2     3     2     4     6
     3     2     1     6     4     2
     3     6     9     4     8    12
     9     6     3    12     8     4
```

Problem 3. Write a function called **`even_indices`** that takes two positive integers, **`n`** and **`m`**, as input arguments (the function does not have to check the format of the input) and returns one matrix as an output argument. The elements of the **`n`**-by-**`m`** output matrix are all zeros except for the ones for which both indices are even: these need to be ones. For example, here is an example run:

```
>> even_indices(5,6)
ans =
     0     0     0     0     0     0
     0     1     0     1     0     1
     0     0     0     0     0     0
     0     1     0     1     0     1
     0     0     0     0     0     0
```

Once again, using the **`zeros`** function can help, but it is not necessary. Do it both ways!

?

Problem 4. Write a function called **alternate** that takes two positive integers, **n** and **m**, as input arguments (the function does not have to check the format of the input) and returns one matrix as an output argument. Each element of the **n**-by-**m** output matrix for which the sum of its indices is even is **1**. All other elements are zero. For example, here is an example run:

```
>> alternate(4,5)

ans =
     1     0     1     0     1
     0     1     0     1     0
     1     0     1     0     1
     0     1     0     1     0
```

Once again, using the **zeros** function can help, but it is not necessary. Do it both ways!

Problem 5. Write a function called **sum_rows** that takes a matrix as input argument (the function does not have to check the format of the input) and returns a vector as an output argument. The elements of the vector are the sums of the elements of the rows of the input matrix. Note that the built-in MATLAB function **sum** returns the sum of the columns. Here is an example run:

```
>> x = [1 2 3; -1 0 6]
x =
     1     2     3
    -1     0     6

>> sum(x)

ans =
     0     2     9
>> sum_rows(x)

ans =
     6     5
```

?

Problem 6. Write a function called **maxmin_rows** that takes a matrix as input argument (the function does not have to check the format of the input) and returns two vectors as output arguments. The elements of the first vector are the maximums of elements of the rows of the input matrix. The elements of the second output vector are the minimums of elements of the rows of the input matrix. Note that the built-in MATLAB functions **max** and **min** return the maximum and minimum of the columns. Consider the following run,

```
>> x = [1 2 3; -1 0 6]

x =

     1     2     3
    -1     0     6

>> max(x)

ans =

     1     2     6

>> min(x)

ans =

    -1     0     3

>> [maxi mini] = maxmin_rows(x)

maxi =
     3     6

mini =
     1    -1
```

Problem 7. Write a function called **pyth** that takes two input arguments **a** and **b**. The inputs are arrays of the same size. For example, if one is a scalar, the other is a scalar, or if one is a 3-by-7 matrix, then the other is too. The function does not have to check the format and size of the input. The function returns one output argument, **c**, which is also the same size as the inputs. Each element of **c** is the hypotenuse of a right triangle, while the corresponding elements of **a** and **b** are the other two sides of the same triangle. The function needs to compute **c** according to the Pythagorean theorem. Note that the built-in MATLAB functions **sqrt** computes the square root. Consider the following run,

```
>> b = [4 3;1 2]

b =
     4     3
     1     2

>> a = [3 4; 1 sqrt(5)]

a =
    3.0000    4.0000
    1.0000    2.2361

>> pyth(a,b)

ans =
    5.0000    5.0000
    1.4142    3.0000
```

?

Problem 8. Write a function called **compound** that takes three scalar input arguments **sum**, **interest** and **years** (the function does not have to check the format of the input). The function returns two output arguments, **total** and **gain**. The function computes how much money we can get by investing **sum** in the first year and then let it vest for **years** years while getting an annual interest rate of **interest** percent. The output **total** is the final amount we'll have and the **gain** is the profit. For instance, by investing $10,000 for 40 years at a rate of 15%, we'll end up with over $2.6 million according to the following run

```
>> [networth profit] = compound(10000,15,40)

networth =
   2.6786e+06

profit =
   2.6686e+06
```

Problem 9. Write a function called **zero_middle** that takes an **n**-by-**m** matrix as an input where both **n** and **m** are odd numbers (the function does not have to check the input). The function returns the input matrix with its center element zeroed out. Check out the following run,

```
>> zero_middle(ones(5))

ans =
     1     1     1     1     1
     1     1     1     1     1
     1     1     0     1     1
     1     1     1     1     1
     1     1     1     1     1
```

?

Problem 10. Write a function called **`cancel_middle`** that takes **`A`**, an **`n`**-by-**`m`** matrix, as an input where both **`n`** and **`m`** are odd numbers and **`k`**, a positive odd integer that is smaller than both **`m`** and **`n`** (the function does not have to check the input). The function returns the input matrix with its center **`k`**-by-**`k`** matrix zeroed out. Check out the following run,

```
>> cancel_middle(ones(5),3)

ans =

     1     1     1     1     1
     1     0     0     0     1
     1     0     0     0     1
     1     0     0     0     1
     1     1     1     1     1
```

Programmer's Toolbox

Objectives

MATLAB has many useful built-in functions. We will explore them in this section.

(1) We will introduce many frequently used built-in functions.

(2) We will learn about polymorphism and how MATLAB exploits it to change a functions behavior on the basis of the number and type of its inputs.

(3) Because random numbers play an important role in computer programming, we will learn how to use the MATLAB random number generator.

(4) We will learn how to get input from the keyboard, to print to the Command Window, plot graphs in a Figure window, and play audio through the speakers.

MATLAB includes hundreds of built-in functions that make programmers' lives much easier.

Every programming language provides basic constructs to create programs. You can create variables, assign values to them, perform arithmetic operations, organize your program into functions, etc. Beyond these basic building blocks, there are many more complex operations that are frequently needed, yet the programming language itself does not provide them. These include mathematical functions such as trigonometry, text manipulation, outputting text and graphics on the screen, getting input from the user through either the keyboard or the mouse, and many many others. It would be extremely wasteful if every programmer were to have to recreate these from scratch. Instead, every programming language comes with a large set of ready-to-use functions for these frequently used operations. These sets of functions are often referred to as **libraries**.

MATLAB is no exception. It provides many hundreds of built-in functions. In the previous section, we have seen **rand** as one of the built-in functions MATLAB provides. Table 2.1 lists a few more useful functions to create matrices in MATLAB.

Table 2.1 Matrix-building functions

FUNCTION	RETURNS AN N-BY-M MATRIX OF
`zeros(N,M)`	zeros
`ones(N,M)`	ones
`eye(N,M)`	zeros except for the diagonal elements that are ones
`rand(N,M)`	random numbers uniformly distributed in the range from 0 to 1

For example, **zeros(n,m)** returns an **n**-by-**m** matrix, all of whose elements equal zero. Similarly, **ones(n,m)**, returns **n**-by-**m** ones. An interesting function is **eye(n,m)**, which returns an **n**-by-**m** matrix that has all zeros, except for those elements on the diagonal, which are equal to one. The **diagonal** of a matrix is the set of elements whose indices are equal to each other. Thus, in the matrix **M**, the diagonal elements are **M(1,1)**, **M(2,2)**, …. Why is it called "eye"? Well, that's because **eye(n,n)** produces an identity matrix, whose symbol in mathematics is *I* (pronounced "eye", get it?). An **n**-by-**n** identity matrix can be multiplied by any other **n**-by-**n** matrix **X**, and the result is **X**:

```
I = eye(3,3)

I =
     1     0     0
     0     1     0
     0     0     1

>> X = [1 2 3;4 5 6;7 8 9]

X =
     1     2     3
     4     5     6
     7     8     9

>> I*X

ans =
     1     2     3
     4     5     6
     7     8     9

>> X*I

ans =
     1     2     3
     4     5     6
     7     8     9
```

Polymorphism

Another useful function takes the square root of its argument:

```
>> sqrt(9)

ans =
     3
```

In the example above, the argument to **sqrt** is a scalar, which is a 1-by-1 array (Note: We will use the terms array and matrix interchangeably when referring to two-dimensional arrays). We could have given **sqrt** an array of any size and shape as an argument. Its behavior in that case is to return an array of the same size and shape, such that each element of the result is equal to the square root of the corresponding element of the argument. Thus, if **X** is a 3-by-4 array, then **Y = sqrt(X)** results in **Y(m,n) = sqrt(X(m,n))** for all elements of **X**:

```
>> X = [1 4 9; 16 25 36]

X =
     1     4     9
    16    25    36

>> sqrt(X)

ans =
     1     2     3
     4     5     6
```

In the general study of programming languages, when the type of an argument used in a function can vary (as for example, from a scalar to a vector to a matrix) from one call of the function to the next, the function is said to be **polymorphic**. And **polymorphism**, which is the title of this subsection, is the property of being polymorphic. As we will see below, polymorphic function may even allow varying numbers of arguments. (The term polymorphic means "having multiple forms", which matches its definition, since it means that the function call can have more than one form.) The characteristic of being polymorphic is called polymorphism. (An alternate terminology is to say that the function's arguments are polymorphic, but the idea is the same.) As can be seen from the **sqrt** example above, MATLAB's support for polymorphism is very powerful, and it makes it possible to handle a huge variety of situations (e.g., a huge variety of different matrix shapes) with relatively few functions. It makes it possible for MATLAB to do a great deal of work very efficiently (i.e., quickly) with very few statements. Polymorphism is not supported by the most popular older languages, such as Fortran, C, Pascal, and Basic. Many modern languages, on the other hand, such as Ada, C++, Java, and other so-called "object-oriented" languages (and even a few of the older languages, such as LISP, which was one of the very first programming languages and remains in use today) do provide support for polymorphism. It is in fact a major feature of these languages, just as it is a major feature of MATLAB.

Unlike most of the other modern languages that support polymorphism, MATLAB's typical function returns an object that is the same shape as the argument that it is given. For example, if **f** is a typical function, then for **y = f(x)**, if the value of **x** is a scalar, then a scalar value will be returned into **y**. If **x** contains a matrix of dimensions **m**-by-**n**, the value returned into **y** will also be an **m**-by-**n** matrix. When the function allows multiple arguments of different types in a single call, the returned type will vary according to the function.

Returning an object whose shape is the same as the argument is not always appropriate however. For example, the function **sum**, when given a row or column vector, returns a scalar—not a row or column vector—that is equal to the sum of the elements of the vector:

```
>> v = [1 -3 5 10];
>> sum(v)

ans =
    13
```

When **sum** is given a two-dimensional matrix, it calculates the sum for each column of the matrix and returns a row vector—not a two-dimensional matrix—of those elements:

```
>> M = [1 10 100; 2 20 200; 3 30 300]

M =
     1    10   100
     2    20   200
     3    30   300

>> sum(M)

ans =
     6    60   600
```

Thus, **sum** is certainly polymorphic, since it accepts either a vector or a two-dimensional matrix as its argument, but it does not behave typically by returning an object whose shape is different than that of the argument.

So far we have concentrated on polymorphism that allows variation in the types of the arguments. There is a second type of polymorphism. A function is also polymorphic if it accepts varying numbers of arguments. So, for example, if a function can be called with one argument or with two arguments, it is polymorphic. The function **sum** exhibits this second aspect of polymorphism as well, as shown by the following call:

```
>> sum(M,2)

ans =
   111
   222
   333
```

The argument **M** in this call is the same matrix as used above, but this time the summation is carried out along each row, and the resulting sums are put into a column vector. The second argument tells **sum** which dimension it is to sum over and whether to return a row vector (second argument equal to 1) or column vector (second argument equal to 2). Since the second dimension is the row index, the call **sum(M,2)** means to sum across the rows. The call **sum(M,1)** means to sum over the column index, so it returns the same vector as that returned by **sum(M)** above. Thus, the second argument is optional, and if it is omitted, the default dimension is 1. When we describe functions in this book, we will not always list all the optional variables, but you can always learn whether there are optional arguments and their meanings by using **help**.

Returning more than one object

Some MATLAB functions can return more than one object. A good example is the function **max**. This function can be used to find the largest element in a vector, as in the following example:

```
>> a = max([1 4 -5 0])
a =
     4
```

In this example, **max** returned one object, the number **4**, which is the largest element, but **max** can do more, as shown in the next example:

```
>> [a b] = max([1 4 -5 0])
a =
     4
b =
     2
```

In this example, **max** returned two objects. The first one is the maximum value; the second one is the index of the first element that contains the maximum value. The two variables, **a** and **b**, in brackets on the left side of the equal sign in the call to **max** are called output arguments in MATLAB, as we have seen in the previous section. This ability to return more than one object is an important feature of MATLAB, and that feature is lacking in the major general-purpose languages: Ada, C, C++, Fortran, Java, and Pascal. Furthermore, the ability to call such a function in different ways so that it returns different numbers of output arguments is a third type of polymorphism, and MATLAB is one of *very* few languages that provide that.

Note that some MATLAB functions return a vector that contains more than one element. Returning a vector is not equivalent to returning more than one object. The vector, despite its multiple values is one object. If one variable can hold it, it is one object. The **size** function, which we encountered before, is such a function. It returns one object, which is a two-element vector. The first

element is the number of rows of the argument; the second is the number of columns. When only one object is returned by a function, it is possible to capture it in one variable, as in the following example, in an individual variable. The syntax is obvious from the following example,

```
>> v = size([1 2 3; 4 5 6]);
>> v
v =
     2     3
```

As can be seen from the examples above, because **max** returns two objects, it is not possible with **max** to capture both the maximum value and the index in a single variable. MATLAB does, on the other hand, make it possible to capture the two elements of the vector returned by **size** in two separate variables, as follows;

```
>> [m n] = size([1 2 3; 4 5 6])
m =
     2
n =
     3
```

Here we see polymorphism in regard to the output arguments. If one output is requested, **size** returns a vector; if two are requested, **size** returns two scalars.

The number of functions provided with a MATLAB installation is huge, and many more are available from the Internet. You have seen a few of them in this book and in Table 2.1. In the following tables (Table 2.2, Table 2.3, Table 2.4), you will find some more of them. To learn more about them and to learn the names of many more functions it is a good idea to use MATLAB's help facility.

Table 2.2 Trigonometric functions

FUNCTION	RETURNS
`acos(x)`	Angle in radians whose cosine equals x
`acot(x)`	Angle in radians whose cotangent equals x
`asin(x)`	Angle in radians whose sine equals x
`atan(x)`	Angle in radians whose tangent equals x
`atan2(y,x)`	Four-quadrant angle in radians whose tangent equals y/x
`cos(x)`	Cosine of x (x in radians)
`cot(x)`	Cotangent of x (x in radians)
`sin(x)`	Sine of x (x in radians)
`tan(x)`	Tangent of x (x in radians)

Table 2.3 Exponential functions

FUNCTION	RETURNS
`exp(x)`	*e* raised to the x power
`log(x)`	Natural logarithm x
`log2(x)`	Base-2 logarithm of x
`log10(x)`	Base-10 logarithm of x
`sqrt(x)`	Square root of x

Table 2.4 Functions that work on complex numbers

FUNCTION	RETURNS
`abs(z)`	Absolute value of z
`angle(z)`	Phase angle of z
`conj(z)`	Complex conjugate of z
`imag(z)`	Imaginary part of z
`real(z)`	Real part of z

Additional useful functions are listed in Table 2.5, Table 2.6, and Table 2.7.

Table 2.5 Rounding and remainder functions

FUNCTION	RETURNS
`fix(x)`	Round x towards zero
`floor(x)`	Round x towards minus infinity
`ceil(x)`	Round x towards plus infinity
`round(x)`	Round x towards nearest integer
`rem(x,n)`	Remainder of **`x/n`** (see help for case of noninteger n)
`sign(x)`	**`1`** if **`x>0`**; **`0`** if **`x`** equals **`0`**; **`-1`** if **`x<0`**

Table 2.6 Descriptive functions applied to a vector

FUNCTION	RETURNS
`length(v)`	Number of elements of v
`max(v)`	Largest element of v
`min(v)`	Smallest element of v
`mean(v)`	Mean of v
`median(v)`	Median element of v
`sort(v)`	Sorted version of v in ascending order
`std(v)`	Standard deviation of v
`sum(v)`	Sum of the elements of v

Table 2.7 Descriptive functions applied to a two-dimensional matrix

FUNCTION	RETURNS A ROW VECTOR CONSISTING OF
`max(M)`	Largest element of each column
`min(M)`	Smallest element of each column
`mean(M)`	Mean of each column
`median(M)`	Median of each column
`size(M)`	Number of rows, number of columns
`sort(M)`	Sorted version, in ascending order, of each column
`std(M)`	Standard deviation of each column
`sum(M)`	Sum of the elements of each column

Random Number Generation

In many scientific and engineering problems random numbers play an important role. MATLAB, like most programming languages, has built-in support for generating random numbers. To be precise, the numbers provided by a random number generator are not truly random since a deterministic algorithm computes them. Hence, they are called "pseudo random" numbers. But for most practical problems, they are sufficient.

We have already introduced the most frequently used MATLAB function for random number generation, **`rand`**, in the previous section. This is another polymorphic function. Calling **`rand`** without an argument returns a single random number, while **`rand(n)`** returns an **`n`**-by-**`n`** matrix of random numbers. Similarly, **`rand(m,n)`** provides an **`m`**-by-**`n`** matrix of random numbers.

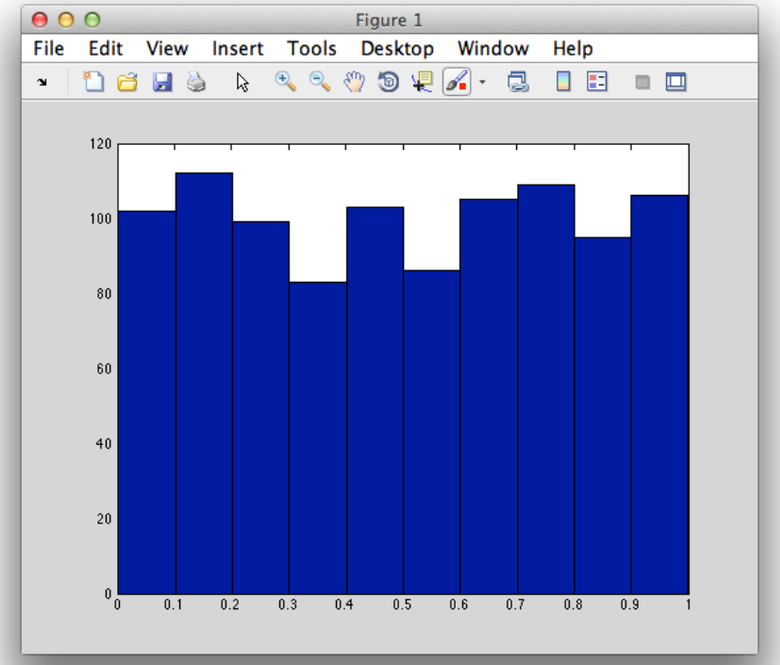
```
>> rand

ans =
    0.8147

>> rand(3)

ans =
   0.9058   0.6324   0.5469
   0.1270   0.0975   0.9575
   0.9134   0.2785   0.9649

>> rand(3,5)

ans =
   0.1576   0.4854   0.4218   0.9595   0.8491
   0.9706   0.8003   0.9157   0.6557   0.9340
   0.9572   0.1419   0.7922   0.0357   0.6787
```

Notice that all the numbers generated by **`rand`** above are between 0 and 1. That is no accident. The function returns numbers strictly larger than 0 and smaller than 1 that are uniformly distributed. What this means is that any number between 0 and 1 has the exact same probability to appear as an output of **`rand`**. To demonstrate this, let's try this:

```
>> hist(rand(1,1000))
```

The function **hist** plots a histogram of the elements in its argument. A histogram shows how many elements fall into given value intervals, and **hist** does this graphically (and other ways too–see **help**). We should see a window like the one in Figure 2.1 pop up.

Figure 2.1 *Histogram of 1000 pseudo random numbers produced by* ***rand***

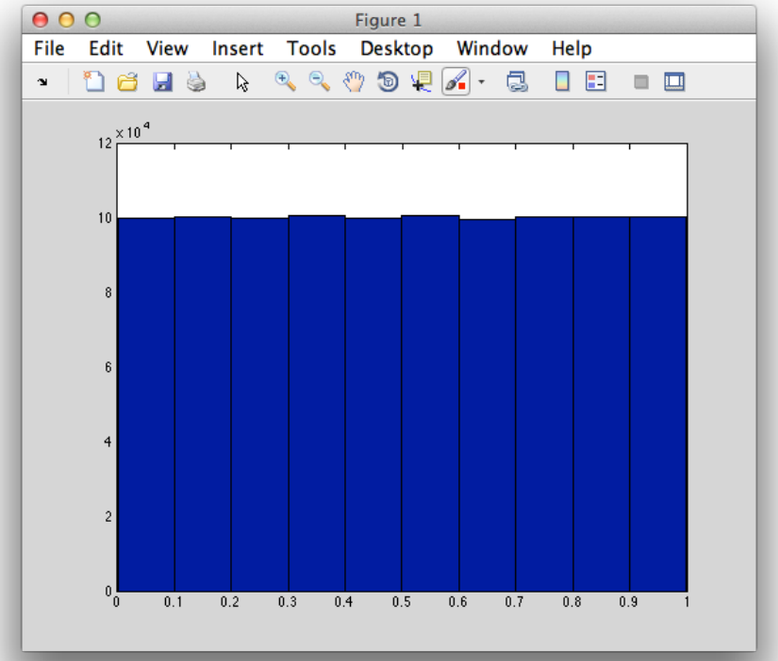


First, we generated a vector of 1000 pseudo random numbers using **rand** and then we plotted its histogram using **hist**. In this case, MATLAB generated 10 evenly spaced intervals between 0 and 1 and plotted the results as a bar chart. The expected value for each interval is 100, since the output of **rand** is uniformly distributed. As we can see, the numbers are close to 100, but there is considerable variance. That is because, 1000 is not a high enough number to give a clear idea of the real distribution. Let's try a million instead:

```
>> hist(rand(1,1000000))
```

As you can see in Figure 2.2, the height of the bars have nicely evened out.

Figure 2.2 *Histogram of a million pseudo random numbers produced by* ***rand***



How can one get pseudo numbers that fall into an interval other than (0, 1), one might ask? It is quite easy. You need to call **rand** and simply use arithmetic operators to modify the results. For example, if you need a 5-by-5 matrix of random numbers between 2 and 8, simply do this:

```
>> rand(5) * 6 + 2

ans =
   4.3534   7.2712   5.6778   4.7812   4.3273
   4.8132   2.9664   3.0216   4.2961   7.1422
   2.6214   5.8898   5.9265   6.9743   2.0767
   6.8485   6.1651   3.9047   3.6829   5.3122
   4.6050   2.5022   4.5134   3.4277   5.1772
```

The interval (2 , 8) is 6 wide, so we need to multiple by 6, and since it needs to start 2 above 0, we add 2.

Normal distribution

In science and engineering, the normal distribution is a very important concept. It is also referred to as Gaussian distribution or the Bell Curve due to its distinctive shape. MATLAB has a function called **randn** that provides pseudo random numbers with a normal distribution with a mean of zero. The output of the function can be any positive or negative floating point number (or zero), but the smaller the absolute value of the number, the higher probability it has of being produced by **randn**. Consider this MATLAB command:

```
>> hist(randn(1,1000000),100)
```

The input arguments to **randn** work exactly the same way as to **rand**. The extra input argument of **100** to the **hist** function above specifies that this time we want **100** intervals as opposed to the default of **10**. The result is shown in Figure 2.3.

Figure 2.3 *The distribution of one million numbers generated by* ***randn***

Random integers

We can use **rand** to generate uniformly distributed, pseudo random integers like this:

```
>> fix(rand(6) * 10) + 1

ans =

     9    10    10     3     2     4
     8     2     1     2     9     5
     1     1     2     6     4     4
     4     2     5     6     3     4
     8     5     1     5     4     3
     4     7     2     8     4     2
```

The function **fix** rounds toward zero. Multiplying the output of **rand** by 10 will give us numbers greater than 0 and smaller than 10, so **fix** will return integers between 0 and 9 inclusive. Hence, adding 1 will generate integers in the range from 1 to 10.

As usual, MATLAB makes our life easier by providing a special function for a common task. In this case, it is the function **randi**, which supplies uniformly

distributed pseudo random integers. The following function is equivalent to the more complicated command above:

```
>> randi(10,6)

ans =
     5     3     9     6     4     6
     1     3     2     5     2     9
     2    10     3     2     6     5
     2     7     4     2     8     1
     1     4     2     9     1     1
     1     5     6    10     9     2
```

The arguments to **randi** are a bit different from those of **rand**. The first argument specifies the interval of the values. In the call above, we wanted integers in the range from 1 to 10, hence we used 10. If we want the interval to start not at 1, but somewhere else, we have to call **randi** like this:

```
>> randi([2,8],4,5)

ans =
     5     5     3     3     5
     3     2     6     8     6
     6     8     4     2     5
     2     8     2     2     8
```

Here, we generated a 4-by-5 matrix of pseudo random integers falling in the range from 2 to 8. The first argument is a vector containing the minimum and maximum values we want. As you can see, the second and third arguments specify the size of the output array similarly to **rand**.

Initializing the random number generator

If we start MATLAB and call **rand** and then restart MATLAB and call **rand** again, we get the exact same "random" number. In fact, if we keep calling **rand** after restarting MATLAB, we will always get the exact same sequence of numbers. At first, this seems pretty bad. What good is a random number generator if it always produces the same numbers? Repeatability is an important concept: many times if you run the same program, you want to get the exact same result. This is especially true while you are developing your code and trying to find you programming errors. If the behavior of the code changes simply because the random number generator gave different numbers, you would have a much harder time finding your coding mistakes. On the flip side, once your code works you may very well want to get different numbers from the random number generator every time you run your program.

Fortunately, there is a simple way. The function **rng** is provided to initialize the pseudo random number generator of MATLAB. It takes an integer argument and once it is called, the sequence of random numbers will immediately change. On the other hand, if you call **rng** with the same argument again, it will initialize the random numbers the same way and you will get the very same sequence. For example,

```
>> rand

ans =
    0.6723

>> rng(1)
>> rand

ans =
    0.4170

>> rand(4)

ans =
    0.7203    0.0923    0.5388    0.8781
    0.0001    0.1863    0.4192    0.0274
    0.3023    0.3456    0.6852    0.6705
    0.1468    0.3968    0.2045    0.4173
```

```
>> rng(1)
>> rand

ans =
    0.4170

>> rand(4)

ans =
    0.7203    0.0923    0.5388    0.8781
    0.0001    0.1863    0.4192    0.0274
    0.3023    0.3456    0.6852    0.6705
    0.1468    0.3968    0.2045    0.4173

>> rng(10)
>> rand

ans =
    0.7713

>> rand(4)

ans =
    0.0208    0.2248    0.0883    0.5122
    0.6336    0.1981    0.6854    0.8126
    0.7488    0.7605    0.9534    0.6125
    0.4985    0.1691    0.0039    0.7218
```

As you can see, different arguments to **rng** initialize (or "seed") the random number generator differently. Also, note that the various functions **rand**, **randi** and **randn** use the same random number generator. They use the same sequence of random numbers (but they may change it, for example, **randi** converts it to an integer).

Of course, we are still not generating truly random numbers: If our code calls **rng** with the same seed, it will generate the same sequence again. Again, this is good for repeatability, for example, we do not have to restart MATLAB to get the same set of numbers, but what if we want our code to behave differently every time we call it? Fortunately, **rng** helps with that too:

```
>> rng('shuffle')
>> randi(10,5)

ans =
     8     4    10     7     1
     3     7     9     8     3
     5     8     4     3     8
     1     5     9     4    10
     7     5     3     2     6

>> rng('shuffle')
>> randi(10,5)

ans =
    10    10     5     2     4
     2     4     3     3     7
     8     2     2     8     3
     5     5     4     6     1
     7     5    10     4     2
```

Calling **rng** with **'shuffle'** uses the system time as the seed. Since it has millisecond resolution, the random number generator will be guaranteed to have been seeded differently. This will create a different random number sequence every time.

Keyboard Input

Some MATLAB programs require input from the user through the keyboard. In a typical situation, the user gives input in response to a message that is printed by the program requesting input from the user. As we have learned, this message is called a prompt. We have already seen that MATLAB's Command Window uses a very simple prompt: **>>**. In the programs that you will write, a prompt may be any sequence of characters you like. It may range from one or two characters to a phrase or complete sentence, but regardless of the form of the prompt, the user must respond to it by typing something

and hitting the Enter key. All this is accomplished by means of the **input** function. For example, suppose an M-file includes the command,

```
fave = input('Type your favorite vector: ');
```

When this command is executed, the program will print this in the Command Window,

```
Type your favorite vector: |
```

with the vertical bar at the right blinking, and it will wait in this state until the user types something and hits Enter. The sequence of characters that begins with the "**T**" in "**Type**" and ends with a space after the colon, is a prompt. Whatever prompt you want **input** to print must be given as an input argument.

While MATLAB patiently waits for input from the keyboard, that blinking vertical bar shows where the first character that the user types will appear. Note that the prompt above ended with a blank character (the space between the colon and the single quote). This blank is "printed" after the colon, meaning that when the user types in a response, there will be one space between the colon and the first character of the response. Why? Well, it just looks better to the user that way (for some reason). If that space is omitted, MATLAB will not include it for you.

Let's suppose the user types in the vector `[1 2 3 4]`. Then the interaction would look like this:

```
Type your favorite vector: [1 2 3 4]
```

Suppose the user types, not a vector, but the matrix `[1 2; 3 4]`. Then the interaction would look like this:

```
Type your favorite vector: [1 2; 3 4]
```

The result is that **fave** would be given the value `[1 2; 3 4]`. Note that MATLAB pays no attention to what is written in the prompt string. Regardless of the type of the object that the user types in—a scalar, a complex number, a vector, a matrix, or even a string—it will be accepted. The fact that the prompt tells the user to type a vector is of no consequence.

Formatted Output

A program's work is of no use to anyone unless it communicates its results to the outside world. So far, we have taken advantage of the fact that a command that is not followed by a semicolon automatically prints any value it computes to the Command Window. We have also seen that it is possible to change the way in which computed results are displayed by using the **format** command.

This method of communication is workable for simple programs, but MATLAB provides a far more sophisticated way for printing information by means of a function, borrowed from the languages C and C++, called **fprintf**. Its name stands roughly for "formatted printing to a file", and the function is versatile enough to allow us to print either to the Command Window or, as we will learn in the File Input/Output section, to a file. It requires at least one argument. Its first argument (when it is being used to print to a file, it is the second argument) is a **format string**, which is a string that specifies the way in which printing is to be done, such as words that must be printed, spacing, the number of decimal places to be used for printing numbers, etc. In the format string, the format of each value that is to be printed is specified individually. So, for example, one value may be printed with four decimal places and another with nine. Additionally, any text that is to be printed along with the values is included in the format string. Following the format string there are typically additional input arguments to

fprintf. These are the values that are to be printed and they must appear as arguments in the order that they are to be printed. For example,

```
>  x = 3; y = 2.71; z = x*y;
>  fprintf('%d items at $%.2f\nTot = $%5.2f\n',x,y,z)
3 items at $2.71
Tot = $ 8.13
```

Conversion characters and escape characters

In this particular format string, the **%d**, **%.2f**, and **%5.2f** are conversion characters. Instead of being printed, a percent sign (**%**) and the characters immediately after the percent sign indicate to MATLAB that the value of one argument after the format string, is to be printed in a specific format. The percent sign acts as a so-called **escape character** , which is a term used in other computer languages as well as MATLAB and which means that the character or characters that follow it have a special meaning. The meaning of the **d** in **%d**, for example, is "If the value is an integer, print it without a decimal point; otherwise print it in scientific notation." For each percent sign, there will usually be exactly one argument after the format string to go with it. One argument's value is printed for each percent sign in the order that they appear. The **f** in **%.2f** is a **format specifier**, which is a character specifying the format in which an object is to be printed (or read, as we will learn in File Input/Output).

See Table 2.8 for additional format specifiers. The format specifier **f** means "Print using fixed-point notation," which in turn means that the number is printed as digits with a decimal point. The **2** after the decimal point specifies the precision. It means that exactly two digits must be included to the right of the decimal point (even if they are all zeros). The **5** in the format **%5.2f** means "Print using at least five spaces." Since **f** is the format specifier, these spaces will include a minus sign if the number is negative, the digits before the decimal place, the decimal point itself, and the digits after the decimal point. The **2** in **%5.2f** has the same meaning as it did in **%.2f**. A meaningful sequence of characters, such as **%5.2f**, that begins with an escape character is called an **escape sequence**. Another escape sequence is **\n**, which means "go to a new line". The backslash, which is one of the division operators, has a different meaning when it occurs inside a string. Here it is an escape character. The special meaning of **n** is "newline". To print a backslash, use two of them with no intervening space: ****. To print a single quote, use two of them: **''**. To print a single percent sign, use two of them: **%%**.

Table 2.8 Format specifiers of ***fprintf***

FORMAT SPEC	DESCRIPTION
c	single character
d	decimal notation (but no decimal if integral)
e	exponential notation
E	exponential notation with an upper case E
f	fixed-point notation
g	shorter of e or f
G	shorter of e or f but with an upper case E
o	unsigned octal notation
s	string
u	unsigned decimal notation
x	hexadecimal notation
X	hexadecimal notation with uppercase

Partial use and recycling of the format string and matrix arguments

If the number of arguments to be printed (i.e., those after the format string) is smaller than the number of percent signs in the format string, then the printing will stop just before the first unused percent sign is encountered. If there are more arguments to be printed than percent signs, the format string will be recycled repeatedly until all of the arguments are printed. If any or all of the arguments is a matrix, then the printing will behave as if the matrix were re-

placed by its elements written as separate arguments in column-major order. Thus, the matrix **A** will be printed in the same way that the column vector **A(:)** would be printed or if all of its elements were provided as separate arguments to **fprintf**.

Imaginary parts ignored

A limitation of **fprintf** is that it cannot be used to print complex numbers properly. If a complex argument is given to it, **fprintf** will ignore the imaginary part! To print complex numbers, you may use another function called **disp**. For example,

```
>> z = 2 + 3i;
>> fprintf('z = '); disp(z);
z =    2.0000 + 3.0000i
```

Alternatively, you can extract the real and imaginary parts of a complex number by using the functions **real** and **imag**. The following example reveals how it can be done:

```
>> fprintf('z = %.2f + %.2fi',real(z),imag(z))
z = 2.00 + 3.00i
```

Plotting

MATLAB has an extremely powerful plotting facility, which is supported by functions, including

- **plot**
- **title**
- **xlabel**
- **ylabel**
- **grid**
- **semilogx**
- **semilogy**
- **loglog**

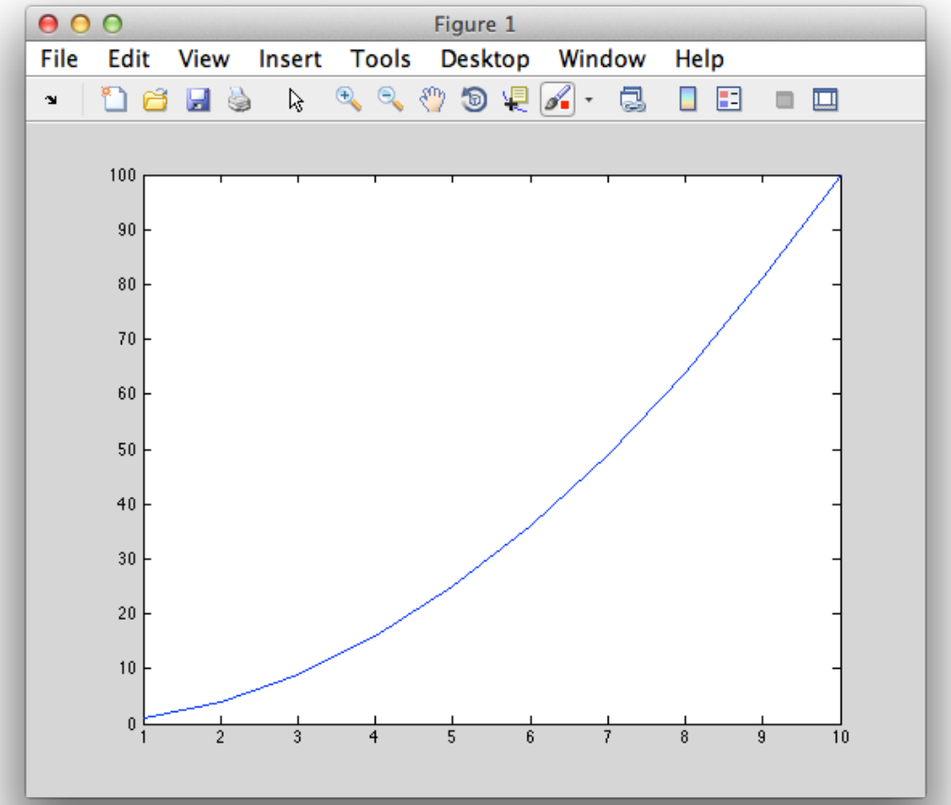
and many others. With these functions it is possible to (a) plot more than one function at a time on a given plot, (b) select distinct symbols, called "markers", for distinct plots, (c) select distinct colors for distinct plots, (d) print a title with a plot, (e) label the axes, (f) provide a "graph-paper" grid, (g) change one or both axes to be scaled according to the logarithms of the values. Many other options are available, including three-dimensional plotting.

The simplest approach to plotting is simply passing **plot** a vector:

```
>> a = (1:10).^ 2;
>> plot(a)
```

A figure window labeled Figure 1 pops up and the plot appears in it, as shown in Figure 2.4.

Figure 2.4 *Plotting a simple quadratic*



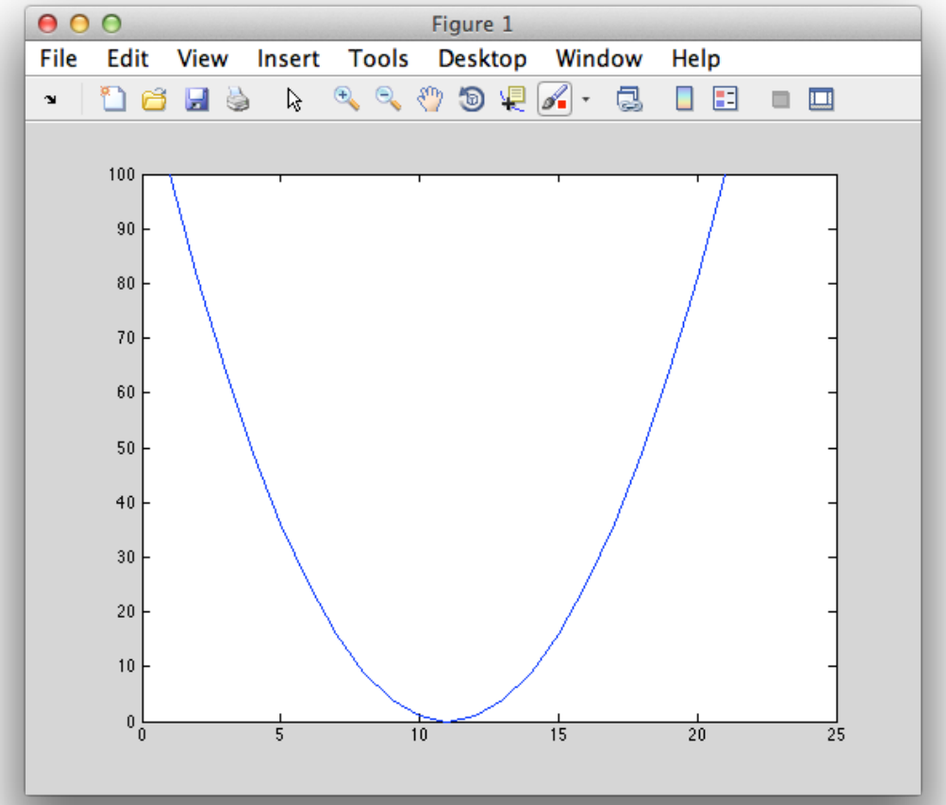
We got the left side of a parabola as expected. Let's get the other side as well. We do that by squaring negative numbers as well:

```
>> b = (-10:10).^2;
>> plot(b)
```

The previous plot is replaced and the result is shown in Figure 2.5.

While the shape looks fine, the axes do not. The square of 11, for example, should not be shown as 0. What happened is that **`plot`** simply uses the index of the vector for the *x* axis if it receives only a single input argument.

Figure 2.5 *Plotting both sides of the parabola*

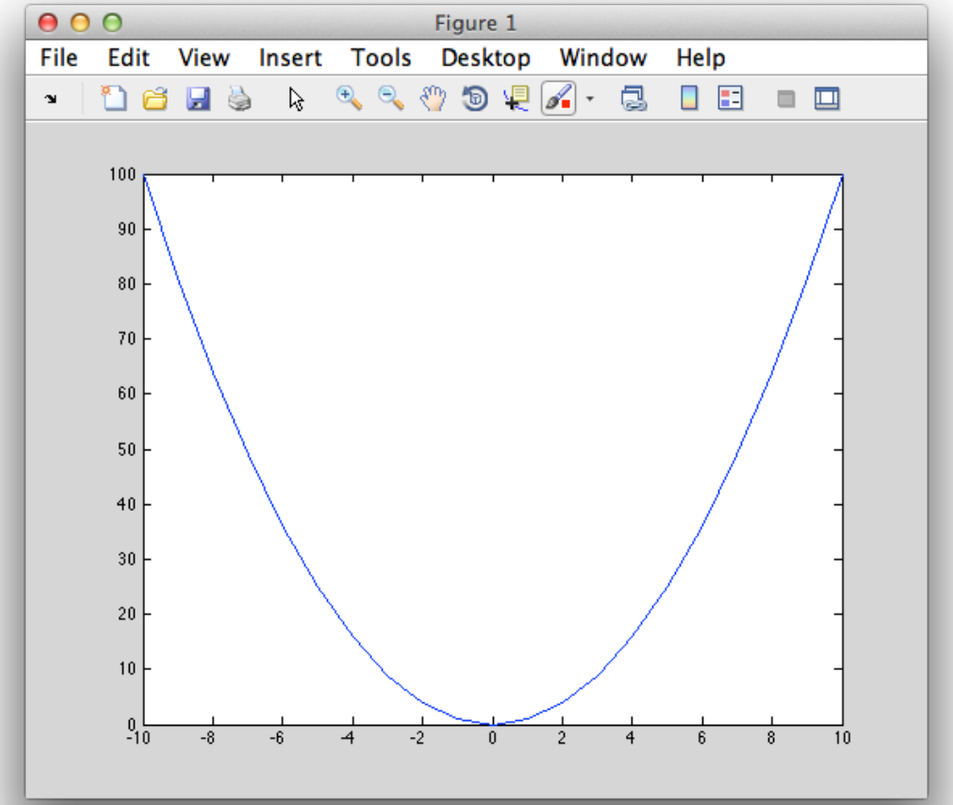


The better approach to plotting is to create two vectors of the same length and give them as arguments to **`plot`**. Thus, if **`x`** and **`y`** are two vectors of length **`N`**, then **`plot(x,y)`** produces a plot of **`N`** points with **`x`** giving the horizontal positions and **`y`** giving the vertical positions. For example,

```
>> t = -10:10;
>> b = t.^2;
>> plot(t,b);
```

which is shown in Figure 2.6.

Figure 2.6 *The correct way of plotting a quadratic*

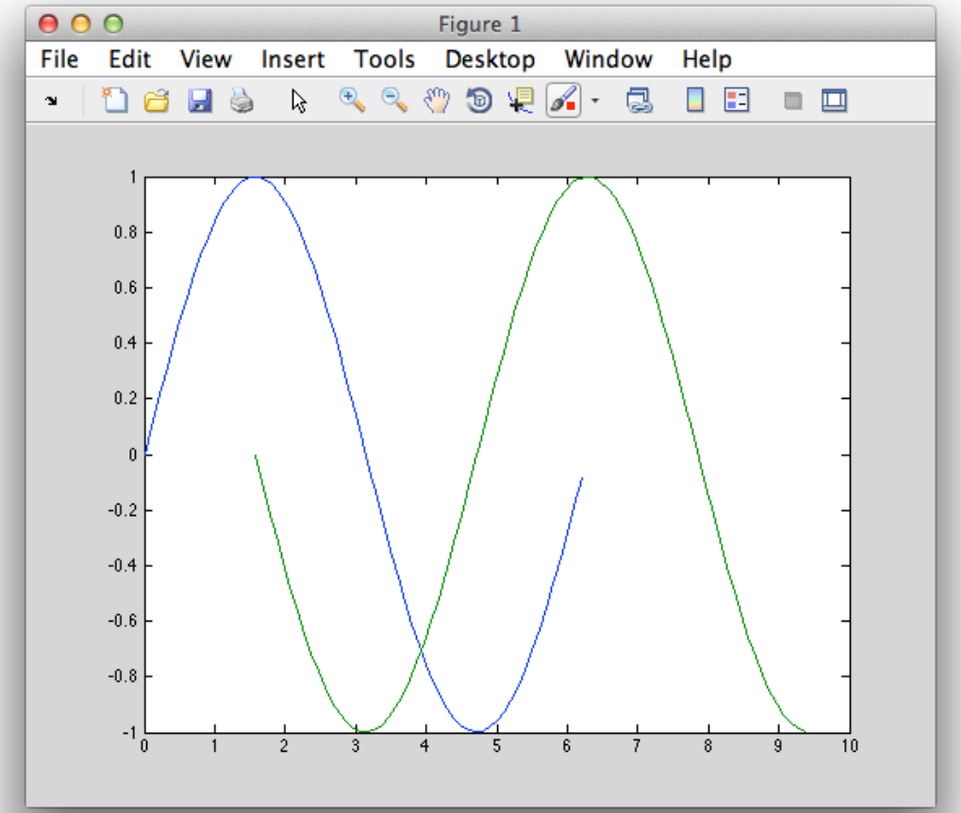


It is easy to extend this idea to put more than one plot in a figure because **`plot`** is polymorphic. It accepts a varying number of arguments. For example,

```
>> x1 = 0:0.1:2*pi;     y1 = sin(x1);
>> x2 = pi/2:0.1:3*pi; y2 = cos(x2);
>> plot(x1,y1,x2,y2)
```

produces the plot shown in Figure 2.7.

Figure 2.7 *A plot of a sine and cosine*



The vector **`y1`** is plotted versus the vector **`x1`** and the vector **`y2`** is plotted versus the vector **`x2`**, separately on the same set of axes. The ranges of the x and y axes are automatically chosen to encompass all the values in the vectors.

So far, we have allowed MATLAB to replace each previous plot with our new one. This is the default, and each one appears in a figure window with the label Figure 1. If we wish to leave the previous plot in Figure 1 and plot the next plot in Figure 2, we can do that by issuing the command,

```
>> figure
```

before calling plot. A blank figure will appear with a number one greater than that of the last figure, and the next call of plot will cause plotting to appear in the new figure.

A number of pairs of vectors may be given to **plot**. It will plot them all on the same set of axes.

It can be seen in Figure 2.7 that the color blue is used for the first plot and the color green for the second plot. It is possible to alter the appearance of the plots by using additional arguments in the form of strings. For example, the command

```
>> plot(x1,y1,'r',x2,y2,'k:')
```

produces the version in Figure 2.8.

Figure 2.8 *Plotting with different colors and markers*

The string **'r'** means "red line". The string **'k:'** means "black dotted line". The **k** means black; the colon means dotted (why not **b** for black? Answer: **b** means blue). One, two, or three characters can be included optionally for every pair of vectors to be plotted. Table 2.9 gives a list of possible symbols and their meanings. Additional line styles can be found by **help plot**.

Table 2.9 *Symbols used to alter plot format*

SYMBOL	COLOR	SYMBOL	LINE STYLE
b	blue	.	point
c	cyan	o	circle
g	green	x	cross
k	black	+	plus
m	magenta	*	star
r	red	s	square
y	yellow	d	diamond
w	white	-	solid (default)
		:	dotted
		-.	dashdot
		--	dashed

It is also possible to plot more than one function on the same axes by means of sequential **plot** functions using the command **hold**. Once the command **hold on** has been issued, all subsequent plots will be made on the same axes with the ranges of the axes being adjusted accordingly. When the command **hold off** is issued, all subsequent calls to **plot** will employ a new figure. The command **hold** (not followed by **on** or **off**) can be used as a toggle to turn plot holding on or off. Thus, the following commands could be used to produce the same plot:

```
>> hold on
>> plot(x1,y1,'r')
>> plot(x2,y2,'k:')
>> hold off
```

There are additional options, not reachable through the **plot** function, that will enhance the visual quality of a plot. The command **grid** turns on a graph-paper like grid. Repeating the command turns it off (and, like **hold**, it may be used with **off** and **on** as well). The **grid** command affects only the currently active figure. (The "active" figure is the one most recently created or the one which has received the most recent mouse click, whichever happens last.)

A plot can be adorned with a title at the top and labels along the *x* and *y* axes by means of the functions,

```
>> title(string)
>> xlabel(string)
>> ylabel(string)
```

each of which takes a single string (the label) as an argument. So, for example, the string might be **'My nifty title'**, or it might be the name of a variable that contains a string as its value. It is also possible to change the ranges of the axes with the function **axis**, which accepts a single four-element vector as its argument:

```
>> axis([xmin xmax ymin ymax])
```

To return the range to its original value the command

```
>> axis auto
```

can be used. Notice that this command does not have the form of the normal function with arguments in parenthesis. Another use of the axis command is to alter the relative lengths of the x and y axes match. The command

```
>> axis square
```

will make the length of the axes be equal, resulting in a square plot area. The command

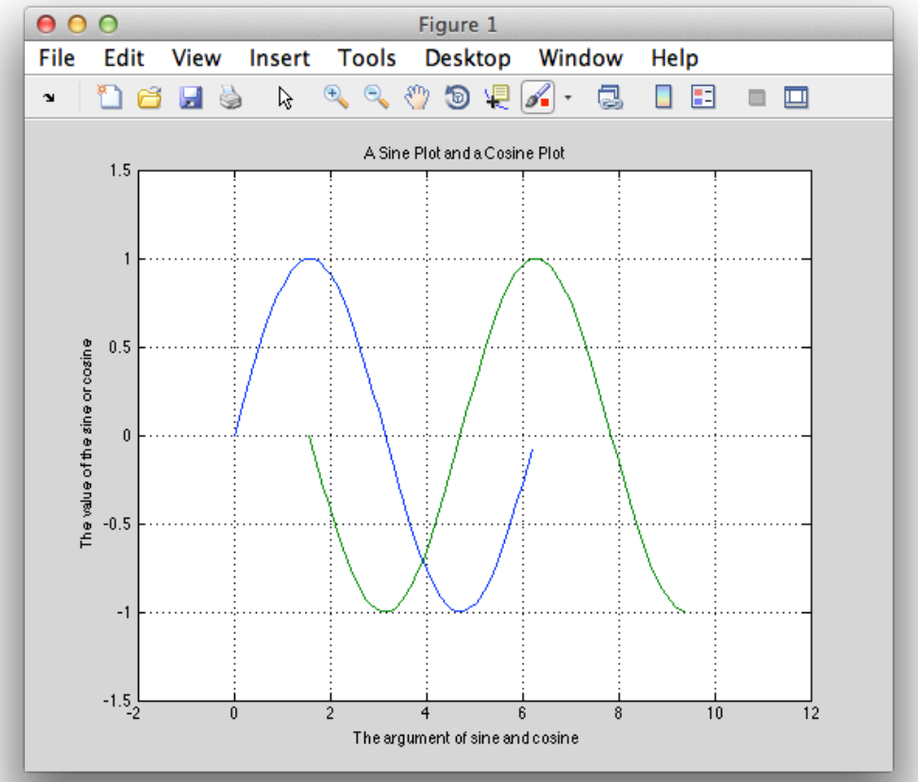
```
>> axis equal
```

will adjust the axes so that equal tick mark increments on the *x*-,*y*- and z-axis are equal in for x and y. To see other uses of the command use **help axis**.

Here is an example of the use of all these functions after a plot has been displayed (the result is shown in Figure 2.9):

```
>> grid
>> title('A Sine Plot and a Cosine Plot');
>> xlabel('The argument of sine and cosine');
>> ylabel('The value of the sine or cosine');
>> axis([-2 12 -1.5 1.5])
```

Figure 2.9 *The result of using the* ***title****,* ***xlabel****,* ***ylabel*** *and* ***axis*** *commands*



Playing Audio

MATLAB supports playing audio through your computer's speakers. The built-in function **sound** takes a vector as an input argument and considers it digitized sound assuming a sample rate provided as a second input argument. To illustrate how to use the function, the following function generates a one second long pure tone:

```
function play_tone(f)
Fs = 8192;
t = 0 : 1/Fs : 1;
tone = sin(f * 2 * pi * t);
sound(tone,Fs);
```

The function takes an input argument **f** that specifies the frequency of the desired tone. First, we set the sampling rate to a typical 8192 Hz. Then we generate a time series: the time between any two sound samples needs to be **1/Fs**. Next we create a pure tone of frequency **f**, that is, a simple sine wave. Finally, we call the sound function using the tone and the sampling rate. You can test the function by running it like this:

```
>> play_tone(147)
```

which is roughly the D below middle C on the piano (called D3, whose exact frequency is 146.832) which is a pretty low-pitched sound or (better turn down your speaker a bit for this next one one)

```
 > play_tone(2093)
```

which is the C that is three octaves above middle C on the piano (C7). This is a high-pitched sound.

The MATLAB distribution comes with a few saved sound files that we can replay using the sound function. Since these files are in the path, we can simply load them and play them. Try this:

```
>> load gong;
>> sound(y,Fs);
```

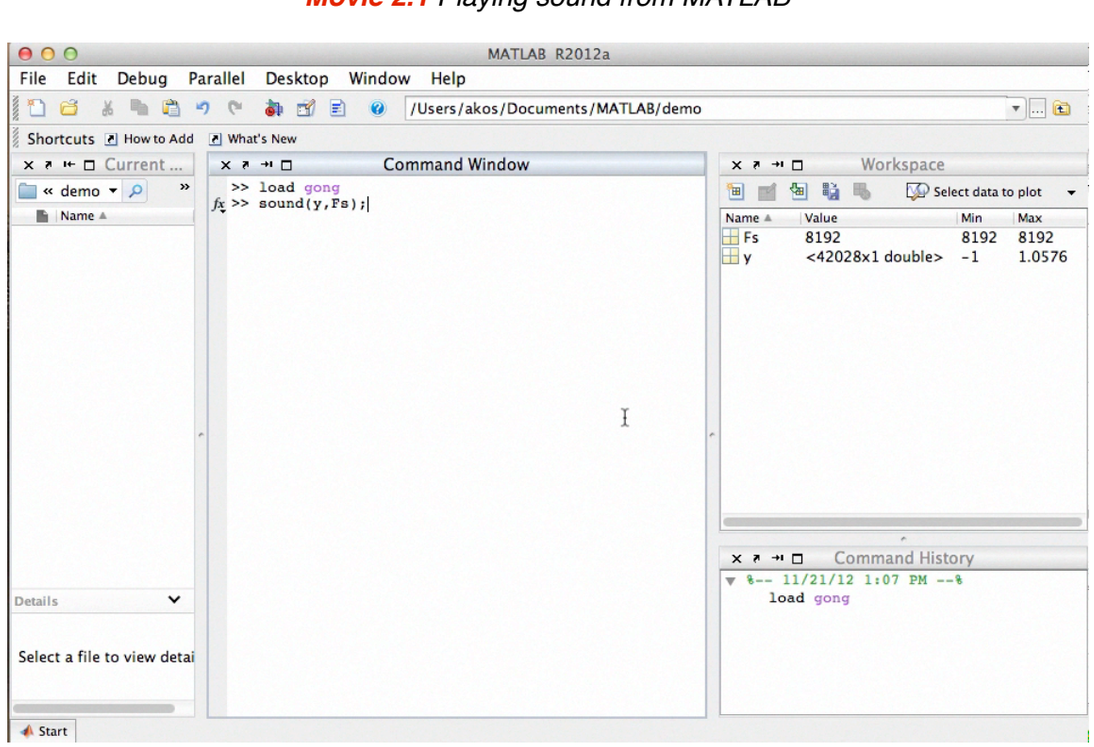
What is this **y** and **Fs**? Let's look at it with **whos**:

```
>> whos
  Name        Size            Bytes  Class    Attributes

  Fs          1x1                 8  double
  y       42028x1            336224  double
```

The **load** function, which we will learn more about in the File Input/Output section, has caused **y** and **Fs** to be loaded from a file called **gong.mat** that is provided with the MATLAB installation and placed in one of the folders on the MATLAB path, which was described in Introduction MATLAB in the subsection Path.

The following video clip (Movie 2.1) demonstrates this with an additional example.

Movie 2.1 *Playing sound from MATLAB*



Debugging

The sad fact of life is that it is very easy to make mistakes when writing computer programs. Many of these mistakes are syntactical. We saw examples of that when we introduced Syntax and Semantics in Introduction to MATLAB, where we learned that syntactical errors happen when the program text is not valid according to the rules of the programming language syntax. For example, if we type

```
>> 23 = x;
 23 = x;
   |
Error: The expression to the left of the equals sign is
not a valid target for an assignment.
```

It is not allowed in MATLAB to assign a value to another value! Values can be assigned only to variables. Another example is when we simply mistype a name:

```
>> ramd(2)
Undefined function 'ramd' for input arguments of type
'double'.
```

There is no function called **`ramd`** defined. We meant to type **`rand`** of course. The good news is that MATLAB will immediately complain when we try to execute a command using illegal syntax. The error message is typically very informative, and hence, the error is easy to fix.

The errors that are more difficult to find involve the logic in our program. These are errors of meaning, known more formally as errors of semantics, and known less formally as "bugs". A **bug** is simply an error in a program that causes it to behave incorrectly. The program text abides by all the syntactical rules of the language, but the program either produces an incorrect result or causes MATLAB to report a runtime error. The latter means that MATLAB tried to execute a command and it caused an error. For example, if we tried to access the sixth element of a vector of five, MATLAB will complain:

```
>> y = rand(1,5)

y =
    0.7577    0.7431    0.3922    0.6555    0.1712

>> y(6)
Index exceeds matrix dimensions.
```

When MATLAB provides such an error message, it makes it easier to find the error. When we simply get an incorrect result, that is typically the hardest type of bug to find.

The procedure of finding and correcting programming errors is called **debugging**. How can we go about doing it? Let's consider a simple example. Here is a function that was meant to create and return an **`n`**-by-**`m`** matrix of random integers and print out the last element as well:

```
function x = rand_int(n,m)

x = randi(n,m);
fprintf('The last element is %d\n',x(n,m));
```

If we run it like this, everything seems fine:

```
>> rand_int(3,3)
The last element is 3

ans =
     3     3     1
     3     2     2
     1     1     3
```

It is somewhat suspicious that the largest integer in the matrix is only 3, but let's set that aside for a moment. However, if we try the function with different arguments, we run into trouble:

```
>> rand_int(3,2)
Index exceeds matrix dimensions.

Error in rand_int (line 4)
fprintf('The last element is %d\n',x(n,m));
```

What is going on here?! In this simple example, we should be able to figure this out just by looking at the two lines of code, but let's say that we can't. What can we do? The simplest approach is to remove the semicolon in key lines of the function and thereby let MATLAB print out the corresponding values. In this case, let us remove the semicolon from the line **`x = randi(n,m)`** and see what happens:

```
>> rand_int(3,2)

x =

     1     3
     2     3

Index exceeds matrix dimensions.

Error in rand_int (line 4)
fprintf('The last element is %d\n',x(n,m));
```

To our surprise we see that the **`x`** is a 2-by-2 matrix as opposed to the 3-by-2 we expected. This should be enough information to realize that we are using the built-in MATLAB function **`randi`** incorrectly. We can type **`help randi`** to see that the first argument it expects is the maximum value of the random integers we wish to generate and not the number of rows of the matrix. The solution is to either add an extra argument to our own **`rand_int`** function to supply to **`randi`** or to simply have a fixed value like this:

```
x = randi(10,n,m);
```

Our new and improved function now works correctly:

```
>> rand_int(3,2)

x =
    10     2
     6     3
     2     9

The last element is 9

ans =
    10     2
     6     3
     2     9
```

It returns a matrix of the correct dimensions with positive integer elements up to 10. It also prints the last element as expected. Of course, we should run the function with many different argument combinations to verify that it indeed is correct. We should also remember to put the semicolon back to suppress printing.

The simple technique of removing semicolons from key lines in our functions does not always work. If the function creates large matrices, for example, we may get thousands of lines of numbers on the screen making finding the problem difficult. Fortunately, there is a better way.

MATLAB, just like most other programming environments, has a built-in tool called the **debugger** to assist in finding bugs. The debugger let's you stop your program in the midst of its execution, so you can look at variables, and it allows you to step through your code line-by-line. Let's see how it works!

The most important feature of any debugger is the ability to put breakpoints in your program. You use a breakpoint to mark a line in your code that you want the debugger to stop executing the code at. The following figures show our **`rand_int`** function (the erroneous version) in the MATLAB text editor with a breakpoint specified at the line

```
x = randi(n,m);
```

Figure 2.10 *Breakpoint in Edit window (R2012a)*

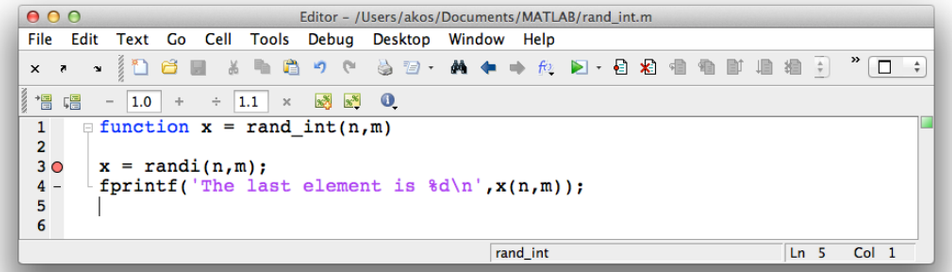
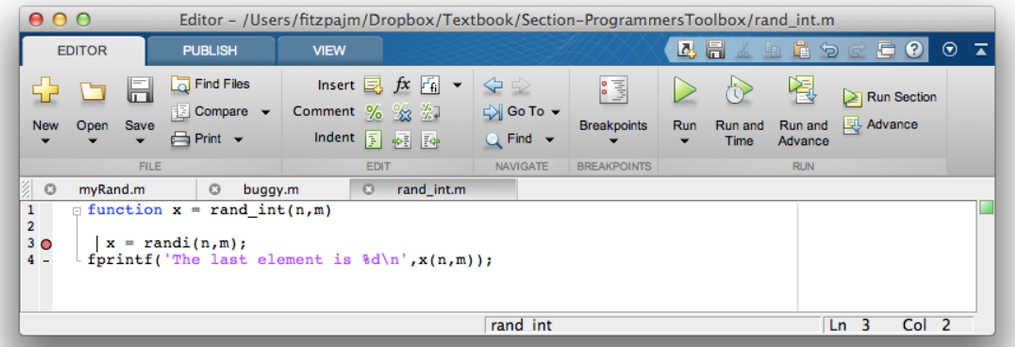


Figure 2.11 *Breakpoint in Edit window (R2012b)*



Figures Figure 2.10 and Figure 2.11 show how MATLAB shows the location of a break point in the Edit window in version R2012a (and earlier versions) and in version R2012b.

The red circle indicates the location of the breakpoint. To create a breakpoint, you simply click in the gray area between the line number and the given line. To remove a breakpoint, simply click on the red circle. Note that if you modify a function, all breakpoints turn gray. This is MATLAB's way of indicating that the breakpoints are inactive and you need to save you function first before running it.

If we run the function, we will see the following in the Command Window:

```
>> rand_int(3,2)
3    x = randi(n,m);
K>>
```

The letter `K` in the prompt indicates that execution is being controlled by the debugger and is stopped. The 3 indicates that the next line to execute will be Line 3, and the command on that line is shown as well. At the same time as this prompt appears in the Command Window, we'll simultaneously see the text editor pop up: In Figure 2.12, and Figure 2.13 the tiny green arrow just to the right of the red circle on Line 3 indicates that execution has stopped at the third line because there is a breakpoint there. To be precise, it stopped before executing the command that the green arrow is pointing at.

Figure 2.12 *Execution stopped at the breakpoint (R2012a)*

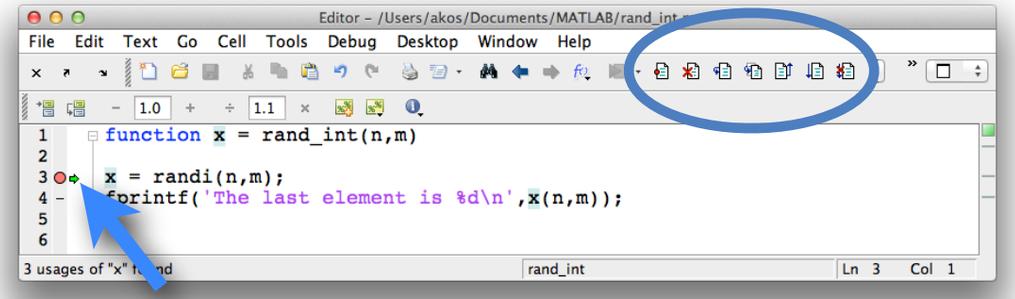
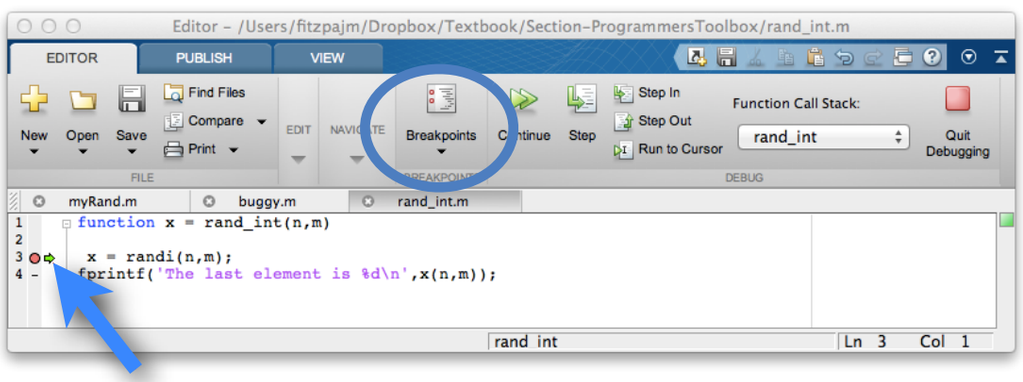
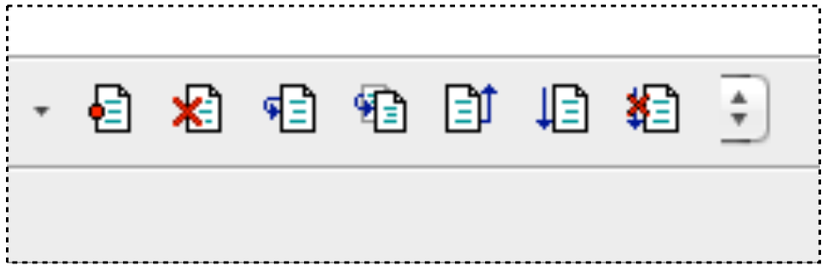


Figure 2.13 *Execution stopped at the breakpoint (R2012b)*



Let's look at the circled buttons in the toolbar of Figure 2.12, which we have magnified in Figure 2.14 .

Figure 2.14 *Magnified view part of Edit window (R2012a)*

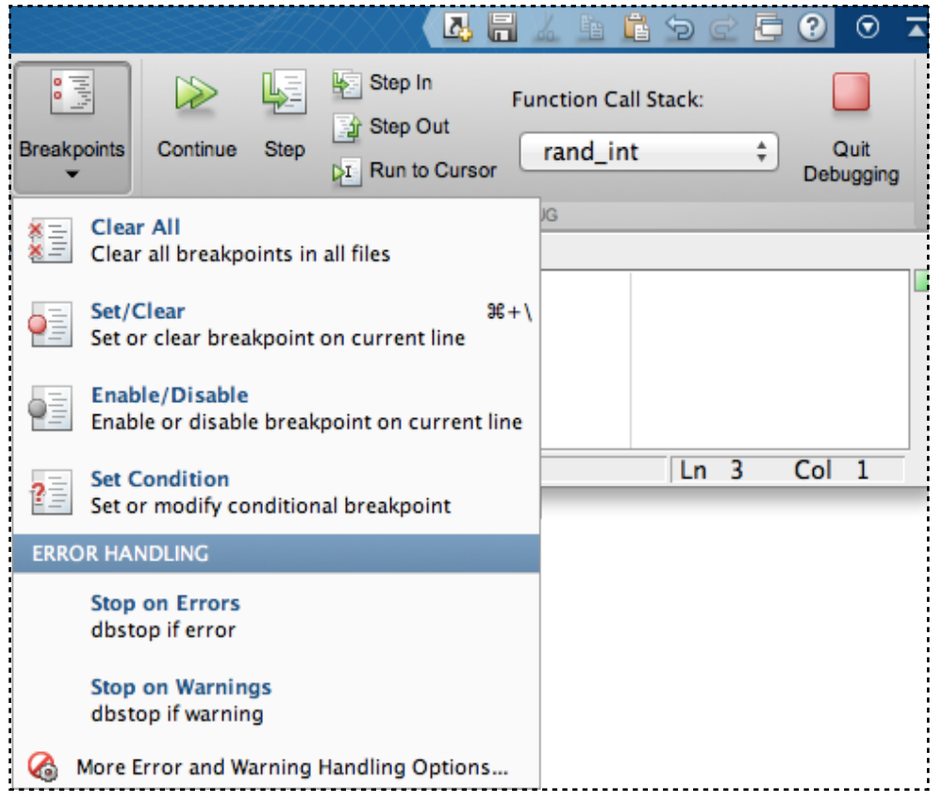


If you allow the mouse cursor to hover on top of the buttons in version R2012a one at a time, a little yellow "tip" box will pop up showing the action of each given button. They are:

1. **Set/clear breakpoint**: toggles the breakpoint at the current line.
2. **Clear breakpoints in all files**: removes all breakpoints.
3. **Step**: executes the command in the current line.
4. **Step in**: if the current line has a call to a user-defined function, it calls the function and stops on its first line.
5. **Step out**: leaves the current function and stops at next line after the function was called (or returns to the Command Window if the function was called directly from there).
6. **Continue**: continues the execution of the function until the next breakpoint is encountered or until completion if no more breakpoints are encountered.
7. **Exit debug mode**: stops the debugger and does not finish the current function.

The same actions are available in version R2012b, but are reached in a somewhat different way. We click the "Breakpoints" button, which is circled in Figure 2.13. We show the result in a magnified the view in Figure 2.15.

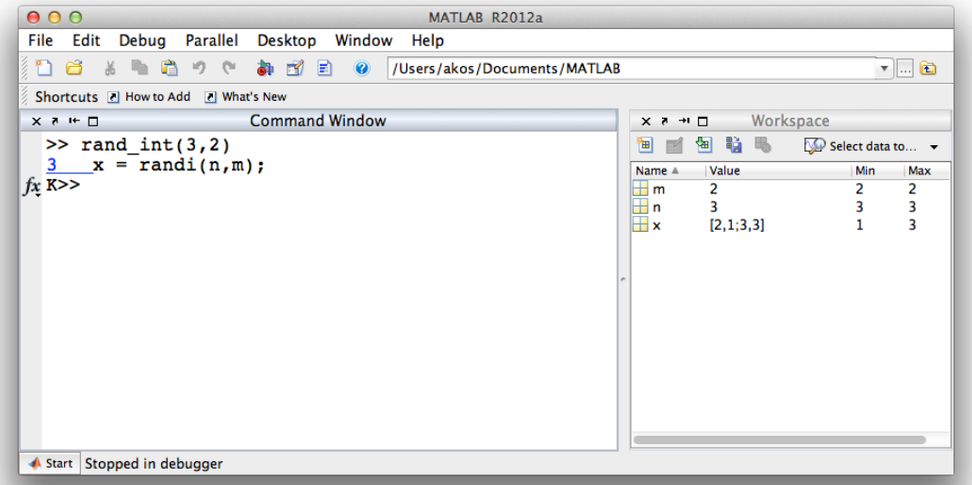
Figure 2.15 *Magnified view of part of Edit window (R2012b)*



Actions 1 and 2 along with others are available by clicking icons in the pop-down menu. Actions 4 through 6 are available via icons with the green arrows in the gray "ribbon", and Action 7 is achieved by clicking the red square with the label "Quit Debugging".

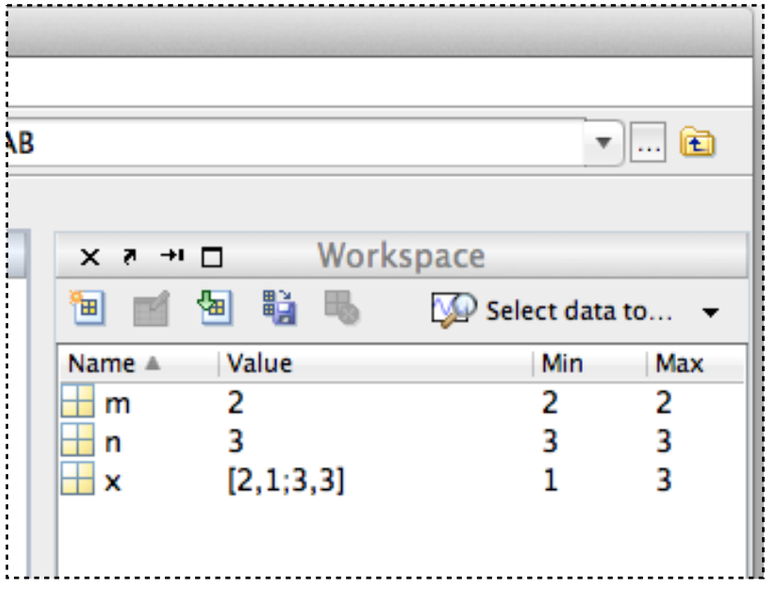
To continue debugging our `rand_int` function, let's press the "Step" button. The little green arrow will now indicate that we have stepped to Line 4 (not shown). Take a look at the MATLAB Desktop at this point, shown in Figure 2.16

Figure 2.16 Command Window in debug mode



(Note: We have closed the Current Folder and History windows). Look closely at the Workspace window, which we have magnified in Figure 2.17.

Figure 2.17 Magnified view of Workspace window



The Workspace window shows something very interesting. The variables listed there are the local variables of our function, and this is the workspace of the function in which execution has been stopped! This is the default workspace that is shown whenever execution is stopped at a break point, and being able to see that workspace with the function's local variables is extremely useful in debugging. From the Value column we can see that the variable **x** is a 2-by-2 matrix, which is not what we expected.

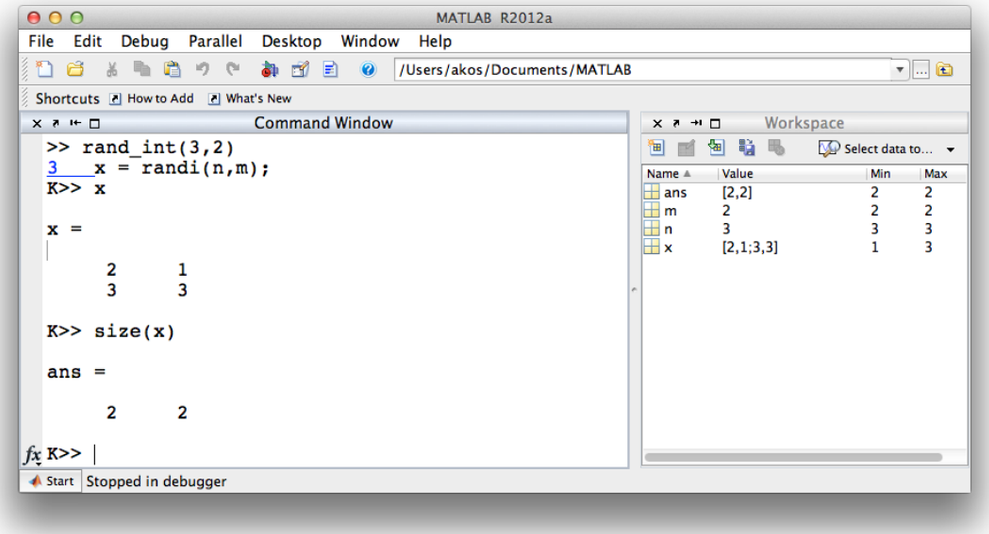
The Command Window is interesting too. We can type commands in the Command Window while we are in debug mode. For example, we can type **whos**:

```
K>> whos
  Name      Size            Bytes  Class     Attributes

  m         1x1                 8  double
  n         1x1                 8  double
  x         2x2                32  double
```

which shows the same variables in the workspace as are shown in the Workspace window, as it must. We can, in fact, type any commands we wish. We can call any function we wish, we can assign values to any variables we wish, we can plot functions, display images—anything! The only difference from the normal situation we have seen before when we are issuing commands in the Command Window is that the workspace now belongs to the active function **rand_int**, and when **rand_int** returns, all the variables in it will vanish!

For this debugging session we might want to print the values of **m**, **n**, or **x**, but the most useful thing to do is probably to use the command **size** to check the dimensions of the array stored in the variable **x**, which shows more clearly that **x** is a 2-by-2 array, as shown in Figure 2.18.

Figure 2.18 *Inspecting variables of the function from the Command Window*



At this point, we should be able to solve the simple problem in our simple function. Once we have done that, we can quit debugging. For more complicated cases, we can continue stepping through the function line by line or hit the "Continue" button. When we want to really finish debugging, we can click the "Exit debug mode" button in version R2012a or the Stop Debugging button in version R2012b. Alternatively, we could type **`dbquit`** in the Command Window (Make sure not to type **`quit`** because it will exit MATLAB altogether!). Once we exit the debugger, the workspace will show the variables at the Command Window level as expected. Now we can clear all the breakpoints, and we are now ready to run the function normally.

From calculator to computer

With this introduction to the Debugger, you have seen the most important tool in the Programmers Toolbox. It is the one we use to correct the errors that we we will inevitably make with the other tools in the box. So far, the tools we have examined are the ones that make MATLAB a very versatile calculator. In the next two sections, we will look at the tools that make MATLAB a true programming language, a language that allows you to write code that can make a computer act like a computer. It's time to get out the power tools.

Additional Online Resources

- Video lectures by the authors:

  Lesson 4.1 Introduction to programmer's Toolbox (7:06)

  Lesson 4.2 Matrix Building (15:11)

  Lesson 4.3 Input / Output (20:47)

  Lesson 4.4 Plotting (17:47)

  Lesson 4.5 Debugging (22:17)

Concepts From This Section

Computer Science and Mathematics:

- library
- bug
- debugger
- breakpoint
- step
- step in
- step out

MATLAB:

- important functions
  - building matrices
  - trigonometric
  - exponential
  - random number generation
  - complex number analysis
  - rounding and remainder
  - descriptive
- **input**
- Formatted output: **fprintf**
  - format string
  - conversion characters
  - escape characters
- **disp**
- **plot**
  - multiple functions on one plot
  - selecting colors and markers
  - titles, labels, log plotting

Practice Problems

Problem 1. Write a function called **`minmax`** that takes a two-dimensional array (matrix) as an input argument (you do not have to check the input argument) and returns the minimum and the maximum element in the matrix. It also needs to print out these values to two decimal point precision according to this example run:

```
>> [x y] = minmax(randn(20,20))
The minimum of the matrix is -3.00
The maximum of the matrix is 2.71

x =
   -2.9962
y =
    2.7081
```

?

Problem 2. Write a function called **`my_size`** that takes a two-dimensional array (matrix) as an input argument (you do not have to check the input argument) and returns the size of the matrix just as the built-in **`size`** function would do. However, **`my_size`** also prints out the dimensions of the matrix as illustrated by this run:

```
>> s = my_size(randi(10,20,30))
This is a 20-by-30 matrix
s =
    20    30
```

Problem 3. Write a function called **`print_square`** that does not take any input arguments, nor does it return any output arguments. Instead it requests the user to input a number and then it proceeds to print it and its square according to this:

```
>> print_square
Give me a number: 5
The square of 5 is 25
```

?

Problem 4. Write a function called **`print_product`** that does not take any input arguments, nor does it return any output arguments. Instead it requests the user to input two numbers one by one and then it proceeds to print them and their product according to this:

```
>> print_product
Give me a number: 4
Give me another number: 7
The product of 4 and 7 is 28
```

Problem 5. Write a function called **`rand_test`** that takes one scalar positive integer input argument **`n`** (you do not have to check the input argument) and returns two output arguments: a column vector of $\mathbf{n}^2$ elements and an **`n`**-by-**`n`** matrix. The two output arguments must contains the exact same set of random numbers (use **`rand`**). Here is a sample run:

```
>> [m v] = rand_test(2)
m =
    0.4170
    0.7203
    0.0001
    0.3023
v =
    0.4170    0.0001
    0.7203    0.3023
```

?

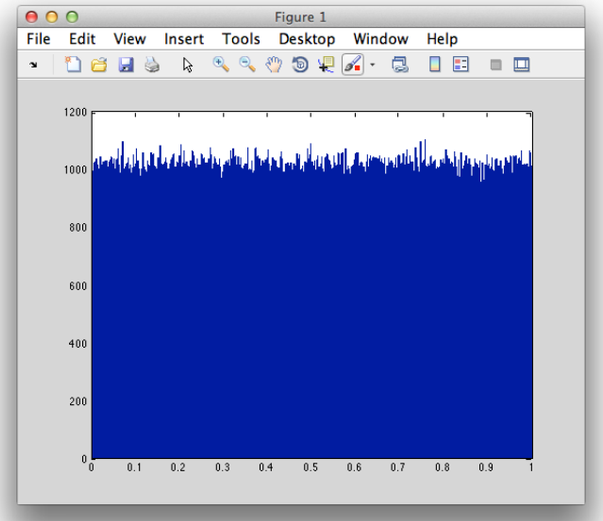
Problem 6. Write a function called **`randi_test`** that takes two scalar positive integer input arguments **`maxi`** and **`n`** (you do not have to check the input arguments) and returns two output arguments: a row vector of $\mathbf{n}^2$ elements and an **`n`**-by-**`n`** matrix. The two output arguments must contains the exact same set of random integers that fall between **`1`** and **`maxi`**. Here is a sample run:

```
>> [m v] = randi_test(5,3)

m =
     3     4     1     2     1     1     1     2     2
v =
     3     2     1
     4     1     2
     1     1     2
```

Problem 7. Write a function called **`uniform_hist`** that takes two scalar positive integer input arguments **`n`** and **`bins`** (you do not have to check the input arguments). The function does not have any output arguments. It needs to generate **`n`** random numbers using **`rand`** and plot their histogram using **`bins`** intervals. For example, the following run generates Figure 2.19 below (yours may not be exactly the same since the random number generator may have been initialized differently):

```
>> uniform_hist(1e6,1000);
```

Figure 2.19 Problem 7

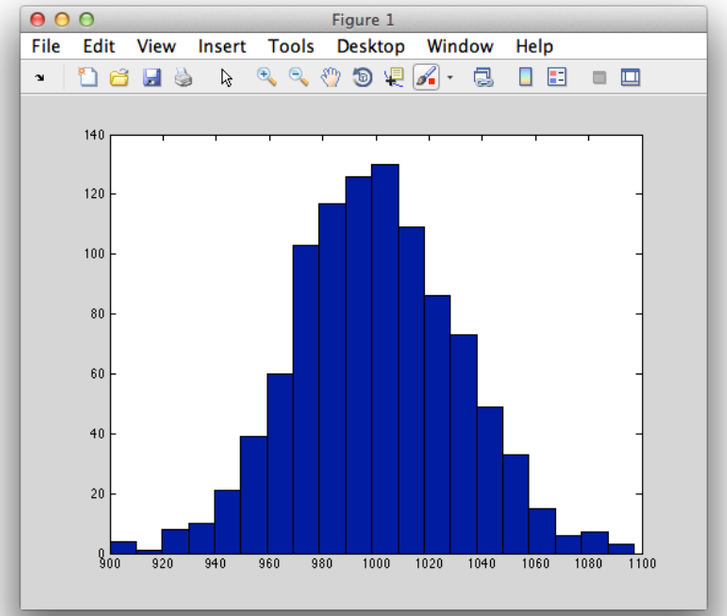


?

Problem 8. Write a function called **`uniform_test`** that takes two scalar positive integer input arguments **`n`** and **`bins`** (you do not have to check the input arguments). The function does not have any output arguments. It needs to generate **`n`** random numbers using **`rand`** and save their histogram using **`bins`** intervals in a variable. Note that the **`hist`** function can provide the histogram in a vector as the output argument. It does not plot the histogram when its output is assigned to a variable. Plot the histogram of this vector using **`bins/50`** intervals. This will show how uniformly the numbers generated **`rand`** are distributed. For example, the following run generates Figure 2.20 below (yours may not be exactly the same since the random number generator may have been initialized differently):

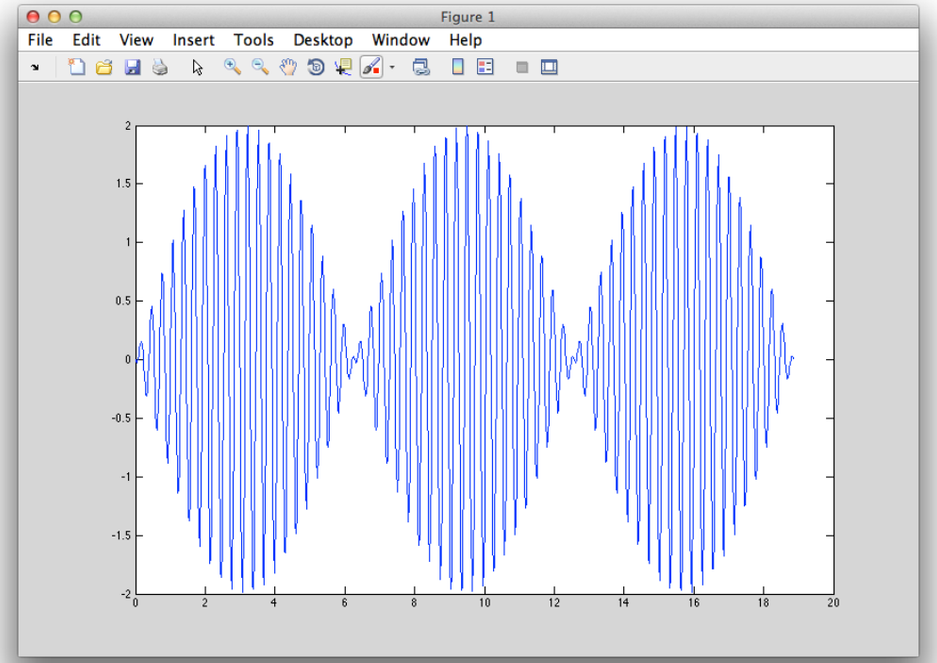
```
>> uniform_test(1e6,1000);
```

Figure 2.20 *Problem 8*



Problem 9. Write a function called **`mix_sines`** that takes two positive scalar input arguments **`f1`** and **`f2`** (you do not have to check the input arguments) that represent the frequency of two sines waves. The function needs to generate these sines waves, add them together and plot the result. If the function is called like this: **`mix_sines(1,0)`** then it displays three full periods of a regular sine wave with amplitude 1. If it is called like this: **`mix_sines(20,21)`**, then it plots Figure 2.21 below (note that if you add a phase shift of **`pi`** to one of the sines, then the result will start at 0 as shown below). Notice that the amplitude is 2. Also, notice that signal has three times 21 periods, but the envelope signal has three times 1 periods. That is because the difference of the two frequencies (21 and 20) is 1.

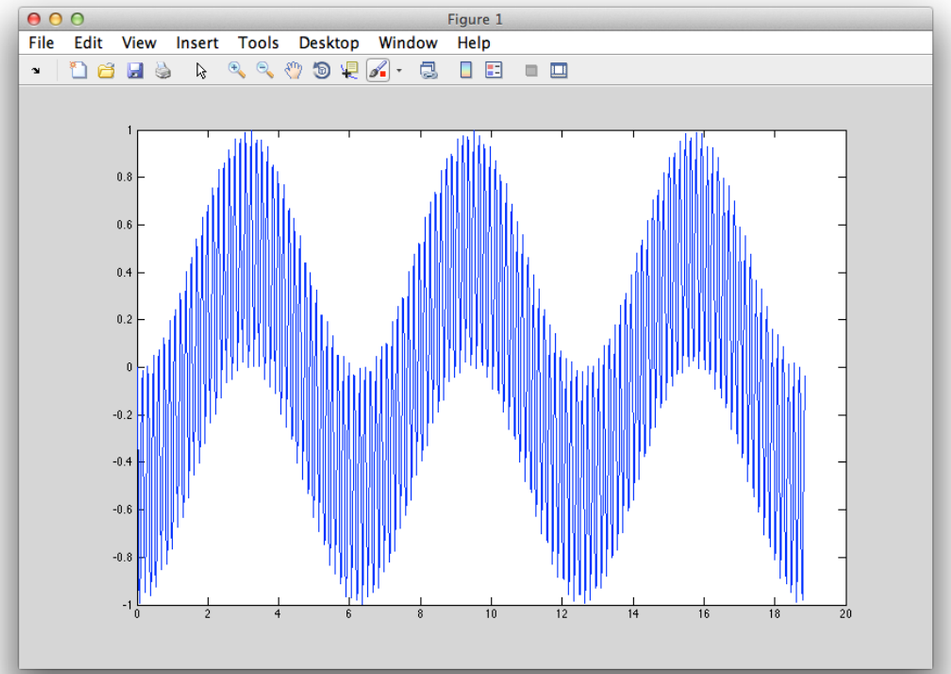
Figure 2.21 *Problem 9*



?

Problem 10. Write a function called **`mul_sines`** that takes two positive scalar input arguments **`f1`** and **`f2`** (you do not have to check the input arguments) that represent the frequency of two sines waves. The function needs to generate three full periods of these sines waves, multiply them together and plot the result. If the function is called like this: **`mul_sines(20,21)`**, then it plots Figure 2.22 (note that if you add a phase shift of **`pi`** to one of the sines, then the result will start at **`-1`** as shown below). Notice that signal has three times **`21+20 = 41`** periods. Also, notice that the envelope signal has three times **`1`** periods. That is because the difference of the two frequencies (**`21`** and **`20`**) is **`1`**.

Figure 2.22 *Problem 10*



Selection

Objectives

Selection is means by which MATLAB makes decisions.

(1) We will learn how to use MATLAB's two versions of selection—the if-statement and the switch-statement.

(2) We will learn how to nest selection statements, which is the placement of if-statements and switch-statements inside each other.

(3) We will learn the relational operators, **==**, **~=**, **>**, **<**, **>=**, and **<=**, and the logical "and", "or", and "not" operators.

(4) We will see the complete precedence table for all MATLAB operators.

Choosing the path to follow through a MATLAB program is called "selection".

So far, every piece of MATLAB code that we have encountered, whether it has been in the Command Window or inside a function, has comprised a sequence of commands, each one of which has been executed immediately after the command that was written immediately before it. This flow of control is handled behind the scenes by the MATLAB **interpreter**, which is a program running in the background that reads the statements that you write and carries them out one by one, allocating space for variables, writing values into those variables, and reading values from them, accessing elements of arrays, calling functions, and displaying results on the screen. All this behind the scenes work is necessary to the execution of statements, and the interpreter's normal approach to the flow of control is to do this work in a

way that results in its executing the statements that you give it in the order that you give them.

Executing the statements in the order that they were written by the programmer is called **sequential control**. Sequential control is the most natural and the most common sequence that occurs in any program written in any programming language, and it is the primary example of a **control construct**. A control construct is simply a method by which the interpreter selects the next statement to be executed after the execution of the current statement has concluded. The programmer tells the interpreter which construct to use by means of the presence or absence of special keywords. So far, such keywords have been absent from our code, and as a result we have been utilizing only one type of control construct—sequential control. In this section we will introduce keywords that signal the interpreter to base its decision as to which statement is to be executed next not only on the order in which the statements are written but also on the basis of the values of expressions. This new control construct is called **selection**, or alternatively, **branching**.

if-statements

The most common selection construct is the **if-statement**. An if-statement is used when the programmer wishes to have the interpreter choose whether or not to executed a statement or set of statements on the basis of the values of variables. We illustrate this idea with a very simple example. Suppose we write a function called **`guess_my_number`** that requires one input and congratulates us when we call it with the "right" input. Here are some example runs:

```
>> guess_my_number(7)
>> guess_my_number(13)
>> guess_my_number(0)
>> guess_my_number(2)
Congrats! You guessed my number.
```

Here is the function:

```
function guess_my_number(x)
if x == 2
   fprintf('Congrats! You guessed my number.\n');
end
```

We have introduced a new keyword here—**if**—and a new operator—the double equals, **==**. We will take up the new keyword first. It indicates that we are using an if-statement, which is the simplest selection construct. Like every selection construct, it begins with a **control statement**, which in this case is the statement, **`if x == 2`**, and ends with the statement, **`end`**. A control statement is a statement that controls the execution of another statement or a set of statements. In this case, it controls the execution of just one statement—the **`fprintf`** function call statement. The meaning should be clear from the code itself: If the value of the variable **`x`** is equal to 2, then the **`fprintf`** statement is executed; otherwise, it is not.

The new operator is symbolized by a sequence of two equal signs (with no space allowed between them), and it means "is equal to". This operator will be examined more closely later in this section, in subsection Relational Operators, but for now it is enough to know that (a) it is a binary operator, which means, as we may recall from Matrices and Operators, that it takes two operands and (b) that it determines whether its first operand, which in this case is the variable **`x`**, is equal to its second operand, which in this case is the number 2.

The if-statement is depicted schematically in Figure 2.23. The black dots represent statements that precede and follow the if-statement. The lines show the possible flow of control from one statement to another with arrows to show the direction. The statement, `if x == 2`, is located at the dot labeled "control statement". The left branch is labeled `x == 2`, meaning that, if `x` is equal to 2, then the flow follows this branch. A **block** is a set of contiguous statements, and each block in the figure is controlled by a control statement. In the example code that we showed above, there is just one statement (the `fprintf` function call statement), but in general there can be any number of statements in a block. In Figure 2.23, (a) shows the layout, while (b) shows the flow of control in red when `x` is equal to 2, and (c) shows the flow of control in red for all other values of `x`.

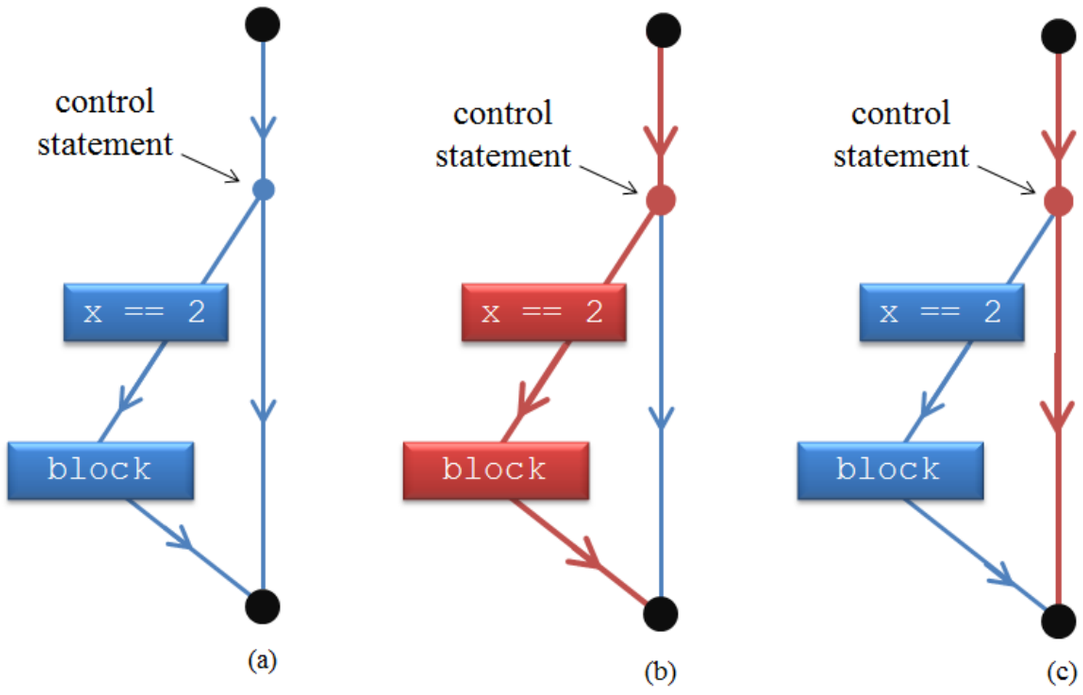


Figure 2.23 *Schematic of if-statement. The flow of control of an if-statement is shown. The black dots represent code before and after the if-statement. Control flows along the lines in the direction of the arrows. The control statement (labeled) is* `if x == 2`*. There are two possible paths as shown in (a) : one labeled* `x == 2`*, in which a block of statements, labeled "block", is executed, and one for every other possibility (the unlabeled vertical line) in which no statements are executed. If* `x` *equals 2, then the left branch is followed and the block of statements is executed, as shown by the red path in (b). Otherwise, the vertical path is followed, as shown in red in (c), and no statements are executed.*

Let's improve our function just a bit by trying to cheer up the poor user who fails to pick the correct number:

```
>> guess_my_number(7)
Not right, but a good guess.
>> guess_my_number(13)
Not right, but a good guess.
>> guess_my_number(0)
Not right, but a good guess.
>> guess_my_number(2)
Congrats! You guessed my number.
```

Here is what the improved function looks like:

```
function guess_my_number(x)
if x == 2
   fprintf('Congrats! You guessed my number!\n');
else
   fprintf('Not right, but a good guess.\n');
end
```

We have used another selection construct—the **if-else-statement**—and to do that we've introduced another keyword—**else**. This statement selects one of two statements to be executed but, as in the case of the if-statement above, each of these statements could be replaced by a block of two or more statements. We depict our if-else statement schematically in Figure 2.24.

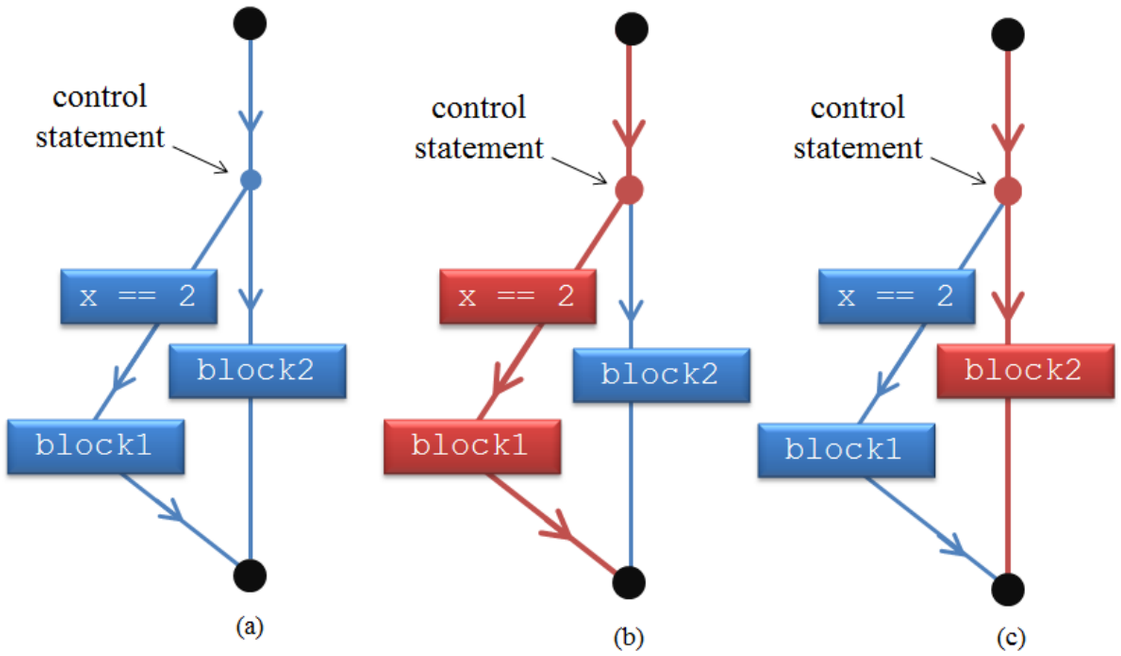


Figure 2.24 *Schematic of if-else statement. The flow of control of an if-else statement is shown. This figure is similar to Figure 2.23, except that there are two blocks of code, instead of one. There are two possible paths as shown in (a): one for* **`x == 2`**, *and one for every other possibility. If* **`x`** *equals 2, then the left branch is followed and* **`block1`** *is executed, as shown by the red path in (b). Otherwise, the vertical path is followed and* **`block2`** *is executed, as shown in red in (c).*

Let's improve our example even more. Let's change the name of the function to **`ultimate_question`** and pick a different answer, say 42. That answer is so obscure that we ought to give the user a hint to make it easier to find it. Here is how we want our improved function to behave:

```
>> ultimate_question(7)
Too small. Try again.
>> ultimate_question (13)
Too small. Try again.
>> ultimate_question(100)
Too big. Try again.
>> ultimate_question(35)
Too small. Try again.
>> ultimate_question(50)
Too big. Try again.
>> ultimate_question(40)
Too small. Try again.
>> ultimate_question(45)
Too big. Try again.
>> ultimate_question(42)
Wow! You answered the ultimate question.
```

It took a bit longer, but it is a lot more fun! Well, maybe it's not all that much fun, but it gives us a chance to sneak in a new keyword. Here is the new improved function:

```
function ultimate_question(x)
if x == 42
   fprintf('Wow! You answered the ultimate question.\n');
elseif x < 42
   fprintf('Too small. Try again.\n');
else
   fprintf('Too big. Try again.\n');
end
```

The new keyword is **elseif** . It is used in a new construct–the **if-elseif-else-statement**, and it allows us to check a second condition. In addition to the previous condition, whether or not **`x`** equals 42, we have chosen to additionally check to see whether the value of **`x`** is less than 42. If it is, then the second **`fprintf`** statement is executed; if not, then the third one is executed. This construct is depicted schematically in Figure 2.25.

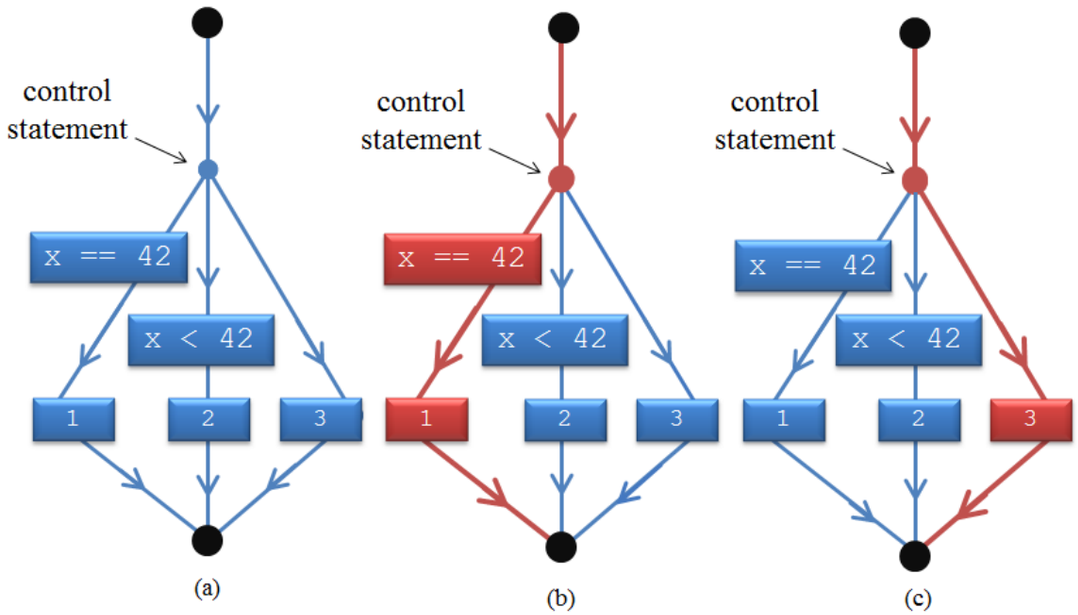


Figure 2.25 *Schematic of if-elseif-else The flow of control of an if-elseif-else-statement is shown. This figure is similar to the previous figure except that there are three blocks of code (labeled 1, 2, and 3), instead of two. There are three possible paths as shown in (a): one for* ***x == 42****, one for* ***x < 42****, and one for every other possibility. If* ***x*** *equals 42, then the left branch is followed and block 1 is executed, as shown by the red path in (b). If* ***x*** *neither equals 42, nor is less than 42 (i.e.,* ***x*** *is greater than 42), then block 3 is executed, as shown in (c). The red path for* ***x < 42*** *is not shown.*

There is one more construct that starts with the keyword **`if`**, the **if-elseif-statement**. It is simply an if-elseif-else statement with the else branch removed. So, for example, if we wanted to require a bit more thinking on the part of the person trying to answer the ultimate question, we could omit the else clause from our last example to get a harder **`ultimate_question`** as follows:

```
function ultimate_question(x)
if x == 42
   fprintf('Wow! You answered the ultimate question.\n');
elseif x < 42
   fprintf('Too small. Try again.\n');
end
```

Now the user who tries to answer the ultimate question would need to realize, that, if the function does not tell us that the answer is too small and doesn't tell us that the answer is correct, then the only remaining possibility is that the number is too big. Here is how it would behave with the same input:

```
>> ultimate_question(7)
Too small. Try again.
>> ultimate_question (13)
Too small. Try again.
>> ultimate_question(100)
>> ultimate_question(35)
Too small. Try again.
>> ultimate_question(50)
>> ultimate_question(40)
Too small. Try again.
>> ultimate_question(45)
>> ultimate_question(42)
Wow! You answered the ultimate question.
```

Note that nothing is printed for the inputs 100, 50, and 45, because none of them is equal to or less than 42.

So far, our examples have selected from among one, two, or three blocks of code, but any number of blocks can be included by including additional elseif keywords. Here is a function called **`day_of_week`**, that contains an if-elseif-else-statement that selects from among seven blocks. It performs the following simple task: Given the number of the day of the week, print the name of the day and identify it as either a week day or a weekend day:

```
function day_of_week(n)
if n == 1
   fprintf('Sunday,');
   day_type = 2;
elseif n == 2
   fprintf('Monday,');
   day_type = 1;
elseif n == 3
   fprintf('Tuesday,');
   day_type = 1;
elseif n == 4
   fprintf('Wednesday,');
   day_type = 1;
elseif n == 5
   fprintf('Thursday,');
   day_type = 1;
elseif n == 6
   fprintf('Friday,');
   day_type = 1;
elseif n == 7
   fprintf('Saturday,');
   day_type = 2;
else
   fprintf('Number must be from 1 to 7.\n');
   return
end
if day_type == 1
   fprintf(' which is a week day\n');
else
   fprintf(' which is a weekend day\n');
end
```

Note also that the blocks in the if-elseif-else-statement above comprise two statements—an **fprintf** statement and an assignment statement, illustrating that a block can indeed include more than one statement.

The return-statement

We have sneaked in a new keyword in the code above. Did you notice it? The new keyword is **return**. This keyword comprises MATLAB's **return-statement**. When a return-statement is executed (in any programming language), it halts the function in which it appears, in this case **day_of_week**, and returns control to the caller of the function. (When **return** is executed in the Command Window, it does nothing; in a script, it causes the script to halt, and control is returned to the Command Window) If **day_of_week** was called from inside another function, then control will return to the calling function. If **day_of_week** was called from the Command Window, then control will return to the Command Window. When a return-statement is executed, output arguments (there are none in **day_of_week**) behave as usual: The most recent value assigned to each before the return-statement is executed will be passed to the caller, just as it is when the function ends in the normal way by executing its last statement.

Let's look at examples of this function in action.

```
>> day_of_week(1)
Sunday, which is a weekend day
>> day_of_week(4)
Wednesday, which is a week day
>> day_of_week(-2)
Number must be from 1 to 7.
```

The inputs **1** and **4** are legitimate inputs and result in the expected behavior. The input **-2** is caught by the else-clause of the if-elseif-else-statement, which executes its **fprintf** with the error message and then executes the return-statement, which halts the function and returns control to the Command Window. If that return-statement had been omitted, then execution would have continued to the final if-else-statement, and this is what would have happened:

```
>> day_of_week(-2)
Number must be from 1 to 7.
Undefined function or variable "day_type".
Error in day_of_week (line 27)
if day_type == 1
```

This ugly result happens because **day_type** is not assigned a value when the input is out of range, so, when the if-statement at the end of the code tries to compare its value against 1, it discovers that **day_type** has no value, and MATLAB is forced to step in and take charge—an embarrassment for any

self-respecting programmer. The return-statement is a convenient way to handle this situation, and indeed it is a convenient way to handle any situation for which there is nothing more that a function can do or needs to do.

The conditional

There is a name for the expression that follows the keywords **`if`** and **`elseif`**. It is called a **conditional**. A conditional is the expression that determines whether or not a block in an if-statement is executed. It can have one of two values—true or false. If it is true, the block is executed; if it is false, the block is not executed. We have seen several simple examples already: **`x == 2`**, **`x == 42`**, **`x < 42`**, **`n == 1`**, **`n == 2`**, ..., and **`day_type == 1`**. We will see more complex examples below in the subsections, Relational Operators and Logical Operators, and we will learn that they play an important role in another construct, the "while-loop", which we will learn about in the Loops section of this chapter.

If-statement summary

We have introduced these four selection constructs:

- if-statement
- if-else-statement
- if-elseif-statement
- if-elseif-else-statement

We refer to these constructs collectively as "if-statements", with the additional branches in the last three referred to as "else-clauses" or elseif-clauses". We might for example say, "The if-statement had an **elseif-clause**, but it did not have an **else-clause**." Or, "The error message was printed by the else-clause," etc.

Here is a summary of the syntax for these four selection constructs:

if-statement:

```
if conditional
   block
end
```

if-else-statement:

```
if conditional
   block
else
   block
end
```

if-elseif-statement:

```
if conditional
   block
elseif conditional
   block
end
```

if-elseif-else statement:

```
if conditional
   block
elseif conditional
   block
else
   block
end
```

Each clause of an if-statement must begin on a new line or be separated from the previous section by a comma or semicolon. The semicolon suppresses printing as usual. Additional elseif-clauses may be inserted without limit, but there can be only one **`if`** and one **`else`** per if-statement with **`if`** coming first and **`else`**, if there is one, coming last.

Note that only one of the blocks of any of these statements will be executed. This is a rule for if-statements of all forms: *When an if-statement is encountered, no more than one block of statements within that if-statement will be executed*. Either nothing will be executed, as when we guessed a number other than 2 in our very first example above, or exactly one of the blocks will be executed, and it will be executed one time. If the first conditional is true, then the first block will be executed once. If there are elseif-clauses and the first conditional is false then the block associated with the first true conditional among the elseif-clauses will be executed once. If there is no else-clause and none of the conditionals is true, all of the blocks will be skipped and the if-statement will do nothing. If there is an else clause, however, it is guaranteed that one block will be executed.

switch-statements

In addition to the many forms of the if-statement, MATLAB provides one additional selection construct—the **switch-statement**. We will introduce the switch-statement by using it to performing a task that we performed above with if-statements. Let's revisit the function from the previous section: **`day_of_week`**, which names the day of the week corresponding to the number we input to it and tells us whether it is a weekday or a weekend day. Here is the switch-statement version:

```
function day_of_week_switch(n)
switch n
   case 1
      fprintf('Sunday,');
      day_type = 2;
   case 2
      fprintf('Monday,');
      day_type = 1;
   case 3
      fprintf('Tuesday,');
      day_type = 1;
   case 4
      fprintf('Wednesday,');
      day_type = 1;
   case 5
      fprintf('Thursday,');
      day_type = 1;
   case 6
      fprintf('Friday,');
      day_type = 1;
   case 7
      fprintf('Saturday,');
      day_type = 2;
   otherwise
      fprintf('Number must be from 1 to 7.\n');
      return
end
if day_type == 1
   fprintf(' which is a week day\n');
else
   fprintf(' which is a weekend day\n');
end
```

There are three new keywords in this switch-statement: **switch**, **case**, and **otherwise**. The keyword, **`switch`**, signifies the beginning of a switch-statement, and that keyword is followed on the same line by the variable **`n`**. Following the **`switch`** line is a series of case-clauses, each beginning with the keyword, **`case`**, followed on the same line by a number. For each case-clause there follows a block of statements, in this case, an **`fprintf`** function call statement and an assignment statement. The **`switch`** line is the control statement. It determines which block of statements will be executed by com-

paring **n** to the number that follows each **case** keyword. The flow of the switch-statement is just like that of the if-elseif-else-statement. The first case whose number matches the value of **n** will have its block of statements executed. When a switch-statement is encountered, no more than one block of statements within that switch-statement will be executed. Either nothing will be executed, or exactly one of the blocks will be executed. In the code above, if none of the numbers after the case keywords matches **n**, then the block of statements following the **otherwise** keyword will be executed. The otherwise-clause may, however, be omitted. If there is no otherwise-clause, then, if none of the numbers matches **n**, then none of the blocks in the switch-statement would be executed. If there is an otherwise-clause, however, it is guaranteed that one block will be executed (just as when there is an else-clause in an if-statement). Regardless of whether or not there is an otherwise-clause, the switch-statement must be be terminated with the keyword **end**.

The switch-statement is actually more flexible than our example might indicate. There are in fact three additional options. First, a simple expression, such as **n+2** or a more complicated one, such as **sqrt(n^3 - 1/n + pi)**, can replace the simple variable **n**. Furthermore, the number following each case keyword can also be an expression. The following summary of the switch-statement syntax shows that expressions can appear in each of these positions.

```
switch switch-expression
case case-expression,
    block
case case-expression,
    block
. . .
otherwise
    block
end
```

Here switch-expression is any scalar expression, and case-expression is any expression at all. Oddly, an array is allowed as a case-expression, but it will never match the switch-expression because the switch-expression must always be a scalar and only a scalar can match a scalar. The commas may be replaced by semicolons. If a block starts on a new line, as in the layout above, neither a comma nor a semicolon is required after the case-expression.

Switch expressions can be strings

The second option for expressions in switch-statements is the string. Here is an example in a function called **number_of_day**, which does the opposite of the function **day_of_week_switch** above. This new function gives us the number of the day when we input the name of the day:

```
function n = number_of_day(day_name)
switch day_name
   case 'Sunday'
      n = 1;
   case 'Monday'
      n = 2;
   case 'Tuesday'
      n = 3;
   case 'Wednesday'
      n = 4;
   case 'Thursday'
      n = 5;
   case 'Friday'
      n = 6;
   case 'Saturday'
      n = 7;
   otherwise
      fprintf('Unrecognized day\n');
      return
end
```

Note that, unlike **day_of_week_switch**, our new function returns something. It returns the number of the day via the variable **n**. Here are some examples of this function being used:

```
>> day_number = number_of_day('Sunday')
 This is a weekend day
day_number =
     1
>> day_number = number_of_day('Wednesday')
 This is a week day
day_number =
     4
>> day_number = number_of_day('Halloween')
Unrecognized day
day_number =
     0
>> day_number = number_of_day('friday')
Unrecognized day
day_number =
     0
```

It works well as long as we give it strings, like **`'Sunday'`** and **`'Wednesday'`**, that we have included as cases in our code, and it is not surprising that it does not recognize **`'Halloween'`** as the name of a day of the week. It may, however, be surprising that it does not recognize **`'friday'`**. The problem here is that the **`'f'`** is not upper case, and as a result **`'friday'`** does not match **`'Friday'`** exactly. There is no wiggle room here. The strings must be exactly the same or there is no match. We will see later how to allow variation between upper and lower case.

A case expression can be a set

The third option is the use of a set as a case-expression. A case-expression can be a set of values or a set of expressions. The set is known as a "cell array", and we will learn about them in the section entitled Data Types, but for now, we will note that a set can be denoted simply by delimiting a vector with braces, instead of brackets, as shown in this formal specification:

```
case {case-expression1,case-expression2,...},
    block
```

The meaning is that, if the switch-expression matches any of the case-expressions in the list, the block will be executed.

Here is an example that includes a set:

```
function weekday_or_weekend(n)
switch n
   case 1
      fprintf('Sunday\n');
   case {2,3,4,5,6}
      fprintf('Weekday\n');
   case 7
      fprintf('Saturday\n');
   otherwise
      fprintf('Number must be from 1 to 7.\n');
end
```

This function prints the name of the day, if we give it 1 or 7 as input, and for the numbers 2, 3, 4, 5, and 6, it prints the word **`'Weekday'`**. The weekday case is handled by giving as the case-expression a set of numbers delimited by braces, instead of a single number. As for a normal vector, the commas separating the numbers between the braces can be replaced by spaces.

Here is the function in action:

```
>> weekday_or_weekend(1)
Sunday
>> weekday_or_weekend(4)
Weekday
>> weekday_or_weekend(9)
Number must be from 1 to 7.
```

As usual, the values can be replaced by expressions. So, for example, **`{2,3,4,5,6}`** could have been **`{2,2+1,5-1,5 2*5-4}`**.

Switch-statements versus if-statements

The switch-statement and the if-statement are equally powerful (each can do what the other can do), but the switch-statement is especially well suited to handle cases with only a finite set of choices. Here is an example of an if-statement and a switch-statement that do the same thing:

```
if x == 1
   fprintf('One\n');
elseif x == 2
   fprintf('Two\n');
elseif x == 3
   fprintf('Three\n');
else
   fprintf('%d\n', x);
end

switch x
   case 1
      fprintf('One\n');
   case 2
      fprintf('Two\n');
   case 3
      fprintf('Three\n');
   otherwise
      fprintf('%d\n', x);
end
```

Each of these statements—the if-statement and the switch-statement—prints the name of the value of **x**, if it equals 1, 2, or 3, and prints digits otherwise. The switch-statement is more appropriate because it avoids the needless repetition of the **==** operator.

Just for fun, here is an example that shows how to construct a switch-statement that handles continuous ranges of values. It proves the power of the switch-statement to duplicate the functionality of the if-statement, but it is a very poor programming example! When a range of values is involved, an if-statement is the right choice.

```
if x < 1
   fprintf('x is small\n');
elseif x > 10
   fprintf('x is large\n');
elseif x > 1
   fprintf('x is medium\n');
else
   fprintf('x equals 1\n');
end
```

```
switch true
   case x < 1
     fprintf('x is small\n');
   case x > 10
     fprintf('x is large\n');
   case x > 1
     fprintf('x is medium\n');
   otherwise
     fprintf('x equals 1\n');
end
```

Relational Operators

The operators, **==**, and **<**, which we have seen in if-statements above, are examples of relational operators. A **relational operator** produces a value that depends on the relation between the values of its two operands. As we noted above, the operator **==** is symbolized by two equal signs (a notation borrowed from C/C++). It is the "is-equal-to" operator, or "equals" operator, and it means exactly what both names imply. When we use it as a conditional in an if-statement, it causes the block it governs to be executed if and only if its first operand is equal to its second operand. Note that this operator is very different from the assignment operator, which is symbolized by just one equal sign and which causes the value of the variable at its left to be set to the value at its right. The operator symbolized by **<** is the "is-less-than" or "less-than" operator, and it also has the meaning we would expect. When we use it as a conditional in an if-statement, it causes the block it governs to be executed if and only if its first operand is less than its second operand.

There are four additional relational operators, and all six of them are given in Table 2.10.

While relational operators usually appear in the conditional expressions of if-statements, when we reach the section entitled Loops, we will find them in another control construct, called the "while-statement" as well. That's not the

Table 2.10 Relational Operators

OPERATOR	MEANING
==	is equal to
~=	is not equal to
>	is greater than
<	is less than
>=	is greater than or equal to
<=	is less than or equal to

only other place they can appear. Newcomers to MATLAB usually find it surprising to learn that relational operators can appear *outside* control statements and even more surprising that a relational operation in fact produces a value! Here are two simple examples in the Command Window:

```
>> 10 == 20
ans =
     0

>> 3 == 35-32
ans =
     1
```

In the first of these two examples, we asked MATLAB to calculate the value of `10==20`, and it told us that the value is 0. In MATLAB, when the operator `==` finds that its first operand is not equal to its second operand, it returns the value zero, which means "false". In the second example, we found that the value of the expression `3 == 35-32` is 1. When the `==` operator finds that its first operand is equal to its second operand, it returns the value 1, which mean "true". (The ideas of using 1 to stand for truth and 0 to represent falseness and of having relational operators produce values is borrowed from the C language.)

In fact, every relational operator returns 0 when its expression is false and 1 when its expression is true. Here is another example:

```
>> x = (45*47 > 2105) + 9
x =
    10
```

In this expression, `45*47`, which equals `2115`, is greater than `2105`, so the relational expression in parentheses evaluates to 1 (true). We then add 9 to 1 and get 10.

The parentheses have an important effect here. If we omit them, we get a different answer:

```
>> x = 45*47 > 2105 + 9
x =
     1
```

This time, the addition operator is executed before the greater-than operator, because the precedence of `+` is higher in than the precedence of `>` so it is carried out first. In fact all the arithmetic operators have higher precedence than all the relational operators. (See the precedence table in Table 2.13.) The result of the addition is `2114`, but the value of the operand on the left of the `>`, which as before is `2115`, is still greater than the value of the operand on the right of the `>`, so the relational operator produces the value 1 (true), as before. Since nothing is added to it this time, the result of the entire expression is `1`.

Here is an example of a meaningful use of relational operators in arithmetic expressions (i.e., not in the conditional of an if-statement, for example). First we note that sometimes, when an expression involves division, there is a danger that we may divide by zero, as in this example:

```
>> x = 16;
>> y = 0;
>> z = x/y
z =
   Inf
```

Most languages show little patience for division by zero and will merely stop everything, print an error, and wait for the user to change the code and rerun the program. MATLAB is far more forgiving (another of its advantages). It simply says that the result is infinity by giving it a special value—**Inf**. Let's suppose, however, that the programmer would rather that **z** be set to **x**, instead of **x/y** when **y** equals zero. We could handle that with an if-statement in the following way:

```
if y ~= 0
   z = x/y;
else
   z = x;
end
```

Using an if-statement is a perfectly good solution, but it is also possible to accomplish the same thing with a single arithmetic expression involving a relational operator as follows:

```
z = x/(y + (y==0)) .
```

Here is a demonstration, using a nonzero value for **y** first and then a value of zero for **y**:

```
>> x = 16;
>> y = 2;
>> z = x/(y + (y==0))
z =
     8
>> y = 0;
>> z = x/(y + (y==0))
z =
    16
```

As with the if-statement version, when **y** is non-zero, **z** is set to **x/y**, and when **y** is zero, **z** is set to **x**. Here is another example. This one sets **z** to **0**, instead of **x**, when **y** is zero:

```
z = (y~=0)*x / (y + (y==0)) .
```

A convenient feature of the relational operators is that they are array operators, like, for example, **.***, which was introduced in the Section, Matrices and Operators, and they obey the same rules as the other array operators. Thus, by giving them two operands that are arrays of the same size and shape, we can compare many pairs of values with just one expression. Here are a couple examples:

```
>> [4 -1 7 5 3] > [5 -9 6 5 -3]
ans =
     0     1     1     0     1
```

which says that **4** is not greater than **5**, **-1** is greater than **-9**, **7** is greater than **6**, **5** is not greater than **5**, and **3** is greater than **-3**,

```
>> [4 -1 7 5 3] ~= [5 -9 6 5 -3]
ans =
     1     1     1     0     1
```

which says that only the 4th elements are equal.

Also, like the array operators, if one operand is a scalar, then the other operand can have any size and shape, allowing us to compare many values to one value, as in these examples:

```
>> [4 -1 7 5 3] <= 4
ans =
     1     1     0     0     1

>> [14 9 3 14 8 3] == 14
ans =
     1     0     0     1     0     0

>> sum([14 9 3 14 8 3] == 14)
ans =
     2
```

The last expression shows an easy way to determine how many elements of a vector are equal to a given value. It is easy to see how to change this example to handle questions like, "How many elements of the vector **v** are greater than **pi**?"

Logical Operators

We have seen that by employing relational operators in conditional expressions, if-statements can be used to determine the flow of control on the basis of the values of variables. This is a powerful idea, but it can be made much more powerful by the addition of another set of operators—the logical operators. A **logical operator** produces a value that depends on the truth of its two operands. There are three of these operators and they are given in Table 2.11.

Table 2.11 Logical Operators

OPERATOR	MEANING
&&	and
||	or
~	not

To understand how they work, let's consider a simple problem. Suppose we want a function that takes three inputs, and it returns 1 if they are in increasing order, -1 if they are in decreasing order, and zero otherwise. Here is a function that does that and uses the logical "and" operator to get it done:

```
function a = order3(x,y,z)
if x <= y && y <= z
   a = 1;
elseif x >= y && y >= z
   a = -1;
else
   a = 0;
end
```

The "and" operator, symbolized by the double ampersand, **&&**, makes its first appearance in the first conditional: **x <= y && y <= z**. Here is how it works: The **&&** operator takes two operands. If both are true (values are nonzero), then it returns true (value of 1). Otherwise, it returns false (value of 0). To be clear, if either one of its operands is false (value of 0), it returns false (value of 0). No real surprise here. This is the normal, everyday meaning of the word "and". However, there is one interesting feature about this operator that we will take up in the next subsection.

Short Circuiting

In the expression, **x <= y && y <= z**, the first operand of the logical "and" operator **&&** (i.e., the operand to its left), **x <= y**, is evaluated first, and if that operand is false, then the **&&** operator returns false (value of 0)—without evaluating its second operand at all! It ignores its second operand when its first operand is false because evaluating the second operand would be a waste of time. Regardless of whether that second operand is true or false, the answer will be false whenever the first operand is false. Skipping the evaluation of the second operand because its value will have no effect on an operator's result is called **short circuiting**. Short circuiting is not always possible. When the first operand is true, then the truth or falsehood of the "and" expression is determined by the second operand. If that operand is also true, then the **&&** operator returns true (value of 1); if the second operand is false, then the **&&** operator returns false (value of 0).

The flow of control within the logical "and" operation above is illustrated in Figure 2.26. The logical "and" operation and its two operands are color coded and their possible outputs—true or false—are shown. The actual outputs are `1` for true and `0` for false (not shown). The path taken when the first operand is false is labeled "short circuit" because it avoids passing through the path of the evaluation of the second operand and takes the direct (short) path (circuit) to false.

Here is a second example of the logical "and" operator:

```
function a = not_smallest(x,y,z)
if x < y && x < z
   a = 0;
else
   a = 1;
end
```

The job of this function is to determine whether its first input argument is smaller than both of its other two input arguments. If it is not smaller than both of them, then it returns `1`, meaning it is not the smallest (hence the name of the function); otherwise it returns `0`. If the first operand is false, then short-circuiting takes place.

There is a second short-circuiting logical operator—the logical "or" operator, symbolized by `||`. It returns true (value of 1) if at least one of its operands is true—the first one, the second one, or both—and (value of 0) if both operands are false. We illustrate its use by rewriting the function above, using `||` instead of `&&`:

```
function a = not_smallest_version_2(x,y,z)
if x >= y || x >= z
   a = 1;
else
   a = 0;
end
```

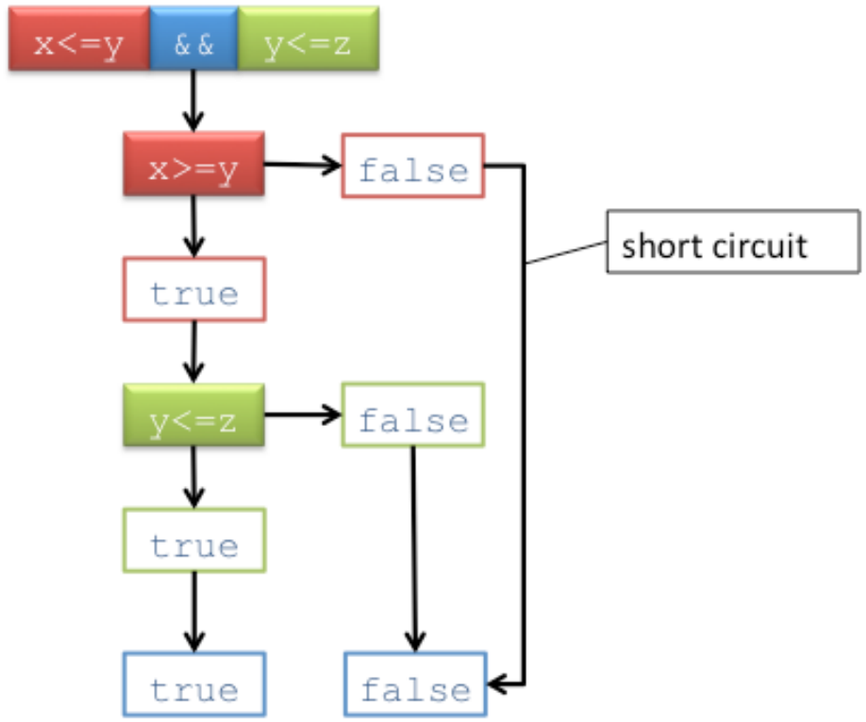


Figure 2.26 *Schematic of short-circuiting for* `&&`*. The flow of control is shown within a short-circuiting logical "and" operation:* `&&`*. The first operand of* `&&` *is red, and its possible outputs–true or false—are outlined in red. The second operand of* `&&` *is green, and its possible outputs are outlined in green. The two possible outputs of* `&&` *are outlined in blue. If the output of the first operand is false, then the path labeled "short-circuit" is followed, bypassing the evaluation of the second operand.*

This function performs exactly the same task as that performed by the "and" version above. Short-circuiting takes place if the first operand is true, as shown in Figure 2.27, because in that case, the `||` operator will return 1, regardless of the value of the second operand.

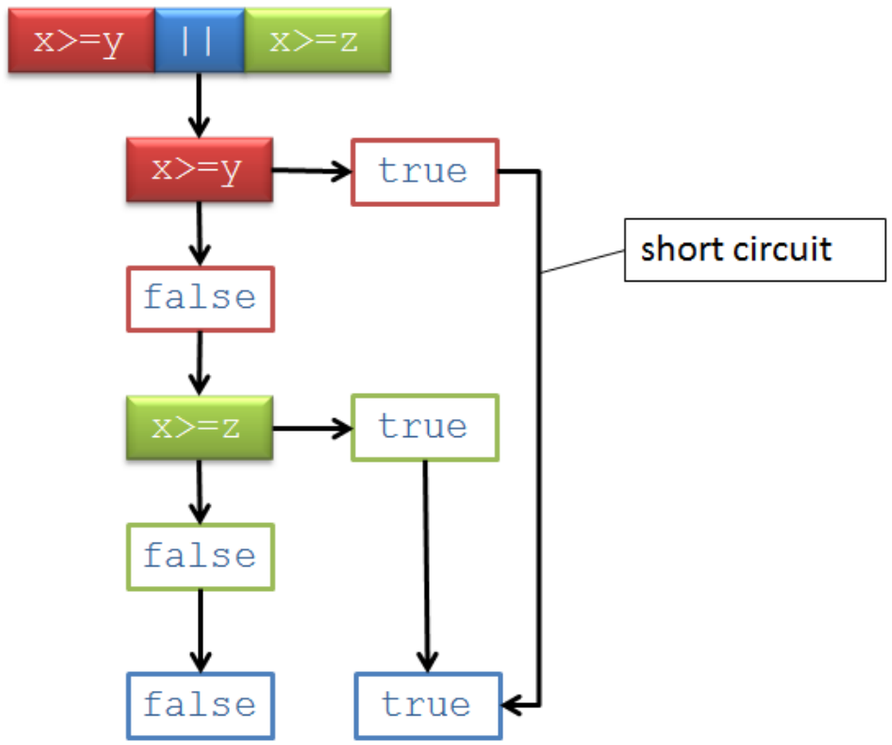


Figure 2.27 *Schematic of short-circuiting for* `||`*. The flow of control is shown within a short-circuiting logical "or" operation:* `||`*. The first operand of* `||` *is red, and its possible outputs –true or false—are outlined in red. The second operand of* `||` *is green, and its possible outputs are outlined in green. The possible outputs of ||are outlined in blue. If the output of the first operand is true, then the path labeled "short-circuit" is followed, by-passing the evaluation of the second operand*

Short circuiting may seem of little importance, because it may seem to be a trivial matter whether or not one operand is evaluated. However, it can save a lot of time if the second operand requires a long time to evaluate. This can happen if a function is involved the operand, as in this example:

```
function a = not_smallest_version_3(x,y,z)
if x >= f(y) || x >= f(z)
   a = 1;
else
   a = 0;
end
```

The only difference between the previous example and this one is that **`y`** and **`z`** have been replaced by **`f(y)`** and **`f(z)`**. Now **`x`** is being compared with the result of the evaluation of the function **`f`** with inputs **`y`** or **`z`**. If the evaluation of **`f`** requires, say, 30 minutes, then when **`x`** is greater than or equal to **`f(y)`**, we definitely do not want to wait around for half an hour while **`f(z)`** is evaluated when we don't care what the result is!

In Table 2.12 the outputs of the logical "and" operator **`&&`** and the logical "or" operator `||` are given for their four possible inputs and the outputs for the related function **`xor`** ("exclusive or") is given as well. There is no operator for **`xor`**. It is a function that takes two inputs as for example: **`xor(x,y)`**.

Table 2.12 *Logical "and" and "or" input/output*

INPUTS		&&	||	XOR
0	0	0	0	0
0	nonzero	0	1	1
nonzero	0	0	1	1
nonzero	nonzero	1	1	0

It should be noticed that the inputs are `0` and "nonzero", instead of `0` and `1`. These are MATLAB's denotations of the values that are treated as meaning false and true when used as inputs to logical operations. Up to now, we have seen only the number `1` used to mean true because that is the value that all the relational and logical operators return when their expressions are true. However, these operators allow any value to be used as ***input***, and among all possible input values, only zero means false. Here are examples of non-zero values being used as input to `&&` and `||`:

```
>> 1 && 1
ans =
     1
>> 17 && 1
ans =
     1
>> -1 && 1
ans =
     1
>> -1 || 0
ans =
     1
>> pi || 0
ans =
     1
```

In every case, a nonzero value as input has the same effect as **1**.

The third operator in Table 2.11 is the logical "not" operator, symbolized by **~** (called "tilde" and pronounced "TILD-ah", and usually found just above the Tab key on the keyboard). The "not" operator is a unary prefix operator, like the unary **+** and unary **-**, which, as we learned in Matrices and Operators, means that it takes only one operand and it comes before its operand. Here is an example of this operator being used in a fourth version of our **not_smallest** function:

```
function a = not_smallest_version_4(x,y,z)
if ~(x < y && x < z)
   a = 1;
else
   a = 0;
end
```

The not-operator appears in the conditional of the if-statement, and its operand is the logical expression in parentheses. The evaluation of the conditional proceeds as follows: (1) the relational operation **x < y** is evaluated, (2) the relational operation **x < z** is evaluated, (3) the logical "and" operation **x < y && x < z** inside the parentheses is evaluated, and (4) the not-operator is applied to the result. If the value produced by the "and" operation is 1 (meaning true), then the not-operation returns 0 (meaning false). If the value of the "and" operation is 0 (meaning false), then the not-operation returns 1 (meaning true). In other words, **~** merely changes a zero to a 1 and a nonzero value to a zero. It might win the award for being MATLAB's simplest operator.

This version of our function seems to match the name of the function, "not_smallest", better than other versions, because it is the only version that uses the "not" operator. This added clarity may seem to be a small thing, but writing logical expressions in an easily understandable form is important because, when we look at our functions later or when someone else looks at them, it will require less time to decipher and will reduce the possibility of a bug being introduced when new code is added. A good programmer always tries to find a clear way to write logical expressions.

Here again, the parentheses are important. If we remove them, then we are left with the expression, **~x < y && x < z**. The precedence of the "not" operator is higher than that of the "less-than" operator, so without the parentheses, the first operation that is carried out is **~x**, which equals either zero or one according to whether **x** is nonzero or zero. Then the "less-than" operator will be carried out, and its first operand will be either one or zero, instead of **x**. This is clearly not what we intended.

Like the relational operators, but unlike **&&** and **||**, the operator **~** is an array operator: Thus, it can be applied to an array producing a "not" operation on each element: Here are two examples:

```
>> ~[1    -1    0    0    pi    0    4]
ans =
     0     0     1    1     0    1     0
```

We have included some nonzero values other than one to emphasize the fact that nonzero (even -1) means true.

```
>> ~[-1<1   4==4   2>3   2~=2   9~=4   6>=7   6<=7]
ans =
     0     0     1     1     0     1     0
```

The output elements are the same as in the previous example, but this time the input elements are relational operations that happen to be true or false in the same places where nonzero and zero values appeared in the input elements of the previous example.

There are two more logical operators—the so-called "element-wise" versions of the logical "and" operator and the logical "or" operator. These operators are symbolized by a single `&` and a single `|`. Like their double-symbol counterparts, they are both binary operators, and they perform the same logical operations as they do, namely, "and" and "or". However, these operators, like the relational operators and `~`, are array operators: They are the array versions of `&&` and `||`, each of which can be applied only to scalars. Like the other binary array operators, they can also take one scalar operand and one non-scalar operand. There is one strange wrinkle to `&` and `|`. When they appear in a conditional of an if-statement (or a while-statement, which we will see in the next section), they short-circuit just like `&&` and `||`, but when they appear outside, they do not.

Here are some examples of the use of `&` and `|` with arrays:

```
>> [4 0 pi -1 0 1/3] & [1 1 -2 0 0 8]
ans =
     1     0     1     0     0     1

>> [4 0 pi -1 0 1/3] | [1 1 -2 0 0 8]
ans =
     1     1     1     1     0     1

>> [1 0 2;0 4 -1] | [0 0 .3;0 4 0]
ans =
     1     0     1
     0     1     1
```

```
>> [1 0 2;0 4 -1] & 7
ans =
     1     0     1
     0     1     1
```

We have now introduced all ten of MATLAB's operators. We gave the precedence of the first five of them (along with parentheses) in the section entitled, Matrices and Operators. Table 2.13 gives the complete **precedences table** for all of them. Remember that a lower precedence *number* means higher precedence, which means earlier evaluation in an expression: operators with lower precedence *numbers* go first. However you can always override the order of operations with parentheses. Before any operations within an expression are carried out, the MATLAB interpreter finds all matching pairs of parentheses (its zero precedence beats all the rest!), and it evaluates the expression inside any matching pair of parentheses separately from the rest of the expression.

Table 2.13 *Operator precedence*

PRECEDENCE	OPERATOR
0	Parentheses: `(...)`
1	Exponentiation `^` and Transpose `'`
2	Unary `+`, Unary `–`, and logical negation: `~`
3	Multiplication and Division (array and matrix)
4	Addition and Subtraction
5	Colon operator `:`
6	Relational operators: `<`, `<=`, `>`, `>=`, `==`, `~=`
7	Element-wise logical "and": &
8	Element-wise logical "or": |
9	Short-circuit logical "and": &&
10	Short-circuit logical "or": ||

Also, as we learned in Matrices and Operators, when there are two or more operators of the same precedence, left-to-right associativity is used, meaning that the order of operation is from left to right. Parentheses can override this associativity rule also.

Memorizing all these rules may seem daunting, but in fact, all of us who have had experience with basic algebra have already learned the rules associated with addition, subtraction, multiplication, and division. For example, you will know without looking at the precedence table when you see **`z/3+y*z`** that **`z`** is divided by 3 and **`y`** is multiplied by **`z`** *before* the addition is carried out, even though that is not the order in which they are written in, and you will know, without peeking at the table, that adding parentheses like this, **`z/(3+y)*z`**, will cause the addition to happen first. The rules for the other operators will be less familiar, but they typically agree with intuition, and if you are in doubt, then anyone who reads your code will probably be in doubt too. In those cases, even if you know the rules, you would do well to add redundant parentheses to make the order of operations perfectly clear. MATLAB makes it easy to review the table. All you have to do is type **`help precedence`** in the Command Window.

Nested Selection Statements

We have seen both types of selection statements: the if-statements and the switch-statements. They have the common feature that each one chooses whether or not a block of statements is executed or selects one block from among two or more for execution. So far, in all our examples, the only statements in these blocks have been assignment statements and **`fprintf`** function call statements, but there is in fact no restriction whatsoever on the type of statements that can be included in those blocks. Other control statements can be included too. We refer to the inclusion of one control construct inside another construct as **nesting**. In this section, we look at examples of nesting in which blocks inside selection statements contain other selection statements.

For our first example, we will use nesting to rewrite a function that we wrote earlier without nesting—**`ultimate_question`**. We repeat that function below:

```
function ultimate_question(x)
if x == 42
   fprintf('Wow! You answered the ultimate question.\n');
elseif x < 42
   fprintf('Too small. Try again.\n');
else
   fprintf('Too big. Try again.\n');
end
```

We used this function to introduce the **`elseif`** clause, and that is the best way to write it, but in order to show how nesting works, here is a version with nesting that does not use **`elseif`**:

```
function ultimate_question_nested(x)
if x == 42
   fprintf('Wow! You answered the ultimate question.\n');
else
   if x < 42
      fprintf('Too small. Try again.\n');
   else
      fprintf('Too big. Try again.\n');
   end
end
```

This function provides an example of nesting because of the if-else statement that begins with "**`if x < 42`**", which lies within the second block of the if-else-statement that begins "**`if x == 42`**". Thus, this is an example of an if-else statement nested inside another if-else statement. This nesting occurred in the second branch of the first if-statement, but nesting can occur in any branch, as for example in the following re-write of this function, which we call (for lack of a good name!) **`ultimate_question_nested2`**:

```
function ultimate_question_nested2(x)
if x <= 42
   if x == 42
      fprintf('Wow! You answered the ultimate question.\n');
   else
      fprintf('Too small. Try again.\n');
   end
else
   fprintf('Too big. Try again.\n');
end
```

This time, the nesting is in the first branch of the first if-else statement.

So far, we have nested if-statements in each other, but, as we mentioned above, there is no limit on what can go inside what. Here is a function, named **`write_digit`**, that prints the name of a digit, and before it does that, it checks its input to be certain that it is a positive, single-digit integer:

```
function write_digit(x)

if floor(x) ~= x % x is not an integer
   fprintf(' %f is not an integer',x);
else
   if x < 1 || x > 9
      fprintf('%d is not in the range 1..9',x)
   else
      switch x
         case 1
            fprintf('one');
         case 2
            fprintf('two');
         case 3
            fprintf('three');
         case 4
            fprintf('four');
         case 5
            fprintf('five');
         case 6
            fprintf('six');
         case 7
            fprintf('seven');
         case 8
            fprintf('eight');
         case 9
            fprintf('nine');
      end
   end
end
fprintf('\n');
```

Here is a look at **`write_digit`** in action:

```
>> write_digit(3)
three

>> write_digit(5)
five

>> write_digit(9)
nine

>> write_digit(4.9)
 4.900000 is not an integer

>> write_digit(10)
10 is not in the range 1..9

>> write_digit(-2)
-2 is not in the range 1..9
```

This example shows that a switch-statement can be nested inside an if-statement, and it also shows that nesting can be deeper than one level. The outer if-else-statement, which begins "`if floor(x) ~= x`", has another if-else statement within its else-clause, which constitutes the first level of nesting. Furthermore, that inner if-else statement has a switch-statement within its else-clause, which constitutes the second level of nesting.

There is no limit to the depth of nesting allowed, and, we will see other types of control constructs nested in one another in the next section. This flexibility in nesting is an important aspect of all modern programming languages, and MATLAB supports it unconditionally.

Additional Online Resources

- Video lectures by the authors:
  - Lesson 5.1 Selection (11:53)
  - Lesson 5.2 If-Statements, continued (8:33)
  - Lesson5.3 Relational and Logical Operators (34:51)
  - Lesson 5.4 Nested If-Statements (2:12)
  - Lesson 5.5 Variable Number of Function Arguments (6:40)
  - Lesson 5.6 Robustness (8:37)
  - Lesson 5.7 Persistent Variables (6:54)

Concepts From This Section

Computer Science and Mathematics:

- control constructs
  - sequential control
  - selection
- conditionals
- relational operators
- logical operators

MATLAB:

- selection statements
  - if-statement
    - **`else`** clause
    - **`elseif`** clause
  - **`switch`**-statement
    - **`case`**
    - **`otherwise`**
  - nesting of control constructs
- relational operators
- logical operators

Practice Problems

Problem 1. [Answer in English] Which type of control construct is most appropriate for determining which operation to perform when the user selects from a menu of operations?

?

Problem 2. [Answer in English] Which type of control construct is most appropriate for determining which operation to perform based on whether an angle is greater than $\pi/2$?

Problem 3. [Answer in English] Name the three keywords that are used with if-statements.

?

Problem 4. [Answer in English] Name the four keywords that are used with `switch`-statements.

Problem 5. Write a function called `two_rows` that takes one input argument and checks its format. If the argument is a two-dimensional array with two rows, it returns the array; otherwise, it prints "I must have two dimensions and two rows!" and returns an array of the same size and shape, but with all its values set to zero. NOTE: A column vector is a two-dimensional array. If it has two elements, then it is a two-dimensional array with two rows. HINT: The function `ndims` may be helpful.

?

Problem 6. Write a function called `chop` that takes a 3-element column vector as an input and returns three output arguments. The function must check the format of its input. Namely, it must determine whether the input satisfies the following two restrictions: (1) it is a column vector and (2) it contains exactly three elements. If either or both of these restrictions are violated, then the function prints "Invalid input!" and returns three zeros. Otherwise, it sets each of the output arguments equal to the value of one of the 3-elements of the input vector in the order that they occur in the vector and prints nothing.

Problem 7. In this section a function was defined named `not_smallest`. Write a function called `not_smallest_expression` that takes three scalars as input arguments (the function does not have to check the format of the input) and returns the same scalar value as output that `not_smallest` does but without using an if-statement or a switch-statement.

?

?

Problem 8. Write a function called `unequal4_expression` that takes one four-element vector as an input argument (the function does not have to check the format of the input) and returns 1 if all the elements are unequal and 0 otherwise. You must accomplish this feat with a single expression, not with an if-statement or a switch-statement. NOTE: The expression may be long. To continue the expression onto a second line, type three successive dots (also known as periods or full stops).

Problem 9. Write a function called **`even_odd`** that takes one scalar argument as input (the function does not have to check the format of the input), returns no output arguments, and uses a switch-statement with three branches to print "Odd" if the argument is 1, 3, or 5, "Even" if the argument is 0, 2, or 4, and "Let me get back to you on that one." for any other value. The output should be printed on a line by itself.

?

Problem 10. Write a function called **`ablmt`** that takes one character as an input argument (the function does not have to check the format of the input), returns one string as an output argument. It uses a three-branch switch-statement to set its output argument to "MATLAB", if the input character is one of the characters in the string, "MATLAB", or set its output argument to "matlab" if the character is one of the characters in the string, "matlab", or set its output argument to "I just don't have it in me", if the character is in neither name.

Problem 11. Write a function called **`inside_outside`** that takes three scalar arguments as input (the function does not have to check the format of the input) and uses an if-else-statement to print "Inside" if the value of the second argument lies between the values of the other two arguments or equals one of them, and "Outside" otherwise.

?

Problem 12. Write a function called **`less3`** that takes two three-element vectors as input arguments (the function does not have to check the format of the input) and returns one output argument. It uses an if-else-statement to return 1 if each element of the first vector is less than the corresponding element of the second vector, and zero otherwise. Thus, it would return 1 for the call **`less3([1 -9 0],[99,-8,0.001])`** but would return 0 for the call **`less3([8,8,7],[9,8,9])`** because the second element of the first argument (**`8`**) is not less than the second argument (**`8`**) of the second argument.

Problem 13. Write a function called **`sort2`** that takes two unequal scalar arguments (the function does not have to check the format of the input or the inequality of the arguments). It uses one or more if-statements to return the two values of these arguments in a two-element row vector in increasing order, i.e., element one of the output vector equals the smaller input argument and element two of the output vector equals the larger input argument. NOTE: The function may not use the built-in function **`sort`**.

?

Problem 14. Write a function called **`sort3`** that takes three unequal scalar arguments (the function does not have to check the format of the input or the inequality of the arguments). It uses if-statements, possibly nested, to return the three values of these arguments in a single row vector in increasing order, i.e., element one of the output vector equals the smallest input argument and element three of the output vector equals the largest input argument. NOTE: The function may not use the built-in function **`sort`**.

Problem 15. Write a function called **`over_pi`** that takes one vector as an input argument and returns one scalar output argument. If it is called this way, **`n = over_pi(v)`**, then it uses a single expression, without using an if-statement or switch-statement (or loop), to set **`n`** equal to the number of elements in **`v`** that are greater than pi.

?

Problem 16. Write a function called **`between`** that takes one scalar and two vectors of the same length as input arguments (the function does not have to check the format of the input) and returns one scalar output argument. If it is called like this, **`n = between(a,u,v)`**, then **`n`** is equal to the number of indices **`ii`** for which the scalar **`a`** is between **`u(ii)`** and **`v(ii)`** or **`a`** is equal to **`u(ii)`** or **`v(ii)`**. Here are some examples for the case in which the length of the vectors is **`4`**:

```
>> n = between(4,[5,0 -7 6], [3, 9, 4, 4])
n =
     4
>> n = between(5,[5,0 -7 6], [3, 9, 4, 4])
n =
     3
>> n = between(6,[5,0 -7 6], [3, 9, 4, 4])
n =
     2
>> n = between(9,[5,0 -7 6], [3, 9, 4, 4])
n =
     1
>> n = between(10,[5,0 -7 6], [3, 9, 4, 4])
n =
     0
```

Problem 17. Write a function called **`bolt_check`** that takes one scalar input argument (the function does not have to check the format of the input) and returns one scalar output argument. The input represents the measured length of a machine screw during the quality-assurance phase of its manufacturing. The purpose of the function is to categorize the measured length into one of six categories. If the measured length is within two percent of one of its five nominal lengths—3/8, 1/2, 5/8, 3/4, or 1 (inch), then the nominal length is returned. Otherwise 0 is returned, signifying that the bolt failed the test.

?

Problem 18. Write a function called **`coin_value`** that takes two scalar input arguments (the function does not have to check the format of the input) and returns one scalar output argument. This function is part of a program used in a coin-operated vending machine that determines the value of a single coin. This function uses the measured diameter and mass of a coin to determine its value. If both the diameter and the mass fall within 5% of their nominal values than the worth of the coin as a fraction of a dollar is returned; otherwise 0 is returned to indicate that the coin failed the test. Here is a table of the nominal diameters, weights, and values of the coins that must be included (copper-alloy coins only):

Name	Value ($)	Diameter (mm)	Mass (g)
Small Cent	0.01	19.05	2.50
Nickel	0.05	21.21	5.00
Dime	0.10	17.91	2.50
Quarter	0.25	24.26	6.25
Half-dollar	0.50	30.61	11.34
Dollar	1.00	26.50	8.10

Problem 19. Write a function named **`day_of_month`** that takes three scalar integer input arguments (the function does not have to check that the inputs are scalars), returns no output arguments, and prints the day of the month to the Command Window. The day of the month must be printed in the form

<*ordinal*> <*day*> in <*month*>

where <*ordinal*> is one of the words, "first", "second", "third", "fourth", or "fifth", <*day*> is one of the words "Sunday", "Monday", …, or "Saturday", and <*month*> is one of the words "January", "February", …, or "December". If the function is called like this, **`day_of_month(n,d,m)`**, then **`n`** determines <*ordinal*>, **`d`** determines <*day*>, and **`m`** determines <*month*>. In this Section, the function **`day_of_week_switch`** is given. A function very similar to **`day_of_week_switch`** must be used as a subfunction in the M-file **`day_of_month.m`** to print <*day*>. The function **`day_of_month`** must check the validity of the values of its three inputs. Unless the first argument falls within the range 1 to 5, the second argument falls within the range 1 to 7, and the third argument falls within the range 1 to 12, the function must print an error message and return. HINT: The built-in function **`floor`** can be used to determine whether a number is a whole number (the function **`isinteger`** cannot be used for this purpose). If the numbers are all in range, then the printout must be consistent with the following examples:

```
>> day_of_month(4,1,1)
Fourth Sunday of January
>> day_of_month(1,3,11)
First Tuesday of November
>> day_of_month(3.4,3,9)
Inputs must be whole numbers
>> day_of_month(3.4,3,9.1)
Inputs must be whole numbers
>> day_of_month(7,3,9)
1st argument must be in the range 1 to 5
>> day_of_month(1,2,13)
3rd argument must be in the range 1 to 12
>> day_of_month(7,33,9)
1st argument must be in the range 1 to 5
2nd argument must be in the range 1 to 7
>> day_of_month(7,33,13)
1st argument must be in the range 1 to 5
2nd argument must be in the range 1 to 7
3rd argument must be in the range 1 to 12
```

?

Problem 20. Write a function named **`write_two_digits`** that takes one scalar input argument (the function does not have to check that the input is a scalar), returns no output arguments, and prints to the Command Window the name of the value of the input. The function must check to determine whether the input is a whole number in the range, -99 to +99. If the number is out of range, the function prints an error message and returns. If the number is in range, then the function prints the integer's name (examples of names are given below). If the input falls in the range -9 to +9, then the printout must be the same as that of **`write_digit`**, which is given in this Section, and that function must be used as a subfunction in the M-file **`write_two_digits.m`**. HINT: The built-in function **`floor`** can be helpful. It can be employed to determine whether a number is a whole number, and it can also be employed to determine the value of the digit in the tens place and the value of the digit in the ones place. The printout of **`write_two_digits`** must be consistent with these examples:

```
>> write_two_digits(0)
zero
>> write_two_digits(3)
three
>> write_two_digits(-3)
minus three
>> write_two_digits(10)
ten
>> write_two_digits(12)
twelve
>> write_two_digits(18)
eighteen
>> write_two_digits(23)
twenty-three
>> write_two_digits(-99)
minus ninety-nine
>> write_two_digits(113)
Input must be an integer from -99 to +99.
>> write_two_digits(24.7)
Input must be an integer from -99 to +99.
```

Loops

Objectives

Loops give computers their power.

(1) We will learn how to use the for-loop and the while-loop.

(2) We will learn how the break and continue-statements work, and we will use nested loops.

(3) We will learn how to make loops more efficient and will learn about MATLAB's powerful implicit loop system.

(4) We will learn about logical indexing and see how to use it to produce implicit loops that are efficient and easy to read.

Marco Silva, Olinda, Brazil, Dreamstime.com, Photo taken April 15th, 2008.

Loops add excitement to roller-coasters and add power to MATLAB.

The former tech journalist and Microsoft Bing Apps program manager, Mitch Ratcliffe, once wrote, "A computer lets you make more mistakes faster than any invention in human history—with the possible exception of handguns and tequila."[1] Why does it let you make mistakes so fast? The answer is simple: because of loops. Without loops computers would still give you opportunities to make mistakes, but not very many or very fast. So loops are bad, right? Well, not all bad. In fact they are very good. Without loops, computers would be little more than fancy calculators. Loops give a computer its greatest power—whether that computer is your laptop, your cell phone, or your automobile's automatic braking system. If we took their loops away, these computers could still consume information, could still do calculations, and could still produce output, but they would lose their

[1]"The Pleasure Machine," MIT Technology Review, Spring 1992.

ability to do these things on such an enormous scale in such tiny fractions of a second. Is it important to get the calculation done in tiny fractions of a second, instead of, say, a minute? Ask yourself that question the next time you hit the brakes at 60 mph and the computer in your car adjusts them quickly enough to stop you from skidding over an embankment or when you use Google to find the address of the office for your job interview from among a billion addresses, just as you leave your apartment.

The mistakes that Mr. Ratcliffe was writing about are loop errors, and those errors come from us—the programmers. What we do with guns and tequila is beyond the scope of this book, but what we do with loops is exactly the subject of this section. We are about to learn how to make a computer go fast—really fast.

The Loop Concept

The loop is a new control construct, and we are about to add it to the constructs already introduced in the previous section Selection. There we learned about the control constructs—**`if`**, **`if-else`**, **`if-elseif`**, **`if-elseif-else`**, and **`switch`**, each of which stipulates whether a block of statements is to be executed one time or zero times. The **loop** construct, as we are about see, involves a far more powerful idea: It is a control construct that can stipulate whether a block of statements is to be executed zero, one, a hundred, a hundred thousand, or any other number of times. We are going to see in this chapter just what "loop" means. However, before we look closely at the loop idea, let's look at the speed-up it can provide with an example. Suppose we want to add up the numbers 1 through 10. You have already seen two simple ways to do this calculation using MATLAB. Here are those two methods as they would be typed into the Command Window:

```
>> 1+2+3+4+5+6+7+8+9+10
ans =
    55

>> sum(1:10)
ans =
    55
```

The first method is the basic calculator approach—type in all the numbers to be added with plus signs between them and hit Enter. The second example is a bit less laborious because it uses loops. In fact it includes two MATLAB features that use loops. The first feature is the expansion of **`1:10`** into the vector **`[1 2 3 4 5 6 7 8 9 10]`** by means of the colon operator. The second feature is the addition of all the elements in that vector by means of the function named **`sum`**. Each of these two features is implemented inside MATLAB (i.e., hidden from the user) by means of a loop. Each loop causes a set of statements to be executed repeatedly, but each loop is an implicit loop. It is implicit because you don't explicitly tell MATLAB what instructions to repeat. When MATLAB sees **`1:10`**, it starts a loop that generates the vector **`[1 2 3 4 5 6 7 8 9 10]`**, which it then uses as the argument in the call of the function **`sum`**. Then, when it evaluates **`sum([1 2 3 4 5 6 7 8 9 10])`**, it starts a loop that adds together all the elements in the argument to the function **`sum`**. This simple example gives a hint as to the power of loops, because it is clearly easier to type the nine characters of the command, **`sum(1:10)`**, i.e., **`s u m ( 1 : 1 0 )`**, than it is to type the twenty characters of the command, **`1+2+3+4+5+6+7+8+9+10`**.

It is also quicker. The use of the **`sum(1:10)`** method is more than twice as fast as the **`1+2`**...**`+10`** method, because 20 characters take more than twice as long to type as do nine characters. And the advantage gets a lot more important when there are more numbers involved. If we want to add, say, 1 through 10,000, then it would take 13½ hours to type in all the numbers and plus signs (at 1 character per second), but it would take only 12 seconds to type **`sum(1:10000)`**. That's a speed-up factor of four thousand. It also takes

MATLAB about a half second longer to do the first calculation after it's typed in than the second because it has to process so many characters, but the big savings here is in the human typing time.

You have been using the colon operator and the function **`sum`** since the section entitled Functions, so you have already seen the power of loops, although you may not have thought about the looping aspect until now. Every expression that involves the colon operator is evaluated with a loop that employs a small set of statements—repeatedly executing them over and over, perhaps hundreds or thousands of times. Every array operation is evaluated with loops; every matrix operation is evaluated with loops; the functions **`max`** and **`min`** use loops; **`sqrt`**(…) uses a loop. In fact almost every MATLAB operator or built-in function that carries out a numerical calculation uses one or more loops. They are implicit loops, so they are hidden, but they are everywhere.

So, what is a loop? Figure 2.28 illustrates a loop for the addition of the integers from 1 to 10. That figure shows a flowchart for the operations required to carry out the addition on a computer, and that flowchart includes a loop, which is outlined in a blue box. The flow of operations enters at the top of the figure, follows the arrows and exits at the bottom of the figure. After the first operation—"total = 0"—is executed, the loop is entered. Inside this loop are two statements. The first is "Repeat for n = 1 to 10". This statement is a control statement, which, as we learned in the Selection, means that it controls the execution of another statement or statements. The second statement in the loop is "Add n to total", which we have highlighted in light green. This statement is the body of the loop. The body of a loop contains the statements that are repeated. There can be many operations in the body, but we are keeping it simple here by having only one. The green statement is repeated 10 times, and then the loop is over. The reason for calling this repetition a "loop" should be obvious from the shape of the lines of flow in the figure: They form a loop. While our green statement is executing over and over, the flow travels around and around through the path that emerges from the side of the green box and re-enters at the top of the box. The flow is forced to loop back through the green box over and over. After the loop ends, the last operation—"Print total"—is executed. There may be many other operations taking place in the program before and/or after those shown in this figure, but we are focusing here on the loop. In fact, the only reason we include "total = 0" and "Print total" is that they show how the loop might be used. They are not part of the loop itself.

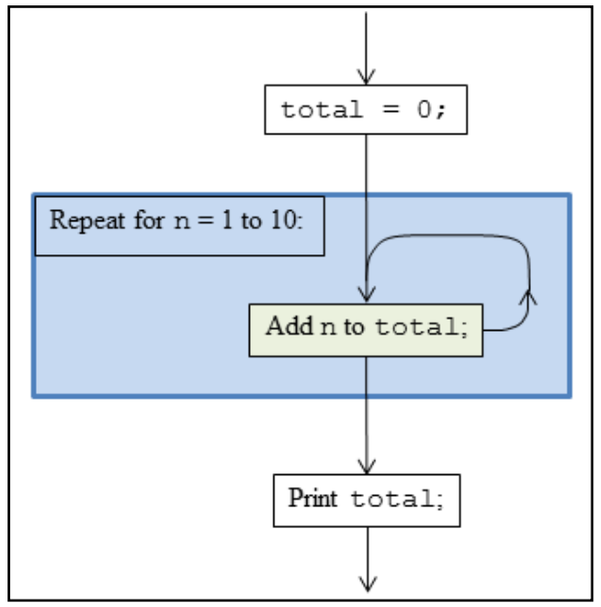


Figure 2.28 *Illustration of a loop. A flowchart is shown for a sequence of operations that calculates and prints the sum: 1 + 2 + 3 + … + 10. Each small box encloses a single operation. The section enclosed in the blue box is a loop. The green box contains the "body" of the loop, which is repeated 10 times before the loop is completed.*

The repetition of the body of the loop in Figure 2.28 continues until a condition has been met: The condition is specified by the control statement, "Repeat for n = 1 to 10", and the condition is that the variable n be set successively to each of the values, 1, 2, 3, …, 10. Here is a formal definition of the word "loop" as it is used in computer programming:

loop :
1a. (noun) a set of statements that is repeated until some condition is met.
1b. (noun) a control construct that causes a block of statements to be executed repeatedly (i.e., zero, one, two, or more times).
2. (non-transitive verb) repeat a set of statements until some condition is met.

Here are examples of each use: 1. "The program that she wrote includes a loop whose statements calculate the voltage for each point in time. 2. "The statements that calculate the voltages will continue to loop until a voltage has been calculated for each point in time."

One execution of the body of a loop is called an **iteration** of the loop. Iteration can also be used to mean looping, as in "Iteration is employed in the calculation of the voltages." Various forms of this word are used to describe a program. We may say that a loop is iterated 10 times or that a loop iterates 10 times, i.e., the verb **iterate** may be transitive or intransitive, and we may describe code that contains a loop as **iterative** code. For example, "Carrying out two hundred calculations requires iterative code to avoid having to type two hundred statements."

As we mentioned above, the body of the loop in Figure 2.28 is controlled by the statement—"Repeat for n = 1 to 10". The body is the workforce of the loop; the control statement is more like the supervisor who keeps the laborers in the workforce repeating their tasks until the job is done. This control statement does something else too. It repeatedly changes the value of the variable n. Here is exactly the sequence of operations it causes to happen:

```
Set n to 1
Execute Add n to total
Set n to 2
Execute Add n to total
Set n to 3
Execute Add n to total
Set n to 4
Execute Add n to total
Set n to 5
Execute Add n to total
Set n to 6
Execute Add n to total
Set n to 7
Execute Add n to total
Set n to 8
Execute Add n to total
Set n to 9
Execute Add n to total
Set n to 10
Execute Add n to total
```

Boring, isn't it? Welcome to the life of a control statement. It tirelessly, robotically, and VERY rapidly repeats statements for you as many times as you ask it to. If we change 10 to 10,000, it will keep executing that same boring statement (i.e., the green one in Figure 2.28) and setting n to the next boring value until it gets to the boring end, and all we have to do to tell MATLAB to carry out all 10,000 of those tedious iterations is to write a couple statements.

The variable n, which is being updated each time through the loop, is a critical part of the loop, and it has a special name. A variable that is changed by the control statement of a loop is called a **loop index**. This is a new use of the word "index", which, up until now, has been used in the programming sense to denote a positive integer that enumerates the elements of a vector or ma-

trix. For example, we might have said, "In the expression `y(n) + 17`, the index `n` specifies a specific element in the vector `y`." That was the old use. We'll continue to use it that way, but here is an example of the new use of "index": "In Figure 2.28, the loop index n starts out equal to 1 and ends up equal to 10." We will use both meanings from now on. Having two meanings for the same word may seem confusing, and sometimes we will find that both meanings apply to the same variable in the same loop, but the context will make the meanings clear, and usually, when we refer to the index of a loop, we will say "loop index" instead of simply "index".

The for-loop

Now we are ready to learn how to write a loop using MATLAB. We will use a control construct called a **for-loop**. The construct works just like the one illustrated in Figure 2.28, and the name "for-loop" is based on the presence of the word "for" in the control statement. The name "for loop" was not invented for MATLAB. It was introduced by the inventors of a language named ALGOL (ALGOrithmic Language) in 1960, and both the construct and the name became so popular that almost every subsequent programming language provides it and uses the same name (e.g., MATLAB, C++, Java all have for-loops). We will later in this section introduce the only other type of loop provided by MATLAB—the "while"-loop.

Here (at last) is the MATLAB implementation of the loop of Figure 2.28:

```
total = 0;
for n = 1:10          <-- control statement
   total = total + n; <-- body
end
fprintf('total = %d\n', total);
```

(the loop = `for` through `end`)

and here is the result of running it:

```
total = 55
```

With this code we have introduced a new MATLAB keyword—**for**. This keyword signifies the beginning of a for-loop. The keyword `end` signifies the end of the loop. The first line of the loop, `for n = 1:10`, is the control statement of the loop. It controls the statement, `total = total + n`, which is the body of the for-loop, and it is forced by the control statement to repeat its task ten times, exactly as in Figure 2.28. Note that there is no semicolon after the control statement. MATLAB never prints the result of the assignment of an element to the loop index, so there is no need for a semicolon here.

The phrase, `n = 1:10`, in the control statement can be a bit confusing at first. It does not mean "Assign the vector `[1 2 3 4 5 6 7 8 9 10]` to `n`, even though that is exactly what the statement `n = 1:10` would mean if it were to appear by itself (i.e., without the keyword `for` in front of it). When `n = 1:10` appears in the control statement of a for-loop, it means this:

```
Assign the first element of [1 2 3 4 5 6 7 8 9 10] to n.
Execute the body of the for-loop.
Assign the next element of [1 2 3 4 5 6 7 8 9 10] to n.
Execute the body again.
Assign the next element of [1 2 3 4 5 6 7 8 9 10] to n.
Execute the body again.
. . .
Assign the last element of [1 2 3 4 5 6 7 8 9 10] to n.
Execute the body for the last time.
```

The general form of the for-loop is as follows:

```
for index = values
   block
end
```

The **block** is the body of the for-loop. The for-loop example above conforms to this general form, as Table 2.14 shows.

Table 2.14 Parts of the `for`-loop

EXAMPLE PHRASE	GENERAL FORM
`n`	`index`
`1:10`	`values`
`total = total + n`	`block`

It might be helpful to note the similarities between the syntax of the for-loop and the syntax of the if-statement. Both constructs begin with a control statement and end with the keyword **`end`**. The control statement for each of them begins with a keyword—**`for`** in the for-loop and **`if`** in the if-statement. The semantics of the two constructs are similar as well in that each one has a body that is controlled by the control statement and each one has a control statement that stipulates the number of times that the body will be executed. The difference is that in the case of the if-statement that number is limited to zero or one, whereas in the case of the for-loop the number is zero or some positive integer.

Let's look at second example:

```
N = 5;
list = rand(1,N); % assigns a row vector of random numbers
for x = list
   if x > 0.5
      fprintf('Random number %f is large.\n',x)
   else
      fprintf('Random number %f is small.\n',x)
   end
end
```

It can be seen that this loop conforms to the general form as well, as all for-loops must. First, **`x`** is the index; second, **`list`** contains the values; and third, the if-statement is the **block**. Here is a sample run of this example:

```
Random number 0.141886 is small.
Random number 0.421761 is small.
Random number 0.915736 is large.
```

```
Random number 0.792207 is large.
Random number 0.959492 is large.
```

(Note that, if you run this same code, you will probably get a different result because the function **`rand`** will probably return a different set of values.) Here is what happened:

1. **`N`** was assigned the number **`5`**.
2. **`rand`** was called with the arguments **`1`** and **`5`**, causing it to generate a row vector of five random numbers, **`[0.14189  0.42176  0.91574  0.79221  0.95949]`**, and that row vector was assigned to **`list`**.
3. The control statement assigned the 1st element of **`list`**, **`0.14189`**, to **`x`**.
4. The if-statement found that **`0.14189`** is less than **`0.5`**, triggering the 1st **`fprintf`**
5. The **`end`** was reached and the flow of execution returned to the beginning of the loop.
6. The control statement assigned the 2nd element of **`list`**, **`0.42176`**, to **`x`**.
7. The if-statement found that **`0.42176`** is less than **`0.5`**, triggering the 1st **`fprintf`** .
8. The **`end`** was reached and the flow of execution returned to the beginning of the loop.
9. The control statement assigned the 3rd element of **`list`**, **`0.91574`**, to **`x`**.
10. The if-statement found that **`0.91574`** is greater than **`0.5`**, triggering the 2nd **`fprintf`**.
11. The **`end`** was reached and the flow of execution returned to the beginning of the loop.
12. The control statement assigned the 4th element of **`list`**, **`0.79221`**, to **`x`**.
13. The if-statement found that **`0.79221`** is greater than **`0.5`**, triggering the 2nd **`fprintf`** .
14. The **`end`** was reached and the flow of execution returned to the beginning of the loop.
15. The control statement assigned the 5th element of **`list`**, **`0.95949`**, to **`x`**.

16. The if-statement found that **`0.95949`** is greater than **`0.5`**, triggering the 2nd **`fprintf`** .
17. The **`end`** was reached and the loop ended because the last element of **`list`** had been assigned to **`x`**.

We can learn some things from this second example:

(1) The values assigned to the loop index do not have to be
- integers,
- regularly spaced, or
- assigned in increasing order.

(2) Another control construct can be used in the body of the for-loop.

The first point can be made stronger: Not only do the values assigned to the loop index not have to be integers, they do not even have to be real. And it can be made stronger still: The values do not have to be scalars. If the list of values in the control statement of a for-loop is not a vector, then the index will be assigned the columns of the array, from first to last.

The second point can be made stronger too: ANY control construct can appear in the body of the loop. In our example, an if-statement is used in the body of the for-loop, but a for-loop can be used in the body of another for-loop as well, as we will see in the next section. We refer to the inclusion of one control construct inside another control construct as nesting. In the example above, the if-statement is nested inside a for-loop. We will return to nesting in the following section.

An interesting question arises when the loop index is assigned a value not only by the control statement but also by a statement within the body of the loop: What is the new value of the index on the next iteration? The answer is that it is the next value in the list of values given in the control statement. Assignments to the loop index inside the body of a loop are temporary. They last only during the iteration in which they take place. There is no effect on that list of values or on the next value to be assigned to the loop index by the control statement for the next iteration. Here is an example:

```
total = 0;
for n = 1:10
   n
   n = n + 1;
   total = total + n;
end
fprintf('%d\n',total);
```

Here we have altered our previous example so that (a) the loop index is printed immediately after the control statement (**`n`** without a print-suppressing semicolon) to allow us to see its value, and (2) the loop index is incremented by **`1`**. Here is the resulting output to the Command Window from the first three iterations:

```
n =
     1
n =
     2
n =
     3
```

We don't need to see the rest of the output, because our question is answered by the time we see the second output, **`n = 2`**. It is clear that, despite the fact that the statement **`n = n + 1`** had incremented the loop index to **`2`** during the first iteration, the control statement behaved just as it would have behaved if the index had not been incremented: It set it to the next value in the list, which is **`2`**. As we can see, the same thing happens at the next iteration, where **`n`** is set to **`3`**. (Note, however, that the result is no longer the sum of the first ten positive integers!)

This rule is ironclad. At the beginning of the **`n`**th iteration of every for-loop, the loop's control statement will assign the loop index the **`n`**th term in its list of values, *regardless of any value that may have been assigned to the loop index within the body of the loop during the previous iteration.*

Now that we have handled that question, we move on to another example involving random numbers just to give you a chance to check your understanding:

```
N = 1000;
list = rand(1,N);  % list gets a row vector of random numbers
N_larger = 0;      % initializes a counter
for x = list
   if x > 0.5
      N_larger = N_larger + 1;
   end
end
fprintf('fraction over 0.5 = %f\n', N_larger/N);
```

Try to determine for yourself what this loop will do. (Spoiler alert: The answer is given in the very next paragraph!)

When the loop starts, the index **x** will first be set equal to **list(1)**. As the iterations proceed, it will be assigned successive elements from **list**, a new element at the beginning of each iteration of the loop, ending with **list(1000)** at the thousandth iteration. However, we cannot know what those values will be by looking at the code because they come from the function **rand**, and the output of that function is determined only at run time (i.e., at the time that the function runs). The body of the for-loop, which corre-sponds to **block** in the general form, is executed once at each iteration. It consists of one if-statement. This if-statement will cause **N_larger** to be incremented if and only if **x** is larger than **0.5**. Since **rand** produces numbers that are spread uniformly over the interval from **0** to **1**, we can expect that about half of them will be greater than **0.5**. Thus, after the loop has ended,
N_larger will equal approximately ½ of **N**. Since **N** is **1000**, we can expect **N_larger** to be about **500**, so the fraction **N_larger/N**, which is printed by the **fprintf** statement, should be about **0.5**.

Let's run it and see what happens. Here is the output from one run:

```
fraction over 0.5 = 0.514000
```

As expected, the fraction is close to 0.5. It is only slightly larger. Here is an example of another run:

```
fraction over 0.5 = 0.497000
```

Again, the fraction is close to **0.5**. This time it is slightly smaller. Since the set of random numbers is different each time, we should not be surprised to find that the fraction is different each time. (Whether these differences lie within the expected range is a matter for a statistical analysis.)

Translating an array operation into a for-loop

As we mentioned before, every array operation involves looping. Let's look at an example. Consider these operations:

```
>> u = [5 4 8 8 2];

>> v = [5 5 7 8 8];

>> w = u - v

w =
     0    -1     1     0    -6
```

Here we formed two 5-element row vectors **u** and **v**. We then used array subtraction to subtract the elements of **v** from the corresponding elements of **u** and assigned the resulting vector to **w**. Since **u** and **v** each have five elements, the array operation required that five subtractions be carried out. Let's write a for-loop that performs the same five operations without array subtraction:

```
for ii = 1:length(u)
   w(ii) = u(ii) - v(ii);
end
```

So what can we learn from this very simple example? First, it is easy to see that every array operation and every matrix operation can be translated into an equivalent for-loop version. There is no reason to do it though, because the array operation will typically run faster and it is easier to program than

the equivalent version using explicit looping, but it does drive home the fact that an array operation always requires looping. Second, we see that, even if MATLAB had omitted one of those handy array or matrix operations that we learned about previously, we could still use explicit loops to force it to carry out the same calculation. It's a good thing that explicit loops can do array and matrix operations, because, if explicit looping could not do them, then languages like C++ and Java, which do not provide any array or matrix operators, could not do them at all! In those languages, writing loops is the *only* way a programmer can get array operations done. You are lucky. You have MATLAB to save you all that typing and debugging by providing implicit looping when an array operation is what you need.

Array operations and matrix operations are very convenient. However, for-loops are more powerful than these operations. In addition to being able to do everything that array and matrix operations can do, for-loops can do many things that array and matrix operations cannot do, and, since we often need to do those additional things, MATLAB includes for-loops as part of its language. Our very second loop example above does something that cannot be done with an array operation: It calls **`fprintf`**.

Here is another example of something that cannot be done with array operations. Suppose we want do produce this series:

`1, 1, 2, 3, 5, 8 ….`

The numbers in the series go on forever, and each one can be generated easily. After the two ones at the beginning, each number in the series is the sum of the two preceding numbers. The resulting sequence, **`1, 1, 2, 3, 5, 8, 13, . . .`** is called the Fibonacci sequence, and we will see it again when we study recursion in Functions Reloaded. There is more than one way to produce this sequence, but we want to do it by the method of adding two successive numbers to get the next. Here is code that uses that method to generate the sequence and puts the first **`10`** numbers of the Fibonacci sequence into the vector **`f2`**:

```
N = 10;
f2(1) = 1;
f2(2) = 1;
for ii = 3:N
   f2(ii) = f2(ii-1) + f2(ii-2);
end
```

The result is: **`f2 = [1, 1, 2, 3, 5, 8, 13, 21, 34, 55]`**.

Nested for-loops

As mentioned in the previous section, MATLAB allows the nesting of any control construct inside any other construct. In this section we look at the case in which for-loops are nested. Nested for-loops often occur when we are doing things with two-dimensional arrays. We already know how to do some things with two-dimensional arrays, such as, for example, the calculations we performed with the array operations that were introduced previously. Here is an example of an array operation on a two-dimensional array. First we generate an array:

```
>> A = randi(10,3,4)

A =
    10     2     8     1
     1    10     9     4
     5     1     9     3
```

Here we used the function **`randi`** to produce a 3-by-4 array of random integers between 1 and 10 and assigned it to **`A`**. Now we perform the array operation:

```
>> P = A.*A

P =
   100     4    64     1
     1   100    81    16
    25     1    81     9
```

We have used array multiplication to calculate the product of each element of **A** with itself and have assigned that product to the corresponding element of a new array **P**. This code is an excellent way to do that calculation, but suppose we want to do it without using array multiplication (we have no good reason to avoid array multiplication except to show how nested for-loops work!). Here is code that uses explicit loops to accomplish the same thing for the same array:

```
for m = 1:size(A,1)
    for n = 1:size(A,2)
        P(m,n) = A(m,n)*A(m,n);   <-- body of inner loop
    end
end                               <-- body of outer loop
```

There are two loops here. The first **for** begins the "outer loop", so called because the second loop lies inside it. We call that second loop, the "inner loop" because it is inside another loop. As indicated, the inner loop is the body of the outer loop. The outer loop uses the loop index **m**. It sets **m** to the values given by **1:size(A,1)**. The function call **size(A,1)** returns the number of elements in the first dimension of **A**, which is the number of rows in **A** and equals **3**. So **m** is set to **1, 2, 3,** respectively. Let's look at the first iteration of this loop, when **m = 1**. The body of the outer loop is executed with **m = 1**. The body of the outer loop is itself a for-loop. Its loop index is **n**. It sets **n** to the values given by **1:size(A,2)**. The function call **size(A,2)** returns the number of elements in the second dimension of **A**, which is the number of columns in **A** and equals **4**. So **m** is set to **1, 2, 3, 4** respectively. The body of this inner for-loop is the statement , **P(m,n) = A(m,n)*A(m,n)**; and it does the hard work. In the first iteration of the inner loop, **n** equals **1**, and since the index **m** of the outer loop is also **1**, the body of this inner loop carries out this operation:

```
P(1,1) = A(1,1)*A(1,1);
```

The next iteration of the inner loop has **n = 2**, and since the index **m** of the outer loop is still 1, the body of the inner loop carries out this operation:

```
P(1,2) = A(1,2)*A(1,2);
```

The inner loop continues, updating its index **n** and executing its body two more times:

```
P(1,3) = A(1,3)*A(1,3);
P(1,4) = A(1,4)*A(1,4);
```

Now the inner loop is done, but the outer loop continues its iteration: the control statement of the outer loop sets its loop index **m** to its next value, which is **2** and forces its body, which is the inner loop to execute again. That inner loop sets its loop index **n** to **1, 2, 3, 4** as before and carries out these operations:

```
P(2,1) = A(2,1)*A(2,1);
P(2,2) = A(2,2)*A(2,2);
P(2,3) = A(2,3)*A(2,3);
P(2,4) = A(2,4)*A(2,4)
```

The inner loop is done once again, the outer loop continues its iteration once again, this time setting its loop index **m** to **3**, and the inner loop caries out these operations:

```
P(3,1) = A(3,1)*A(3,1);
P(3,2) = A(3,2)*A(3,2);
P(3,3) = A(3,3)*A(3,3);
P(3,4) = A(3,4)*A(3,4);
```

(and you thought the first example was boring!). Again the inner loop is done, but this time the outer loop is done too. If we now print **P**, we find that it has the same elements as it did when we calculated them with the array operation **P = A.*A**, namely:

```
P =
   100     4    64     1
     1   100    81    16
    25     1    81     9
```

So what have we learned? We have learned that, even if MATLAB had not provided array operations to carry out operations on two-dimensional (i.e. non-vector) arrays, we could still use nested for-loops to carry out the same calculations. In fact, we can always write an explicit loop to do anything that an array operation does, regardless of the number of dimensions of the arrays.

In the example above, the inner loop is the only statement in the body of the outer loop, but the body can contain more than one statement. Here is code that accomplishes the same array operation as the one above but also informs us at each iteration of the outer loop which row it is operating on:

```
for m = 1:size(A,1);
   fprintf('Working on row %d...\n', m);
   for n = 1:size(A,2)
      P(m,n) = A(m,n)*A(m,n);
   end
end
```

Here is what we see when we run it:

```
Working on row 1...
Working on row 2...
Working on row 3...
```

As we mentioned above, for-loops can do things that array operations cannot do, such as call **fprintf**, as this example does. Here is another example. This time **fprintf** occurs in both an outer and inner loop:

```
N = 7;
for mm = 1:N
   for nn = 1:mm
      fprintf('*');
   end
   fprintf('\n');
end
```

Running this program produces this output in the Command Window:

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

The first **fprintf** prints one asterisk each time it is executes. Note that the inner loop in which this **fprintf** lies has an index **nn** that goes from **1** to **mm**. Thus, as **mm** gets larger, the inner loop is repeated more times, and as a result more asterisks get printed in each successive row. The second **fprintf**, which executes only after the inner loop is completed, causes a new-line character to be "printed" to the Command Window, which means that a new line is begun and a new row of asterisks can be printed for this new value of **mm**.

An important area in which array operations are typically too limited to do everything we want to do is image processing. **Image processing** is the generation of a new image from one or more existing images. It is the sort of thing that is done by Photoshop®. Image processing can make new images that are brighter or darker than the input images, increase their contrast, enhance their colors, change their colors entirely, remove noise, make them sharper or more blurry, or rotate them, mirror them, squeeze, or stretch them, combine parts of two images into one new image, and manipulate them in countless other ways. There is almost no limit to the operations that Photoshop can perform, but for-loops can do everything that Photoshop can do—and far more. We will illustrate the idea with a couple examples.

MATLAB makes it easy to work with images by providing some basic functions that allow you to read and write images in standard image-file formats, such as JPEG (Joint Photographic Experts Group), which is a format typically stored in a file named with the extension **jpg**, PNG (Portable Network Graphics), which is typically stored with the filename extension, **png**, GIF (Graphic Interchange Format), stored with the extension **gif**, and several others as well. A complete list can be easily seen by giving the command **`help imwrite`**.

Suppose we have a photo of a bluebird stored in JPEG format in the file **`bluebird_photo.jpg`**. We can view the image this way:

```
>> bluebird = imread('bluebird_photo.jpg');
>> image(bluebird)
>> axis equal
>> axis tight
```

The image appears in Figure 2.29, and we see a bluebird about to enter his house.

The first command reads the image from the file into an array named **`bluebird`**. The second command displays the image, then **`axis equal`** forces the display to have the correct aspect ratio, and finally **`axis tight`** forces the display to omit "empty" white areas above, below, left, or right of the image. The aspect ratio is the ratio of horizontal to vertical distances, and **`axis equal`** sets them properly so that circles in the image file look like circles in the display, instead of ellipses.

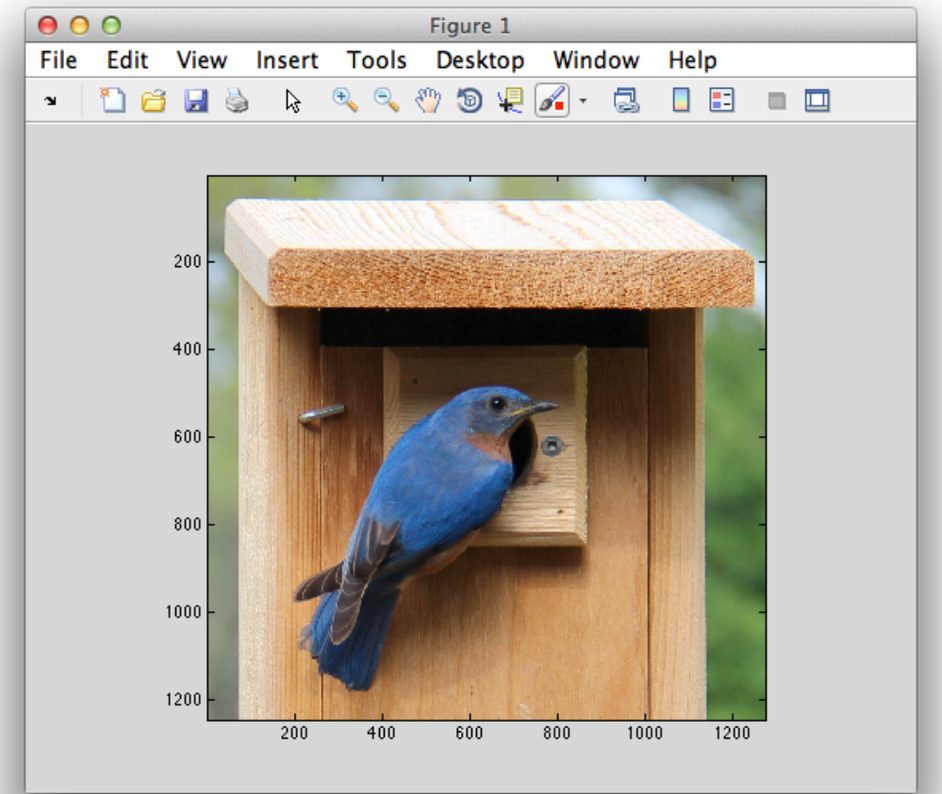
It is interesting to look at the dimensions of **`bluebird`**:

```
>> dims = size(bluebird)

dims =
        1246        1276           3
```

It can be seen that **`bluebird`** is a three-dimensional array. So, to look at individual values within the array, we need three indices. For example,

```
>> bluebird(526,582,1)
ans =
   50

>>  bluebird(526,582,2)
ans =
   78

>> bluebird(526,582,3)
ans =
  141
```

Figure 2.29 Displaying an image in MATLAB



These three numbers give the values of the red component, the green component, and the blue component of the color of the pixel at location 526, 582 (row 526 from the top, column 582 from the left). These components are commonly called the red channel, the green channel, and the blue channel. As we learned at the end of the Introduction to MATLAB, in Chapter 1, a **pixel** is one square piece of a mosaic of colors that make up a digital image, in this case a 1246-by-1276 image, and every pixel consists of three numbers, representing the intensities of red, green, and blue light that together create the color of the pixel in the human eye. Each of these numbers falls in the range 0 to 255, with higher values meaning higher intensity. A black pixel has the values 0,0,0, the brightest pure red pixel has the values 255,0,0, the brightest pure green has the values, 0,255,0, the brightest pure blue pixel has the values 0,0,255, and the brightest pure yellow pixel has the values 255,255,0. Gray pixels have all three channels with equal values, and a pure white pixel has the values 255,255,255. The pixel at location 526, 582, whose three color channels we looked at above, is located within the blue area of the bird, and, not surprisingly, the blue value, 141, is considerably higher than the red and green values of 50 and 78.

Bluebirds are beautiful, but suppose we wanted a red bird. We can do a little image processing to change the bluebird into a red bird. All it takes is a nested for-loop:

```
redbird = bluebird;
dims = size(bluebird);
for ii = 1:dims(1)
   for jj = 1:dims(2)
      if bluebird(ii,jj,3) > ...
                  1.2*mean(bluebird(ii,jj,:))
         redbird(ii,jj,1) = bluebird(ii,jj,3);
         redbird(ii,jj,2:3) = 0;
      end
   end
end
```

Let's see how this code works. The first statement copies all the pixels of **`bluebird`** into **`redbird`**. Then, the nested for-loops allow us to inspect each pixel to see whether it is blue enough to be part of the bluebird's blue feathers. We use the if-statement to determine how blue a pixel is. That if-statement determines whether the blue channel is more than 20% larger than the mean of the three color channels:

```
if bluebird(ii,jj,3) > 1.2*mean(bluebird(ii,jj,:))
```

If so, it sets the red channel's value in redbird's pixel to be equal to the blue channel of bluebird's pixel:

```
redbird(ii,jj,1) = bluebird(ii,jj,3);
```

It then sets the green and blue channels in redbird's pixel to zero, so that there no hint of green or blue (we want some serious red!):

```
redbird(ii,jj,2:3) = 0;
```

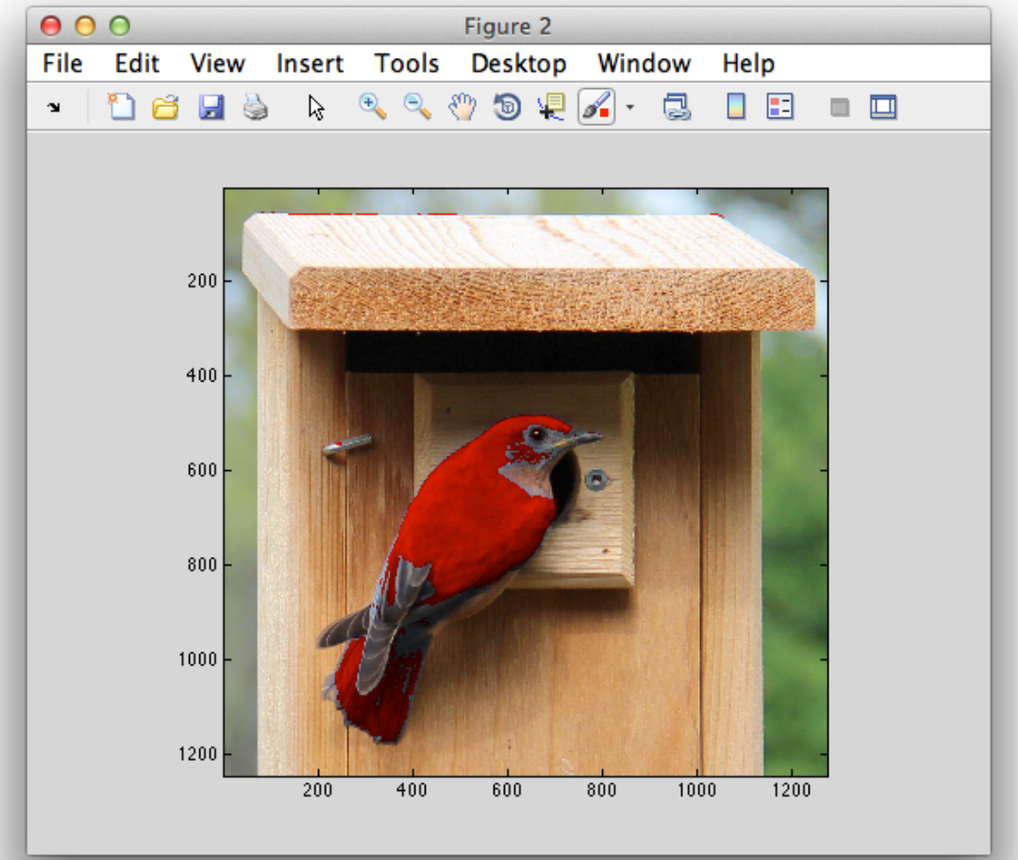
These loops take a bit of time to complete—a few seconds depending on your computer. We can display the new image similarly to the old image:

```
>> image(redbird), axis equal, axis tight
```

Note that we have put the two **axis** commands on the same line with the **image** command, using commas to separate them. The effect is the same as that of issuing commands on separate lines.

The result is shown in Figure 2.30.

Figure 2.30 *Changing colors in nested* **for***-loops*



Now, we can save our work into a standard PNG image file using another handy MATLAB function:

```
>> imwrite(redbird,'red_bluebird.png','png')
```

Caution! This function will overwrite a file that already exists without warning! The first argument to **imwrite** is the image matrix, the second is the file name in single quotes, and the third is the desired file format, also in single quotes. In this case, we decided to use the Portable Network Graphic format—just to show that we can.

This blue-bird-to-red-bird operation can in fact be accomplished with implicit array operations, which we will learn about later in the subsection, Logical Indexing, but it is a bit tricky. It is included as an exercise at the end of this section. If you implement it, you will find that it runs about 30 times faster than the for-loop version, but it is important to understand the for-loop version, because this is the way more sophisticated image processing operations must be handled. Either way, we can alter this simple example to make green birds, yellow birds, black birds, or white birds. In fact we can give that bird any of the 16,777,216 possible colors that can be gotten with the 256 x 256 x 256 combinations of red, blue, and green intensities that are available in the standard image-encoding scheme.

And we can do far more than that. For example, we can perform geometrical operations. A simple operation that is often handy when images are used in publications, is stretching. Suppose, for example, that we have a picture of a flower and we want it to be 20 % wider. Figure 2.31 is the input image showing a tulip, while Figure 2.32 is the output.

Figure 2.31 Input image

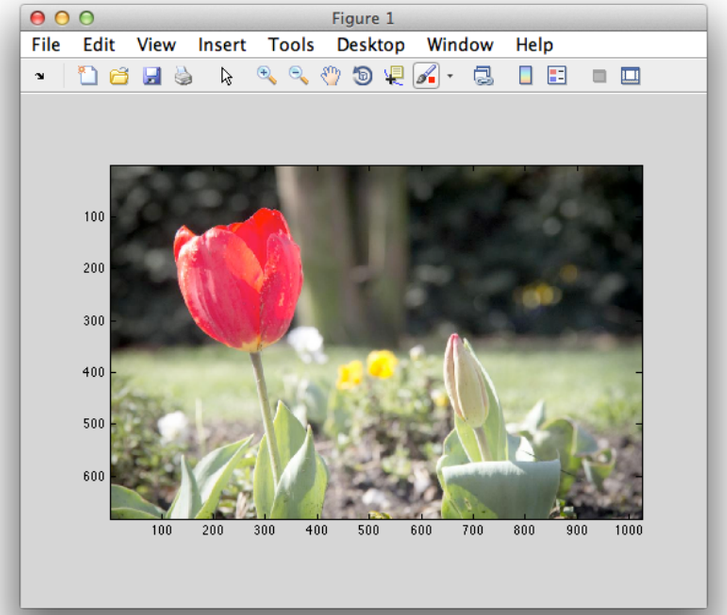
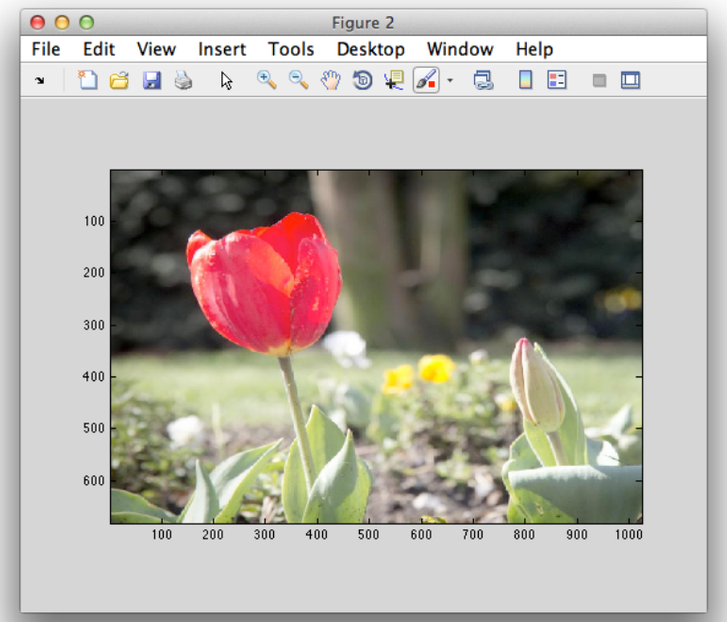


Figure 2.32 Output image



In the output image we have a nice fat tulip, and everything else is wider as well—except for the picture itself. Its pixel dimensions (683 x 1024) are the same as those of the input image. Here is the code that fattened things up:

```
dims = size(flower);
for ii = 1:dims(1)
   for jj = 1:dims(2)
      wide_flower(ii,jj,:) = ...
                flower(ii,round(jj/1.2),:);
   end
end
```

The outer loop with index **`ii`** runs through each horizontal line of the image. The inner loop copies one pixel from **`flower`** into **`wide_flower`** at each horizontal position on the line. That horizontal position is indexed by **`jj`**, and the pixel that is copied into **`wide_flower`** at that position is a pixel from another position on that line of **`flower`**. It is a pixel that is a bit to the left. Here is an example. Suppose we are on line **`10`** (**`ii = 10`**), and we are at horizontal position **`120`** (**`jj = 120`**) on that line. The pixel we choose to copy from **`flower`** is the one on line **`10`** at horizontal position **`100`**. We get that position by dividing **`120`** by **`1.2`** (**`jj/1.2`**), which gives **`100`**, and then rounding **`100`** to the nearest integer, which is **`100`**. No rounding was necessary in this case, but for most pixels rounding is necessary. For example, if we are at horizontal position **`200`** (**`jj = 200`**), dividing by **`1.2`** (**`jj/1.2`**) gives **`166.67`**. If we were to try to use **`166.67`** as an index, we would get a MATLAB error, because all array indices must be integers. When we give **`166.67`** to the function **`round`**, it returns the value **`167`**, which is a perfectly good index. So, the three-element color that is copied into the pixel at position **`10,200`** of **`wide_flower`** is the color of the pixel found at position **`10,167`** of **`flower`**.

Here again, it is possible to produce this result with array operations, but as in the example above, it is important understand the nested for-loop approach to these problems, not only because it will help you see how to design the more efficient array-operation approach but also because it is the only way to perform more sophisticated image-processing tasks.

We have seen two image processing operations here—color changing and size changing, but the number of image-processing operations that we can do is unimaginably large: For a one mega-pixel color image, which is relatively small by today's standards, merely writing down the number of possible operations would require over 100 million digits! Clearly, when it comes to image processing, we are limited only by our imaginations, and we can produce any image we imagine with nested for-loops.

The while-loop

As we have seen, the for-loop is more powerful than array and matrix operations. Now, we are about to meet an even more powerful construct—the **while-loop**. Let's return to the addition of integers that we started with in this section. In our first example, we wanted to know the sum of the first 10 positive integers. The sum turned out to be 55. Suppose instead we wanted to sum the positive integers until we reached the first sum that is greater than 50. Unless we knew that we needed to stop at 10, we could not do this with a for-loop. What we need to solve this problem is a while-loop. Figure 2.33 shows how a while-loop works.

As in the case of the for-loop in Figure 2.28, the while-loop in Figure 2.33 has a control statement, which in this case is "While total <= 50". Once again we have highlighted the body of the loop in green. The two statements in the body are repeated as long as the value of the variable total remains less than or equal to 50. Also, as in Figure 2.28, after the loop ends, the last operation—"Print total"—is executed. In Figure 2.33, however, a second print statement is included. That statement prints the final value of n, which is not known for the while-loop until it has completed its execution. This situation is different from that of the Figure 2.28 because the final value of n can be seen for the for-loop simply by noting the last value in the list of values that it is assigned by the for-loop's control-statement. The value of total that is printed after the while-loop terminates is 55. The value printed for n is 10.

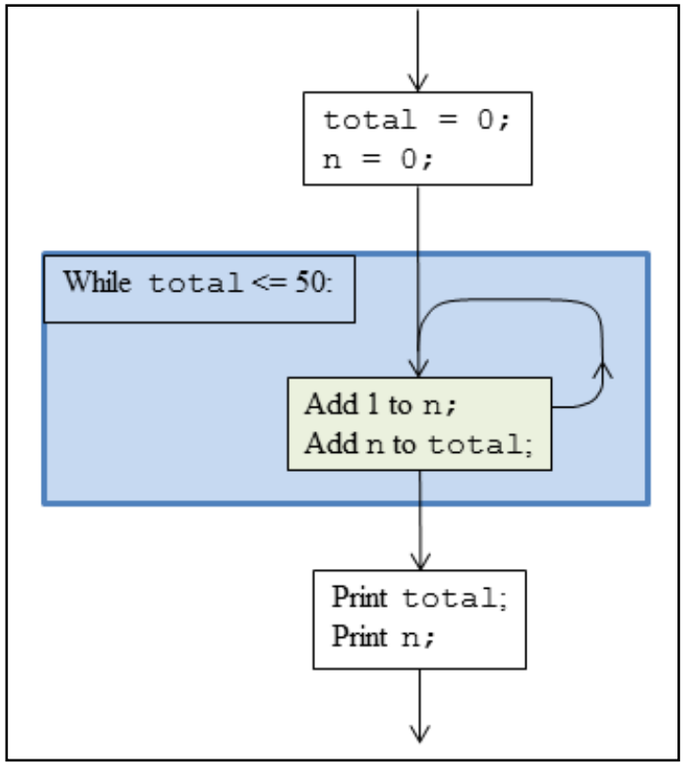
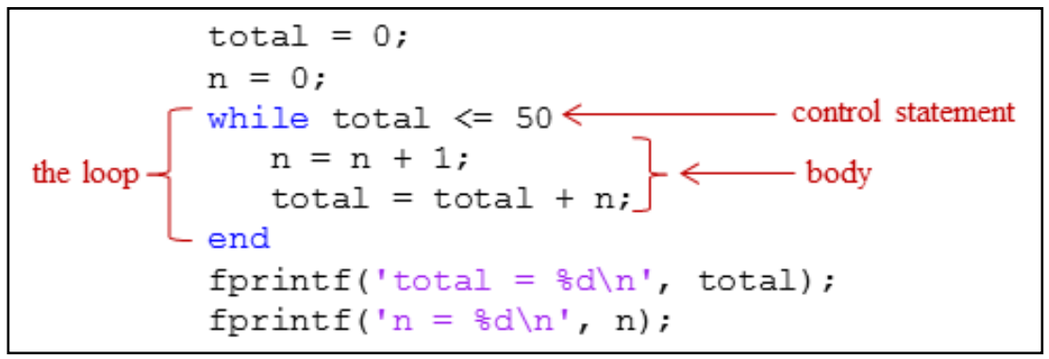


Figure 2.33 *Illustration of a while-loop. A flowchart is shown for a sequence of operations that calculates and prints the smallest sum of the form: 1 + 2 + 3 + … that exceeds 50. Each small box encloses a single operation. The section enclosed in the blue box is a loop. The green box contains the "body" of the loop, which is repeated while the sum is less than or equal to 50.*

A comparison between the for-loop of Figure 2.28 and the while-loop of Figure 2.33 reveals the major difference between the two: Unlike the for-loop, the while-loop has no formal loop index. While we have chosen to use the same variable n in both the for-loop and the while-loop, it is formally a loop index only in the for-loop. In the while-loop, it may be convenient to think of n as a loop index, and we will sometimes refer to a counter like n that enumerates the number of iterations as a "loop index", but it is not part of the syntax of the while-loop, as it is for the for-loop. In particular, it is not a required part of the control-statement of the while-loop. As a result, there are three important differences between the two types of loops concerning a counter, such as n: (1) n must be initialized before the while-loop is entered,

whereas initialization is unnecessary for the for-loop; (2) n must be incremented in the body of the while-loop, whereas incrementing is unnecessary in body of the for-loop; (3) the control-statement of the while-loop does not assign values to n, whereas the control-statement of the for-loop does.

Here is the MATLAB implementation of the loop in Figure 2.33:



and here is the result of running it:

```
total = 55
n = 10
```

With this code we have introduced a new MATLAB keyword—**while**. This keyword signifies the beginning of a while-loop. The keyword **end** signifies the end of the loop. The first line of the loop, "**while total <= 50**", is the control statement of the loop. It controls the body of the loop, which comprises the two statements "**n = n + 1;**" and "**total = total + n;**". The general form of the while-loop is as follows:

```
while conditional
   block
end
```

where **block** is the body of the loop. This form is very similar to the simple if-statement, whose form was given in the previous section and is repeated here:

```
if conditional
   block
end
```

As for the if-statement, the conditional in the while-statement determines whether the statements in the body will be executed. The key difference between the if-statement and the while-statement is that, after the body is executed, the if-statement ends, whereas in the while-statement, the conditional is evaluated again. As long as the conditional is true (i.e., is nonzero), the body will be executed, and re-executed, and re-executed, etc.

While-loops often have nothing that resembles a loop index. Here is an example:

```
y = x
while abs(y^2 - x) > 0.001*x
   y = (x/y + y)/2
end
```

We have omitted semicolons so that the values assigned to y will be printed to the Command Window. Let's set **format long** so that we can see lots of digits and run the code when the value of **x** is **43**:

```
y =

    43

y =

    22

y =

  11.977272727272727

y =

   7.783702777298602

y =

   6.654032912679918

y =

   6.558139638647883
```

To understand what this loop has accomplished, we compute the square of the last **y**:

```
>> y^2

ans =
  43.009195520004582
```

Since the square of **y** is approximately equal to **x**, we see that the loop sets **y** equal to an approximation of the square root of **x**. The quality of the approximation is determined by the conditional. It calculates the absolute value of the difference between **y**-squared and **x**, which should be close to zero if **y** is close to the square root of **x**, and it compares that absolute value to `0.001*x`. If the absolute value of the difference is greater than `0.001*x`., which means that **y**-square differs from **x** by more than one-thousandth part of **x**, then it continues to the next iteration. If not, it stops. We can improve the approximation by reducing the acceptable error from one-thousandth of **x** to, say, one-ten millionth of **x**, i.e. from `0.001*x` to, say, `0.0000001*x`. To do that, we change the control statement to this:

```
while abs(y^2 - x) > 0.0000001*x
```

and rerun. The result is:

```
y =
    43
y =
    22
y =
  11.977272727272727
y =
   7.783702777298602
y =
   6.654032912679918
y =
   6.558139638647883
y =
   6.557438561779193
```

Again calculating the square of the last y, we get:

```
>> y^2

ans =
  43.000000491508771
```

which is closer to **x** than the previous **y**-square. Thus, this new final **y** is a better approximation to the square root of **x**, as we had hoped.

Infinite Loops And Control-c

In the previous section, we got better approximations to the square root by reducing the acceptable level of error. We might hope to get an ideal square root by setting the acceptable level to zero. We can try that by changing the control statement to this:

```
while abs(y^2 - x) > 0
```

Unfortunately, this idea does not work because the value of `abs(y^2 - x)` will never get to zero. What happens is that the value of **y** gets to `6.557438524302000` and then repeats forever, because, when the command, `y = (x/y + y)/2`, is carried out, the result of the calculation on the right of the assignment statement, `(43/6.557438524302000 + 6.557438524302000)/2` is `6.557438524302000`. As a result, **y** is assigned the same value that it had before. The value remains the same instead of getting closer to the exact square root because of the limited accuracy (about 16 decimal places) of the numbers stored in the computer, and it will happen again and again and again. So **y** does not change, and it never will.

MATLAB continues gamely to carry out the body of the loop over and over and over, hundreds of thousands of times, or, if nothing is done to stop it, hundreds of millions of times, for hours, for days, for as long as the power remains on and the computer doesn't wear out, until the sun expands to become a red giant.... Well, you get the picture. A loop that continues iterating

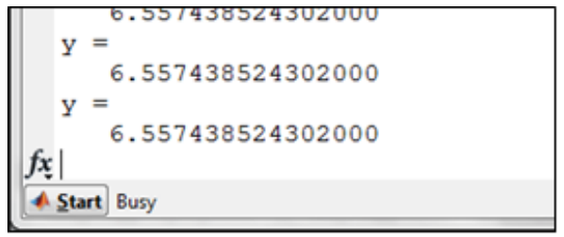
without any possibility of stopping is called an **infinite loop**. When MATLAB is caught in an infinite loop and nothing is being printed to the Command Window, it may at first glance appear that the MATLAB system has died. As a result, the user may attempt to close its window by clicking the icon to close it down. That usually will not work. Of course, there are more extreme steps one can take, such as shutting MATLAB down using the Task Manager in Windows or use Force Quit from the Apple® menu on a Mac. However, such harsh measures are rarely necessary. Typically MATLAB is still alive and working properly, but it has a long (possibly infinitely long) task to perform and is performing its work silently.

Fortunately, in this circumstance MATLAB displays a small sign of life that shows that is still with us. While MATLAB is working on your program, whether it is making good progress or is caught in an infinite loop, it displays the word "Busy" just to the right of the Start button at the bottom left of the Command Window as a signal that it is working. Figure 2.34 is a screenshot of the bottom part of the Command Window while the infinite loop above is being processed. This infinite loop causes repeated printing to the Command Window, but because the same value is printed in exactly the same position over and over, the screen appears to be static, and once again the only way to see that something is happening inside that MATLAB brain is to notice that little word "Busy".

Now that we know MATLAB is not dead, what can we do to get control of it again? Well, any time that we suspect that it is wasting time instead of making progress, we can tell MATLAB to stop running the program that we have given it without killing MATLAB itself. It is done with a control-c. A control-c command is issued by holding the Ctrl key and hitting the c key. Its meaning is roughly, "Abort!".

An infinite while-loop is actually a common occurrence during program development because of programming errors. The most common error being that the programmer forgets to include a statement to increment a counter in the body of the loop. For example, if the statement `n = n + 1` had been inadvertently omitted in Figure 2.33 , the loop would continue indefinitely, this time without anything being printed, and once again we would know that MATLAB was working because of that telltale word "Busy". For either of these infinite loops or for any infinite loop, or if you just want to stop for lunch before MATLAB completes a long program and you need to take your laptop with you—for any case in which MATLAB is busy and you want it to tell it to stop what it is doing, you can do it with control-c.

Figure 2.34 *MATLAB may be stuck in an infinite loop*



Changing The Loop Flow With `break` And `continue`

Sometimes during an iteration of a for-loop or a while-loop, part of the calculation should be skipped. Suppose, for example, that we want to set all the values in the vector named **`readings`** to zero until we reach the first value that exceeds **`100`**. This might be the situation when a sequence of outputs from some device are small values of random noise until a true reading is encountered. For example, suppose the vector has these values:

```
readings = [32 100, 0, 8, 115, 123, 277 92, 14, 87 0 8];
```

The first value that exceeds 100 is 115, so in this case we would want to set the values before 115 to zero:

```
readings = [ 0,  0, 0, 0, 115, 123, 277 92, 14, 87 0 8];
```

We could do this with a while-loop as follows:

```
ii = 1;
while ii < length(readings) && readings(ii) <= 100
   readings(ii) = 0;
   ii = ii + 1;
end
```

but there is better approach.

The break-statement

The above while-statement works, but it seems a bit awkward. Here is a for-loop approach that uses a new construct called the **break-statement**.

```
for ii = 1:length(readings)
   if readings(ii) > 100
      break;
   else
      readings(ii) = 0;
   end
end
```

Here, it appears that we are going to process each element of the vector because of the phrase **`ii = 1:length(readings)`** at the beginning, but that does not happen. As long as the if-statement finds that the value of **`readings(ii)`** is less than or equal to **`100`**, the else-clause dutifully sets that element to zero, but when it reaches the element **`115`**, it enters its if-clause, and there we find a new keyword: **break**. The break-statement can appear only inside a loop. In fact, MATLAB will stop the program and print an error message, if it encounters the keyword **`break`** anywhere that is not inside a loop. (This strict rule is in contrast to C++, which uses the keyword "**`break`**" both as a means to stop a loop and as part of the syntax of its **`switch`**-statement). The meaning of the break-statement is that the loop is ended and control continues at the next statement following that loop. In this case, the loop ends while **`ii`** equals 5, and **`ii`** maintains that value after the loop terminates. So if we wanted merely to know where the first value greater than **`100`** occurred, we could use this code:

```
for ii = 1:length(readings)
   if readings(ii) > 100
      break;
   end
end
fprintf('First reading above 100 is at index %d.\n', ii);
```

The break-statement can be used in a while-statement as well. Here is an alteration of the while-statement above that incorporates a break-statement to accomplish the same goal:

```
ii = 1;
while ii <= length(readings)
   if readings(ii) <= 100
      readings(ii) = 0;
   else
      break;
   end
   ii = ii + 1;
end
```

The difference between this while-statement and the previous while-statement is that the break-statement handles the checking for the magnitude of the **`readings`**, instead of the conditional phrase in the first line of the while-statement. The semicolon after **`break`** is optional.

A common misunderstanding involving the break-statement appears when it is used in nested-loops. The **`break`** applies only to the innermost loop, meaning that it will cause the loop it appears in to terminate, but the outer loop will continue. Here is an example. Suppose we have the following array.

```
A =
    81    10    16    15    65    76    70    82
    90    28    97    42     4    74     4    69
    13    55    95    91    85    39    28    32
    91    95    49    79    93    65     5    95
    63    96    80    95    68    17    10     4
```

We wish to look at each element in row-major order, and set them to zero until we find the first value that is greater than **`90`**. Here is the result we want:

```
A =
     0     0     0     0     0     0     0     0
     0     0    97    42     4    74     4    69
    13    55    95    91    85    39    28    32
    91    95    49    79    93    65     5    95
    63    96    80    95    68    17    10     4
```

It can be seen that, since there are no numbers greater than **90** on the first row, we must set all of its values to **0**. When we get to the second row, we set the first and second elements on the row to zero, and then we encounter **97** at **A(2,3)**. Since this value is greater than **90**, we are done. We leave all the rest of that row and all the following rows as they were.

To look at each element in row-major order, we need a nested for-loop, so let's write one and include a break-statement to stop the processing when we reach a value greater than **90**:

```
for ii = 1:size(A,1)
   for jj = 1:size(A,2)
      if A(ii,jj) <= 90
         A(ii,jj) = 0;
      else
         break;
      end
   end
end
```

The outer loop moves from one row to the next; the inner loop moves across each row, and when we hit a value greater than **90**, we break out of the loop. Sounds good, but which loop do we break out of? Only the inner loop. This does not accomplish what we wanted. Here is the result:

```
A =
     0     0     0     0     0     0     0     0
     0     0    97    42     4    74     4    69
     0     0    95    91    85    39    28    32
    91    95    49    79    93    65     5    95
     0    96    80    95    68    17    10     4
```

How did those erroneous zeros on the third and fifth row get there? Here is what happened: The outer loop starts with **ii = 1**, and the inner loop sets each element of the first row to zero. The outer loop then sets **ii = 2**, and the inner loop sets each element of the second row to zero until the **97** is encountered, causing the break-statement to be executed, which terminates the inner loop. At this point we want the changes to the array to stop, but unfortunately control continues to the next statement after the inner loop, which is the end-statement of the outer loop. The outer loop then loops back to its control statement, which sets **ii** to to **3**. Then the inner loop runs again, setting the first two elements of the third row to zero, after which the break-statement is hit again at **95**. Then **ii** becomes **4** and the break-statement is hit immediately when the value **91** is encountered, leaving the fourth row unchanged. Finally **ii** is set to **5**, and the first element of the fifth row is set to zero, after which the value **96** is encountered, the break-statement terminates the inner loop again, and the outer loop is complete.

What we need to do is terminate *both* loops when the inner loop encounters that **97** on the second row. MATLAB has no mechanism for specifying that multiple loops be terminated when a break-statement is encountered. The only way to cause the outer loop to terminate is to include a statement in the inner loop that writes a note and another statement in the outer loop that reads it. That note, which in common programming terminology is called a **flag**, which is a value indicating a special condition. In this case the special condition is that the value we have been looking for has been found. Here is how we do it:

```
found = false;
for ii = 1:size(A,1)
   for jj = 1:size(A,2)
      if A(ii,jj) <= 90
         A(ii,jj) = 0;
      else
         found = true;
         break;
      end
   end
   if found
      break;
   end
end
```

We have introduced the built-in functions **`false`** and **`true`**. Neither takes any arguments; false returns 0; true returns 1.They are used to show that the values 0 and 1 are being used to signify false and true, and they are often used to set flags. Our flag here is called **`found`**, and we begin by setting it to **`false`**, since nothing has been found before we start looking for it. Then, after we have found the first value that is greater than **`90`** in the inner loop, but just before we break out of the inner loop, we set that flag to **`true`**. A new if-statement, which we have added just after the inner loop, includes a second break-statement, which terminates the outer loop, and we get the result we want.

This flag-setting scheme will always work for you when you want to break out of more than one loop. If the loops are nested three deep, then you need two additional if-statements containing break-statements; deeper nesting requires more if-statements and break-statements, but only one flag is needed, no matter how deep the nesting.

The continue-statement

The break-statement has a complementary construct, called the **continue-statement**. The continue-statement, which consists of the single keyword **continue**, causes the innermost loop to continue to the next iteration without completing the current one. It is used when it is determined, while executing the statements in the body of the loop, that all the subsequent statements in the body of the loop should be skipped. Unlike the break-statement, the continue-statement has no effect on the number of iterations that are carried out. As with the break-statement, the semicolon after **`continue`** is optional.

Suppose, for example, that we want to print five powers of each of the numbers in a list. For each number **`x`** we wish to print **`x`**, **x^2**, **x^4**, **$x^{1/2}$**, and **$x^{1/4}$**. However, we are not interested in complex numbers (for some reason), so, if the number is negative, since the last two values would be complex, we do not want to print (or calculate) them. Here is way to do it without using a continue-statement:

```
for ii = 1:length(numbers)
   x = numbers(ii);
   fprintf('x = %d\n', x);
   fprintf('    x^2 = %d\n', x^2);
   fprintf('    x^4 = %d\n', x^4);
   if x >= 0
      fprintf('    x^(1/2) = %f\n', x^(1/2));
      fprintf('    x^(1/4) = %f\n', x^(1/4));
   end
end
```

In this simple example, the first three **`fprintf`** statements will be executed for each number in the list, but the last two will be executed only if **`x`** is non-negative. Here is how we do it with a continue-statement:

```
for ii = 1:length(numbers)
   x = numbers(ii);
   fprintf('x = %d\n', x);
   fprintf('    x^2 = %d\n', x^2);
   fprintf('    x^4 = %d\n', x^4);
   if x < 0
      continue;
   end
   fprintf('    x^(1/2) = %f\n', x^(1/2));
   fprintf('    x^(1/4) = %f\n', x^(1/4));
end
```

This difference is that, instead of putting the last two **fprintf** statements inside an if-statement that checks to see whether **x** is non-negative, we have used a continue-statement inside an if-statement that checks to see whether **x** is negative. If it is negative, then the rest of the statements in the body of the for-loop (i.e., the last two **fprintf** statements) are skipped.

For example suppose we have this vector,

```
numbers = [7 -2 0 -4 5];
```

Here is what is printed (the version without the continue-statement gives identical output):

```
x = 7
    x^2 = 49
    x^4 = 2401
    x^(1/2) = 2.645751
    x^(1/4) = 1.626577
x = -2
    x^2 = 4
    x^4 = 16
x = 0
    x^2 = 0
    x^4 = 0
    x^(1/2) = 0.000000
    x^(1/4) = 0.000000
x = -4
    x^2 = 16
    x^4 = 256
x = 5
    x^2 = 25
    x^4 = 625
    x^(1/2) = 2.236068
    x^(1/4) = 1.495349
```

For the non-negative elements—the first, third, and fifth elements—all four powers of **x** were printed. For the negative elements—the second and fourth elements—the continue-statement causes the remainder of the body of the loop—the printing of **x^(1/2)** and **x^(1/4)**—to be skipped.

Logical Indexing

Many loop applications involve performing the same operation on each of the elements of an array or on selected elements of it. For-loops and while-loops provide highly versatile means for carrying out such tasks, but those tasks can often be implemented far more simply and more efficiently by means of logical indexing. With logical indexing, the programmer can instruct MATLAB to carry out the equivalent of a for-loop or while-loop without explicitly using the loop syntax at all and with a far simpler set of statements that are easier to program, easier to read, and less prone to error. And they run faster. These simpler statements cause MATLAB to execute an equivalent loop, but in these statements there is no "**for**" or "**while**" keyword, no loop index, no loop body, not even an end-statement. A loop like this, which is invoked without the use of explicit for-loop or while-loop syntax, is called an **implicit loop**, and in this subsection we show how to implement implicit loops via a new concept called logical indexing and a new type of array called the logical array.

Logical indexing with vectors

Logical indexing is a bit easier with vectors than with non-vectors, so we will use only vectors at the beginning. We'll start with an example. Suppose we are given two vectors of the same length, **speed** and **valid**. The elements of **speed** are speeds measured by means of radar of cars selected at random on a busy highway; the elements of **valid** are ones and zeros. Because of the limitations of the radar detector, only some of the readings in **speed** are correct, and the vector **valid** identifies the correct readings. If the value of an element of **valid** is **1**, then the corresponding value at that position in **speed** is a true reading; if the value of the element in **valid** is **0**, then the corresponding speed at that position in **speed** is a false reading and should be discarded. We wish to produce a vector **valid_speed** that contains only the true readings from **speed**.

Let's assign some values to **speed** and **valid**:

```
>> speed = [67, 13, 85];
>> valid = [1, 0, 1];
```

According to the meaning that we have chosen for **valid**, since only the first an third elements of **valid** have the value 1, only the first and the third values in the vector **speed** are true readings, so we would want to put only those values into the vector **valid_speed**. For such a small list of speeds, we could do that explicitly without any fancy loops as follows:

```
valid_speed  = [67, 85]
```

It is important to note that the vector **valid_speed** can be shorter than the vector **speed**. We are not setting invalid speeds to zero here; we are omitting them entirely. Thus, if there are any invalid speeds (at least one value in **valid** is zero), **valid_speed** will be shorter than **speed**. If all of the elements in **valid** had been zeros, then we would have set **valid_speed = []** (the empty matrix), which has a length of zero. If all of them had been ones, we would have set **valid_speed = speed**, which has a length of three. Any other combination of zeros and ones in **valid** would have resulted in **valid_speed** having an intermediate length.

Here is a first attempt at a for-loop to perform this task:

```
for ii = 1:length(speed)
   if valid(ii)
      valid_speed(ii) = speed(ii);
   end
end
```

Let's check the result:

```
>> valid_speed

valid_speed =
    67     0    85
```

This is not what we wanted. Instead of omitting the invalid speed at the second position, we simply set it to zero. How did this happen? There is no zero anywhere inside the for-loop, so we certainly did not set the second element of **valid_speed** to zero on purpose. In fact, we didn't set the second element to anything at all. Since **valid(2)** is equal to zero, the if-statement did nothing on the second iteration. So, we assigned values to only the first and third values of **valid_speed**. However, since we assigned a value to the third element, there must be three elements in this vector and each element must have a value, so MATLAB filled in the missing value. As we saw previously in Chapter 1, it fills in missing values with zeros.

Here is a second attempt:

```
count = 0;
for ii = 1:length(speed)
   if valid(ii)
      count = count + 1;
      valid_speed(count) = speed(ii);
   end
end
```

This time, we added the variable **count** to keep track of the number of elements that we have put into **valid_speed**. When we find a valid entry, we increment **count** and use it as an index in **valid_speed** so that we place the new value immediately after the last element in **valid_speed**.

To check the result, we clear **valid_speed** from the memory, so that we will not be confused by the values that the previous execution gave it (**clear valid_speed**) and then run the new code. Checking the result,

```
>> valid_speed
valid_speed =
    67    85
```

shows that we got what we wanted. However, there is still a subtle bug in our code. What happens if there are no valid speeds? According to our description, **valid_speed** should be an empty matrix. Let's see what happens

in that case by keeping the same speeds, but declaring all of them to be invalid.

```
>> speed = [67, 13, 85];
>> valid = [0, 0, 0];
```

Again we clear **valid_speed** from the memory. Then we run the code again and check the results:

```
>> valid_speed
Undefined function or variable 'valid_speed'.
```

The problem is that, since there are no non-zero values in valid, the body of the if-statement is never executed, so the assignment statement

```
valid_speed(count) = speed(ii);
```

is never executed, so nothing is assigned to **valid_speed**. Since we have cleared **valid_speed** from the memory and since variables remain undefined until something is assigned to them, **valid_speed** is left undefined in this case. We solve this problem easily by adding one line before the loop begins that sets **valid_speed** to the empty matrix, and we include a comment to explain what would otherwise seem to be a useless command:

```
valid_speed = []; % in case there are no valid speeds
count = 0;
for ii = 1:length(speed)
   if valid(ii)
      count = count + 1;
      valid_speed(count) = speed(ii);
   end
end
```

Let's clear **valid_speed** from the memory again, run the code, and check the results for the case in which there are no valid speeds:

```
>> valid_speed
valid_speed =
     []
```

To be sure that the other case works, we rerun with the original **valid = [1, 0, 1]** and check again:

```
>> valid_speed
valid_speed =
    67    85
```

At last we have a loop that works correctly, and it is relatively simple, but here is an even simpler, two-command solution that is much easier to write and much less prone to error:

```
valid_new = logical(valid);
valid_speed = speed(valid_new);
```

There is no for-loop; there is no special case when there are no valid speeds. It seems too simple to work! Let's run this code and check the result:

```
valid_speed =
    67    85
```

So far so good. Now to check the case of no valid speeds. First we clear **valid_speed** from the memory again and set all elements of valid to zero again:

```
>> valid = [0, 0, 0];
```

We run the code again and check the result:

```
>> valid_speed

valid_speed =
   Empty matrix: 1-by-0
```

Success! And success came much easier this time, because we used an implicit loop instead of an explicit one. Here is how it works

```
valid_new = logical(valid);
```

This command converts **valid** into a **logical array** of the same size and shape as **valid** by using the function **logical**, and it assigns the logical array to **valid_new**. The conversion does two things:

1. It replaces any non-zero value in **valid** with the value **1**, leaving only ones and zeros in the array. (Both nonzero values in our example were equal to one, so this step accomplished nothing in this case.)
2. It changes the "type" of the array to **logical**.

We will discuss types a bit more below, and we will learn about types in detail when we get to the section entitled Data Types. For the moment though, we note only that the conversion to the type **logical** is required before the next command is executed.

```
>> valid_speed = speed(valid_new);
```

The syntax of this command makes it appear to be using **valid_new** as a set of indices in **speed**, but, since the type of **valid_new** is **logical**, MATLAB instead treats **valid_new** as a set of indicators to tell it which values of speed are wanted. Each **0** in **valid_new** causes the corresponding element in **speed** to be ignored; each **1** in **valid_new** causes the corresponding element in **speed** to be used.

It might be tempting to try logical indexing with the vector **valid** instead of **valid_new** with its fancy new logical type:

```
>> valid_speed = speed(valid);

Subscript indices must either be real positive integers
or logicals.
```

We got our hands slapped. Let's look in detail at what happened here. Since **valid** is equal to **[1, 0, 1]**, this command is equivalent to

```
valid_speed = speed([1, 0, 1]);
```

The syntax is fine. As we learned in the section entitled Matrices and Operators, it is equivalent to

```
valid_speed = [speed(1), speed(0), speed(1)].
```

The problem is not with the syntax but with the value of the second index—**0**. There is no such thing in MATLAB as **speed(0)** because in MATLAB all indices of all arrays must be real positive integers, and **0** is not positive. However, if we use an array whose type is **logical**, then we are telling MATLAB not to use the values of its elements directly as indices, but instead to use **logical indexing**, which means that a subset of **speed** is selected according to which of the logical indices are zeros and which are ones.

There are other ways to create **logical** arrays. The most important method is the use of the relational operators, **<**, **>**, **==**, **<=**, **>=**, and the logical operators, **&&**, **&**, **||**, **|**, and **~**, which are most commonly used in **if**-statements (see the section entitled Selection). For example, the command:

```
>> c = [2>1, 2<1, ~(3>2 && 4>5)]

c =
     1     0     1
```

makes **c** equal to **[1, 0 1]**, and its type is **logical**. The type of **c** is **logical** because every relational and logical operator produces the logical value **1** when its expression is true and produces the logical value **0** when its expression is false. For the purposes of logical indexing, the type that these operators produce is as important as the value. The type of the values produced by relational and logical operators is always **logical**.

The type of a variable determines the way in which it can be used, as we have seen above when we tried to use the variable **value**, whose type is not **logical**, for logical indexing. Up until now all the variables that we have dealt with have been of a type called **double**. The name "double" has historical origins that are explained in later in the section entitled Data Types, which

is devoted to types and provides detailed definitions of all the standard MATLAB types, but the meaning of **double** in MATLAB is roughly "number". Since MATLAB's applications are typically numerical, its variables typically hold numbers, so most of the time we want variables whose type is **double**, and most of the time that is what MATLAB gives us. It is easy to determine a variable's type by using the function named **class**, as, for example,

```
>> class(c)

ans =
logical

>> class(speed)

ans =
double
```

MATLAB responds by spelling out the type: **double**.

Logical indexing and logical and relational operators

The operators **&** and **&&** each signify the "and" operation and the operators **|** and **||** each signify the "or" operation, but there are three differences:

As we have already learned in the subsection Short Circuiting of Selection, the operators **&**, and **|** unlike **&&** and **||**, do not short-circuit.

Now we will learn another difference: Both **&** and **|** can operate on arrays, whereas **&&** are and **||** can operate only on scalars:

```
>> [1 0 1] & [0 0 1]

ans =
     0     0     1

>> [1 0 1] && [0 0 1]

Operands to the || and && operators must be convertible
to logical scalar values.

>> [1 0 1] | [0 0 1]

ans =
     1     0     1

>> [1 0 1] || [0 0 1]
Operands to the || and && operators must be convertible
to logical scalar values.
```

In fact both **&** and **|** are array operators, meaning that they obey the same rules given in Matrices and Operators for the "dot operators", **.***, **./**, **.^**, etc. regarding the required shapes of their operands, the element-by-element method of evaluation, and the shapes of their outputs. For example, **[1 0 1] & [0 0 1]** above returned **[0 0 1]**, because, according to the rules of array operations, it is equivalent to the element-by-element operation, **[1&0, 0&0, 1&1]**, and **1** and **0** are treated as being equivalent to true and false. Similarly, **[1 0 1] | [0 0 1]** returned **[1 0 1]**, because, according to the rules of array operations, it is equivalent to the element-by-element operation, **[1|0, 0|0, 1|1]**.

A common use of logical indexing is illustrated by the next example. Suppose we have a vector **a = [12, 3, 45]**. We wish to form a new vector consisting of those elements of **a** that are greater than **10**. We do it as follows:

```
>> b = b(a > 10)
b =
    12    45
```

What has happened is that a new vector is formed by the operation **a > 10**. That vector looks like this **[1, 0, 1]** and it is a logical vector (i.e., its type is **logical**). That vector then provides logical indexing into **a**, and the result is that a new vector is formed, as in the example **speed(valid_new)**, consisting of the two elements of **a** that appear at those indices where the values of the logical vector are nonzero (positions 1 and 3). The result, **b**, is not a logical vector. It is of the same type as **a**, which in this case is **double**.

It can be seen from this example how logical indexing can simplify the code. Here is a loop version of `b = a(a>10)`:

```
b = [];% in case no values are greater than 10
jj = 0;
for ii = 1:length(x)
   if a(ii) > 10
      jj = jj + 1;
      b(jj) = a(ii);
   end
end
```

Logical indexing can be used on the left side of the equal sign as well. We introduce left-side logical indexing with a simple example. Once again, we let `a = [12, 3, 45]`. Now we execute the following command:

```
>> a(a>10) = 99

a =
    99     3    99
```

The meaning here is that all elements of `a` that are greater than `10` are set to `99`. The rest are unchanged. Here we note a difference between righthand logical indexing (i.e., to the right of the equal sign) and lefthand logical indexing. With righthand logical indexing, the number of elements may be reduced; with lefthand indexing, the number of elements remains the same.

Once again logical indexing is much simpler than explicit looping, as can be seen by comparing `a(a>10) = 99` with the following semantically equivalent loop:

```
for ii = 1:length(a)
   if a(ii) > 10
      a(ii) = 99;
   end
end
```

In each of the last two examples—one with logical indexing on the right of the equal sign and one with logical indexing on the left of the equal sign, this explicit loop reveals something that is hidden by the far more succinct form of logical indexing: namely, that each element of the vector must be accessed and operated on. With the syntax of logical indexing, it appears that an entire vector is being operated on at once. That is not true. It is not possible with conventional computers, such as the ones that MATLAB typically runs on. MATLAB commands may appear to operate on entire vectors, but in fact they cause hidden loops to be activated by the MATLAB interpreter behind the scenes, and these loops operate on each element of the array—one-by-one.

Logical indexing with arrays

So far, we have used logical indexing only with vectors. Let's look at an example of logical indexing with an array. Our task is to find the elements in the matrix `M` that are greater than 0.1 and replace each one with its square root. First we give a version employing explicit looping:

```
[m,n] = size(M);
for ii = 1:m
   for jj = 1:n
      if M(ii,jj) > 0.1
         M(ii,jj) = sqrt(M(ii,jj));
      end
   end
end
```

Here is a version using logical indexing:

```
M(M > 0.1) = sqrt(M(M > 0.1));
```

This example shows both that logical indexing works with arrays and that logical indexing can be used on both sides of the equal sign in the same command. Note, however, that the number of elements selected on the left must equal the number of elements selected on the right. Equality is guaranteed when the logical expression is identical on both sides, as it is in this example, where the expression is `M > 0.1`.

So far, we have used logical indexing to compare each element in an array against a single scalar value, for example, `0.1` above. We can also compare arrays of elements against arrays of the same size, as in this example:

```
>> A
A =
    89    82    11    53
    33     5    59    42
>> B
B =
    34    44    52    64
    62    73    58    99
>> A((A>B)) = A(A>B) - B(A>B)
A =
    55    38    11    53
    33     5     1    42
```

where each element in **A** that is larger than the element at the corresponding position in **B** is replaced by the difference between the two elements. At this point, it may seem that differences when performing logical indexing are minor between vectors and non-vector arrays (i.e., arrays that are not not two-dimensional with one dimension equal to 1). Unfortunately they are not. With logical indexing, the situation with arrays is more complicated, as can be seen with the following example:

```
>> A = [1 2 3; 4 5 6]

A =
     1     2     3
     4     5     6

>> B = A(A>2)

B =
     4
     5
     3
     6
```

As expected, only those elements of **A** that are greater than **2** made it into **B**, but surprisingly, we find that **B** is a column vector! What is the reason for this change of shape, and how did MATLAB choose the order of the elements in **B**? The answers have to do with the requirement that arrays must be rectangular. In other words, each row must have the same length. If that were not required, then we might well have seen something like this:

```
>> B = A(A>2)
B =
                    3
      4     5       6
```

NOT VALID MATLAB!

In this fictitious output, the first row of **B** has only one element and it is "sitting" at the far right (whatever sitting means!), and the second row of **B** has three elements. This scenario cannot work because later operations with **B** would be poorly defined, so MATLAB does not behave this way. Instead, when logical indexing is applied to an array on the right side of the equal sign, if the array is not a vector, then that array is treated as a column vector in which the elements of the array occur in column-major order. As we learned in subsection Functions for transforming matrices in Matrices and Operators column-major order means that all elements in one column of an array are processed before the elements of the next column. We also learned that linear indexing could be used to access the elements of an array in column-major order and that the index **:** enumerates all of them in that order, as for example,

```
>> A(:)

ans =
     1
     4
     2
     5
     3
     6
```

In the command `B = A(A>2)`, `A` is treated as just such a column vector, and it is this column vector from which elements are selected for `B`. Only those elements that are greater than `2` are selected. Thus `1` is omitted, `4` is selected, `2` is omitted, and `5`, `3`, and `6` are selected. The result is, as we saw above, is

```
B =
     4
     5
     3
     6
```

A difficulty arises when logical indexing is used to compare elements within the array to other elements in it. Of course, we can easily set to zero all the elements in some matrix `X` that are smaller than, say, the second element on the third row, as in this example:

```
X =
    35    35    34    23    17
    18     3    31     6    10
    37    31    30    41    31

>> X(X<X(2,3)) = 0

X =
    35    35    34     0     0
     0     0    31     0     0
    37    31     0    41    31
```

But, if you wish to use logical indexing to set every element of the array `A` to zero that is the smaller than, say, the second element in its own row, you are going to need help from a built-in function. Here is a way do it by using the function `repmat`, which we introduced in the section Matrices and Operators of Chapter 1:

```
function A = zero_small(A)

A2ii = repmat(A(:,2),1,size(A,2));
A(A<A2ii) = 0;
```

The variable `A2ii` is an array containing only copies of the second column of `A`. The copying is done by `repmat`, which makes its living by copying vectors and arrays (`rep`licating `mat`rices). It is helpful in situations like this where it is necessary to compare every element in a matrix to elements in one column of the same matrix or of another matrix, or to elements in one row or in some rectangular tile of a matrix. This code is much simpler than that required to solve the problem with nested for-loops, and it is shorter, but it can be even shorter than this. One line is all that is required:

```
A(A<repmat(A(:,2), 1, size(A,2))) = 0;
```

For comparison, here is a version that uses explicit loops:

```
function A = zero_small_explicit(A)
for ii = 1:size(A,1)
   for jj = 1:size(A,2)
      if A(ii,jj) < A(ii,2)
         A(ii,jj) = 0;
      end
   end
end
```

There is no appreciable savings in execution time realized by using implicit looping for this problem, but the implicit version can save both programming time and debugging time, which in many cases is more important to the programmer. Of course, as this example shows, in order to avoid explicit looping, we sometimes have to be resourceful, and that requires experience and a facility with vectorization.

Vectorization

As we mentioned at the beginning of this section, MATLAB provides many methods for invoking an implicit loop, including the colon operator, array and matrix operations, and built-in functions that operate on arrays. Logical indexing is just our latest example of implicit looping.

In the previous subsection, we encountered commands that operate on entire vectors and entire arrays. Examples include `b = a(a>10)`, `B =A(A>2)`, and our latest example, `A(A<repmat(A(:,2), 1, size(A,2))) = 0`. A command that operates on an entire vector or entire array is said to be a **vector command** (the more general term "array" command is not used), and the translation of code from a version that uses explicit looping into one that uses implicit looping via a vector command is called **vectorization**. In every case of vectorization, at least one explicit loop is avoided. Looping is still happening, because, unless parallel processing is used, only one element of an array can be accessed by the CPU at a time (the interested reader can look up MATLAB's versatile parallel-processing capability via `help parfor` and `help gpu`). However, implicit looping via vectorization can save considerable execution time relative to explicit looping because the MATLAB interpreter does not have to slavishly follow the programmer's explicit loop. Instead, it parcels out the task into one or more built-in functions that have been optimized to run very efficiently on the CPU.

As we saw in the last example of the previous subsection, the function `repmat` can be very helpful when we want to vectorize our code. Another function that helps with vectorization is `ndgrid`. To see how it works, we will use an extended example. Let's start with the picture of a dog and some flowers shown in Figure 2.35.

Let's suppose the dog's name is Obie. If we wanted to focus the viewer's attention on Obie, we might want to crop the picture, which means that we would remove all of the picture except, say, the portion around Obie's face. Let's suppose that we have determined that the center of his face is on row 900 and column 2500, and that we want to get a rectangular region whose top and bottom edges are a vertical distance of 850 pixels from that center position and whose left and right edges are a horizontal distance of 650 pixels from it. That can be done easily by loading the picture, which, as noted in Figure 2.35, is stored in a file named **dog_and_flowers.jpg**, into an array, which

Figure 2.35 dog_and_flowers.jpg

we might call `dog_flowers`, and then copying a rectangular portion of that array into another array, which we might call `dog_cropped`, as follows:

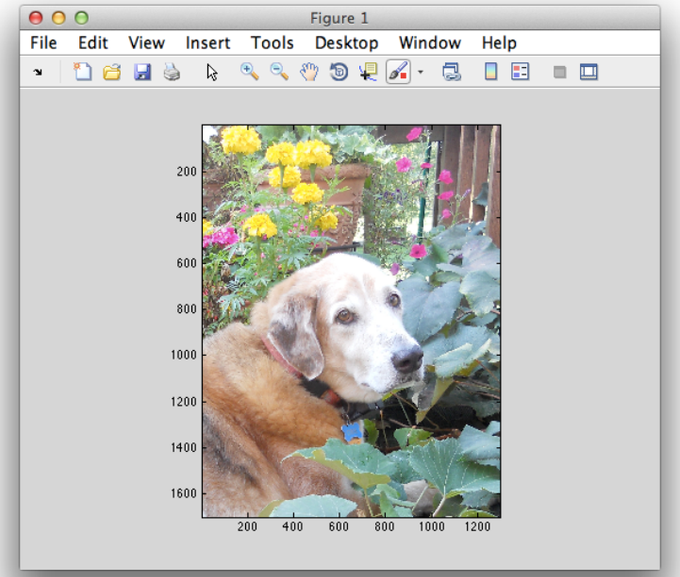
```
>> dog_flowers = imread('dog_and_flowers.jpg');
>> c1 = 900; c2 = 2500; r1 = 850; r2 = 650;
>> dog_cropped = dog_flowers(c1-r1:c1+r1, c2-r2:c2+r2, :);
>> figure(1); image(dog_cropped);
>> axis equal; axis tight;
```

Here, `c1` and `c2` are the row and column indices of the center pixel, `r1` is the distance of the top and bottom of the cropped rectangle we want from the center pixel, and `r2` is the distance of its left and right edges from the center. pixel.

(Note that `axis equal` and `axis tight` have been explained earlier in this chapter). The result is shown in Figure 2.36.

Not bad, but this is just not artistic enough for us. If we were to use a fancy image-processing tool like, say, Photoshop, we could put an elliptically shaped mask around Obie. But wait! Didn't we say that we can do anything

Figure 2.36 dog_cropped



with MATLAB that we can do with Photoshop? Well, here is a function that does elliptical masking:

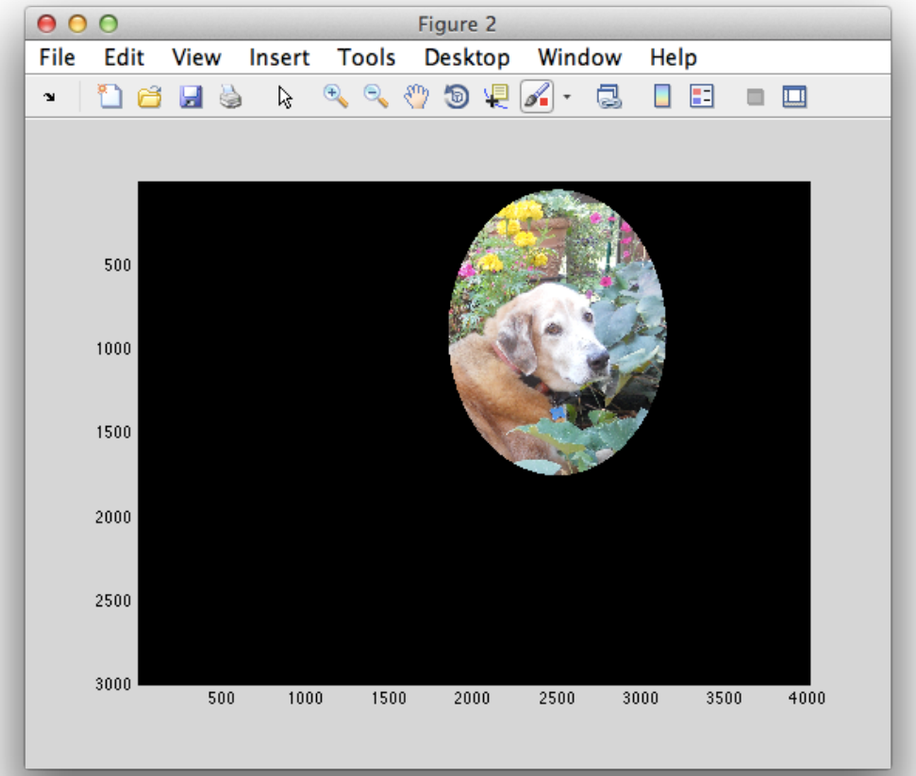
```
function im = elliptical_masker(im,c1,c2,r1,r2)
[M,N,K] = size(im);
for i1 = 1:M
   for i2 = 1:N
      if ((i1-c1)/r1)^2 + ((i2-c2)/r2)^2 >= 1
         im(i1,i2,:) = 0;
      end
   end
end
```

Let's use it first and then explain what it does:

```
>> dog_masked = elliptical_masker(dog_flowers,c1,c2,r1,r2);
>> figure(2); image(dog_masked);
>> axis equal; axis tight;
```

The result is shown in Figure 2.37. What we have done is set all pixels to black (red, green, and blue channels equal to zero) outside an ellipse centered on element 900,2500 of the array **dog_flowers** using a vertical radius (semi-major axis), **r1**, of 850 and a horizontal radius, **r2**, of 650. (By the way, MATLAB, like Photoshop, will let you click on the image to specify positions and shapes, instead of forcing you enter numbers. You can do do it easily with a built-in function called **ginput**, which is introduced in the Practice Problems at the end of this section). The determination that we are outside is done by calculating **((i1-c1)/r1)^2 + ((i2-c2)/r2)^2**. If the result is greater than or equal to 1, then we are outside the ellipse. That determination is made by the if-statement. We can now crop, as we did before, and save the result:

Figure 2.37 dog_masked



```
>> dog_masked_cropped = dog_masked(c1-r1:c1+r1,c2-r2:c2+r2,:);
>> imwrite(dog_masked_cropped,'Obie.jpg');
```

and we end up with the nice result displayed in Figure 2.38.

***Figure 2.38** Obie.jpg*

We chose a black background, but we could make it white too, by changing `im(i1,i2,:) = 0` to `im(i1,i2,:) = 255` (any color is possible). So, MATLAB ≈ Photoshop.

There is just one small problem. This code is a bit slow. For the array **dog_flowers**, whose dimensions are 3000-by-4000-by-3, `elliptical_masker` takes 15.4 seconds on a MacBook Air with a 1.7 GHz CPU and 4 GB of memory running Mac OS X. This is a tolerable length of time, given that we are processing a 12-megapixel image, but we can speed it up. One trick would be to crop first and mask second. That would exploit vectorization (fast) to reduce the size of the array that we need to send to `elliptical_masker`, thereby reducing the number of iterations of its explicit loops (slow). Instead, we will keep the order the same and completely vectorize `elliptical_masker`, thereby eliminating *all* explicit looping.

We wish to assign zero to a subset of array elements without using explicit loops. Well, we have done just that via logical indexing in the previous subsection, in the function `zero_small`. In that function, we employed vectorization in the statement `A(A<A2ii) = 0`, which zeroed every element in the array `A` that was smaller than the element at the corresponding position in the array `A2ii`. That statement fits a common vectorization pattern using logical indexing: (1) `A(B) = x`, where `x` is some scalar value, (2) `A` and `B` have the same dimensions, and (3) `B` is a logical array. For the specific statement, `A(A<A2ii) = 0`, we have `B` equal to `A<A2ii`, and therefore the value of `B(i1,i2)` (true or false) is determined by whether or not the value of `A(i1,i2)` is less than `A2ii(ii,jj)`. In our image-processing problem, we want to do a similar thing: (1) `im(B) = 0`, where (2) `im` and `B` have the same dimensions and (3) `B` is a logical array. So what's new? Plenty!

There is an important difference between this problem and the problem solved by `zero_small`, and, for that matter, all the previous logical-indexing problems that we have seen. In our current problem, the value of the element `B(i1,i2)` (true or false) has nothing whatever to do with the *value* of the corresponding element `im(i1,i2)`. Instead, it depends only on the *position* of the element in `A`. In other words, it depends only on `i1`, and `i2`.

What we need is a matrix `B`, that is the same size as `im` and for which `B(i1,i2)` equals true if `((i1-c1)/r1)^2 + ((i2-c2)/r2)^2 >= 1` and equals false otherwise. This is where `ndgrid`, comes in. We can make `ndgrid` return arrays whose element have values that depend only on their positions within the array. Before we do that, though, let's look at an example that demonstrates the basic functionality of `ndgrid`:

```
>> [a1, a2] = ndgrid([4, 2, 56, 5], [-9, 0.3, 8])

a1 =
     4     4     4
     2     2     2
    56    56    56
     5     5     5
a2 =
   -9.0000    0.3000    8.0000
   -9.0000    0.3000    8.0000
   -9.0000    0.3000    8.0000
   -9.0000    0.3000    8.0000
```

The function **ndgrid** takes any number of vectors (row or column) as input arguments, in this case two, and it returns the same number of arrays as output arguments by duplicating the input vectors. The output arguments all have identical dimensions, and those dimensions are equal to the lengths of the input vectors, in this case 4-by-3, because the lengths of the input vectors are 4 and 3. Let's call those input vectors **v1** and **v2**. Then the output arrays, **a1** and **a2**, have these values:

a1(i1,i2) = v1(i1).

a2(i1,i2) = v2(i2).

By looking at **a1** and **a2** printed above, we can see how **ndgrid** gets its name: for **n d**imensions it produces a **grid** from each input vector. Pretty simple. And, as we about to see, we can use this simple functionality to make **ndgrid** give us arrays, each of whose elements has a value that depends on its position in the array. Let's look at an example that moves us further in that direction:

```
>> [I1,I2,I3] = ndgrid(1:5,1:7,1:2)

I1(:,:,1) =
     1     1     1     1     1     1     1
     2     2     2     2     2     2     2
     3     3     3     3     3     3     3
     4     4     4     4     4     4     4
     5     5     5     5     5     5     5
I1(:,:,2) =
     1     1     1     1     1     1     1
     2     2     2     2     2     2     2
     3     3     3     3     3     3     3
     4     4     4     4     4     4     4
     5     5     5     5     5     5     5
I2(:,:,1) =
     1     2     3     4     5     6     7
     1     2     3     4     5     6     7
     1     2     3     4     5     6     7
     1     2     3     4     5     6     7
     1     2     3     4     5     6     7
I2(:,:,2) =
     1     2     3     4     5     6     7
     1     2     3     4     5     6     7
     1     2     3     4     5     6     7
     1     2     3     4     5     6     7
     1     2     3     4     5     6     7
I3(:,:,1) =
     1     1     1     1     1     1     1
     1     1     1     1     1     1     1
     1     1     1     1     1     1     1
     1     1     1     1     1     1     1
     1     1     1     1     1     1     1
I3(:,:,2) =
     2     2     2     2     2     2     2
     2     2     2     2     2     2     2
     2     2     2     2     2     2     2
     2     2     2     2     2     2     2
     2     2     2     2     2     2     2
```

This time we gave **ndgrid** three input vectors, and, since the dimensions of each output array comprise the lengths of three input vectors, the output arrays are three-dimensional: 5-by-7-by-2. We made another change from our first **ndgrid** example too: The input arguments are vectors of the form **1:N**.

The significance of this choice is that the elements of such vectors can represent sequenced array indices. Let's look at the outputs. **I1** , **I2**, and **I3** are each 5-by-7-by-2 arrays because 5, 7, and 2 are the lengths of the three input vectors. Let's denote the input vectors by **v1**, **v2**, and **v3**. Then, since **v1 = 1:5**, **v2 = 1:7**, and **v2 = 1:2**, we find that

I1(i1,i2,i3) is equal to **v1(i1)**, which equals **i1**.

I2 (**i1,i2,i3**) is equal to **v2(i2)**, which equals **i2**.

I3 (**i1,i2,i3**) is equal to **v3(i3)**, which equals **i3**.

So, we have achieved our goal of producing arrays whose values depend on their positions in the array, and, as we will see, this particular dependence is very versatile. Let's use it on an array of random numbers:

```
>> rng(0); A = randi(99,5,7,2)
A(:,:,1) =
    81    10    16    15    65    76    70
    90    28    97    42     4    74     4
    13    55    95    91    85    39    28
    91    95    49    79    93    65     5
    63    96    80    95    68    17    10
A(:,:,2) =
    82    44    49    28    50    75    95
    69    38    45    68    96    26    55
    32    76    64    65    34    51    14
    95    79    71    17    58    70    15
     4    19    75    12    23    89    26
```

We have used **randi** to produce a 5-by-7-by-2 array **A** of random integers chosen from the range 0 to 99. It is no coincidence that we gave **A** the same dimensions as the output arrays **I1**, **I2**, and **I3**. We chose the inputs to **ndgrid** carefully to produce output arrays that would have exactly the same dimensions that we had planned to use for **A**. Now, without further ado, let's use these arrays to put vectorization to work on **A**.

First vectorization example: Set all the elements of the first four columns on both pages of **A** equal to 0:

```
>> A(I2<=4) = 0

A(:,:,1) =
     0     0     0     0    65    76    70
     0     0     0     0     4    74     4
     0     0     0     0    85    39    28
     0     0     0     0    93    65     5
     0     0     0     0    68    17    10
A(:,:,2) =
     0     0     0     0    50    75    95
     0     0     0     0    96    26    55
     0     0     0     0    34    51    14
     0     0     0     0    58    70    15
     0     0     0     0    23    89    26
```

Perhaps you are not impressed, since **A(:,1:4,:) = 0** would accomplish the same thing via the colon operator and is also vectorized. Granted. But let's see you do our second vectorization example with the colon operator.

Second vectorization example:

```
>> rng(0); A = randi(99,5,7,2);
>> A(I2<I1) = 0

A(:,:,1) =
    81    10    16    15    65    76    70
     0    28    97    42     4    74     4
     0     0    95    91    85    39    28
     0     0     0    79    93    65     5
     0     0     0     0    68    17    10
A(:,:,2) =
    82    44    49    28    50    75    95
     0    38    45    68    96    26    55
     0     0    64    65    34    51    14
     0     0     0    17    58    70    15
     0     0     0     0    23    89    26
```

The logical expression **I2<I1** produces a 5-by-7-by-2 array whose elements

have the value true only if they are located at a position in the array for which the first index is less than the first. Using it in `A(I2<I1) = 0` allows us to set all the elements in the lower triangle below the diagonal to zero, a feat that the colon operator cannot match. Still not impressed? Man, you are a tough audience. OK, our third example has got to impress you (hopefully!):

Third vectorization example:

```
>> c1 = 3; c2 = 4;
>> D = (I1-c1).^2 + (I2-c2).^2
D(:,:,1) =
    13     8     5     4     5     8    13
    10     5     2     1     2     5    10
     9     4     1     0     1     4     9
    10     5     2     1     2     5    10
    13     8     5     4     5     8    13
D(:,:,2) =
    13     8     5     4     5     8    13
    10     5     2     1     2     5    10
     9     4     1     0     1     4     9
    10     5     2     1     2     5    10
    13     8     5     4     5     8    13
```

To set the values of the 5-by-7-by-2 array `D`, we have used the formula for the square of the distance of each element within a page from the element in the center of that page (whose first two indices are `c1 = 3` and `c2 = 4`). That squared distance starts at zero at the center of the page and gets larger as we radiate outward from the center making it all the way up to 13 at the corners. Now we form a logical array by comparing each value in `D` to 1:

```
>> D>1
ans(:,:,1) =
     1     1     1     1     1     1     1
     1     1     1     0     1     1     1
     1     1     0     0     0     1     1
     1     1     1     0     1     1     1
     1     1     1     1     1     1     1
ans(:,:,2) =
     1     1     1     1     1     1     1
     1     1     1     0     1     1     1
     1     1     0     0     0     1     1
     1     1     1     0     1     1     1
     1     1     1     1     1     1     1
```

The array returned by the expression `D>1` is a logical array for which elements that are within one unit of distance from the center of the page have the value false (`0`), while the elements greater than one unit of distance away have the value true (`1`). Now let's use logical indexing to set the distant elements of our matrix `A` to 0:

```
>> rng(0); A = randi(99,5,7,2);
>> A(D>1) = 0

A(:,:,1) =
     0     0     0     0     0     0     0
     0     0     0    42     0     0     0
     0     0    95    91    85     0     0
     0     0     0    79     0     0     0
     0     0     0     0     0     0     0
A(:,:,2) =
     0     0     0     0     0     0     0
     0     0     0    68     0     0     0
     0     0    64    65    34     0     0
     0     0     0    17     0     0     0
     0     0     0     0     0     0     0
```

We have "masked" the elements outside a circle. That is almost what we wanted to do with the image of Obie. The only differences are that (a) the array holding the image is a lot bigger than `A` and (b) we want our mask to be elliptical. Let's extend our third example to our image-processing problem by

rewriting **elliptical_masker** to use **ndgrid**, incorporating a formula for an ellipse:

```
function im = elliptical_masker(im,c1,c2,r1,r2)
[M,N,K] = size(im);
[I1,I2] = ndgrid(1:M,1:N,1:K);
D = ((I1-c1)/r1).^2 + ((I2-c2)/r2).^2;
im(D >= 1) = 0;
```

Here again we form arrays of appropriate dimensions by giving **ndgrid** input vectors, **1:M**, **1:N**, and **1:K**, that correspond to the dimensions **M**, **N**, and **K**, of the array, **im**, that we wish to mask. The masking formula is just a bit more complicated, because an ellipse is just a bit more complicated than a circle, but, other than that, the process is exactly the same as the one we applied above to **A**. Let's use our vectorized version of `elliptical_masker`:

```
>> dog_masked = elliptical_masker(dog_flowers,c1,c2,r1,r2);
>> figure(2); image(dog_masked);
>> axis equal; axis tight;
```

The result is exactly the same as before (Figure 2.37), but this time, because we have used vectorized code, the execution time is 4 seconds instead of the 15.4 seconds that was required with the explicit looping of the non-vectorized code—a speed-up factor of 3.85.

Before we leave this dog, let's use **ndgrid** to apply a second popular embellishment: vignetting. Vignetting is the gradual reduction of the intensity of an image with the increase in distance from some point in the image. We will apply vignetting to the dog image using the same center as before and with the same elliptically shaped pattern for the reduction of intensity. Here are two versions of a function that accomplishes this vignetting—the first one using explicit looping and the second one using vectorization via **ndgrid** to avoid explicit looping.

```
function im = vignetter(im,c1,c2,r1,r2)
[M,N,K] = size(im);
for i1 = 1:M
   for i2 = 1:N
      d = ((i1-c1)/r1).^2 + ((i2-c2)/r2).^2;
      dim_factor = exp(-1.5*d);
      im(i1,i2,:)= ...
            uint8(double(im(i1,i2,:))*dim_factor);
   end
end
```

```
function im = vignetter(im,c1,c2,r1,r2)
[M,N,K] = size(im);
[I1,I2] = ndgrid(1:M,1:N,1:K);
D = ((I1-c1)/r1).^2 + ((I2-c2)/r2).^2;
Dim_factor = exp(-1.5*D);
im = uint8(double(im).*Dim_factor);
```

The output is identical for the two functions and is shown in Figure 2.39. The final cropped version, saved in JPEG format, is displayed in Figure 2.40.

Figure 2.39 *dog_vignetted*

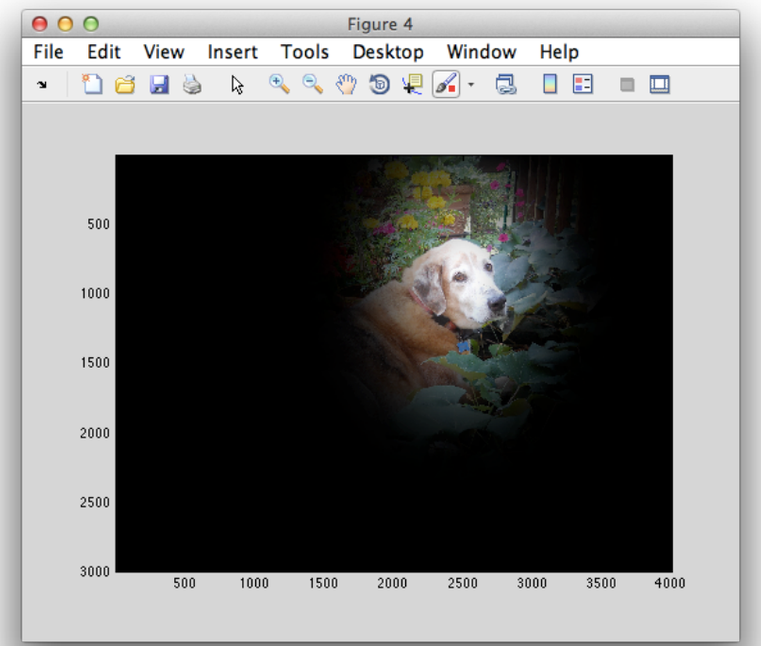
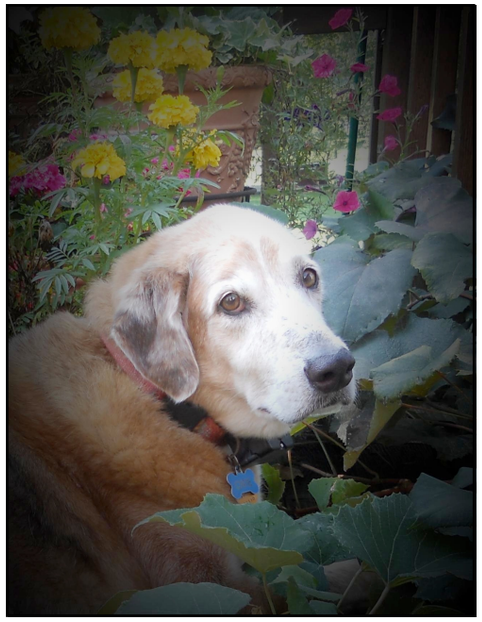


Figure 2.40 **Obie vignette.jpg**



We must briefly digress from our explanation of the vectorization aspect of these functions to provide a note of explanation for the appearance of **uint8** and **double** in these two functions. These are so-called conversion functions, which are used to convert a number from one "data type" to another. These functions are formally introduced in Data Types, where we will learn that there are many types of numbers and that multiplication of two different types is not allowed. Ordinary numbers, like those in **dim_factor** and **Dim_factor**, are of the type **double**, which we discussed in the previous subsection, while the type of image arrays like **im** is called "**uint8**", so, before the multiplication can be done, one of them must be converted. The type **uint8** allows only whole numbers, so the best choice is to convert **im** to **double**. The resulting **double** type must be converted back to **uint8** before it is stored back into **im**.

We have used the same formula for the ellipse that we used in **elliptical_masker** above, but, instead of masking by means of logical indexing, we have multiplied the values in **im** by a dimming factor. In the first version of **vignetter**, that factor is a scalar, **dim_factor**, which is calculated for each pixel and applied to it inside nested for-loops, whereas in the second version it is an array **Dim_factor**, which contains all the factors and is applied to the entire image by means of array-multiplication. The speed-up factor is more dramatic here: the vectorized version requires 5.26 seconds, while the version with explicit looping requires 78.1 seconds. Here vectorization has produced a sped-up factor of 14.9.

Before we leave **ndgrid**, we will point out that it has a sister function called **meshgrid**, which behaves the same, except that the output arguments are transposed. It is a bit of a weak sister, because it works only with two-dimensional and three-dimensional arrays. However, since higher dimensional arrays are rare, its weakness is not serious. Its (slight) advantage is that the transposed outputs allow the programmer to use the first input argument to represent the horizontal direction and the second to represent the vertical direction, which is the traditional *x*-*y* order for images.

The functions **ndgrid** and **meshgrid** are not the only functions that will help you vectorize your code. MATLAB provides an assortment of functions to assist you. Some of the more helpful ones are listed in Table 2.15.

Table 2.15 *Helpful functions for vectorization*

FUNCTION	DESCRIPTION
`all`	determine whether all elements are zero
`any`	determine whether any elements are zero
`find`	find indices and values of nonzero elements
`ind2sub`	convert linear index to multidimensional indices (subscripts)
`meshgrid`	generage arrays by duplicating vectors
`ndgrid`	generate arrays by duplicating vectors
`permute`	rearrange dimensions of an array
`repmat`	replicate an array
`reshape`	rearrange the elements of an array into a new shape
`shifdim`	shift the dimensions of an array

Limitations of vectorized commands

Code that relies on vectorized commands can provide dramatic speed-ups, but such commands share one common limitation: They cannot replace commands whose results depend on the order in which the elements are processed. Thus, for example, in `sqrt([1 2 3])` the CPU may find the square root of `1`, then `2`, and then `3`, or it may find them in the reverse order—`3`, `2`, `1`—or in any order—with exactly the same result. The order in which they are presented when they are returned is important `(1, 2, 3)`, but not the order in which the operations on them are performed. Another way to look at this limitation is that it must be permissible (if not practical) to perform all the operations simultaneously—each one on a separate CPU. This restriction means that operations on arrays in which the results depend on the order in which the elements are processed must be carried out with explicit loops. A simple example is a while-loop to find the first element in a vector that is greater than 10. Each element must be compared with 10, but the determination of the first one that is greater requires sequential operations, not simultaneous operations. In fact, while there are many for-loops that cannot be replaced by an implicit loop, it is a good rule of thumb that while-loops are a bit less likely than for-loops to be replaceable by implicit loops.

While there are limitations to vectorization, there are many instances in which it results in a significant simplification in the code required to accomplish a task. Furthermore, the vectorized version typically runs faster than the loop version, because the vectorized version takes advantage of highly efficient, built-in, implicit looping and also takes advantage of multiple processors (cores), when they are available. Because of the simplification in code and the resulting speed-up in programming, the easy accessibility to vectorization in MATLAB represents a major advantage of this language in numerical applications. Skill at writing vectorized MATLAB code is essential for time-consuming applications, and that skill comes both with practice and by the observation of experienced MATLAB programmers. (A slight disadvantage of vectorized code is that it sometimes requires more memory.)

Saving Memory Allocation Time

Often during the execution of a loop, an array is built up one element at a time, as in the following example, which creates an array of products of integers:

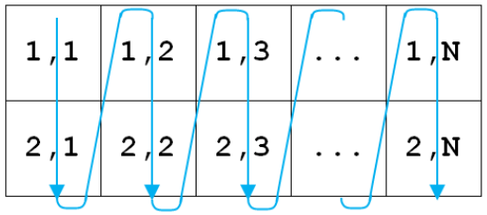
```
N  = 3000;
for ii = 1:N
   for jj = 1:N
      A(ii,jj) = ii*jj;
   end
end
```

There are nine million multiplications here. That's a lot of work for a human, but it is a mere trifle for MATLAB running on a typical computer. On even a low-end laptop, which carries out billions of operations in a second, nine million multiplications should take much less than a second. However, on a 2.67 GHz Dell Latitude E6410 with 7.8 gigabytes of usable memory (used for timings throughout this subsection), these six lines required almost 25 seconds! Something seems very wrong here, and indeed something is very wrong. This code is very inefficient. The problem has to do with memory allocation for the two-dimensional array. It is a very common problem, and to avoid it, we must understand it. We will see what is happening in the next subsection, and then we will learn two ways to avoid the problem, not only in this example but in any application involving two-dimensional arrays.

Wasting time with re-allocation

Let's examine what happens each time the command `A(ii,jj) = ii*jj` is executed. First, the values of `ii` and `jj` must be retrieved from memory, then they must be multiplied, and finally, the result must be stored at the position `A(ii,jj)` in the array. Nine million of those operations would take only about one-tenth second, but there is another matter that must be attended to each time a new value is stored in `A(ii,jj)`: the array must get larger, and that is what is taking so much time. Let's consider what happens after, say,

the second row is completed. At this point the array is laid out in the memory in the order shown by the blue line below,

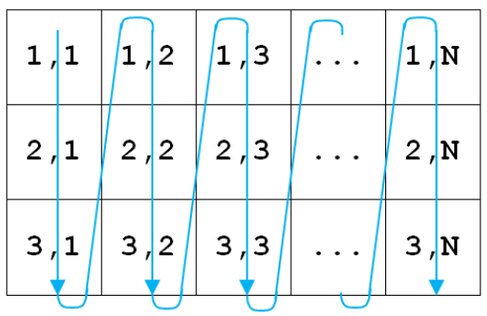


where `N = 3000`.

In other words, the elements of the two-row array are laid out in consecutive memory locations in this order:

```
(1,1)
(2,1)
(1,2)
(2,2)
(1,3)
(2,3)
...
(1,3000)
(2,3000)
```

Now we are ready to add element `(3,1)`, the first element on the third row. But where does it go? To see where it belongs, we must look at the layout of an array with three rows. An array with three rows is laid out like this:



In other words, the three-row array is laid out in consecutive memory locations in this order:

```
(1,1)
(2,1)
(3,1)
(1,2)
(2,2)
(3,2)
(1,3)
(2,3)
(3,3)
...
(1,3000)
(2,3000)
(3,3000)
```

where the red type indicates places where new elements should be inserted between the existing elements of the two-row array. Unfortunately such insertions are impossible because there is no room between consecutive memory locations. Instead, space must be re-allocated for the array. To do that, it is necessary to set aside enough room for a new 3000-by-3 array and then to copy all 6,000 elements of the 3,000-by-2 array into the new array. Only after that is done can the newly calculated first element on the third row be written into its appropriate position, followed by the other elements on the third row. This process must be repeated for each iteration of the outer loop. In other words, in addition to the calculations that are obvious in the code, a 3000-column array must be copied to a new array, not once, not twice, but 2,999 times!

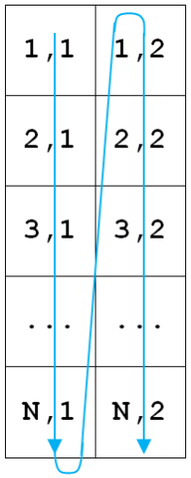
There are two ways to make this process more efficient—far more efficient: (1) Change the order of the for-loops or (2) Pre-allocate the array `A` (or both). We will take up these two approaches in turn.

Re-organizing to reduce re-allocation time

The insertion problem above happened because the array is stored in column-major order while in the code the new elements are added in row-major order. We can very easily re-organize the loops to switch to column-major order, and that can be accomplished as follows:

```
N  = 3000;
for jj = 1:N
   for ii = 1:N
      A(ii,jj) = ii*jj;
   end
end
```

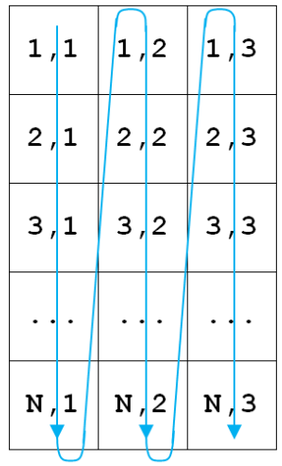
All we did was change **ii** to **jj** in the control statement of the first for-loop and **jj** to **ii** in the control statement of the second for-loop. As a result of this simple change, the code takes just over one-half second—a speed-up by a factor of about 45. To see why, we note that this time, the inner loop is moving down the columns. Let's consider what happens after the second column is completed. At this point the array is laid out in the memory like this:



In other words, the two-column array is laid out in consecutive memory locations in this order:

```
(1,1)
(2,1)
(3,1)
...
(3000,1)
(1,2)
(2,2)
(3,2)
...
(3000,2)
```

Now we are ready to add element **(1,3)**. Let's see where it goes. A three-column array is laid out like this:



In other words, the three-column array is laid out in consecutive memory locations in this order:

```
(1,1)
(2,1)
(3,1)
...
(3000,1)
(1,2)
(2,2)
(3,2)
...
(3000,3)
(1,3)
(2,3)
(3,3)
...
(3000,3)
```

where the red type indicates places where new elements must be inserted. So in this case, no insertion of new elements between the existing elements is required, because each new element is added at the end. Space is easier to add at the end of the array, and there is no need to copy the old array into a new one. Avoiding all that copying saves all that copying time.

Pre-allocation

Sometimes it may not be feasible to re-arrange nested loops for more efficient space allocation. Fortunately there is an easier way, and it is even more efficient: **pre-allocation**. Pre-allocation is the allocating of space for an entire array before calculating the values to put into the array. It is done in MATLAB by calling a function, typically the function named **`zeros`**, which was introduced at the beginning of Programmer's Toolbox. Here is pre-allocation in action:

```
N = 3000;
A = zeros(N);
for ii = 1:N
   for jj = 1:N
      A(ii,jj) = ii*jj;
   end
end
```

Except for the insertion of the statement **`A = zeros(N)`**, this code is identical to our original code for this problem (i.e., with the original inefficient loop-index order). The function **`zeros`** allocates space for an array and sets each element to zero. As we learned before, if there is one input argument, as in this case, the allocated array is square, in this case **`N`**-by-**`N`**. If there are two arguments, **`M`** and **`N`**, it returns an **`M`**-by-**`N`** array of zeros. Thanks to this pre-allocation of the entire array **`A`** before our calculations begin, there is no need whatever for re-allocation during the loop iterations. This code runs even faster than the code above—about 0.19 seconds, which is a speed-up factor of about 130. The use of **`zeros`** is the most common (by far) way to accomplish pre-allocation. Its use is a hallmark of the experienced MATLAB programmer.

Combining pre-allocation with the efficient for-loop indexing order produces the fastest code, because there is greater efficiency in the way that MATLAB accesses arrays in column-major order (i.e., even when no re-allocation is necessary). The code looks like this:

```
N = 3000;
A = zeros(N);
for jj = 1:N
   for ii = 1:N
      A(ii,jj) = ii*jj;
   end
end
```

and it runs in 0.06 seconds for a speed-up factor of almost 400.

Pre-allocation is always a good idea, and it is crucial when large arrays are involved, but it is not always possible, because it is not always possible to know in advance (i.e., before the loops run) what the size of the array will be. This can happen when an array is being constructed inside a while-loop, as in the following example:

```
N = 3000;
ii = 0;
while rand > 1/N
   ii = ii + 1;
   for jj = 1:N
      A(ii,jj) = ii + jj^2;
   end
end
```

Here, the outer loop continues to iterate as long as the random numbers returned by **`rand`** are greater than 1/3000, and each of those iterations adds a new row to the matrix, causing the problem of re-allocation inefficiency that we faced for the nested for-loop examples. We had two ways to solve the problem then—(1) re-order the loops and (2) pre-allocation. This time we can do (1), but we cannot do (2) because, while we know that each row of the matrix **`A`** will have 3,000 elements, we cannot know how many rows it requires. Thus, we know that we need to allocate a matrix of size *M*-by-3000, but we don't know *M*. The conditional **`rand > 1/3000`** in this example has no special meaning. It merely represents an unpredictable conditional—one whose outcome cannot be known until the code runs. When the command **`rng(2)`** is issued to initialize the random-number generator before this code is exe-

cuted, there are 7,810 iterations of the while loop, meaning that we would have needed to use `A = zeros(7810,3000)` as our pre-allocation statement. Unfortunately, for an unpredictable conditional, we cannot know what to allocate until the program has ended. Without pre-allocation, this loop takes over two and a half minutes (160 seconds)! Clearly we are having allocation problems again!

When pre-allocation is not possible and allocation inefficiency rears its head, it is even more important to order the loops to employ column-major order if possible. It is possible here, and here is how to do it:

```
N = 3000;
ii = 0;
while rand > 1/N
   ii = ii + 1;
   for jj = 1:N
      A(jj,ii) = ii + jj^2;
   end
end
A = A';
```

The only differences are that (a) `A(ii,jj)` has been changed to `A(jj,ii)`, to change re-allocations to column-major-order but produces a transposed version of the matrix and (b) the matrix is transposed after the looping is done. This version runs in 1.5 seconds—a speed-up factor over 100. Thus, even when pre-allocation is not possible, understanding the re-allocation problem allows us to alter our code to reduce that problem dramatically.

In this last example, we managed to get our computer to do 46.9 million arithmetic operations done and stored in 1.5 seconds—a rate of over 31 million per second, and when we were able to do pre-allocation, in our double for-loop example, we managed to get nine million multiplication done and stored in 0.06 second. That's a stunning rate of 150 million operations per second! Loops are clearly a powerful way to get things done, but, as we have seen, to get the most efficient code, it is necessary to understand a bit of what is going on behind the scenes and then to know how to take advantage of that knowledge. Now that we have done that, we know how, as promised at the beginning of this section, to make a computer go fast—really fast.

Additional Online Resources

- Video lectures by the authors:
  - Lesson 6.1 For-Loops (38:50)
  - Lesson 6.2 While-Loops (20:16)
  - Lesson 6.3 Break Statement (29:31)
  - Lesson 6.4 Logical Indexing (37:29)
  - Lesson 6.5 Preallocation (8:59)

Concepts From This Section

Computer Science and Mathematics:

- loops
- iteration
- implicit looping
- loop index
- for-loop
- while-loop
- infinite loop
- nested loops
- break-statement
- continue-statement
- vectorization
- row-major order
- column-major order
- re-allocation of arrays
- preallocation of arrays

MATLAB:

- **`for`**-loop
- **`while`**-loop
- **`break`**-statement
- **`continue`**-statement
- logical arrays
- the **`logical`** function
- logical indexing
- column vector version of **`A`** using **`A(:)`**

Practice Problems

for-loops

Problem 1. Write a function called **`vector_square`** that takes one vector as an input argument (the function does not have to check the format of the input) and returns one row vector as output. If it is called like this,

```
v2 = vector_square(v1)
```

then **`v2 = v1.^2`**. However, this function must not use an array operation. It must instead use a for-loop.

?

Problem 2. Write a function called **`vector_multiply`** that takes two vectors of the same length as input arguments (the function does not have to check the format of the input) and returns one row vector as output. If it is called like this, **`v3 = vector_multiply(v1,v2)`**, then **`v3 = v1.*v2`**. However, this function must not use an array operation. It must instead use a for-loop.

Problem 3. Write a function called **`vector_algebra`** that takes three vectors of the same length as an input arguments (the function does not have to check the format of the input) and returns one row vector as output. If it is called like this, **`a = vector_algebra(x,y,z)`**, then **`a = x.^2 + y.*z`**. However, this function must not use any array operations. It must instead use a for-loop.

?

Problem 4. Write a function called **summit** that takes three vectors of the same length as input arguments (the function does not have to check the format of the input) and returns a scalar as output. If it is called like this,

```
a = summit (x,y,z)
```

then

```
a = x(1)^2 + y(1)*z(1) + x(2)^2 + y(2)*z(2) + . . . +
x(N)^2 + y(N)*z(N),
```

where **N** equals the length of **x**. This function must not use an array operation, and it must not use the function **sum**. Instead, it must instead use a for-loop.

Problem 5. Write a function called **is_between** that takes three vectors of the same length as input arguments (the function does not have to check the format of the input) and returns 1 or 0 as output. If it is called like this, **a = is_between (x,y,z)**, then **a** equals 1, if each element of **y** lies between the corresponding elements of **x** and **z** and **a** equals 0 otherwise. The function must use a for-loop and must include either a break-statement or a return-statement to make the loop efficient. Here are four examples of the function in action:

```
>> is_between([5 -33 9],[5.5 2 40],[6, 2.3, 41])
ans =
     1
>> is_between([5 -33 9],[4.5 -34 8],[0 -34.6 5])
ans =
     1
>> is_between([5 -33 9],[5.5 3 40],[6, 2.3, 41])
ans =
     0
>> is_between([5 -33 9],[5 2 40],[6, 2.3, 41])
ans =
     0
```

?

Problem 6. Write a function called **exp_approx** that takes two scalars as input arguments, the second of which is a non-negative integer (the function does not have to check the format of the input) and returns a scalar as output. If it is called like this, **ea = exp_approx(x,N)**, then **ea** is equal to the following approximation of **exp(x)**:

```
ea = sum(x.^(0:N)./factorial((0:N))).
```

However, this function must not use an array operation. It must instead use a for-loop.

Problem 7. Write a function called **cosine_approx** that takes two scalars as input arguments, the second of which is a non-negative integer (the function does not have to check the format of the input) and returns a scalar as output. If it is called like this, **ca = cosine_approx (a,N)**, then **ca** is equal to the following approximation of **cos(a)**:

```
ca = sum((-1).^(0:N).*a.^(2*(0:N))./ ...
                                  factorial(2*(0:N)))
```

However, this function must not use an array operation. It must instead use a for-loop.

?

Problem 8. Write a function called **`nat_log_approx`** that takes two scalars as input arguments, the first of which is positive and is less than two and the second of which is a non-negative integer (the function does not have to check the format of the input) and returns a scalar as output. If it is called like this, **`nla = nat_log_approx(x,N)`**, then **`nla`** is equal to the following approximation of **`log(x)`**:

```
nla = sum((-1).^(2:N+1).*(x-1).^(1:N)./(1:N))
```

However, this function must not use an array operation. It must instead use a for-loop.

Problem 9. Write a function called **`sine_approx`** that takes two scalars as input arguments, the second of which is a positive integer (the function does not have to check the format of the input) and returns a scalar as output. If it is called like this, **`sa = sine_approx (a,N)`**, then **`sa`** is equal to the following approximation of **`sin(a)`**:

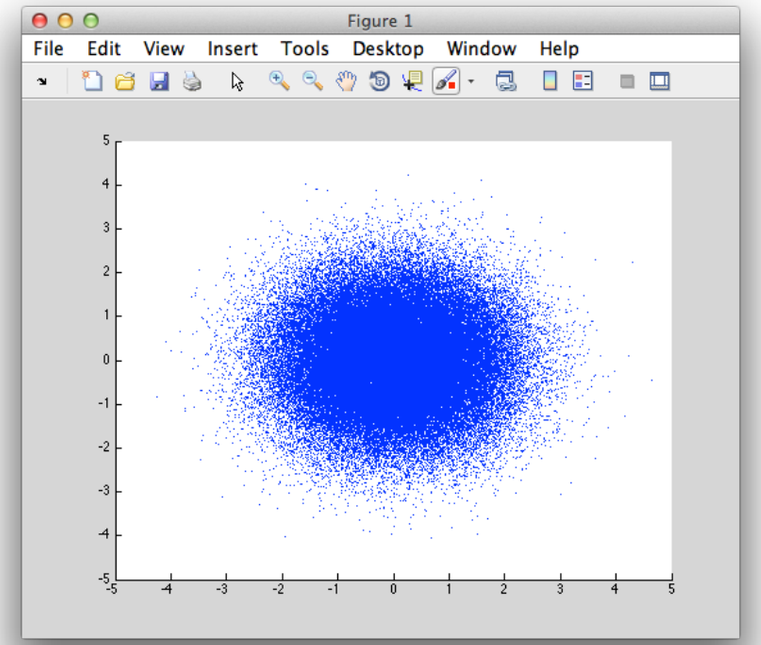
```
sa = sum((-1).^(0:N).*a.^(2*(0:N)+1)./ ...
                          factorial(2*(0:N)+1))
```

However, this function must not use an array operation. It must instead use a for-loop.

?

Problem 10. Write a function called **`point_cloud`** that takes one scalar as an input argument (the function does not have to check the format of the input) and has no output argument. If it is called like this, **`point_cloud (100)`**, then it plots 100 points. Each point has a random x coordinate and a random y coordinate, each of which is gotten by a call to **`randn`**, which uses a normal distribution with a standard deviation equal to 1. The range of the plot axes should be −5 to 5 in both the x and y dimensions. The grid should be turned off. The points should be plotted and displayed one at a time by calling **`plot`** with only one point specified and, following the call of **`plot`**, by a call of **`drawnow`**, which causes the point to be plotted immediately. The command **`hold on`** should be included so that all previous points are retained when a new point is plotted. Figure 2.41 shows an example view of the plot after **`point_cloud(100000)`** has completed its point-by-point plotting on a Mac. (Note that on Windows the points are much larger. Also note that it takes a long time to plot this many points with **`drawnow`**. Finally, try zooming in the middle.)

Figure 2.41 *Problem 10*



while-loops

Problem 11. Write a function called **`steps`** that takes one scalar as an input argument (the function does not have to check the format of the input) and returns one scalar as an output argument. If it is called this way, **`n = steps(d)`**, then it sets **`n`** equal to the minimum number of steps required to move from a distance of one foot away from a wall to a distance less than **`d`** feet from the wall. The first step is ½ foot. If a second step is required, it is ¼ foot. If a third step is required, it is 1/8 foot, and so forth, with each subsequent step being half the distance of the step before. Here is a sample run:

```
>> n = steps(0.001)
n =
    10
```

?

Problem 12. Write a function called **`leaps`** that takes one scalar as an input argument (the function does not have to check the format of the input) and returns one scalar as an output argument. If it is called this way, **`n = leaps(d)`**, then it sets **`n`** equal to the minimum number of leaps required to move from a wall to a distance at least **`d`** feet from the wall. The first step is 1 foot. If a second step is required, it is 2 feet. If a third step is required, it is 3 feet. If a fourth step is required, it is 4 feet, and so forth. Here are some sample runs:

```
>> leaps(0)
ans =
     0
>> leaps(1)
ans =
     1
>> leaps(100)
ans =
    14
```

Problem 13. Write a function called **`just_enough`** that takes one scalar as an input argument (the function does not have to check the format of the input) and returns one scalar as an output argument. If it is called this way, **`n = just_enough(x,N)`**, then it sets **`n`** equal to the smallest non-zero integer such that **`n*exp(x) ≥ N^x`**. Here are two sample runs:

```
>> n = just_enough(1,10)

n =
     4

>> n = just_enough(4.5,10)

n =
   352
```

?

Problem 14. Write a function called **`just_enough_logs`** that takes one scalar as an input argument (the function does not have to determine whether the input is a scalar) and returns two scalars as output arguments. If it is called this way, **`[n, total] = just_enough_logs(x)`**, it checks the value of **`x`**, and if it is not greater than 1, it prints an error message (see below), sets **`n`** and **`total`** equal to zero, and returns; otherwise, it sets **`n`** equal to the smallest integer such that **`n*log x > 5`** and sets **`total`** equal to **`n*log x`**. Here are two sample runs:

```
>> [n,total] = just_enough_logs(0.4)
Input must be > 1.

n =
     0

total =
     0

>> [n,total] = just_enough_logs(2.1)

n =
     7

total =
       5.1936
```

Problem 15. Write a function called **`big_abs_normal`** that takes one scalar as an input argument (the function does not have to check the format of the input) and returns one integer as an output argument. If it is called this way,

```
n = big_abs_normal(d)
```

then it repeatedly calls the function **`randn`** counting the number of calls it must make until the absolute value of the random number is at least as large as **`d`**. Here are some sample calls. Before the sample calls are made, the random number generator is initialized by means of **`rng`**. In this case, it is initialized with the number **`41`** (non-negative integer). The purpose of this use of **`rng`** is to make it possible to compare your results with those below.

```
>> rng(41)
>> big_abs_normal(0)

ans =
     1

>> big_abs_normal(3)

ans =
   686

>> big_abs_normal(4)

ans =
        1494
>> big_abs_normal(5)

ans =
       94520
```

?

Problem 16. Write a function called **one_so_small** that takes no input arguments and returns three scalars as output arguments. If it is called this way, **[a,b,c] = one_so_small**, then it repeatedly gets three numbers from the function **rand** until the numbers pass this test: ten times one of the numbers is smaller than the product of the other two numbers. For example, if the three random numbers are 0.4, 0.03, and 0.9, then ten times the second number is 0.3 and the product of the first and third numbers is 0.36, so ten times one of these three numbers is smaller than the product of the other two numbers. On the other hand, if the second number were 0.038, these three numbers would not pass the test. Here are two sample calls. Before the sample calls are made, the random number generator is initialized by means of **rng**. In this case, it is initialized with the number 12 (non-negative integer). The purpose of this use of **rng** is to make it possible to compare your results with those below.

```
>> rng(12)
>> [a,b,c] = one_so_small
a =
     0.014575
b =
      0.91875
c =
      0.53374
>> [a,b,c] = one_so_small
a =
     0.033421
b =
      0.95695
c =
      0.90071
>>
```

Problem 17. Write a function called **guess_my_number** that takes no input arguments and returns no output arguments. Instead, it gets it input via the function **input**. It repeatedly asks for a number with the phrase, "Try to guess my number: " until the user enters the number 42. If the number is low, the message "Higher" is printed on one line. If the number is high, the message, "Lower" is printed on line. If the user enters 42, the message "That's it!" is printed, the loop is ended, and the function returns. Here is an example run:

```
>> guess_my_number
Try to guess my number: 32
Higher
Try to guess my number: 45
Lower
Try to guess my number: 42.000000001
Lower
Try to guess my number: 42
That's it!
```

?

Problem 18. Write a function called **`number_pattern`** that takes no input arguments and returns no output arguments. Instead it gets it input via the function **`input`**. It asks for a number repeatedly until the user puts in the same number twice in a row. The loop that it uses must be a while-loop. Here is an example run:

```
>> matching_number
Please input a number: 4
Please input number (I'm looking for a pattern): 5
Sorry, that's not what I'm looking for.
Please input number (I'm looking for a pattern): 6
Sorry, that's not what I'm looking for.
Please input number (I'm looking for a pattern): 7
Sorry, that's not what I'm looking for.
Please input number (I'm looking for a pattern): 7
That's it. Well done!
```

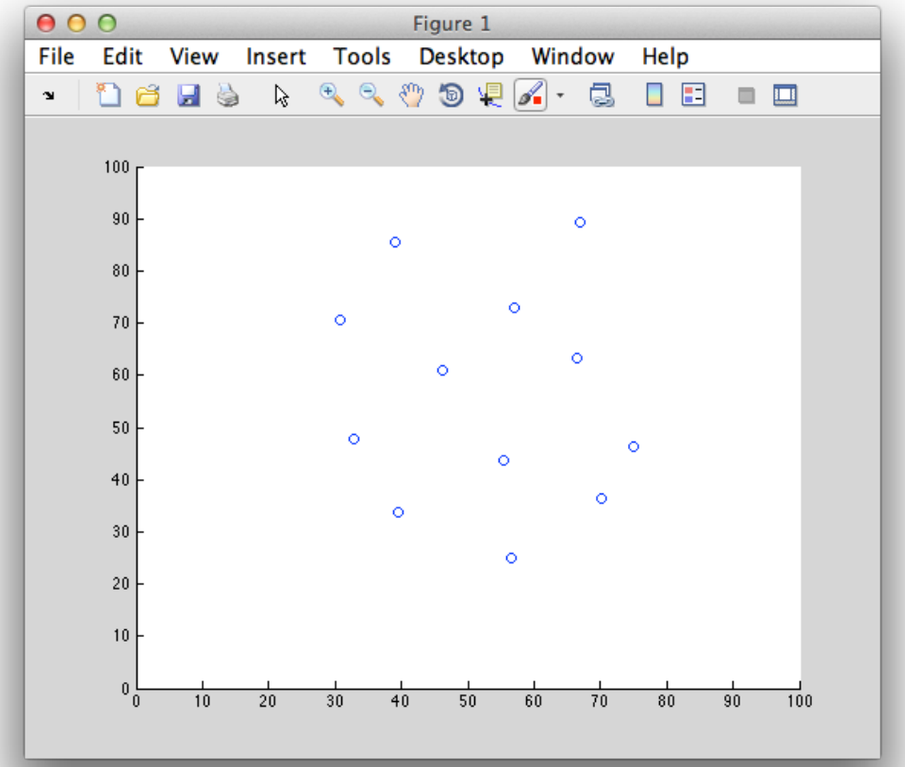
The function should behave just as in the example, using exactly the same phrasing in its output (e.g., "That's it. Well done!")

Problem 19. Write a function called **`click_and_plot`** that takes no input arguments and returns no output arguments. Instead it gets it input via the function **`ginput`**, which does nothing until the user either clicks the mouse or hits a key on the keyboard. The help function gives detailed information about **`ginput`**, but for this problem, it should be called this way: **`[x,y,button] = ginput(1)`**. After each call, when the user moves the mouse into an active figure and clicks the left mouse button, **`button`** is set to **`1`**, and **`x`** and **`y`** are set to the horizontal and vertical positions in the figure. When the user moves the mouse into an active figure and clicks a key on the keyboard, **`button`** is set equal to the character for that key. For example, if the G-key is hit, then **`button`** will be set equal to **`'g'`**. If Shift is held down while the G-key is hit, button will be set equal to **`'G'`**. So, if it is necessary to determine whether the G-key has been hit, this **`if`**-statement will do the trick:

```
if button == 'g' || button == 'G'
    . . . % do something
end
```

The function **`click_and_plot`** must first use the function **`axis`** to set the horizontal and vertical range of a figure each to be 0 to 100 and give the command, **`hold on`**, so that points will be added cumulatively to the plot. Then, it must enter a while-loop to repeatedly plot one point, via the function **`plot`**, using the plot symbol **`'o'`** (plot symbols are explained by **`help plot`**) every time the user clicks somewhere inside the figure, until the user hits the Q-key (with or without pressing the Shift-key) while the mouse is inside the figure. The loop must then end, and the function must return. Figure 2.42 shows the result after the user has clicked the mouse twelve times before hitting the Q-key.

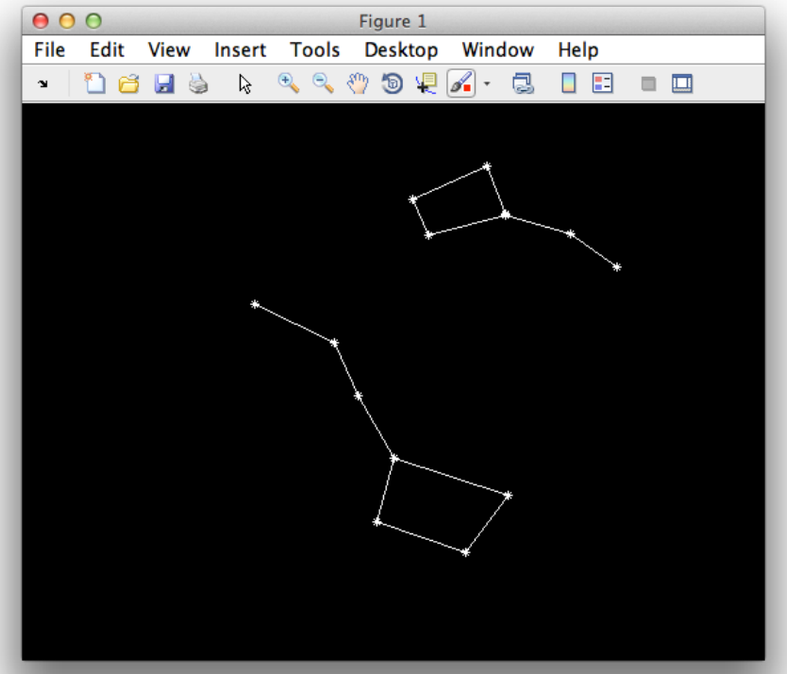
Figure 2.42 *Problem 19*



?

Problem 20. Write a function called **`draw_constellations`** that takes no input arguments and returns no output arguments. Instead, it obtains its input via the function **`ginput`**. See the previous problem for an explanation of how **`ginput`** works. Like the function in the previous problem, this function allows the user to plot points by clicking in a figure whose horizontal and vertical ranges are set to be 0 to 100, but with this function the plot symbol **`'*'`** is used, and the points are joined by straight lines. Furthermore, the color of the plotted symbols and lines must be white, and before plotting begins, the function must use the following two commands to set the background to black: **`set(gcf,'color','k')`** and **`set(gca,'color','k')`**, and it must issue the command **`axis square`**. Then, the function must enter a while-loop to plot the first point and then plot subsequent points with lines joining each point to the previous point. These connected points represents a constellation (more precisely, a star pattern). If the user clicks the N-key, with or without the Shift key depressed, then a new set of connected points is begun—points that are not connected by a line o the previous connected pattern. Finally, when the user hits the Q-key (with or without Shift), the loop must end and the function must return. Figure 2.43 is an example of the result of the use of **`draw_constellations`** to draw the constellations Ursa Major and Ursa Minor.

Figure 2.43 Problem 20



Nested loops

Problem 21. Write a function called **`mult_table`** that takes one positive integer as an input argument (it does not have to check the format of the input) and returns a two-dimensional array. If it is called like this, **`A = mult_table(N)`**, then **`A`** is an **`N-by-N`** multiplication table, which means the entry with indices **`ii`** and **`jj`** is equal to the product of **`ii`** and **`jj`**. The function must not use array operations but instead must use nested loops.

?

Problem 22. Write a function called **`div_table`** that takes one positive integer as an input argument (it does not have to check the format of the input) and returns a two-dimensional array. If it is called like this, **`A = div_table(N)`**, then **`A`** is an **`N-by-N`** division table, which means the entry with indices **`ii`** and **`jj`** is equal to **`ii`** divided by **`jj`**. The function must not use array operations but instead must use nested loops.

Problem 23. Write a function called **`make_waves`** that takes two positive integers as input arguments (it does not have to check the format of the input) and returns a two-dimensional array. If it is called like this, **`A = make_waves(M,N)`**, then **`A`** is an **`N`**-by-**`N`** array of products of sine waves whose amplitude is **`1`** and whose period is **`N/M`**. The result can be achieved by means of the following expression:

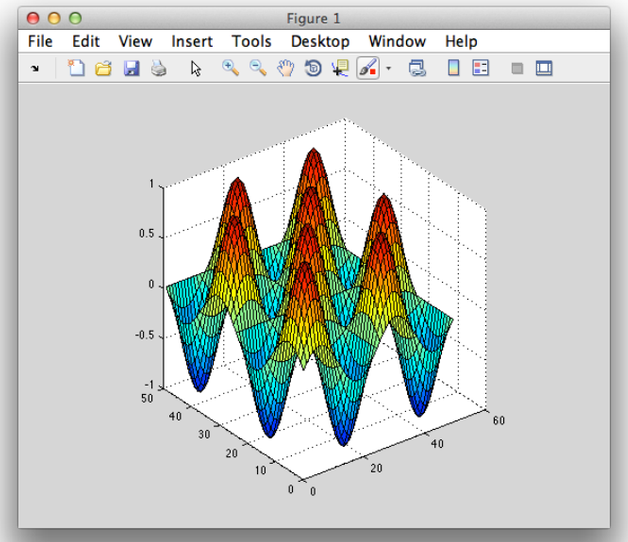
```
A = sin(2*M*pi/N*(1:N))'*sin(2*M*pi/N*(1:N))
```

However, the function must instead use nested loops. The result can be visualized by means of the function **`surf`**, which renders a surface whose distance above the **`x`**-**`y`** plane is equal at each **`x = ii`** and **`y = jj`** to **`A(ii,jj)`**. Here is an example:

```
>> surf(make_waves(2,50));axis square
```

which produces Figure 2.44.

Figure 2.44 *Problem 23*



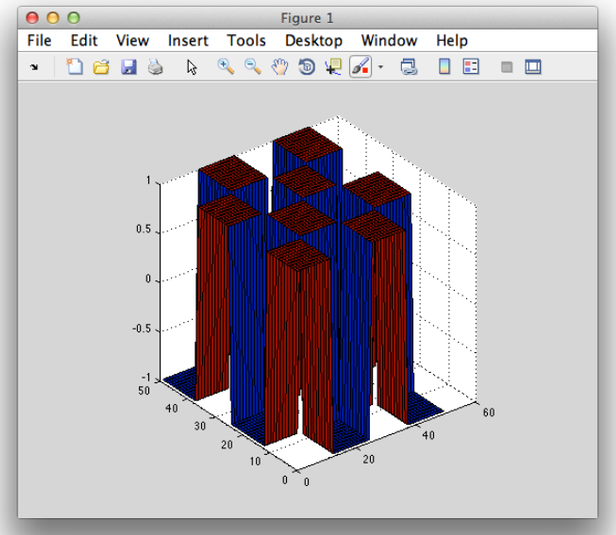
?

Problem 24. Write a function called **`make_square_waves`** that takes two positive integers as input arguments (it does not have to check the format of the input) and returns a two-dimensional array. If it is called like this **`A = make_square_waves(M,N)`**, then **`A`** is an **`N`**-by-**`N`** array (*not* **`M`**-by-**`N`**) of products of square waves whose amplitude is 1 and whose period is **`N/M`**. The result can be achieved by modifying the function **`make_waves`**, which is described in the previous problem. The modification involves the use of the built-in function **`sign`**. Here is an example:

```
>> surf(make_square_waves(2,50));axis square
```

which produces Figure 2.45.

Figure 2.45 *Problem 24*



Problem 25. Write a function called **exp_approx_sequence** that takes a vector as input (the function does not have to check the format of the input) and returns vector as output. If it is called like this, **[v,max_ns] = exp_approx_sequence(u)**, then **v(ii)** = an approximation of **exp(u(ii))**. The approximation is obtained by means of a while-loop that adds terms in the sequence:

$$x^0 + x^1 + x^2/2! + x^3/3! + x^4/4! + \ldots\ x^n/n! + \ldots\ ,$$

where x = **u(ii)**. The while-loop must stop when it has added a term $x^n/n!$ whose absolute value is smaller than 1/10,000. For each **ii**, **max_ns(ii)** equals the value of **n** in the last term of the sequence in the while-loop's calculation of the approximation. The while-loop must be nested inside a for-loop that progresses through the elements of **u**.

?

Problem 26. Write a function called **sine_approx_sequence** that takes a vector as input (the function does not have to check the format of the input) and returns vector as output. If it is called like this,

[v,max_ns] = sine_approx_sequence(u),

then **v(ii)** = an approximation of **sin(u(ii))**. The approximation is obtained by means of a while-loop that adds terms in the sequence:

$$x^1 - x^3/3! + x^5/5! - x^7/7! + \ldots + (-1)^n x^{2n+1}/(2n+1)! + \ldots,$$

where **x = u(ii)**. The while-loop must stop when it has added a term

$$(-1)^n x^{2n+1}/(2n+1)!$$

whose absolute value is smaller than 1/10,000. For each **ii**, **max_ns(ii)** equals the value of n in the last term of the sequence in the while-loop's calculation of the approximation. The while-loop must be nested inside a for-loop that progresses through the elements of **u**.

Problem 27. Write a function called **forward_backward** that takes two positive integers as input arguments (it does not have to check the format of the input) and returns a two-dimensional array. If it is called like this,

A = forward_backward(M,N)

then **A** is an **M**-by-**N** array with the numbers 1 to **N** in forward order on the first row. If there is a second row, then it has the numbers from **N** down to 1 (i.e., backward order) on the second row. For subsequent rows, the numbers 1 to **N** are in forward order on odd-numbered rows and in backward order on even-numbered rows. The function must use nested loops, as opposed to a single loop that uses a colon operator to generate a row.

?

Problem 28. Write a function called **`double_fibonacci`** that takes two integers, each greater than one, as input arguments (it does not have to check the format of the input) and returns a two-dimensional array. If it is called like this, **`A = double_fibonacci(M,N)`**, then **`A`** is an **`M`**-by-**`N`** array that has the Fibonacci series on its first row, has the same series in its first column, and for rows **`n = 2`** to **`M`**, row **`n`** contains the Fibonacci numbers that begin with **`n`**th number in the series. Here is an example showing the output when the input integers are **`6`** and **`9`**, **`A = double_fibonacci(6,9)`**:

```
A =
    1     1     2     3     5     8    13    21    34
    1     2     3     5     8    13    21    34    55
    2     3     5     8    13    21    34    55    89
    3     5     8    13    21    34    55    89   144
    5     8    13    21    34    55    89   144   233
    8    13    21    34    55    89   144   233   377
```

The first row of **`A`** gives the first nine numbers in the Fibonacci series. The series starts with two ones, and each subsequent number is equal to the sum of the two numbers that precede it.

Problem 29. Write a function called **`blue_to_red`** that takes two input arguments and returns one output argument. The first input argument is a string containing the name of a file (it does not have to check the format of the input). That file should contain a color image (i.e., a three dimensional array with three pages containing the red, green and blue intensities) in one of the image formats, such as JPEG, supported by **`imread`**, which is the built-in function that should be used to read the image from the file into an array. A list of these formats can be found with the command **`help imwrite`**. Suppose the file, 'bluebird_photo.jpg', contains such an image. If the function is called like this:

`new_image = blue_to_red('bluebird_photo.jpg')`,

then **`new_image`** will contain an image that is identical to the one in the file, except that every pixel whose blue intensity is more than 120 percent of the mean of the red, green, and blue intensities will be changed to a pixel whose red channel is equal to the blue channel of the input image and whose red and green channels are equal to zero. Use explicit, nested loops. Check your work by displaying the input image and the output image, as in the following example:

```
>> old_image = imread('bluebird_photo.jpg');
>> figure(1); image(old_image); axis equal; axis tight;
>> new_image = blue_to_red('bluebird_photo.jpg');
>> figure(2); image(new_image); axis equal; axis tight;
```

NOTE: The solution was given previously in this section.

?

Problem 30. Write a function called **`replace_blue`** that takes two input arguments and returns one output argument. The first input argument is a string containing the name of a file (it does not have to check the format of the input). That file should contain a color image (i.e., a three dimensional array with three pages containing the red, green and blue intensities) in one of the image formats, such as JPEG, supported by **`imread`** (listed by **`help imread`**). The second input argument is a three-element vector containing values in the range 0 to 255. Suppose the file, 'bluebird_photo.jpg', contains such an image. If the function is called like this,
`blue_replaced=replace_blue('bluebird_photo.jpg',[r,g,b])`, then **`blue_replaced`** will contain an image that is identical to the one in the file, except that every pixel whose blue intensity is more than 120 percent of the mean of the red, green, and blue intensities will be changed to a pixel with the red channel equal to **`r`**, the green channel equal to **`g`**, and the blue channel equal to **`b`**. Use explicit, nested loops. Check your work by displaying the input image and the output image, as in the following example:

```
>> old_image = imread('bluebird_photo.jpg');
>> figure(1); image(old_image); axis equal; axis tight;
>> new_image =
blue_replaced('bluebird_photo.jpg',[255,255,0]);
>> figure(2); image(new_image); axis equal; axis tight;
```

Logical Indexing

Problem 31. Write a function called **`picking_nits`** that takes one vector as an input argument (it does not have to check the format of the input) and returns one vector as an output argument. If it is called like this, **`[v_clean,nits]= picking_nits(v)`** then **`v_clean`** is identical to **`v`** except that every element whose absolute value is less than **`1/100`** has been removed (*not* set to zero), and **`nits`** contains the elements that have been removed from **`v`**. The function must use logical indexing instead of explicit looping.

?

Problem 32. Write a function called **`trim10`** that takes two vectors of the same length as input arguments (it does not have to check the format of the input) and returns two row vectors of the same length as the input vectors. If it is called like this, **`[v_trimmed,trimmings] = trim10(v1,v2)`**, then **`v_trimmed`** is identical to **`v1`** except that every element **`v1(ii)`** that is greater than **`v2(ii)+10`** must be trimmed, which means that it must be replaced by **`v2(ii)+10`**. Each element of **`trimmings`** is equal to the amount by which each element has been trimmed. The function must use logical indexing instead of explicit looping. Here is an example of the function being used:

```
>> v1
v1 =
    36    26     4    17  -100    90

>> v2
v2 =
    34    15   -20     0     6    80
```

```
>> [v_trimmed,trimming] = trim10(v1,v2)
v_trimmed =
    36    25   -10    10  -100    90
trimming =
     0     1    14     7     0     0
```

Problem 33. Write a function called **blue_to_red_implicit** that performs the same operation as the function **blue_to_red** described in Problem 29 above but uses logical indexing instead of explicit looping.

?

Problem 34. Write a function called **replace_blue_implicit** that performs the same operation as the function **replace_blue** described in Problem 30 above but uses logical indexing instead of explicit looping.

Making explicit loops efficient

Problem 35. Write a function called **so_fast** that takes one positive integer as an input argument (it does not have to check the format of the input) and returns one two-dimensional, square array as an output argument. If it is called like this **A = so_fast(N)**, then **A** is an **N**-by-**N** array of random numbers gotten from the built-in function **rand**. Such an array can be produced most simply and quickly as follows **A = rand(N)**. However, **so_fast** is required to make a separate call, **rand(1)** (the argument is optional), for each element of **A** and to use nested for-loops to handle all N^2 elements. This function must use both of the techniques given in this section to save memory allocation time. On a Dell Latitude E6410, with 7.8 gigabytes of usable memory, the difference between using both techniques and using neither is a factor of 78 in execution time.

?

Problem 36. Consider the following function:

```
function A = plodding(N,d)
for ii = 1:N
   jj = 1;
   A(ii,jj) = randn;
   while abs(A(ii,jj)) < d
      jj = jj + 1;
      A(ii,jj) = randn;
   end
end
```

Rewrite this function to eliminate the allocation problem that is slowing it down. Call the new function, **cruising**. On a Dell Latitude E6410, with 7.8 gigabytes of usable memory, eliminating the allocation problem produces a speed-up factor of 7.

Data Types

Objectives

Computers operate on bits, but humans think in terms of numbers, words, and other types of data. Like any good language, MATLAB organizes bits into convenient data types. We will study those types in this section.

(1) We will learn that there are ten types of numbers and that there are conversion functions to change one type into another.

(2) We will learn much more about strings and how the characters in them are encoded as numbers.

(3) We will learn how to produce heterogeneous collections of data via data types called **`structs`** and **`cells`**.

(4) We will learn two new concepts from computer science: the symbol table and the pointer.

Inkaphotoimage, Dreamstime.com, Photo taken February 15th, 2010.

Data comes in all shapes and sizes.

Every modern programming language provides various means for storing numbers in variables, operating on them, and printing them. MATLAB is no exception, and we have seen many examples of numbers being stored, operated on, and printed. The examples suggest that these numbers are equivalent to the real numbers and complex numbers of mathematics, where there is no upper limit to the absolute value of any number and no smallest absolute value of a non-zero number. On the other hand, since the space of mathematical numbers is infinite, while the memory of a computer is finite, it is clear that the use of a computer must impose some limits on the numbers it can store. It does. There is an upper limit to the size of a number on the computer, and there is a smallest value for the absolute value of a non-zero

number. The set of numbers that can be represented by a MATLAB variable is finite.

We are about to learn that that there are many types of numbers in MATLAB and that each of these types provides its own set of allowed values. We will also learn that there are types that hold non-numeric data. These other types cannot be used in arithmetic operations but do allow other operations. A data type is in fact completely determined by the set of all possible values it can have and the set of operations that can be performed on it. Indeed, in the formalism of computer science, a **data type**, or simply, a **type**, is defined as a set of values and a set operations that can be performed on those values.

It is common to employ variables of several different types in a single function. It is even possible to use variables of different types in the same operation. MATLAB does however enforce one rule of uniformity: ***All elements of a given array must be of the same type.*** We call the type of these elements the **elementary type** of the array. Thus, when we speak of the type of a variable that is not a scalar, it is not necessary to specify the type of each element in the array. We need to specify only one elementary type because an array has only one elementary type. However, to specify the type of an array, more information is required than its elementary type. A complete description of the type of an array must include these things: its number of dimensions, its size in each dimension, and its elementary type. In MATLAB, the number of dimensions is always greater than or equal to two, so there are always two or more sizes (in MATLAB even a scalar is a one-by-one matrix!). Example descriptions of types are

- **`x`** is a 1-by-1 matrix with elements of type **`double`**
- **`x`** is a scalar of type **`double`**
- **`y`** is a row vector with 10 elements of type **`single`**
- **`y`** is a 1-by-10 vector with elements of type **`single`**
- **`A`** is a 2-by-3 matrix with elements of type **`int8`**
- **`B`** is a 2-by-3-by-4 by-5 array with elements of type **`logical`**

The `class` *function*

To determine the data type of a MATLAB variable, you can use the function **`class`**. For example,

```
>> x = 14
x =
    14
>> class(x)
ans =
double
```

(Why **`class`** instead of **`type`**? Well, MATLAB reserves the name **`type`** for a command that prints out the contents of a file.)

Values have types, even when they are not stored in a variable. Thus,

```
>> class(14)
ans =
double

>> class('Mary had a little lamb.')
ans =
char

>> class(class(14))
ans =
char
```

The first example shows that the type of a number is by default **`double`**. The second, shows that the type of a string is **`char`** (It is a matter of taste, whether "char" is pronounced like the first syllable of "charcoal" or like the word "care"). The third example shows that the type that is returned by **`class`** itself is of type **`char`**. That is because **`class`** returns a string that spells out the name of the type.

While MATLAB supports complex numbers, it does not distinguish among real, imaginary, and complex numbers when classifying them into types.

```
>> class(sqrt(-1))
ans =
double
```

Here we see that a clearly imaginary number, $i = \sqrt{-1}$, is classified as type **double**.

Numeric Types

MATLAB uses many different internal representations to store numbers. Each representation is a numeric type. Most of the time a MATLAB programmer need not be concerned with the specifics of the types of numbers used in a calculation. The default data type used by MATLAB to store a number is **double**, and double is so versatile that it is capable of handling almost any numerical problem that comes up in engineering and science applications. The name "double" has historical roots. It was introduced as the name of a numeric data type in the 1960s in programming languages that supported two types of numbers, one of which required twice as much memory space per number as the other. The one with twice the space was called "double precision". The other one was called "real" and was said to employ "single precision". In the 1970s these two types were renamed "double" and "float" in some new languages. The latter name is meant to indicate that a "floating-point" format is used to represent the number. (The specifics of the numeric formats is beyond the scope of this book.) Double uses floating-point representation as well, and, thanks to increases in memory sizes and CPU speeds, it is now the default representation in most languages. The language C uses this name, and C++ and Java use it as well. Today, if a computer supports any of these languages—MATLAB C, C++, and/or Java—it will use the same amount of memory space for a number stored using the **double** type for programs written in any of these languages. C, C++, and Java employ the name "float" for single-precision numbers, but MATLAB has adopted the more specific term "single" for them.

Table 2.16 includes all of MATLAB's numeric data types along with the range of values that each supports. The names are mnemonic: The word **int** embedded in all but two of the names means "integer"; a leading "**u**" means unsigned, so **uint** means "unsigned integer". The numbers in the names indicate the number of bits used for storage, as explained below.

Table 2.16 *Numeric data types*

DATA TYPE	RANGE OF VALUES
`int8`	-2^7 to 2^7-1
`int16`	-2^{15} to $2^{15}-1$
`int32`	-2^{31} to $2^{31}-1$
`int64`	-2^{63} to $2^{63}-1$
`uint8`	0 to 2^8-1
`uint16`	0 to $2^{16}-1$
`uint32`	0 to $2^{32}-1$
`uint64`	0 to $2^{64}-1$
`single`	$-3.4\text{x}10^{38}$ to $3.4\text{x}10^{38}$, `Inf`, `NaN`
`double`	$-1.79\text{x}10^{308}$ to $1.79\text{x}10^{308}$, `Inf`, `NaN`

If you don't have this table handy, the maximums and minimums can always be gotten from MATLAB by using these four functions: **intmax**, **intmin**, **realmax**, and **realmin**. Each takes one string as input. The string for the first two is the name of the integer type of interest. The string for the last two is 'double' or 'single'. Note, however, that **realmin** gives the number with the smallest absolute value that can be represented and not the negative number with the greatest absolute value. For example,

```
>> intmax('int32')
ans =
  2147483647

>> intmin('int32')
ans =
 -2147483648
```

```
>> realmax('single')
ans =
  3.4028e+038

>> realmin('single')
ans =
  1.1755e-038
```

The "is" functions

MATLAB provides a set of functions that allow the user to check for a specific type. The name of each function begins with "**is**". Examples include **isinteger**, **isfloat**, **issingle**, **isnumeric**, and **ischar**. Each of these functions takes an array as an input argument and returns either true or false. The name of the function reveals its meaning: **isinteger(x)** returns true if and only if **x** is of one of the integer types; **isfloat(x)** returns true if and only if **x** is of floating point type (i.e., **single** or **double**); **isnumeric(x)** returns true if and only if **x** is one of the types in Table 2.16. There is also a generic function called **isa**. It takes two input arguments, an array whose type is being checked and a string that spells out the type. For example **isa(x,'uint32')** returns true if and only if the type of **x** is **uint32**. A complete list of the "is" functions is given by **help isa**.

Conversion functions

To produce a numeric value of a specific type MATLAB provides conversion functions. The conversion function for a given data type has the same name as that type. A **conversion function** takes one input argument of any numeric type and returns one output argument of its specified type. Let's see some examples of conversion functions:

```
>> x = 10
x =
    10

>> class(x)
ans =
double
```

From the commands above, we see that the default numeric data type is **double**. Let's convert **x** to another type:

```
>> x = int8(10)
x =
   10
>> class(x)
ans =
int8
```

From the two commands above, we see how the conversion function **int8** can be used to produce the numeric type **int8** as output when given an input type of **double**.

```
>> y = x
y =
   10

>> class(y)
ans =
int8
```

From the two commands above, we see, as we would expect, that the data type of **y** becomes the same as the data type of the value that is assigned to it. If we want **y** to be of type **double**, we must convert it:

```
>> y = double(y)
y =
    10

>> class(y)
ans =
double
```

From the two commands above, we see that the conversion function **double** can be used to produce an output of type **double**.

With the exception of **int64** and **uint64**, the advantage of each of the first 9 data types over **double** is that they require less memory space per number.

As we learned at the end of the subsection, Issuing commands in Introduction to MATLAB, computer memory is measured in bits and bytes, where a bit is the smallest unit of memory on a computer and a **byte** is a group of eight bits. All but the last two of the data types in Table 2.16 include a number as part of their names. As mentioned above, that number designates the number of bits required to store a single number of that type. The last two types do not include a number in their names, but `single` requires 32 bits and `double` requires 64 bits. Computers manipulate their memories most efficiently by accessing a byte at a time or a number of bytes that is equal to a power of two, and most computers perform comparisons between objects and arithmetic operations on them most efficiently when the power is 0, 1, 2, or 3. For that reason each type in MATLAB and all other modern programming languages supports these numerical variable sizes—1 byte, 2 bytes, 4 bytes, and 8 bytes.

All but two of the data types include "int" as part of their names. These are integer types, which means that they are capable of storing only integers. Thus, the integer types cannot store fractional parts. An attempt to convert a number with a fractional part to an integer type will result in rounding, as in the following examples:

```
>> int8(9.5)
ans =
   10

>> int8(9.4)
ans =
    9

>> int8(-9.4)
ans =
   -9

>> int8(-9.5)
ans =
  -10
```

Furthermore, if a conversion function is given a value outside its range, it will return the value that lies at the closest end of its range. For example,

```
>> int8(128)
ans =
  127

>> int8(-1000)
ans =
 -128
```

As mentioned above, some of the integer types have names that begin with the letter "u", which stands for "unsigned". These types can store only non-negative integers, so converting a negative number to an unsigned type always results in zero, which is the value at the end of any unsigned range that is closest to all the negative numbers:

```
>> uint8(-13)
ans =
    0
```

Arithmetic operations

We have seen many examples of arithmetic in MATLAB, all involving the operators, `+`, `-`, `*`, `/`, `\`, `^`, `.*`, `./`, `.\`, and `.^`. However, up to now, the operands have always been of `double` type. All other numeric types can be used in arithmetic operations in MATLAB as well. However, there are important restrictions on these operations. When the two operands of a binary arithmetic operator (recall that "binary" means "taking two operands") are of different types, the resulting operation is called **mixed-mode arithmetic**. In MATLAB there are severe restrictions on mixed-mode arithmetic. For the expression `x op y`, where `op` is an arithmetic operator, the following list of categories gives in braces the set of allowed operators for each given pair of operand types:

1. **x** and **y** are of floating-point types: {any arithmetic operator}
2. **x** is of integer type and **y** is a floating-point scalar: {**+**, **−**, **.*** , **./** , **.** , **.^** , ***** , **/**}
3. **x** is a floating-point scalar and **y** is of integer type : {**+**, **−**, **.*** , **./** , **.** , **.^** , ***** , ****}
4. **x** and **y** are of the same integer type: { **+**, **−**, **.*** , **./** , **.** , **.^** }
5. Category 4 when **y** is a scalar: { **+**, **−**, **.*** , **./** , **.** , **.^** , ***** , **/** }
6. Category 4 when **x** is a scalar: { **+**, **−**, **.*** , **./** , **.** , **.^** , ***** , **** }
7. **x** and **y** are of different integer types: NONE!

The last category shows that arithmetic operations involving differing integer types are always illegal.

A question arises immediately in mixed-mode operations: What is the type of the result? The answer has two cases: When an operation of Category 1 is carried out, the result has the type,

- **double**, if both operands are of type **double**.
- **single**, if either of the operands is of type **single**.

When an operation involves an operand of integer type, which includes all categories from 2 to 6, the result has that same integer type. Note from these two cases that MATLAB uses the rule that the output type of an arithmetic operation is the same as the type of the input type with the smaller range. (This behavior is opposite that of C, C++, and Java. Their rule is that the output type is the same as the type of the input with the larger range.)

Clearly, since the output of a mixed-mode operation in MATLAB has the type with the narrower range, the likelihood is increased, relative to C, C++, and Java, that the result will fall outside the range supported by the type of the output. So we are led to the next question: What is the benefit?

The answer is that it saves memory space—in some cases a lot of it—because types with narrower ranges can be stored in fewer bytes. Let's suppose that you have decided to use an array **A** with the type **int16**. You may have chosen that type because each element of an array of that type requires only two bytes of storage, **A** is going to be very large, and the numbers in **A** will never lie out of the range -2^{15} to $2^{15}-1$ (Table 2.16). This is exactly the case, for example, if you are writing image-processing algorithms for medical images, such as computed tomography (CT) or magnetic resonance images. These arrays are three-dimensional, their values fall within this range, and they are big. A typical CT, for example, might have the dimensions 512-by-512-by-100, which will occupy 512x512x100x2 bytes. That's 50 megabytes for one array! At some step in your algorithm you might need to increase all the values in **A** by, say 50, which can easily be done this way: **A = A + 50**. If MATLAB gave the output the type with the wider range, it would in this case be of type **double**, because that is the type of any literal number, i.e., a number like **50** that is typed in. The assignment of that output back to **A** would change it to a **double**, increasing its size by a factor of four. **A** would now weigh in at a whopping 200 megabytes! Of course, this problem could be avoided by writing **A = A + int16(50)**, but this would make the program hard to write and hard to read. Instead, since in MATLAB the output of the addition operation has the narrower type, **A** remains the same size in memory.

A third question pops up: What value is given when the correct answer is out of range of the type? The answer to that one is that it depends. It depends on the correct answer and on the type. MATLAB gives the expression a special value according to the type of the result (**Inf** represents "infinity"):

For the types **double** and **single**, if the correct answer is positive, the value is **Inf**. If the correct answer is negative, the value is **-Inf**.

For any integer type, if the correct answer is greater than the maximum of the range, the value is the maximum of the range. If the correct answer is smaller than the minimum of the range, the value is the minimum of the range. For example, if **x** is of type **int8**, and **x** equals 100, then **x + 50** yields 127, be-

cause 127 is the maximum of the range of **int8**. If **x** equals 100, then **x - 250** yields -128 because -128 is the minimum of the range of **int8**. If **x** is of type **uint8**, and **x** equals 100, then **x + 500** yields 255 because 255 is the maximum of the range of **uint8**, and if **x** equals 100, then **x - 500** yields 0, because 0 is the minimum of the range of **uint8**.

One question remains: What value is given when the result is undefined, as, for example in **x = 0/0**, **y = Inf/Inf**, or **z = 0*Inf**? For integer types, the result is 0, for **single** and **double**, the result is a special value named **NaN**, which means "Not a Number".

Relational operations

The relational operators, **==**, **~=**, **<**, **>**, **<=**, and **>=** all allow mixed-mode operands of any numeric types. They return a value of type **logical**. For example,

```
>> int8(4) < single(4)
ans =
     0

>> class(ans)
ans =
logical
```

Strings

Strings were introduced in this book in the very first section of Chapter 1 and have been used repeatedly after that, for example to provide a prompt to the user, to hold filenames, to specify plotting styles in **plot**, or to hold the format string in **fprintf**, which tells **fprintf** what to print and how to print it. There is, however, much more to strings. Strings contain numeric values, that can be assigned to a variable and manipulated by functions.

The ASCII encoding scheme

As we saw in an example at the very beginning of this section, a string is of type **char**. More specifically a string is a row vector of numbers of type **char**, each number being a code that represents one character. Part of the scheme for encoding characters as numbers is shown in Table 2.17. This part of the scheme is called **ASCII** (American Standard Code for Information Inter-

Table 2.17 ASCII codes

(nul)	0	(sp)	32	@	64	`	96
(soh)	1	!	33	A	65	a	97
(stx)	2	"	34	B	66	b	98
(etx)	3	#	35	C	67	c	99
(eot)	4	$	36	D	68	d	100
(enq)	5	%	37	E	69	e	101
(ack)	6	&	38	F	70	f	102
(bel)	7	'	39	G	71	g	103
(bs)	8	(	40	H	72	h	104
(ht)	9	)	41	I	73	i	105
(nl)	10	*	42	J	74	j	106
(vt)	11	+	43	K	75	k	107
(np)	12	,	44	L	76	l	108
(cr)	13	-	45	M	77	m	109
(so)	14	.	46	N	78	n	110
(si)	15	/	47	O	79	o	111
(dle)	16	0	48	P	80	p	112
(dc1)	17	1	49	Q	81	q	113
(dc2)	18	2	50	R	82	r	114
(dc3)	19	3	51	S	83	s	115
(dc4)	20	4	52	T	84	t	116
(nak)	21	5	53	U	85	u	117
(syn)	22	6	54	V	86	v	118
(etb)	23	7	55	W	87	w	119
(can)	24	8	56	X	88	x	120
(em)	25	9	57	Y	89	y	121
(sub)	26	:	58	Z	90	z	122
(esc)	27	;	59	[	91	{	123
(fs)	28	<	60	\	92	|	124
(gs)	29	=	61	]	93	}	125
(rs)	30	>	62	^	94	~	126
(us)	31	?	63	_	95	(del)	127

change). It was designed in the 1960s to encode the so-called "Latin alphabet", the 10 decimal digits, and punctuation. In that table the abbreviations in parentheses have special meanings to computer systems that use communication protocols that are of no interest here, except for these: (nl) means "newline", (ht) means "horizontal tab", and (sp) means space. These have meaning to **`fprintf`**, and there is a special escape sequence for each of them: **`\n`** produces (nl), and **`\t`** produces (ht). We saw the first one in the subsection Conversion characters and escape characters in Programmer's Toolbox. The encoding scheme assigns a number from 0 to 127 to each character. In the table the encoding number is given to the right of each character. For example, the number that encodes an uppercase "A" is 65, while the number that encodes a lowercase "a" is 97.

ASCII was augmented in subsequent years to include non-Latin characters. Those augmented versions must be stored in two or more bytes. Today, there are several standards, but all of them include ASCII as a subset. MATLAB uses a two-byte code that varies with the installation. The name of the encoding scheme can be gotten with the command
`feature('DefaultCharacterSet')`,
which returns the name of the scheme as a string.

A string is a vector

The sequence of numbers that encode a string is stored as a standard row vector. The length of the vector is equal to the number of characters in it, and it can be determined by using the same **`length`** function that we have applied before to numeric vectors. Furthermore, each individual element can be accessed using the usual indexing scheme. Here is an example:

```
>> book_title = 'MATLAB for Smarties'
book_title =
MATLAB for Smarties
```

Note that, when MATLAB prints the value of a string, it omits the quotes. Thus, it prints **`MATLAB for Smarties`**, instead of **`'MATLAB for Smarties'`**.

```
>> length(book_title)
ans =
    19
```

There are 19 characters in **`'MATLAB for Smarties'`**, and that is the value returned by **`length`**. Among these characters are two spaces, and, if there were any punctuation marks, they would have been counted too. Spaces and punctuation marks are all part of the string, and each one is encoded with its own number.

```
>> book_title(1)
ans =
M

>> book_title(4:16)
ans =
lab for Smart

>> book_title(7:11)
ans =
 for
```

Note that the word **`for`** on the last line is shifted slightly to the right. That is because the first character printed by the last command is a space. There is a trailing space there too, but it is invisible. Checking the ASCII table above we see that the number that encodes a space (sp) is 32. We can use that information to look for spaces in a string. We can easily see the numerical codes of any of the characters in any string. MATLAB enables us to do that by converting from the type **`char`** to a numeric type, such as **`double`**. Let's do that by using the conversion function **`double`**:

```
>> double(' ')
ans =
    32
```

Here, we have given **`double`** an argument that is a one-element character string consisting of one space. Now let's look for spaces in **`book_title`**:

```
>> double(book_title)
ans =
  Columns 1 through 9
   77   97  116  108   97   98   32  102  111
  Columns 10 through 18
  114   32   83  109   97  114  116  105  101
  Column 19
  115
```

As expected there are two 32s. The first one is followed by 102, which is the ASCII code for the letter "f". The second one is followed by 83, which is the code for "S". Digits and punctuation marks are treated no differently. If they occur in a string, they are encoded in ASCII:

```
>>  pi_digits = '3.14159'
pi_digits =
3.14159

>> double(pi_digits)
ans =
    51    46    49    52    49    53    57
```

Here we have assigned the 7-character string `'3.14159'` to `pi_digits`. These are the first digits of the number π, but, in this case, instead of assigning the number 3.14159, which would be of type `double`, we have, by enclosing 3.14159 in quotes, assigned a vector consisting of seven elements, each of which is the ASCII code for one character. Since there are seven elements in this vector, the result of applying the function `length` to `my_number` should not be surprising:

```
>> length(my_number)
ans =
     7
```

Like other vectors, new strings can be made from parts of existing ones:

```
>> s = 'Two is company, three a crowd.';
>> ssub = s(13:end)
ssub =
ny, three a crowd.
```

Also, new strings can be made by concatenating existing strings:

```
>> a = 'Three, two, one';
>> b = ', ';
>> c = 'BLASTOFF!';
>> count_down = [a,b,c]
count_down =
Three, two, one, BLASTOFF!
```

Strings can even be put together to make arrays with more than one row and column:

```
>> first = 'abcdefgh'
first =
abcdefgh

>> second = '12345678'
second =
12345678

>> both = [first;second]
both =
abcdefgh
12345678

>> double(both)
ans =
    97    98    99   100   101   102   103   104
    49    50    51    52    53    54    55    56

>> both'
ans =
a1
b2
c3
d4
e5
f6
g7
h8
```

Here we have made an array with two rows in which all the elements are of type `char`. It is important, though, to remember that arrays must be rectangu-

lar. That is, each row must have the same number of elements. In this case, the variables **first** and **second** are both of length 8, so each row of **both** has 8 elements. It is not possible to make a "ragged" array, in which rows contain strings of differing lengths, as the following attempt shows:

```
>> long = 'MATLAB'; short = 'Java';
>> languages = [long; short]
??? Error using ==> vertcat
All rows in the bracketed expression must have the same
number of columns.
```

(We've seen this same error involving **vertcat** on earlier occasion in which we also tried to produce a non-rectangular array.)

Arithmetic operators

It may seem odd at first, but, like the numeric data types, strings can be used in arithmetic operations! When we do arithmetic on them, MATLAB treats them as a vector of ASCII numbers, instead of a string of characters. Like the integer types, a string can be used in mixed-mode arithmetic, and, also like the integer types, they can be used in this way, if and only if the other operand is one of the floating point types: **single** or **double**. A major difference, though, is that when a string is used in mixed-mode arithmetic, the result of the operation is of type **single** or **double**, corresponding to the type of the other operand, whereas, as pointed out before, when an integer variable is used in mixed-mode arithmetic, the result is of the same integer type. Here is an example of arithmetic on strings:

```
>> language = 'MATLAB'
language =
MATLAB

>> double(language)
ans =
    77    65    84    76    65    66

>> MATLABplus = language + 1
MATLABplus =
    78    66    85    77    66    67

>> class(MATLABplus)
ans =
double
```

We see that when we add 1, which is of type **double**, to **language**, which is of type **char**, the result is of type **double**, as predicted above. Thus, arithmetic operations involving the **char** type produce an output of the wider type, instead of the narrower one, as happens for the integer types.

```
>> char(MATLABplus)
ans =
NBUMBC

>> char(MATLABplus - 1)
ans =
MATLAB
```

This last command produces the original string again by subtracting the incremental 1 and converting once again to the type **char**. We have our favorite language back.

This forward-backward conversion suggests a simple (and easy to crack) coding scheme for secret messages: Any operation that can be undone is suitable to encode a message. Here is another example,

```
>> secret_plans = '1pm: Stairs behind door at right to
basement';

>> coded_plans = char(158 - secret_plans)
coded_plans =
m.1d~K*=5,+~<9650:~://,~=*~,576*~*/~<=+9190*
```

Well, this is certainly a nice, cryptic mess! Decrypting it is easy though:

```
>> decoded_plans = char(158-coded_plans)
decoded_plans =
1pm: Stairs behind door at right to basement
```

We are back to our original message again.

Relational operations

MATLAB's relational operators, **==**, **~=**, **<**, **>**, **<=**, and **>=**, all allow mixed-mode operands of any numeric type and of type **`char`**, and they treat a string as a vector of ASCII numbers, instead of a string of characters. For example,

```
>> int8(97) < 'a'
ans =
     0

>> single([97 97]) < 'ab'
ans =
     0     1
```

Here each element of the vector **`[97 97]`** is compared in turn with the corresponding element of the vector of ASCII numbers **`[97 98]`** that encode "a" and "b". The following example shows how this ability to compare on the basis of ASCII codes might be useful:

```
>> 'agbe' < 'ah f'
ans =
     0     1     0     1
```

Thanks to the sensible order of the ASCII codes, letters that come earlier in the alphabet also have smaller numbers. Thus, a program that is required to alphabetize words can compare the ASCII numbers in order to determine which letters come first. It is interesting to note that the upper case letters all precede the lower case ones. It is for this reason that many computer-sorted word lists put all capitalized words before all the uncapitalized words. It may not be the preferred order for a given application, and it is not necessary to order them that way. However it is the way that requires the least complicated program. For good or ill, programmers have often done it that way.

String functions

While it is possible to do almost anything we wish with strings simply by using the ordinary tools for manipulating vectors, such as indexing, the colon operator, concatenation, arithmetic, relational operators, etc., there are many tasks that have already been written as functions for us by the MathWorks programmers. MATLAB provides a number of built-in functions that include strings as input arguments and/or output arguments. Table 2.18 gives a partial list.

Table 2.18 String functions

FUNCTION	DESCRIPTION
`char`	converts type to char
`findstr`	finds the positions of a substring in a string
`ischar`	returns 1 if argument is a character array and 0 otherwise
`isletter`	finds letters in string
`isspace`	finds spaces, newlines, and tabs in string
`isstrprop`	finds characters of specified type in string
`num2str`	converts number to string
`length`	determines the number of letters in string
`lower`	converts string to lower case
`sprintf`	writes formatted data to string (compare with fprintf)
`strcmp`	compares strings
`strcmpi`	like strcmp but independent of case
`strmatch`	search array for rows that begin with specified string
`strncmp`	like strcmp but compares only first n characters
`strncmpi`	like strncmp but independent of case
`str2num`	converts string to number
`upper`	converts string to upper case

One of these functions is **`sprintf`**. It is similar in functionality to **`fprintf`**, a function that we have already learned about. Both functions permit you to stipulate in detail how values stored in variables are to be represented with letters and digits. The difference is that, while **`fprintf`** causes its output to

appear directly in the Command Window, **sprintf** puts its output into a string. Here is a simple example that reveals the difference:

```
>> x = pi;

>> fprintf('x:\npi to three decimals: %6.3f\n', x);
x:
pi to three decimals:  3.142

>> s = sprintf('x:\npi to three decimals: %6.3f\n', x);

>> s
s =
x:
pi to three decimals:  3.142
```

We see by this example that **fprintf** itself forms a string, but it is not possible to capture the string in a variable, as it is with **sprintf**. We see also that with both **fprintf** and **sprintf**, it is possible to use the escape characters **\n** to produce the newline character.

The following example makes that clearer:

```
>> x = 7;
>> s1 = sprintf('hello'), x = 7
s1 =
hello
x =
     7
>> s2 = sprintf('hello\n'), x = 7
s2 =
hello

x =
     7
```

From this example we note that a line is skipped in the second command. If we were to check the last element of **s2** we would see why. Its value is 10, which, as can be seen by checking Table 2.17 is the ASCII code for newline (nl).

For the remaining functions in Table 2.18, it is recommended that, when you have a task to perform with strings, such as alphabetizing a list, looking something up in a database of strings, or using strings in a graphical-user-interface, you consult this table, find possibly useful functions, and use the MATLAB **help** command to learn the details of their operations.

Structs

An array in MATLAB, C++, Java, Fortran, and most other languages, consists of a collection of elements, each of which has the same elementary data type. The array is said to be homogeneous with regard to the types of its element. It is not possible, therefore, to have a matrix whose first element is of type **int16** and whose second element is of type **char**. Such an array, were it legal, would be heterogeneous with respect to the types of its elements. An attempt to set up any heterogeneous array will be thwarted by MATLAB. Here is an example of what happens,

```
>> A = [int8(1),int16(2);int32(3),uint32(4)]

A =
    1    2
    3    4

>> class(A)
ans =
int8

>> class(A(1,1)),class(A(1,2))
ans =
int8

ans =
int8
```

```
>> class(A(2,1)),class(A(2,2))
ans =
int8

ans =
int8
```

MATLAB imposes its homogeneity restriction by converting all but the first element in the list to be of the same class as that first element, which in this particular case is **int8**. The homogeneity restriction is actually quite reasonable because of the important feature of arrays that allows an operation to be specified simultaneously for all elements of the array, as for example **x = int8(2)*A**. This operation would be legal for **A** above, since the elements of **A** are all of type **int8**, but it would legal for only **A(1,1)**, if MATLAB allowed heterogeneous arrays.

One might argue about the importance of homogeneity with respect to the functionality of arrays, but it is also important from an efficiency standpoint. Accessing individual elements of an array can be handled internally by MATLAB, or any other language for that matter, in an especially efficient way, if the types of the array's elements are all the same. However, efficiency of the inner workings of MATLAB is beyond the scope of this book, and when array-wide operations are not important, then the homogeneity restriction becomes onerous. If, for example, we wish to store information about a person in an array, we might want to put both their social security number and their name in the same data structure. In this case, a first row of type **int16** and a second of type **char** would work very well–if it weren't illegal! Fortunately, for applications in which array-wide operations are not required and heterogeneous elements are desired, MATLAB (like C, C++, and Java) supports a perfect data type. It is called a **struct**. Like an array, a struct consists of a set of elements, but unlike an array those elements can be of differing data types. There is another important difference as well. Each element is indexed by user-defined name instead of a numerical index. Let's look at our first example of a struct:

```
>> r.ssn = 568470008
r =
    ssn: 568470008

>> class(r)
ans =
struct

>> class(r.ssn)
ans =
double
```

Here we have created a user-defined name, **ssn**, to serve as an index for an element of a variable **r**. The dot between **r** and **ssn** indicates that the name to the right of it is the name of an element of the variable to the left of it. A name used as an index is called a **field name**, and the element of the struct associated with that field name is a **field**. The variable **r** is now a struct. It has one field, whose name is **ssn** and whose type is double. We can add as many fields as we like:

```
>> r.name = 'Homer Simpson'
r =
     ssn: 568470008
    name: 'Homer Simpson'
>> r
r =
     ssn: 568470008
    name: 'Homer Simpson'
```

Each field can be of any type. It can, for example, be another struct:

```
>> r.address.street = '742 Evergreen Terrace'
r =
         ssn: 568470008
        name: 'Homer Simpson'
    address: [1x1 struct]
```

We have added a new field to **r**, called **address** and we have made the type of that field be a struct. That inner struct has one field, called **street**, and its value is the string **'742 Evergreen Terrace'**. Notice that when the

structure gets this complex and MATLAB is commanded to display the contents, it resorts to abbreviating the inner structure a bit. Thus, instead of showing the structure of the inner struct, it simply shows that it is a struct. It shows that by means of the word **struct** in the phrase, **[1x1 struct]**. (The "1x1" does not mean that the struct has just one field. Its meaning will be made clear below.) We can add an additional field to the inner struct:

```
>> r.address.city = 'Springfield'
r =
        ssn: 568470008
       name: 'Homer Simpson'
    address: [1x1 struct]
```

There is no visible change in MATLAB's display of the contents of **r** because, while we have added another field to the struct within the third field of **r**, we have not changed the fact that the third field of **r** is a struct. If we want to look at the inner struct itself, we can do it as follows:

```
>> r.address
ans =
    street: '742 Evergreen Terrace'
      city: 'Springfield'
```

Arrays and structs

The value of any field of a struct can be an array of any dimension. Thus, for example,

```
>> r.years_on_air = 1989:2012;
>> r
r =
             ssn: 568470008
            name: 'Homer Simpson'
         address: [1x1 struct]
    years_on_air: [1x17 double]
```

Here we see that, when MATLAB displayed the contents of the last field, it shows only that it is a 1-by-17 array of type **double**. The contents of the field address is shown as a 1-by-1 array of type **struct**. MATLAB chooses to show only the types when the values would occupy too much space on the Command Window. Smaller arrays are displayed in full, as for example:

```
>> pixel.position = [567, 688];
>> pixel.rgb = [145, 155, 134];
>> pixel
pixel =
    position: [567 688]
         rgb: [145 155 134]
```

Since short numeric vectors fit conveniently on one line, they are displayed in full. As a special case of this sort of display, strings, which are stored as row vectors of type char, are displayed in full, as was, for example **'Homer Simpson'** above.

Accessing individual elements of an array that is a field of a **struct** is done as usual. Thus,

```
>> pixel.rgb(1:2)
ans =
   145   155
```

Structs can themselves be elements of an array, as long as the homogeneity of the array is maintained. Thus, if an element of an array is a **struct**, then all elements of that arrays must be **structs** and all must have exactly the same structure:

```
>> clear pict;

>> pict(2,2) = pixel
pict =
2x2 struct array with fields:
    position
    rgb
```

The **clear** command is included above to show that **pict** is non-existent before the second command assigns it a value. As we saw in the subsection of

Matrices and Operators entitled, Accessing Parts of a Matrix, when a value is assigned to an element of variable on a column or row that did not previously exist for that variable, then a new row and column is created along with all the rows and columns necessary to produce a rectangular array of values. In doing that, MATLAB creates elements in addition to the one specified in the command, and it gives each of them an initial value. If the type of the array is numeric, the initial value is zero. If the type is struct, then the initial value is a struct with the value for each field set to the empty matrix, and that is what we find in the elements other than the one to which we assigned a value. For example,

```
>> pict(1,2)
ans =
    position: []
         rgb: []
```

We have seen that structs can have fields that are structs or arrays and that arrays can have elements that are structs. In fact, such nesting involving arrays of structs can extend to any depth.

Arithmetic and Relational operations

Objects of type struct cannot be used as operands by any of the arithmetic operators or relational operators or as the arguments of a functions that expect numeric arguments. However, it is perfectly legal to apply these operators or functions to any *field* of a struct, if it is of numeric type. Here are two examples:

```
>> pict(1,2).position = pict(2,2).position + 1;

>> pict(1,2).position
ans =
   568   689

>> pict(1,2).rgb = round(pict(2,2).rgb./[1 2 3])

>> pict(1,2).rgb
ans =
   145    78    45
```

Dynamic Field Names

When writing functions that manipulate structs, it is sometimes desirable to create field names during the execution of the function. Creation of field names during execution requires additional functionality. One convenient approach is to use MATLAB's dynamic field-naming operation. Here is an example of this operation,

```
>> zoo.lions = 3;

>> zoo.('tigers') = 2;

>> zoo
zoo =
     lions: 3
    tigers: 2
```

The first command is a conventional struct assignment. The second one is the dynamic operation. Note the dot before the left parenthesis. The string `'tigers'`, is read by the "dot-parentheses" operator and converted into a field name. To see how this operator might be used in a program, consider the following function, which allows the user to create a customized structure during its execution,

```
function s = create_struct_dynamic
% S = CREATE_STRUCT_DYNAMIC Create a struct from field names
% and values that are input by the user
while 1
    field_name = ...
       input('Enter a field name (zero to quit): ');
    if field_name == 0
        break;
    end
    field_value = ...
       input('Enter value for this field: ');
    s.(field_name) = field_value;
end
```

Then, we use **`create_struct_dynamic`** to create a structure,

```
>> birth_years = create_struct_dynamic;
Enter a field name (zero to quit): 'Katherine'
Enter value for this field: 1984
Enter a field name (zero to quit): 'John'
Enter value for this field: 1986
Enter a field name (zero to quit): 0
```

and we check to see that the fields that we input are indeed the names of the fields of our new struct:

```
>> birth_years
birth_years =
    Katherine: 1984
         John: 1986
```

Struct functions

There are a number of useful functions designed specifically for working with structs. A list of the most useful ones is given in Table 2.19.

Several of the descriptions in the table include the phrase, "a field whose name is specified by a string". Like the dynamic field-naming operation, these functions allow for the possibility of a string that is formed during the execution of a program to be used with structures . Here is a function that produces the same result as **`create_struct_dynamic`** above, but it does it by

Table 2.19 Struct functions

FUNCTION	DESCRIPTION
`getfield`	returns the value of a field whose name is specified by a string
`isfield`	true if a struct has a field whose name is specified by a string
`isstruct`	returns true if argument is of type struct
`orderfields`	changes the order of the fields in a struct
`rmfield`	removes from a struct a field whose name is specified by a string
`setfield`	assigns a value to a field whose name is specified by a string -or- if the field is not present, adds it and assigns it the value
`struct`	create a struct with fields whose name are specified by strings

means of two of the functions in the table, instead of the dot-parentheses operator:

```
function s = create_struct
% S = CREATE_STRUCT Create a structure from field names
% and values that are input by the user
first_field = 1;
while 1
    field_name = ...
       input('Enter a field name (zero to quit): ');
    if field_name == 0
        break;
    end
    field_value = ...
        input('Enter value for this field: ');
    if first_field
        s = struct(field_name, field_value);
        first_field = 0;
    else
        s = setfield(s, field_name, field_value);
    end
end
```

Note that we must use the function **`struct`** to create the struct initially, but once the struct has been created, we must use **`setfield`** to add fields to it.

Cells

We noted above that structs and arrays can be nested within each other and structs can be nested within structs. What about arrays within arrays? MATLAB provides that option as well, albeit indirectly. It is provided by means of a new data type, called a "cell". To understand the cell, we must first understand a bit about the way in which a program references its variables.

The symbol table

As we saw in the subsection Variable Scope of the section entitled, Functions, when we were discussing the idea of local variables, each variable has its own location in the computer's memory. The operating system (e.g., Mac OS, Linux, or Windows) allocates portions of the computer memory for MATLAB's exclusive use while it is running, and MATLAB, in turn allocates subparts of its portion to variables as they are used. Each time MATLAB encounters a name in a command, either in the Command Window or during the running of a function, it consults a table that is in computer-science terminology called a "symbol table", to determine whether it has already been defined. A **symbol table** is simply a table maintained automatically during the execution of code in some programming languages, including MATLAB, in which the names (or "symbols") of functions and variables and information about them is kept. If the name is not in the symbol table, then it checks the command to see whether it requires that the variable's value be looked up or a function be executed, as it would for `x` in an expression, such as, for example, `x + 5`. If no such variable appears in the table and no function named `x` can be found on the path, then MATLAB halts execution of the program and issues an error message, as for example,

```
??? Undefined function or variable 'x'.
```

If, on the other hand, the command assigns a value to a variable that has not already been defined, for example in `x = 5`, it defines it immediately by placing its name into the symbol table and choosing an area of memory in which to store the variable's values. This is how MATLAB brings a variable into "existence". The area of memory chosen for the variable is a set of bytes, and each byte in the memory has its own unique address. MATLAB will choose a contiguous set of bytes and will put the address of the first byte into the table. In that contiguous set of bytes it puts the value 5, but it also writes a descriptor, which in MATLAB terminology is called a "header", that encodes the type, number of dimensions (two in this case) and the lengths of each dimension (one row and one column in this case). It reserves this area for this variable. Every variable, while it exists, has its own unique area of memory.

You can look at part of the contents of the symbol table. You see it by issuing the command `whos`. For example,

```
>> a = 5;
>> b = int8([5 6 + 2i;-9 9]);
>> c = 'Tiger';
>> whos
  Name      Size      Bytes  Class     Attributes

  a         1x1           8  double
  b         2x2           8  int8      complex
  c         1x5          10  char
```

Here we see that three variables are defined. In the terminology of computer science the set of defined variables is called a "stack frame" or "activation record", or in MATLAB terminology, a workspace, as we learned in Chapter 1. All these terms mean the same thing—either the set of variables that are visible during the execution of statements in the Command Window or the set of variables that are visible during the execution of the statements in a function. What is displayed above is the workspace of the Command Window. Each function also has its own workspace ("stack frame", "activation record") while it is running. We will return to the concept of multiple workspaces in

the section named Functions Reloaded when we study recursion. The dimensions of the array stored for each variable in the table are given in the second column of the printout from **whos**, labeled "Size"; the total amount of memory occupied by the all the values in the variable (more than one value unless the variable is a scalar) is given in the "Bytes" column, and its type is given under "Class". The number of bytes in the variable's header is not included in its byte count. If the variable has a special feature it is listed under Attributes. The addresses in the symbol table are not shown by the **whos** command.

When a command includes a variable that is already in the table, MATLAB finds the name in the table, obtains the elemental type, the dimensions, and address associated with the name, and either uses the value that it finds at that address or, if the variable appears on the left side of an assignment statement, gives the variable a new value. The following command gives an example of both uses,

```
>> c = a
c =
     5
>> whos
  Name      Size      Bytes  Class     Attributes

  a         1x1           8  double
  b         2x2           8  int8      complex
  c         1x1           8  double
```

Here, MATLAB looks up the variable **a**, finds it in the table, uses the address that it finds there for that variable to find its header, discovers that **a** is a 1-by-1 array of type **double**, and uses that information to interpret the bytes that are located after the header to find that its value is 5. It then looks up the variable **c**, finds it in the table, discovers that it is a 1-by-5 array of type **char**, and replaces that value by a scalar 5 of type **double** and changes its header to reflect that new type, shape, and value. (*Notes to experts: First, we hope you are enjoying this book! Second, we are knowingly and purposely omitting some esoteric details in this model, such as reference copying, which are beyond the scope of an introductory text and would be of no help to the non-expert.*) If the new value is too large to fit within the space already reserved for that variable, then MATLAB allocates for **c** enough new space to hold the new value or values, puts the value(s) there, and changes the address in the symbol table for **c** to that of the new space. That would happen, for example, if the new value were a 10-by-10 array of type **double**, as in the command,

```
>> c = rand(10);

>> whos c
  Name        Size              Bytes  Class     Attributes

  c          10x10                800  double
```

It is possible at any time, both while issuing commands in the Command Window, and within a running function, to determine whether a variable exists in the table by using the function **exist**, which returns 1, meaning **true**, if and only if the variable whose name is input as a string is found in the symbol table, as for example,

```
>> exist('c')
ans =
     1
```

It is also possible to remove a variable from the table at any time using the command **clear**:

```
>> clear('c')
>> exist('c')
ans =
     0
```

Pointers

As mentioned above, a very important piece of information about each variable is missing form the printout provided by the command **`whos`**: the variable's address. That information is never revealed by **`whos`** or by any other MATLAB command. MATLAB is different in this regard from some languages, such as C and C++, which provide an operator that returns the address of any variable whose name is given an operand. These languages also permit one variable to hold the address of another variable, which is another practice that MATLAB does not allow. In these languages the former variable is said to hold a "pointer" to the other variable. In fact, in the parlance of programming languages the term **pointer** means simply the address of an object in memory. A variable that holds an address is called a **pointer variable**.

Secret addresses

In C and C++ a pointer's value can be set to any desired value, printed, assigned as the value of a variable, and generally treated like any other number. MATLAB's implementation is a bit more like that of Java. They both keep the addresses of their variables secret. There are debates about language designers' decisions to reveal or sequester the addresses used during the running of a program. When they are revealed, the program has the power to select specific addresses for special uses, as for example input and output to hardware devices using hardware registers with specific addresses. The ability to specify in the code a specific address is a particularly important feature for "embedded" applications, which are applications that run on special-purpose computers in appliances, cars, etc. Having addresses open to direct manipulation is advantageous in these applications, but there are disadvantages too. When addresses are inaccessible to the programmer, it is far less likely that a change in the portion of memory addresses allocated to a program while it runs will affect the outcome of that program. Thus, a MATLAB or Java program that runs properly on one computer is made more likely to run properly on another computer because changes in the address space from one computer to another cannot affect its behavior.

Indirect implementation of heterogeneous arrays

MATLAB uses pointers as a means to provide an indirect implementation of heterogeneous arrays. The indirect implementation is accomplished through the use of the "cell" data type. The **cell** is the type of pointer variables in MATLAB, and only a variable of type **`cell`** can hold a pointer. The name "cell" is used because a single memory location is often called a memory cell. A cell variable may be created in one of three ways. The first way is by means of the function called **`cell`**:

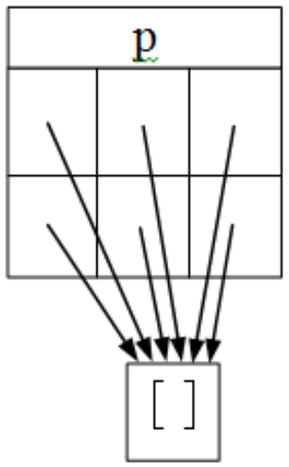
```
>> p = cell(2,3)

p =
     []     []     []
     []     []     []

>> class(p)
ans =
cell
```

The function **`cell`** creates an array of type **`cell`** and initializes each element to contain the address of an empty matrix. (This operation is analogous to the use of the function **`zeros`** to produce a numeric array of zeros.) The result is depicted schematically in Figure 2.46.

Like any matrix in MATLAB, **`p`** is homogeneous: All its elements are of the same type, and in this case the type is cell. Also, like any matrix, we can access any element of **`p`** by putting the indices of the element in parentheses:

```
>> p(1,2)
ans =
    {[]}
```

Figure 2.46 Depiction of pointers to an empty matrix



The set of curly braces around the empty matrix indicates that `p(1,2)` is, as we know it must be, of type `cell`. Those braces are MATLAB's way of telling us that `p(1,2)` contains the address of the empty matrix. The value inside the braces, in this case the empty matrix, is the object whose address is stored in `p(1,2)`.

Curly braces are featured in much of the syntax involving arrays of type `cell`, as we will now see. For example, curly braces are part of the syntax that allows us to gain access to the object to which `p(1,2)` points. We do that by replacing the parentheses by curly braces in the indexing operation,

```
>> p{1,2}
ans =
     []
```

The phrase `(ii,jj)` refers to the pointer that is stored at position `ii,jj`, while the phrase `{ii,jj}` refers to the object to which it points.

Each of the six elements of `p` contains a value that represents a pointer to a location in memory that stores an empty matrix, as Figure 2.46 shows. A more useful array would include pointers to matrices with something in them. Let's put a single scalar into the first element,

```
>> p(1,1) = 17
??? Conversion to cell from double is not possible.
```

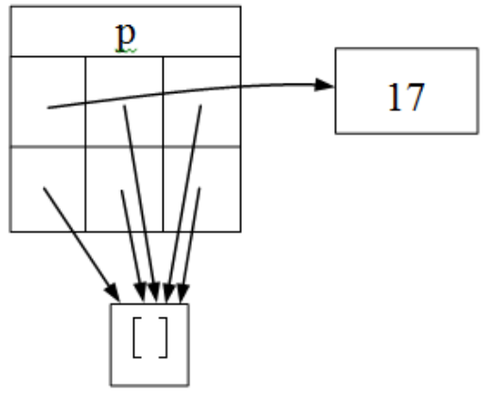
Oops. MATLAB caught us trying to set the address that is stored at `p(1,1)` to 17. Nice try, but it is not going to happen. As mentioned above, MATLAB does not allow us to choose the value of a pointer (or even to know that value that MATLAB chooses). The notation that we use to let MATLAB choose the address, while letting us tell MATLAB what to put at the address, is a pair of curly braces

```
>> p{1,1} = 17
p =
    [17]      []      []
      []      []      []
```

This example shows syntax for assigning a value to an element of a `cell` array. The value that is assigned is not the number 17. Instead, as is illustrated by Figure 2.47, it is a pointer to a secret location in memory where MATLAB will put the number 17.

We get to stipulate what array the element points at, but we must leave it to MATLAB to choose the address at which to place that array and to write that address into `p(1,1)`. MATLAB does not reveal the address at which the new

Figure 2.47 Pointer to a scalar array assigned



array is stored, but it will tell us how much memory it requires. We can do that with another **whos** command,

```
>> whos p
  Name        Size                Bytes  Class     Attributes

  p           2x3                   160  cell
```

The number of bytes depends on the memory model used by the particular MATLAB installation. This example was run on a 64-bit model. Why are so many bytes needed? Well, a detailed look at memory management is beyond the scope of this book, but we will give a glimpse of what is involved. First there are 8 bytes required to hold the value 17, because it is by default of type **double**. Then, there are 112 bytes required to hold the header for the element (1,1) (32-bit installations require only 60). Finally, 8 bytes were reserved for each of the empty-matrix entries by the cell command, and there are five of those remaining: 8+112+5*8 = 160.

Now, let's put a pointer to something else at element (1,2), say a two-element row vector of type double,

```
>> p{1,2} = [-8 pi]
p =
    [17]    [1x2 double]      []
      []              []      []
```

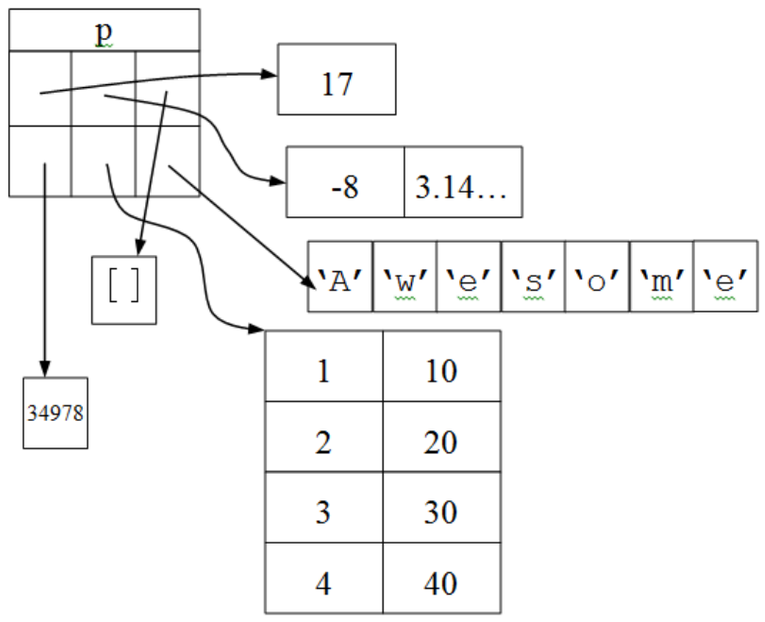
This time, MATLAB gives an abbreviated description of the object that is pointed at by the second element, just as it did when it abbreviated the fields of a struct in the previous subsection, entitled, Structs.

Now, just to get an idea of the great variety of possible objects that can be pointed at by elements of a cell array, let's make the first element on the second row a pointer to a **uint16**, the second element a pointer to a 2-by-4 array of **int8**, and the last element on the second row a pointer to a string,

```
>> p{2,1} = uint16(34978);
>> p{2,2} = int8([1 10;2 20;3 30;4 40])
>> p{end,end} = 'Awesome!'

p =
    [   17]    [1x2 double]              []
    [34978]    [4x2 int8  ]    'Awesome!'
```

With this example we can see clearly how MATLAB achieves its "indirect" implementation of heterogeneous arrays. Figure 2.48 illustrates the situation. The array **p** is certainly not heterogeneous, since all its element are of the same type, **cell**, (a point that is debatable, since pointers to different types are themselves considered to be of different types in C, C++, and Java), but the pointers in those elements point to objects of differing types, and that is why we say that the cell array is an indirect version of a heterogeneous array. In this regard MATLAB is more flexible than C, C++, or Java. These three languages certainly provide for arrays of pointers, but they each require that all

Figure 2.48 *Cell array pointing to multiple types*



pointers in the same array point to objects of the same type (can be overridden in C and C++ but not in Java).

Throughout this subsection, we have been using curly braces to manipulate a cell array that we created by means of the `cell` function. There are three other ways to create a cell array, each involving curly braces. Here are examples showing the syntax of each,

First:

```
>> A = {1, 'hello'; int8(7), [1 2; 3 4]};
```

Second:

```
>> A(1,1) = {1};
>> A(1,2) = {'hello'};
>> A(2,:) = {int8(7), [1 2; 3 4]};
```

Third:

```
>> A{1,1} = 1;
>> A{1,2} = 'hello';
>> A{2,1} = int8(7);
>> A{2,2} = [1 2; 3 4];
```

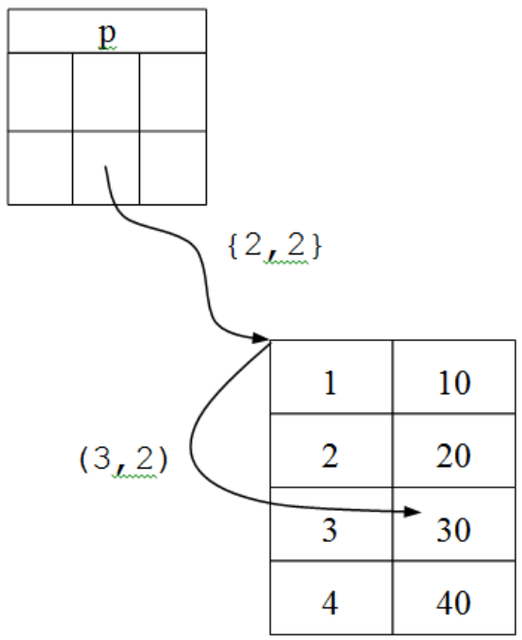
All three of these examples produce the same cell array,

```
>> A
A =
    [1]    'hello'
    [7]    [2x2 double]
```

Accessing subparts of cell arrays

As we have seen, we can access individual elements of `cell` arrays by using subscripting, and there are two options for enclosing the subscripts—parentheses and curly braces. The first option gives us a pointer; the second gives us the object pointed at. What if that object is an array, and we wish to access individual elements within it? We use a concatenation of indices in curly braces followed by indices in parentheses. Here is an example, which is depicted in Figure 2.49,

Figure 2.49 Concatenation of subscripts




```
>> p{2,2}(3,2)
ans =
    30
```

The first set of indices, which are enclosed in curly braces, identify element `2,2` in the `cell` array. The second set of indices, which are enclosed in parentheses, identify element `3,2` within the array pointed at by cell element `2,2`.

The colon operator can be used to assign a subarray, as usual. It can be used directly, as for example,

```
>> q = p(1:2,2:3)
q =
    [1x2 double]              []
    [4x2 int8  ]    'Awesome!'
```

and it can be used in the concatenated parentheses,

```
>> q{2,2}(2:4)
ans =
wes
```

Pointer model forbids two cells from pointing at the same object.

The MATLAB pointer model differs from pointers in other languages, such as C, C++, and Java (and many others), in an important way: It does not allow two cells to point to the same object. (*Note to experts: We are talking about the MATLAB pointer model.*) Here is an example to show what that means:

First we produce a cell object called **c1**:

```
>> c1 = {[1 2],[10,20]}
c1 =
    [1x2 double]    [1x2 double]
```

It consists of two elements, each pointing to a 1-by-2 array of type **double**. Now consider the following statement:

```
>> c2 = c1
c2 =
    [1x2 double]    [1x2 double]
```

It looks like a simple assignment operation, so we might assume that the pointers contained in **c2** are the same as those in **c1**. This would mean that they both point to the same objects. Indeed that is how it appears if we look at those objects:

```
>> c1{1,1}
ans =
     1     2
>> c2{1,1}
ans =
     1     2
>> c1{1,2}
ans =
    10    20
>> c2{1,2}
ans =
    10    20
```

But what happens if we change one of the objects? Let's change object at which **c1(1,1)** is pointing:

```
>> c1{1,1} = 'Cupcakes'
c1 =
    'Cupcakes'    [1x2 double]
```

The first element of **c1** is now pointing at some nice cupcakes. If **c2(1,1)** contains the same pointer as **c1(1,1)**, then when we ask MATLAB to show us what it is pointing at we should find those same cupcakes:

```
>> c2{1,1}
ans =
     1     2
```

Instead, we find that the object pointed at by **c2(1,1)** is unchanged. What happened?

The answer is that the pointer model enforced by MATLAB does not allow **c2** to point to the same object as **c1**. In its version of pointer assignments, when MATLAB carries out the assignment **c2 = c1**, it makes copies of all the objects pointed at by **c1**, and makes **c2** point at the copies. As a result it is impossible to change an object pointed at by one cell by manipulating it via another cell. This is a very stringent limitation. It means that cells are not useful for applications involving so-called "linked lists", in which chains or networks of pointers are formed from one object to another (typically involving structs as well as pointers). However, (1) linked lists are not typically needed or encountered in numerical applications, so it is not a serious drawback for this language, which is after all designed primarily for numerical applications, and (2) MATLAB supports linked-lists via its Object-Oriented Programming feature, which we will see in Chapter 3 in the section entitled, appropriately enough, Object-Oriented Programming.

Distinguishing between a pointer and the object at which it points

MATLAB uses various notations to indicate that the elements of a **cell** array are not actually matrices, but instead are pointers to matrices. In its depiction of the contents of **p** above, for example, it encloses all numeric matrices in square brackets and it encloses a string in single quotes. That notation is supposed to say to the reader, "pointer to" the respective objects. The distinction is a bit clearer in the following examples:

```
>> q = p(1,1)
q =
    [17]
```

Here we have copied the pointer in **p(1,1)** into the variable **q**, making **q** of type **cell**. MATLAB indicates that the value stored in **q** is a pointer by enclosing 17 in square brackets. We can verify the type of **q** with the class function,

```
>> class(q)
ans =
cell
```

If we want access the object that is pointed at by **p(1,1)**, instead of the pointer that is stored in **p(1,1)**, we use curly braces:

```
>> r = p{1,1}
r =
    17
```

The type of the object retrieved by the using **p{1,1}** is **double**, and it is assigned to **r**. Therefore, **r** is of type **double** , not **cell**, as can also be verified with the function **class**,

```
>> class(r)
ans =
double
```

Finally, compare the way MATLAB displays the contents of **p(2,3)**,

```
>> p(2,3)
ans =
    'Awesome!'
```

This element contains a pointer to a string. To indicate that it is a pointer to a string, and not a string, MATLAB encloses the string in single quotes. When we access the string itself, MATLAB omits the quotes. Thus,

```
>> p{2,3}
ans =
Awesome!
```

The characters in the string are printed without quotes and without indentation, as they are for the display of any string,

```
>> s = 'Simply amazing!'
s =
Simply amazing!
```

Uses for cell arrays

Cell arrays are useful when it is necessary to combine sets of data of differing types and/or sizes and it is desired to use numerical indices instead of field names, as are required for structs. For example, if a user is inputting a set of strings that will later be operated on, it is convenient to put pointers to each of them in a **cell** vector:

```
n = 0;
while 1
    s = input('Name please: ');
    if isempty(s)
        break;
    else
        n = n+1;
        name_list{n} = s;
    end
end
fprintf('Here is a numbered list\n');
for ii = 1:n
    fprintf('%3d: %s\n', ii, name_list{ii});
end
```

Here is an example of this code in action,

```
Name please: 'Marge'
Name please: 'Homer'
Name please: 'Maggie'
Name please: 'Lisa'
Name please: 'Bart'
Name please: []
Here is a numbered list
  1: Marge
  2: Homer
  3: Maggie
  4: Lisa
  5: Bart
```

Arithmetic and relational operators

Neither arithmetic operators nor relational operators allow the type cell for their operands.

Cell functions

MATLAB provides a number of functions designed to assist you in using cell arrays. Table 2.20 gives a partial list.

Table 2.20 *Cell functions*

FUNCTION	DESCRIPTION
`cell`	create an array of type cell
`celldisp`	show all the objects pointed at by a cell array
`cellfun`	apply a function to all the objects pointed at by a cell array
`cellplot`	show a graphical depiction of the contents of a cell array
`cell2struct`	convert a cell array into a struct array
`deal`	copy a value into output arguments
`iscell`	returns true if argument is of type cell
`num2cell`	convert a numeric array into a cell array

Additional Online Resources

- Video lectures by the authors:
  - Lesson 7.1 Introduction to Data Types (20:27)
  - Lesson 7.2 Strings (29:04)
  - Lesson 7.3. Structs (14:51)
  - Lesson 7.4 Cells (21:47)

Concepts From This Section

Computer Science and Mathematics:

- data type, or type
- mixed-mode arithmetic
- strings
  - encoded characters
  - ASCII encoding
- **`structs`**
  - heterogeneous structure
  - element is called a field
  - indexed by field names
- symbol table
- pointers

MATLAB:

- the function **`class`** returns the type of its argument
- conversion functions
- numeric types
  - floating point
    - **`double`**, **`single`**
  - integer
    - **`int8`**, **`int16`**, **`int32`**, and **`int64`**
    - **`uint8`**, **`uint`**, **`uint`**, and **`uint`**
- string
  - a row vector
  - type is **`char`**
- **`struct`** type
- **`cell`** type
  - holds pointers
  - created with function **`cell`**
  - syntax of commands employs curly braces
  - **`cell`** array may hold pointers to arrays of differing types

Practice Problems

Problem 1. Write a function named **`safe_int8`** that takes one array as an input argument and returns one array as an output argument. If it is called like this: **`B = safe_int8(A)`**, then **`B`** is set equal to an array of the same size and shape as **`A`**, and, if all the elements of **`A`** are integers and all of them fall within the range of **`int8`**, then **`B`** is of type **`int8`**. Otherwise, **`B`** has the same type as **`A`**. Here are some examples of its behavior:

```
>> A
A =
     1     0     3
     4     5     6
>> B = safe_int8(A),class(B)
B =
    1    0    3
    4    5    6
ans =
int8
>> A
     1     0   345
     4     5     6

>> B = safe_int8(A),class(B)
B =
     1     0   345
     4     5     6
ans =
double
>> A
A =
            1          1.05             3
            4             5             6
>> B = safe_int8(A),class(B)
B =
            1          1.05             3
            4             5             6
ans =
double
```

?

Problem 2. Write a function named **safe_int** that takes one array as an input argument (the function does not have to check the format of the input) and returns one array as an output argument. If it is called like this:

```
B = safe_int(A)
```

then **B** is set equal to an array of the same size and shape as **A**, and, if all the elements of A are integers and all of them fall within the range of one or more of the signed integer types, then the type of **B** is the signed integer type that has the smallest range that contains all the values of **A**. Otherwise, **B** has the same type as **A**. Here are some examples of its behavior:

```
>> B = safe_int([1 345]),class(B)
B =
      1         345
ans =
int16
>> B = safe_int([1 -2^31]),class(B)

B =
      1 -2147483648
ans =
int32

>> B = safe_int([1 2^31]),class(B)
B =
      1  2147483648
ans =
int64

>> B = safe_int([1 2^63]),class(B)
B =
   1.0e+18 *
    0.0000    9.2234
ans =
double
```

Problem 3. Write a function named **odd_shift** that takes one string and one integer as input arguments (the function does not have to check the format of the input) and returns one string as an output argument. If it is called like this:

```
s2 = odd_shift(s1,shift)
```

then **s2** is a string that is the same as **s1** except that the characters with odd indexes have their codes increased by **shift**. Here is an example,

```
>> s2 = odd_shift('Conrad_and_Bean',12)
s2 =
OozrmdkazdkBqaz

>> double('Conrad_and_Bean')
ans =
  Columns 1 through 12
    67   111   110   114    97   100    95    97   110
100    95    66
  Columns 13 through 15
   101    97   110

>> double(s2)
ans =
  Columns 1 through 12
    79   111   122   114   109   100   107    97   122
100   107    66
  Columns 13 through 15
   113    97   122
```

?

Problem 4. Write a function named **`cyclic_shift`** that takes one string and one vector of integers as input arguments (the function does not have to check the format of the input) and returns one string as an output argument. If it is called like this: **`a = cyclic_shift(s,shifts)`**, then **`a`** is a string whose characters are the result of adding numbers to the codes of the characters in **`s`**. Suppose **`s`** is of length **`M`** and **`shifts`** is of length **`N`**. If **`M <= N`**, then for **`m = 1`** to **`M`** the code of **`a(m)`** has been shifted from that of **`s(m)`** by **`shifts(m)`**. Otherwise, for **`m = 1`** to **`N`** the code of **`a(m)`** has been shifted from that of **`s(m)`** by **`shifts(m)`** and then **`shifts`** is recycled, **`a(N+1)`** has been shifted by **`shifts(1)`**, **`a(N+2)`** has been shifted by **`shifts(2)`**, etc. Here are two examples:

```
>> a = cyclic_shift('Isaac',[1,6,4,2,1,7,2,7])
a =
Jyecd

>> double('Isaac')
ans =
    73   115    97    97    99

>> double(a)
ans =
    74   121   101    99   100

>> a = cyclic_shift('Principia',[2, -3, 0, 7])
a =
Roiuefppc

>> int16('Principia')
ans =
     80    114    105    110     99    105    112    105
97

>> int16(a)
ans =
     82    111    105    117    101    102    112    112
99
```

Problem 5. Write a function called **`raising_the_bar`** that takes one string (i.e., a row vector of type **`char`**) as an input argument (it does not have to check the format of the input) and returns one string as an output argument. If it is called like this, **`s2 = raising_the_bar(s1)`** then **`s2`** is identical to **`s1`** except that every underscore (**`_`**) has been changed to a dash (**`-`**). Here is an example of the function being used:

```
>> raising_the_bar('1966_12_18--A Day in the Life')
ans =
1966-12-18--A Day in the Life
```

?

Problem 6. Write a function called **`jumping_the_shark`** that takes one string (i.e., a row vector of type **`char`**) as an input argument (it does not have to check the format of the input) and returns one string as an output argument. If it is called like this, **`s2 = jumping_the_shark(s1)`** then **`s2`** is identical to **`s1`** except that every occurrence of the string **`'shark'`** has been removed. Here is an example of the function in action:

```
>> s1
s1 =
The only good shark is a dead shark, excepting sand
sharks.
>> safe_to_go_in = jumping_the_shark(s1)
safe_to_go_in =
The only good is a dead  excepting sand.
```

Problem 7. Write a function called **mean_max** that takes one two-dimensional array as an input argument (it does not have to check the format of the input) and returns one column vector of structs as an output argument. If it is called like this, **mm = mean_max(A)** then there is one element of **mm** for each row of **A**, and each element of **mm** has two fields with the field names **mean** and **max**. The type of the field named **mean** is **double** regardless of the type of **A**, and **mm(ii).mean** equals the mean of row **ii** of **A**. The type of the field named **max** is the same as the type of **A**, and **mm(ii).max** is equal to the maximum value on row **ii** of **A**. Here is an example of the function in action:

```
>> A
A =
    69   123    11   116   111    80    31     7    43
   127     1    51    24    74    45    16   115   115
    10    99    34    34    70    66    24   120    47
    57   104   102    19    19    52    31    63    15
    14   111    55    18   109    10    53    63   100
>> mm = mean_max(A);
mm =
5x1 struct array with fields:
    mean
    max
>> class(mm(1).mean)
ans =
double

>> class(mm(2).max)
ans =
int8
```

```
>> mm(1),mm(2),mm(3),mm(4),mm(5)
ans =
    mean: 65.6667
     max: 123
ans =
    mean: 63.1111
     max: 127
ans =
    mean: 56
     max: 120
ans =
    mean: 51.3333
     max: 104
ans =
    mean: 59.2222
     max: 111
```

?

Problem 8. Write a function called **`sparse_array_struct`** that takes one two-dimensional array as an input argument (it does not have to check the format of the input) and returns one column vector of structs as an output argument. If it is called like this,

```
As = sparse_array_struct(A)
```

then each element of the output vector **`As`**, represents one of the non-zero elements of **`A`**. Each element of **`As`** has three fields: **`row`**, which is of type **`int16`** and contains the row number of a non-zero element of **`A`**, **`col`**, which is also of type **`int16`** and contains the column number of the same non-zero element of **`A`**, and **`val`**, which is of the same type as **`A`** and contains the value of that non-zero element of **`A`**. The non-zero values of **`A`** appear in column-major order in **`As`**. If **`A`** is empty, then **`As`** is empty. Here is an example of the function in action:

```
>> A
A =
    81    28     0     0     0     0
    90    55    49     0     0     0
     0    95    80    65     0     0
     0     0    15     4    39     0
     0     0     0    85    65    10
     0     0     0     0    17    82

>> As = sparse_array_struct(A);

>> size(As)
ans =
    16     1
```

```
>> As(1),As(2),As(3),As(4),As(5)
ans =
    row: 1
    col: 1
    val: 81
ans =
    row: 2
    col: 1
    val: 90
ans =
    row: 1
    col: 2
    val: 28
ans =
    row: 2
    col: 2
    val: 55
ans =
    row: 3
    col: 2
    val: 95
```

Problem 9. Write a function called `price_list` that takes no input arguments and returns one two-dimensional cell vector as an output argument. If it is called like this, `items = price_list`, then each row of `items` contains the name and the price of one item on a price list. The names and the prices are entered by the user. The function gives a general prompt and then repeatedly prompts for the name, which is entered without single quotes, and the price. The required form of the prompts is given in the example below. The user hits Enter after each entry. If after either of the prompts, the user hits Enter without typing anything, the function stops and returns `items`. HINT: The following format of the call of the function `input`, as shown in the help system, will be helpful: `STR = input(PROMPT,'s')`. Here is an example of the function in action:

```
>> items = price_list
Enter items and prices (empty item ends list)
Item name: Ink jet paper 500 sheets
Price of Ink jet paper 500 sheets: 11.49
Item name: Lexmark Black Ink Cartridge
Price of Lexmark Black Ink Cartridge: 19.99
Item name: Manila folders, pack of 100
Price of Manila folders, pack of 100: 6.99
Item name: Stapler
Price of Stapler: 13.99
Item name: Desk tape dispenser
Price of Desk tape dispenser: 3.99
Item name:
items =
    'Ink jet paper 500 sheets'        [11.4900]
    'Lexmark Black Ink Cartridge'     [19.9900]
    'Manila folders, pack of 100'     [ 6.9900]
    'Stapler'                         [13.9900]
    'Desk tape dispenser'             [ 3.9900]
```

?

Problem 10. Write a function called **`sparse_array_cell`** that takes one two-dimensional array as an input argument (it does not have to check the format of the input) and returns one column vector of **`structs`** as an output argument. If it is called like this, **`Ac = sparse_array_cell(A)`** then each element of the output vector **`Ac`**, represents one of the non-zero elements of **`A`**. Each row of **`Ac`** has three elements: The first, which is of type **`int16`**, contains the row number of a non-zero element of **`A`**. The second, which is also of type **`int16`**, contains the column number of the same non-zero element of **`A`**. The third, which is of the same type as **`A`**, contains the value of that non-zero element of **`A`**. The non-zero values of **`A`** appear in column-major order in **`Ac`**. If **`A`** is empty, then **`Ac`** is empty. Here is an example of the function in action:

```
>> A
A =
    81    28     0     0     0     0
    90    55    49     0     0     0
     0    95    80    65     0     0
     0     0    15     4    39     0
     0     0     0    85    65    10
     0     0     0     0    17    82

>> Ac = sparse_array_cell(A);

>> class(Ac)
ans =
cell>> size(Ac)
ans =
    16     3

>> Ac{1,:}
ans =
     1
ans =
     1
ans =
    81

>> Ac{2,:}
ans =
     2
ans =
     1
ans =
    90

>> Ac{3,:}
ans =
     1
ans =
     2
ans =
    28

>> Ac{4,:}
ans =
     2
ans =
     2
ans =
    55

>> Ac{5,:}
ans =
     3
ans =
     2
ans =
    95
```

File Input/Output

Objectives

Files are named areas in permanent memory for storing data that can be used as input or output to MATLAB and to other programs.

(1) We will be introduced to MATLAB's most important methods for reading and writing files.

(2) We will learn how to create, read from, and write to MAT-files, Excel® files, text files, and binary files.

(3) We will learn how to navigate among folders with MATLAB commands.

Reading and writing data in permanent memory is an essential task for many programs.

So far, whenever we have needed to supply input to a function, we have passed data to it via its input arguments or, in rare cases, by means of the function **`input`**, through which data is entered by the user typing on the keyboard. Similarly, when we wanted a function to produce output, we used output arguments, or in rare cases, employed **`fprintf`** to send the output to the Command Window for the user to read. This sort of input/output scheme works well for many self-contained projects, but it is often desirable to get input from files and to send output to them. A **file** is an area in permanent storage, typically residing on a disk drive, that can be named, renamed, moved from one folder to another and from one computer to another, inspected by users, and accessed by other programs, and it is all managed by the operating system. The key word in this long description is "per-

manent", by which we mean that the data in a file is not lost when the computer is shut down—unless disaster strikes and there is no backup! (Nothing after all is truly permanent.) When referring to input and output, it is common to abbreviate them as **I/O**, which stands for input/output. Since this section deals with input and output to files, it is entitled, "File Input/Output".

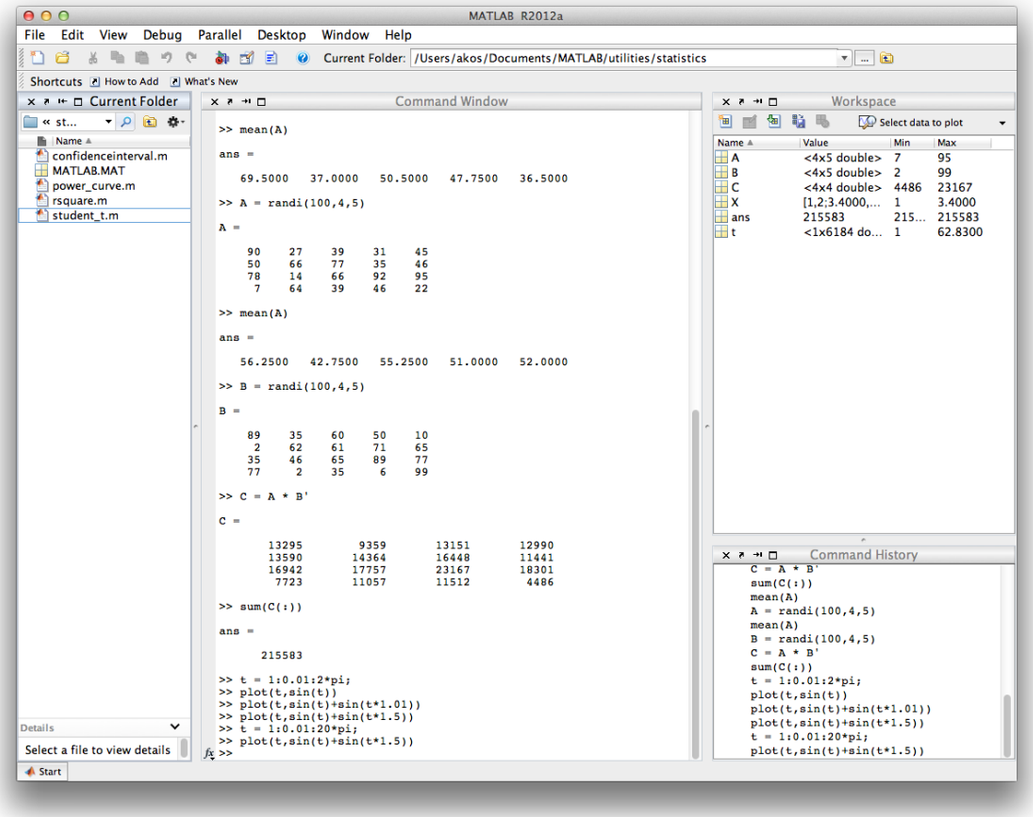
Input from files is important not only because it is permanent but also because the data to be processed can come from another program that has written its data into files. Output to files also has an importance beyond permanence: output written into a file by MATLAB programs can serve as input to other programs that read their input from files. MATLAB supports several approaches to file I/O, and we will learn how to exploit the most important ones in this section. Each approach is defined by the type of file used for I/O, and MATLAB provides sets of functions to work with several types, the most important being Mat-files, Excel files, text files, and binary files. We treat each of these file types in separate sections below, but first we will learn how to find the files we need.

Moving Around

Files are created, named, renamed, moved, and deleted by the Operating System of the computer on which they reside. The three most common operating systems are Windows®, Mac OS®, OS X®, which is the name for Mac OS 2012, and Linux®. These operating systems are different in many respects, but they behave very similarly with regard to files. One difference is that, while Windows and Mac OS (we will use "Mac OS" for all versions including OS X) refer to a named collection of files as a "folder", Linux refers to it as a "directory". MATLAB uses both terms, and so does this book. We will often use the unbiased term "folder-directory" to emphasize their equivalence.

As we saw in Introduction to MATLAB, when MATLAB is run without changing the default layout of its Desktop, the Command Window is situated between a "Current Folder" window on its left and a vertical stack of two windows on its right—"Workspace" and "Command History". An example is shown in Figure 2.50.

Figure 2.50 *MATLAB desktop with the Current Folder window on the left side.*



That window on the left gives the name of the "Current Folder", known as the "Current Directory" in Linux, and it displays a list of files in that folder-directory. You already know that the current folder is the first place that MATLAB looks to find M-files (it looks in the list of folders in the Path if it doesn't find what it is looking for there). The current folder has another important meaning: It is the first folder-directory in which MATLAB looks for files that are to be read for input, and, it is the place that it will put any new file that is created to hold output data (unless we tell it explicitly to put the new file elsewhere).

It is possible to change directories using the Current Folder window, using mouse clicks just as you would when using your operating system's normal folder-directory navigation windows, but it is also possible to do it by means of the MATLAB command **cd** , which stands for "**c**hange **d**irectory". Furthermore, it is possible to see the name of the current folder-directory with the command **pwd** ("**p**rint **w**orking **d**irectory", where "working" means the same thing as "current"), and it is possible to list the names of the files in the current directory using the command **ls** ("**ls**t files). Here is an example of the use of these commands:

```
>> pwd

ans =

/Users/akos/Documents/MATLAB/utilities/statistics
```

The command **pwd** returns a string containing the complete path of the current folder. The slashes separate names of folder-directories, and this string tells us that the current drive contains the folder-directory **Users**, which contains the folder-directory **akos**, which contains **Documents**, which contains **MATLAB**, which contains the folder-directory **utilities**, which (finally!) contains the folder-directory **statistics**. A parent-child metaphor is often used instead of the "container" description: Thus, we may say for example that the folder **utilities** is the "parent" of **statistics**, and **statistics** is the "child" of **utilities**. A parent folder-directory can have any number of children, as with real parents and children, but a child can have only one parent folder-directory. In that sense, the analogy to the real world isn't perfect. Also, there is a metaphor involving altitude. Since the parent is a step higher in the hierarchy than the child, we tend to think of **utility** as being a step higher than **statistics** with the drive itself sitting at the very top. Note that on Windows, drives are named by letters, such as C or D, so the path above will look like this if MATLAB is running under Windows:

```
>> pwd

ans =

C:/Users/akos/Documents/MATLAB/utilities/statistics
```

Since **statistics** is the last name in the string, it is the name of the current folder-directory.

It is common to say that "we" are "in" the current folder-directory (the "container" metaphor), so, let's say it: We are now in **statistics**. Now let's have a look around. We can do that with the command **ls**:

```
>> ls
.
..
confidenceinterval.m
matlab.mat
power_curve.m
rsquare.m
student_t.m
```

This example was run on Windows, for which the command **ls** returns a two-dimensional array of characters, with one name on each row in alphabetical order (padded with blanks on the right where necessary) with each filename that it finds in the current folder-directory (in this case in statistics) occupying one row. On Mac OS, the output looks similar, but it is a row vector of characters as opposed to a two-dimensional array.

The list of names also includes the names of any folder-directories that it finds in there. There happen to be no folders inside **statistics**, so there are none listed. If there were, they would look like files with no dot and no file extension after the dot. Some files have no dot in the name. In that case there is no indicator to tell us whether a name belongs to a file or not. If we compare the list returned by **ls** with the list of files shown for the Current Folder in Figure 2.50, we see that they agree, but the first two lines returned by **ls**, the single dot and the double dot, are absent from the list in the figure.

The single dot stands for the current directory, and the double dot stands for the parent directory, which in this case is the directory named **utilities**.

Suppose that we decide that we would rather be up in the parent directory **utilities** instead of in **statistics**. We have a couple options: the hard way and the easy way. The hard way looks like this:

```
>> cd '/Users/akos/Documents/MATLAB/utilities'
```

Note that we used single quotes around the string this time. They are necessary only when the filename includes one or more blank characters. Let's go back down into statistics for a minute, so we can go back up again:

```
cd statistics
```

Now let's move up to the parent directory the easy way. We still use **cd**, but we specify **..** (two dots) instead of a name:

```
>> cd ..
```

This command moved us up to the parent directory, and to prove it, we use **pwd** again:

```
>> pwd

ans =

/Users/akos/Documents/MATLAB/utilities
```

As expected, we are now in **utilities**.

As you might have guessed, since the single dot stands for the current folder-directory, the command **cd .** accomplishes nothing! Its presence in the list returned by **ls** makes little sense in another way too: it suggests that a directory is inside itself. We will leave this deep philosophical question to those who like to work on deep philosophical questions.

Other commands allow the creation of a new folder-directory—**mkdir**, the deletion of a directory—**rmdir**, the moving of a file from one place to another—**movefile**, the copying of a file—**copyfile**, and the deletion of a file—**delete**. Each of these commands is relatively simple to use, as can be seen by reading their descriptions with the **help** command.

That is all we need to know about moving around in folder-directories. It's time to learn how to manipulate the files inside them.

MAT-files

As we found in the very first section of Chapter 1, Introduction to MATLAB, in the subsection named, Saving variables, when we complete a session of programming with MATLAB, we often find that there are numerous variables in the Workspace with values that we would like to save for our next session. We can see the names of these variables with the command **whos**, and we can look at the values of the individual variables by typing their names into the Command Window, or printing them with **fprintf**, but unless we write those numbers down on paper or type them into some file, they will be lost when we shut down MATLAB. For example, suppose we type **whos** and see this:

```
>> whos
  Name          Size                Bytes  Class     Attributes

  A             1x5                    40  double
  B             1x5                    40  double
  C             1x5                    40  double
  a           140x42                11760  char
  ans           1x61                  122  char
  b             1x1                     8  double
  c             1x1                     8  double
  d             1x1                     8  double
  ii            1x1                     8  double
  x             1x1                     8  double
  y             1x1                     8  double
```

As in Saving variables, we can save these variables into a MAT-file simply by issuing the command **save**,

```
>> save
Saving to:
/Users/akos/Documents/MATLAB/Project 8/Plan 9/matlab.mat
```

MATLAB responds with a comforting message that tells us that it has saved all our variables. Specifically, it has created a MAT-file, named it, imaginatively enough, **matlab.mat**, placed it in the current folder-directory, which happens to be called **Plan 9**, and saved every variable including the variable's name, type, size and shape, and the values of all its elements, in that file. We can now exit MATLAB and, if we want to, shut down our computer, and when we restart MATLAB, we can get these variables back by putting MATLAB back into the directory

```
/Users/akos/Documents/MATLAB/Project 8/Plan 9/
```

and typing the command **load**, which, as we also saw in Saving variables, tells us that it read the data from a MAT-file,

```
>> load
Loading from: matlab.mat
```

All our variables are right back in the workspace, as can easily be verified with **whos**, and they all have the same values they had when we last used save in this directory.

If you try to inspect a MAT-file with any word processor or editor, including the MATLAB editor, you will see nothing but a hodgepodge of strange characters. That happens because, in order to save space, MATLAB uses a compressed format that is not compatible with word processors and text editors. Many word processors use their own special formats, while text editors, like MATLAB's m-file editor, TextEdit on OS X, or Notepad on Windows work with text files. We will see that MATLAB can save data into text files, and we will learn what text files are in a subsection below, aptly entitled, "Text files".

You can repeat this process as many times as you wish: work, save, load…. It is important to know though that each time you save into **matlab.mat**, all previous variables recorded in that file are removed and replaced by the current set. It is not a cumulative save. On the other hand, **load** *is* cumulative. When you load from a MAT-file, the current variables in the workspace are not removed. If, for example only the variables **Eros** and **Tanna** exist in the workspace and only the variables **Jeff** and **Paula** exist in **matlab.mat**, then **load** will cause **Jeff** and **Paula** to be added to the workspace, and **Eros** and **Tanna** will still be there. There is no danger of losing a variable from the workspace. The only caveat is that if the same variable exists in both the MAT-file and the workspace, then the current value of the variable and its type, shape, and size in the workspace will be lost. They will be replaced by the value, type, shape, and size that they have in the MAT-file.

Many options are available with the two commands, **save** and **load**. You can read about them in the Help facility, but we will look at the two handiest ones.

First, you can specify a specific MAT-file name.

```
>> save Plan9
```

This time, MATLAB saves the workspace to a file named **Plan9.mat** in the current directory, but it does not tell us what it did (because we do not need to be told a name that we just typed). The workspace in **Plan9** can be restored with the command

```
>> load Plan9
```

Second, when you specify the MAT-file name, you can also specify that only a subset of the variables in the workspace be saved into the MAT-file by listing them (without commas). For example,

```
>> save Plan9 x y
```

Excel® Files

The program Microsoft Excel® is a widely used data-analysis tool that organizes data into spreadsheets, also known as "worksheets". Excel utilizes special file formats that allow the user to store a set of these spreadsheets, which is called a "workbook," in one file. There are ten or so formats in which Excel can store a workbook, partly to allow different features to be supported, but also to maintain backward compatibility with previous versions of the program. Excel is used so heavily throughout the world that many other data-analysis application programs now provide the option of storing their results in one or more of Excel's formats, and of reading files written by Excel and Excel files written by other applications. MATLAB is one such application.

Reading Excel files

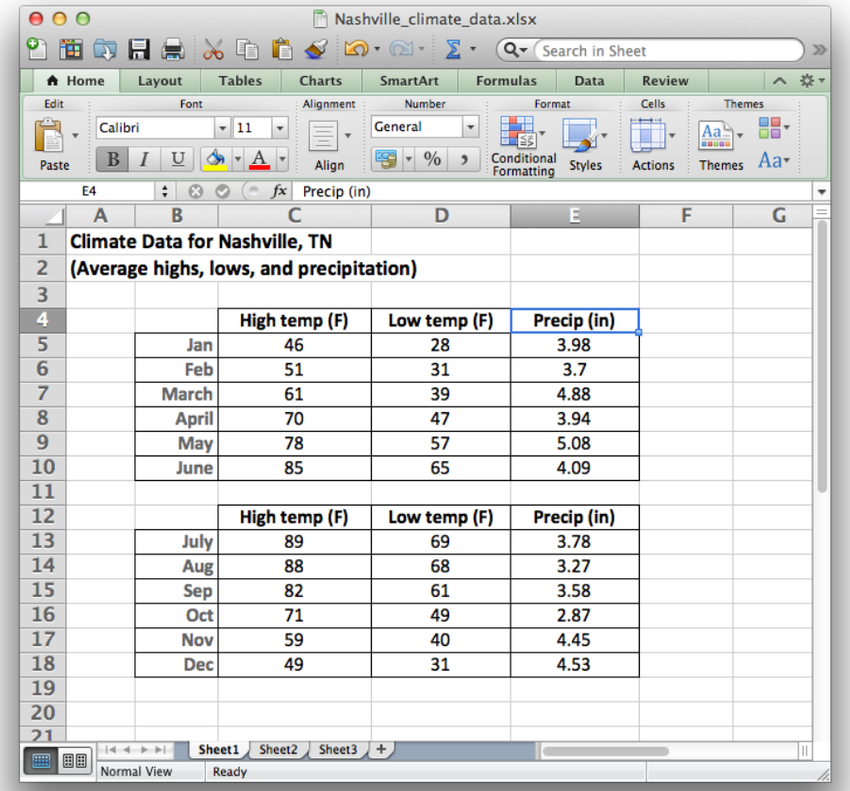
Let's suppose that some application has generated an Excel file containing a workbook with just one spreadsheet containing the average monthly high and low temperatures and average monthly precipitation for Nashville, Tennessee, or that we have used Excel to enter the information by hand. Figure 2.51 shows how the spreadsheet appears while it is open in Excel.

We can tell MATLAB to read the data in that file by using the function **xlsread**. Here is an example session:

```
>> [num,txt,raw] =
xlsread('Nashville_climate_data.xlsx');
```

The input argument is the name of the Excel file in the form of a string. Note that the file name in this example has the file extension **xlsx**. This extension identifies the version of Excel file being read. The version is xlsx, and it is stored in an xlxs-file. MATLAB can read all xlsx-files and xls-files, and, if Excel is installed on the computer, then MATLAB can read any version of Excel file that is supported by the installed version of Excel.

Figure 2.51 *Excel spreadsheet*



The first output argument, **num**, will receive the values of only those cells that contain numbers; the second argument, **txt**, will receive the values of only those cells that contain text. The last argument, **raw**, will receive all the data in the spreadsheet. The types of these output arguments differ. The first argument is always of type **double**:

```
>> class(num)
ans =
double
```

The other two are of type **cell**:

```
>> class(txt)
ans =
cell

>> class(raw)
ans =
cell
```

Let's take a look at the contents of the outputs, starting with the first output variable, **num**:

```
>> num
num =
          46          28        3.98
          51          31         3.7
          61          39        4.88
          70          47        3.94
          78          57        5.08
          85          65        4.09
         NaN         NaN         NaN
         NaN         NaN         NaN
          89          69        3.78
          88          68        3.27
          82          61        3.58
          71          49        2.87
          59          40        4.45
          49          31        4.53
```

The size of **num** is 14-by-3. MATLAB determines these dimensions by finding the smallest rectangular array of cells that contains all the numbers in the spreadsheet, ignoring cells that contain non-numeric values. In this case the rectangle comprises the cells in rows 5 through 18 and columns C through E. It then copies the contents of the cell in the upper left corner of that rectangle, which is cell C5, into **num(1,1)**, and copies the rest of the cells in that rectangle into elements in the array that have the corresponding horizontal and vertical offsets from their respective upper left corners. If a cell within this rectangle does not contain numeric data, either because there is non-numeric data in it or because it is empty, then the value put into the corresponding element of **num** is **NaN**, which, as we learned in Data Types, means "Not a Number". In this case, the **NaN**s in **num(7,:)** are there because of the empty cells of Row 11 of the spreadsheet, and the **NaN**s in **num(8,:)** are there because the values in the cells of Row 12 are the strings, "High temp (F)", "Low temp (F)", and "Precip (in)" and hence, are non-numeric.

Next, we look at the second output variable, **txt**:

```
>> txt
txt =
  [1x30 char] ''      ''               ''              ''
  [1x40 char  ''      ''               ''              ''
  ''          ''      ''               ''              ''
  ''          ''      'High temp (F)'  'Low temp (F)'  'Precip (in)'
  ''          'Jan'   ''               ''              ''
  ''          'Feb'   ''               ''              ''
  ''          'March' ''               ''              ''
  ''          'April' ''               ''              ''
  ''          'May'   ''               ''              ''
  ''          'June'  ''               ''              ''
  ''          ''      ''               ''              ''
  ''          ''      'High temp (F)'  'Low temp (F)'  'Precip (in)'
  ''          'July'  ''               ''              ''
  ''          'Aug'   ''               ''              ''
  ''          'Sep'   ''               ''              ''
  ''          'Oct'   ''               ''              ''
  ''          'Nov'   ''               ''              ''
  ''          'Dec'   ''               ''              ''
```

The variable **txt** is an 18-by-5 array of cells, and MATLAB determines these dimensions in a manner similar to the way it determined them for **num**, but this time it finds the smallest rectangular array of cells that contains all the strings in the spreadsheet, ignoring cells that do not contain strings. It copies the string in the upper left corner of the rectangle into **txt(1,1)** and places the rest of the strings in the corresponding relative positions in **txt**. If a cell within this rectangle does not contain a string, either because it contains non-string data or because it is empty, then an empty string is placed in the corresponding element.

Finally, we look at the third output variable, **raw**:

```
>> raw
raw =
    [1x30 char] [  NaN] [ NaN] [ NaN] [  NaN]
    [1x40 char] [  NaN] [ NaN] [ NaN] [  NaN]
    [      NaN] [  NaN] [ NaN] [ NaN] [  NaN]
    [      NaN] [  NaN] 'High temp (F)' 'Low temp (F)'  Precip (in)'
    [      NaN] 'Jan'   [  46] [  28] [ 3.98]
    [      NaN] 'Feb'   [  51] [  31] [  3.7]
    [      NaN] 'March' [  61] [  39] [ 4.88]
    [      NaN] 'April' [  70] [  47] [ 3.94]
    [      NaN] 'May'   [  78] [  57] [ 5.08]
    [      NaN] 'June'  [  85] [  65] [ 4.09]
    [      NaN] [  NaN] [ NaN] [ NaN] [  NaN]
    [      NaN] [  NaN] 'High temp (F)' 'Low temp (F)' 'Precip (in)'
    [      NaN] 'July'  [  89] [  69] [ 3.78]
    [      NaN] 'Aug'   [  88] [  68] [ 3.27]
    [      NaN] 'Sep'   [  82] [  61] [ 3.58]
    [      NaN] 'Oct'   [  71] [  49] [ 2.87]
    [      NaN] 'Nov'   [  59] [  40] [ 4.45]
    [      NaN] 'Dec'   [  49] [  31] [ 4.53]
```

This variable, which contains all the data in the spreadsheet, is an 18-by-5 array of cells. (Its dimensions happen to be the same as those of **txt**, but that is not always the case.) MATLAB determines these dimensions by finding the smallest rectangle that contains all the data in the spreadsheet (the rectangle does not necessarily start with cell A1. In fact, it will include that cell if and only if there is data in the first column and/or the first row). In this case the rectangle is the one whose upper left cell is A1 and whose lower right cell is F19. After forming **raw**, MATLAB copies the contents of every cell in the spreadsheet into the corresponding cell in **raw**, placing **NaN** where ever the corresponding spreadsheet cell is empty.

If only one of the second or third outputs is desired, one of the following formats can be used,

```
>> num = xlsread(filename);
>> [~,txt] = xlsread(filename);
>> [~,~,raw] = xlsread(filename);
```

Often, we want to read from a specified area within an Excel workbook. That can be done by including an optional second argument, or optional second and third input arguments, to specify the spreadsheet within the workbook and the place within the spreadsheet that the data are to be written. Here is an example:

```
>> num = xlsread('Nashville_climate_data',1,'D15')
num =
     61
```

The second argument instructs **xlsread** to read sheet 1, which is always the first sheet in a workbook. The third argument instructs **xlsread** to read only the contents of cell D15 in that sheet. We note that Excel specifies the column of a cell first (via one or more letters), and the row second. For example, D15 means column four and row fifteen, whereas MATLAB uses the opposite order: (15,4) means row fifteen and column four.

The third argument can also specify a range of cells, using the same notation that is used in Excel. For example,

```
>> num = xlsread('Nashville_climate_data',1,'D15:E17')
num =
          61         3.58
          49         2.87
          40         4.45
```

Note that the colon in the third argument is not MATLAB's colon operator, but merely a character in the string. It indicates that **'D15:E17'** specifies a rectangular array whose upper left corner is the cell D15 and whose lower right corner is E17.

A modification of the second argument is allowed. Instead of a number (positive integer) it may be a string. In that case it is treated as the name of a worksheet. Since the user has not renamed the first sheet in **Nashville_climate_data.xlsx**, it has the default name, Sheet1. Here is an example, in which the sheet name is used:

```
>> num =
xlsread('Nashville_climate_data','Sheet1','D15:E17')
num =
          61         3.58
          49         2.87
          40         4.45
```

If a non-existent sheet is specified, MATLAB returns the empty matrix (**[]**). There is no error message, because specifying a non-existent sheet is not an error. There is nothing there, so nothing is returned.

Writing Excel files

If Excel is installed on a Windows computer in which MATLAB is being run, then MATLAB can generate an Excel spreadsheet with the function **xlswrite**. (For Macbook users, MATLAB provides a somewhat limited alternative: **xlswrite** writes text files instead of Excel files. Only numeric arrays can be written. Each row is written on a separate line and the numbers on a row are separated by commas. This format, which is called the "comma-separated value" format, or CSV format, is written and read by many applications including Excel. More general writing of text files is taken up in the next section.)

```
>> xlswrite('foo.xlsx',raw)
```

As for **xlsread**, the first input argument is a file name. It is the name of the Excel file into which we wish to write data. The second input argument is a cell array. In this case we are using **raw**, which is the same cell array that we just generated above with our call to the function **xlsread**. If the file, **foo.xlsx**, does not exist in the current folder-directory, it will be created there, and the data in **raw** will be written into it. If the file **foo.xlsx** does exist in the current folder-directory, then the data in **raw** will be written into it with no need to create a new file. Only those elements in **raw** that do not contain **NaN** will be written, and each such element in **raw** will be written into the cell located at the same position relative to the upper left corner of the spreadsheet (cell A1) as the corresponding element in **raw** relative to **raw(1,1)**. The rest of the cells in the spreadsheet will remain as they were before this call of **xlswrite**.

For example, if **raw** is the same cell array that we saw in the previous section, then the cell A1 (column 1, row 1) in **foo.xlsx** will receive the value stored in **raw(1,1)** (row 1, column 1), which happens to be a 1x30 string. If there is data there already, it will be replaced. The cell B2 (column 2, row 1) will remain unchanged, because the element of **raw** at that cell's corresponding position, **raw(1,2)** (row 1, column 2) contains **NaN**. Thus, if there is data there already, it will remain there, and if it is empty, it will remain empty.

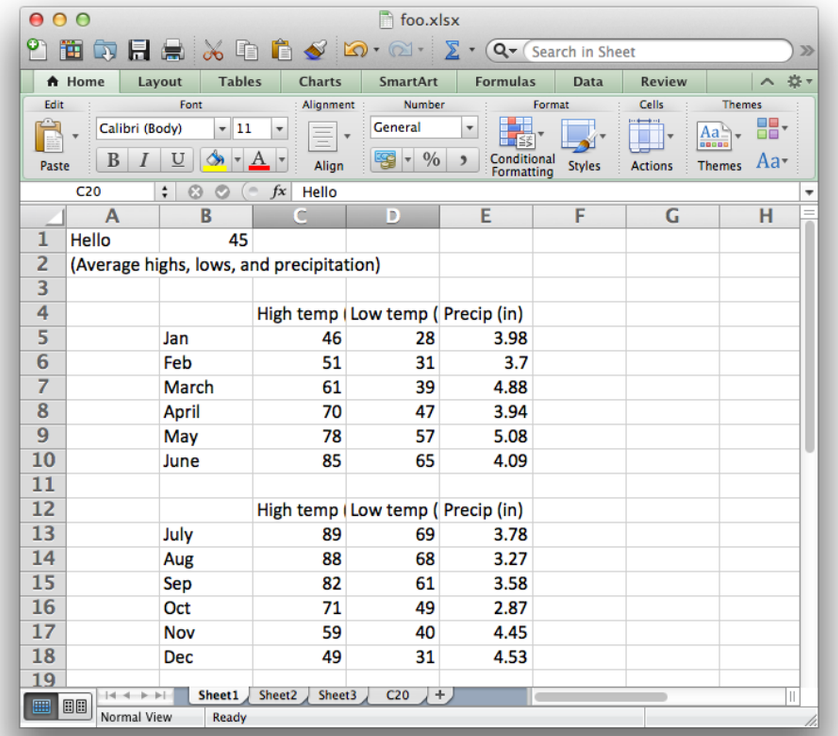
If the spreadsheet being written into is empty, as it would be if the file **foo.xlsx** did not exist before it was created by **xlswrite**, then the contents of **foo.xlsx** will be identical to that of the spreadsheet **Nashville_climate_data.xlsx**, from which **raw** was obtained using **xlsread**.

To understand better what happens when we overwrite data, let's assume that the command above has been carried out. The resulting spreadsheet will look like that of Figure 2.51, except that there will be no formatting of the cells, such as the grid of lines drawn around some of the cells in that original spreadsheet. Suppose we now give this command:

```
>>xlswrite('foo.xlsx', {'Hello',45});
```

The resulting spreadsheet is shown in Figure 2.52

Figure 2.52 Generated Excel spreadsheet



It can be seen that the string that originally appeared in cell A1 has been replaced by the string, "Hello" and that the formerly empty cell, B1 now contains the number 45. The rest of the cells remain as they were written by the previous command.

When we write a file with **`xlswrite`**, we may want to write only into a specified area. That can be done with optional input arguments three and four, which specify the spreadsheet within the workbook and the upper left corner of the rectangular area to be written. These arguments have the same form and meaning as the optional second and third arguments of **`xlsread`**: argument three is either a positive integer that gives the sheet number or a string that gives the sheet name, and argument four is a string that specifies the range of cells to be written in the spreadsheet. There is an important difference, however. If there is no colon (:) in the fourth argument, then it is interpreted to mean the upper left corner of the range that will be written. In other words, when there is no colon, the third argument specifies the starting point of the range to be written instead of the range itself. For example, if we wished to write Hello, 45 into cells C20 and C21 of the first sheet, we could give this command:

```
xlswrite('foo.xlsx',{'Hello',45},1,'C20');
```

If a colon is included and the range is smaller than required to accommodate the array being written, then only that part of the array that fits into the specified range will be written. Thus, the following function call has the same effect as the one above.

```
xlswrite('foo.xlsx',{'Hello',45},1,'C20:D20');
```

The following results in only Hello being written into cell C20:

```
xlswrite('foo.xlsx',{'Hello',45},1,'C20:C20');
```

The third argument specifies that we wish to write into the first sheet in the workbook. As with **`xlsread`**, this argument can also be a string containing a sheet name. If a sheet with that name exists, the data will be written into it. If not, a new sheet will be created in the workbook **`foo.xlsx`** with the specified name, and the data will be written into that new sheet.

There are other variations on the use of **`xlsread`** and **`xlswrite`**, which can be found by consulting MATLAB's Help system, but the approaches that we have shown here will handle most tasks quite well. After trying an example or two, you should find not only that Excel spreadsheets provide an easy way to communicate with other applications but also that with the help of Excel itself, you can do additional analysis of the data between MATLAB sessions.

Text Files

A **text file** is a file that contains characters (i.e., letters, numbers, and punctuation marks) encoded in a standard format, such as ASCII, Unicode, or UTF-8. We discuss the ASCII encoding scheme in detail in the subsection named The ASCII encoding scheme within the section Data Types. Fortunately, however, it is not necessary to know the encoding scheme in order to read and write text files with MATLAB because the encoding is handled for you behind the scenes. ASCII is the oldest of the schemes for encoding characters and is a subset of all of the other commonly used schemes. Because it is always part of any encoding scheme for text files, it is common to refer to a text file as "an ASCII file", even if the particular encoding scheme is unknown or is known to be different from ASCII. Newer encoding schemes provide thousands of codes, whereas ASCII is limited to only 128 codes. Each of these codes can be stored in a single byte (with the first bit always equal to zero). A mere 128 codes was enough for those who developed ASCII originally because they used only the English alphabet, but it is inadequate for handling other languages, and, as a result, new schemes have been developed over the intervening years, involving two, three, four, or more bytes. MATLAB uses the default scheme provided by the operating system on which it is running. (You can determine the name of that scheme with this command, `feature DefaultCharacterSet`.)

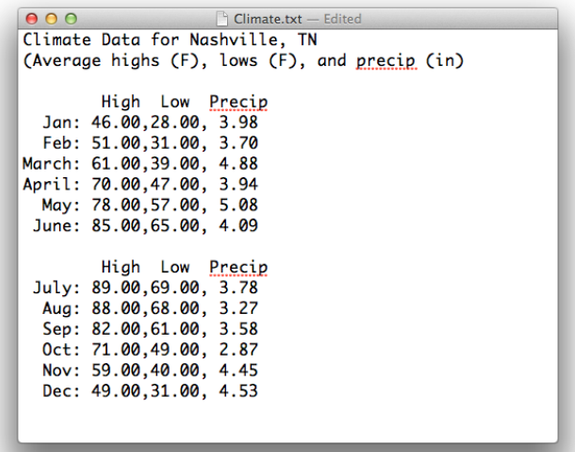
Opening a text file

A key difference between text files and both MAT-files and Excel files is that, before you read or write into a text file, you must "open" the file, and when you are through reading and writing to it, you must "close" it. As usual MATLAB provides a function for everything you need to do, including opening and closing files. These functions are easy to use. However, they mean little in isolation from the functions for writing and reading text files, so we will explain the functionality of opening, writing, reading, and closing text files via examples that show how to use all of these functions together. Our first example is a function named **`write_temp_precip_txt`**, which opens a text file whose name is passed to the function as an argument, and which writes climate data into that text file. Here is an example of the function in action:

```
>> write_temp_precip_txt('Climate.txt')
>>
```

We show the Command prompt after the command to make it clear that it writes nothing to the Command Window. Instead, it has written data into a file named **`'Climate.txt'`**. We chose the file extension **`txt`**, not because the function requires it, or because it means anything special to MATLAB, but because it is universally recognized as an extension that means that the file is a text file. So, how can we look at the contents? Well, we will see below how to do that with MATLAB, but we can also do it with any other application designed to read text files. On Mac OS, TextEdit is the default program for looking at **`.txt`** files. When we open the file, we see the window shown in Figure 2.53.

Figure 2.53 *Text file open in TextEdit*



If you look carefully at the data being written, you might recognize it as the climate data used in the Excel examples above. Now lets look at the code that generated this file:

```
function write_temp_precip_txt(filename)

Title_1 = 'Climate Data for Nashville, TN';
Title_2 = '(Average highs (F), lows (F), and precip (in)';
Label_1 = ' High ';
Label_2 = ' Low  ';
Label_3 = 'Precip';
Mo_1 = {'Jan','Feb','March','April','May','June'};
Mo_2 = {'July','Aug','Sep','Oct','Nov','Dec'};

Data_1 = [
   46 28 3.98
   51 31 3.7
   61 39 4.88
   70 47 3.94
   78 57 5.08
   85 65 4.09
   ];
Data_2 = [
   89 69 3.78
   88 68 3.27
   82 61 3.58
   71 49 2.87
   59 40 4.45
   49 31 4.53
   ];
fid = fopen(filename,'w+t');
if fid < 0
   fprintf('error opening file\n');
   return;
end
fprintf(fid,'%s\n',Title_1);
fprintf(fid,'%s\n',Title_2);
fprintf(fid,'\n');
fprintf(fid,'       %s%s%s\n', ...
                    Label_1,Label_2,Label_3);
for ii = 1:size(Data_1,1)
   fprintf(fid,'%5s: ',Mo_1{ii});
   fprintf(fid,'%5.2f,%5.2f,%5.2f\n',Data_1(ii,:));
end
fprintf(fid,'\n');
fprintf(fid,'       %s%s%s\n', ...
                    Label_1,Label_2,Label_3);
for ii = 1:size(Data_2,1)
   fprintf(fid,'%5s: ',Mo_2{ii});
   fprintf(fid,'%5.2f,%5.2f,%5.2f\n',Data_2(ii,:));
end
fclose(fid);
```

The first new feature in this example is the statement, **`fid = fopen(filename,'w+t')`**. The function **`fopen`** requires at least one input argument, and it must be a string. That string contains the name of the file to be opened. The returned value is an integer. We have captured this integer in the variable **`fid`**, which is mnemonic for "**f**ile **id**entifier". This number is needed when we call functions to write to this file or read from it.

The second input argument tells **`fopen`** whether the file is going to be written into or not. If there is no second argument, then the file will be opened as a "read-only" file. In that case, MATLAB will not allow any writing to the file. This is a good rule, because there is often precious data in a file to be read, and it might be disastrous to change its contents by accidentally calling a function that will write over the current contents. In this case, however, we want to write into the file, so we give as a second argument **`'w+t'`**. The second argument to **`fopen`** is called the file "permission", and it tells us whether the file will be read or written, whether its previous contents will be erased, whether a non-existent file should be created, and whether the file should be treated as a text file or a binary file. In this case, the '**`w`**' means that we want to be permitted to write to the file, discarding any previous contents, the '**`+`**' means that if the file does not exist we want it to be created, and the '**`t`**' means that we want the file to be treated as a text file.

Table 2.21 Permissions for opening text files indicated by second argument to **`fopen`**

2ND ARGUMENT	PERMISSION
`'rt'`	open text file for reading
`'wt'`	open text file for writing; discard existing contents
`'at'`	open or create text file for writing; append data to end of file
`'r+t'`	open (do not create) text file for reading and writing
`'w+t'`	open or create text file for reading and writing; discard existing contents
`'a+t'`	open or create text file for reading and writing; append data to end of file

Table 2.21 shows six important file permissions that can be requested for text files with **fopen**. They all end with '**t**', which is mnemonic for "**t**extfile."

The permission argument actually represents a request—which may not be granted! When the MATLAB interpreter executes **fopen**, it passes the requested permission over to the operating system. If the operating system can, it will grant the request, if not, it will not. Success or failure can be determined by examining the value of the returned file identifier (**fid** in this case). If the file exists in the current directory, or in any directory on the MATLAB search path, it will be found and opened for writing. If the file is found or if the file is created, and if the operating system allows the file to be opened with the requested permission, then the open is successful and a positive integer will be returned as the file identifier. If not, then the file name may be illegal (contains a character that the operating system does not allow in file names), the operating system may have locked the directory against file writing or creation, or some other operating-system-level problem may be keeping **fopen** from procuring the right to write to the file.

If the file is not opened successfully, then **fopen** will return the number −1, which is not a file identifier, but instead is a signal denoting that no file was opened. It is always a good idea to check for this event, as we have done above with the if-statement: `if fid < 0`. Otherwise, the behavior of the program will often be quite mysterious, and a long debugging sessions may be required before it is realized that the problem is that no file was ever opened, so nothing was written or read.

If the file is opened successfully, then an integer greater than 2 will be returned: 3 for the first file that is opened, 4 for the second one, etc. The value 1 means Command Window. The values 0, and 2 are reserved.

Writing to a text file

Now it is time to learn how to write data into a text file. Good news! You already know how to do it! Well, almost. Writing to a text file is done with **fprintf**, and all you have to do in order to tell it to write to a text file instead of the Command Window is to give it one extra argument. The extra argument is given first. It is the file identifier, which in our example is contained in the variable **fid**. It can be seen that in the example above every **fprintf** statement includes **fid** as the first argument, and therefore, everything will be written into the file that was opened with this file identifier, which in our case is **Climate.txt**. Everything else is the same. In fact, while you are developing an application that writes to a text file, it is possible to see how the text is formatted without having to look in a file by temporarily setting `fid = 1`, which means that the "file" being written is not a file at all, but is the Command Window.

Closing a text file

Once we are through writing or reading a text file, we must "close" it. Closing a file is done with the function **fclose**. It requires one argument. In the example above, it can be seen that the argument is **fid**, which means that we are closing the file opened with the file identifier stored in **fid**. Any file that has been opened must be closed, because, if we leave it open, then some other applications may not be able to access it. For example, if we want to look at our text file with TextEdit, and we have omitted the call to **fclose**, we will see an error saying that it is locked by another user or is open in another application. In the event that this happens, you can determine which files are open in MATLAB by calling **fopen** with the argument '**all**'. It will return a list of the open files. You can then close them one at a time with **fclose**. Alternatively, `fclose('all')` will close all open files.

Displaying a text file with MATLAB

We saw above how to display the contents of a text file using a program provided by the operating system (TextEdit). It is also easy to display a text file in MATLAB. That can be done with a simple command named **type**, as follows:

```
>> type Climate.txt

Climate Data for Nashville, TN
(Average highs (F), lows (F), and precip (in)

        High  Low  Precip
  Jan: 46.00,28.00, 3.98
  Feb: 51.00,31.00, 3.70
March: 61.00,39.00, 4.88
April: 70.00,47.00, 3.94
  May: 78.00,57.00, 5.08
 June: 85.00,65.00, 4.09

        High  Low  Precip
 July: 89.00,69.00, 3.78
  Aug: 88.00,68.00, 3.27
  Sep: 82.00,61.00, 3.58
  Oct: 71.00,49.00, 2.87
  Nov: 59.00,40.00, 4.45
  Dec: 49.00,31.00, 4.53
```

The function **type** prints one blank line followed by the line-by-line contents of the file. To get an idea of how this process of reading from a text file to the Command Window works, we provide a simple function that we have adapted from an example provided by the MATLAB Help system, which produces exactly the same output to the Command Window as the command **type**:

```
function view_text_file(filename)
fid = fopen(filename,'rt');
if fid < 0
   fprintf('error opening file\n');
   return;
end
fprintf('\n');

% Read file as a set of strings, one string per line:
oneline = fgets(fid);
while ischar(oneline)
    fprintf('%s',oneline) % display one line
    oneline = fgets(fid);
end
fclose(fid);
```

As can be seen from the second line of the function, we are using the permission string '**rt**' in **fopen**, meaning that we are requesting that the file be opened for reading (**r**) as a text file (**t**). Before the contents of the file are read, the command prints a blank line, just as the function **type** does. The actual reading is then done by the function **fgets**. This function gets one line from the file and returns it as a string, which we have captured in the variable **oneline**. The first line is read just before the while-statement begins, then **fgets** is called repeatedly in the body of the loop. Without our telling it to do so, **fgets** skips to the next line of the file each time we call it, moving line-by-line through the file putting the most recent line read into **oneline**, until it gets to the end of the file. When it finds that it is at the end of file, it returns the number -1, encoded as a double precision number instead of a string, so the type of **oneline** is **double** instead of **char**. This change of type is used by the control statement of the while-loop—**while ischar(oneline)**— to determine whether the end of file has been reached.

The only task remaining after a line is read is to display it in the Command Window. That is done with **fprintf**. In the body of the loop, **oneline** is printed to the Command Window using the format string **'%s'**.

Reading data from a text file into variables

As we have seen, it is simple to display the contents of a text file. However, if it is desired to store data from the file into MATLAB variables for processing, things are more complicated. Many people are surprised to discover that reading the contents of a text file into variables (other than copying lines as strings directly into **oneline** as in the previous example) can be much more difficult than going the other way, i.e., writing the contents of variables into a text file! The reason for the difference is that, when we are writing the contents of variables into a text file, we have much more information readily available. The information we have comprises the type, size, and shape of the variables that we are writing to the file, but when we wish to the read the contents of a text file into variables, all we have is a stream of characters from the

file. The text file, unlike, say, an Excel file, contains no other information. We must have additional information to know how to parse that stream of characters into meaningful parts: strings and numbers. It is impossible to know where the boundaries are between one datum and the next in the file or what types of variable they should be stored in. Reading the text file into variables requires detailed knowledge of the order of types of data in the file, and that knowledge must be incorporated into the code that reads the file. We will show an example that exploits such knowledge to read parts of a text file into strings and parts of it into variables of a numeric type.

Our example is a function designed specifically to read files in the format of **`Climate.txt`**. Here is what we mean by "format":

- Lines 1-4: each contains a string, the third line being blank.
- Lines 5-10: each contains a string followed by three numbers.
- Lines 11 and 12: each contains a string, the first being blank.
- Lines 13-18: each contains a string followed by three numbers.

Our function for reading a file written in this format is shown below. The lines are numbered for easier reference:

```
1.  function contents = read_temp_precip_txt(filename)
2.  fid = fopen(filename,'rt');
3.  if fid < 0
4.    fprintf('error opening file\n');
5.    return;
6.  end
7.
8.  % Read file as a set of strings, one per line:
9.  line_number = 1;
10. oneline{line_number} = fgetl(fid);
11. while ischar(oneline{line_number})
12.   line_number = line_number + 1;
13.   oneline{line_number} = fgetl(fid);
14. end
15. fclose(fid);
16.
17. % Parse the lines:
18. Title_1 = oneline{1};
19. Title_2 = oneline{2};
20. Labels  = oneline{4};
21. for ii = 1:6
22.   [Mo_1{ii},~,~,n] = sscanf(oneline{ii+4},'%s:');
23.   Data_1(ii,1:3)   = sscanf(oneline{ii+4}(n:end),'%f,');
24. end
25. for ii = 1:6
26.   [Mo_2{ii},~,~,n] = sscanf(oneline{ii+12},'%s:');
27.   Data_2(ii,1:3)   = sscanf(oneline{ii+12}(n:end),'%f,');
28. end
29.
30. % Put the parsed data into one output argument:
31. contents{1} = Title_1;
32. contents{2} = Title_2;
33. contents{3} = Labels;
34. contents{4} = Mo_1;
35. contents{5} = Data_1;
36. contents{6} = Mo_2;
37. contents{7} = Data_2;
```

We run it like this:

```
>> contents = read_temp_precip_txt('Climate.txt');
```

If we check the contents of the individual cells in **`contents`**, we find the following:

```
contents{1} =
'Climate Data for Nashville, TN'
contents{2} =
' (Average highs (F), lows (F), and precip (in) '
contents{3} =
'        High  Low  Precip'
contents{4} =
'Jan:'    'Feb:'    'March:'    'April:'    'May:'    'June:'
contents{5} =
[
           46           28         3.98
           51           31          3.7
           61           39         4.88
           70           47         3.94
           78           57         5.08
           85           65         4.09
]
contents{6} =
'July:'    'Aug:'    'Sep:'    'Oct:'    'Nov:'    'Dec:'
contents{7} =
[
           89           69         3.78
           88           68         3.27
           82           61         3.58
           71           49         2.87
           59           40         4.45
           49           31         4.53
]
```

The code in **read_temp_precip_txt** starts out like that of **view_text_file**. In fact, **lines 2-6** are identical to those of **view_text_file**. **Lines 9-15** are similar, but there are three differences. First, there is no printing (no **fprintf**). Second, **oneline** is now a cell vector, and each line is read into a separate element of it. After the entire file has been read, it is closed on **line 15**. After it is closed parsing of information begins. Third, **fgets** has been replaced by **fgetl**, which discards new-line characters.

This function separates the reading of the information in the file from the parsing of the information in the file. While these operations can be combined, it is easier to keep them separate, so we first capture every line of the file in **oneline** and then close the file and begin parsing the contents of **oneline**. At this point each element **n** of **oneline** contains the corresponding line **n** of the file, except that the last element **oneline(end)** contains the number -1, which, is returned by **fgetl(fid)** when it has reached the end of the file whose file identifier is stored in **fid**, just as **fgets(fid)** does.

Parsing begins on **line 18** of the function. **Lines 18-20** of the function copy elements 1, 2, and 4 of **oneline**, which contain lines 1, 2, an 4 of the file, as strings in the variables, **Title_1**, **Title_2**, and **Labels**. **Line 3** is blank according to our format, so we ignore it.

Lines 21 through **24** of the function parse the contents of elements 5 through 10 of **oneline**, with **lines 22** and **23** parsing each line into two parts. We look at these two lines separately.

Line 22: `[Mo_1{ii},~,~,n] = sscanf(oneline{ii+4},'%s:');`

This line extracts the first part of **oneline{ii+4}**. That first part is the month abbreviation followed by a colon (':') and it is assigned to **Mo_1{ii}**. This parsing is accomplished with the help of a MATLAB function named **sscanf**, which means "**s**tring **scan** (i.e., read) using a **f**ormat". This function reads a string, which is its first input argument, and compares it with a format string, which is its second input argument. The format string tells **sscanf** how to interpret the characters in its first argument to produce values to place into its first output argument. A similar function named **fscanf** does the same thing, but takes as its first input argument a file identifier. It interprets the file as a text file and interprets the characters in the file in the same manner that **sscanf** interprets the characters in its first input argument. In their format strings, both **sscanf** and **fscanf** use the same escape characters and the same format specifiers with roughly the same meanings that are used by **fprintf**, whose functionality is described in Programmer's Toolbox, when it prints the values of variables to the screen. The meanings of the most useful format specifiers for **sscanf** and **fscanf** are given in Table 2.22.

Table 2.22 Format specifiers for **sscanf** and **fscanf**

SPECIFIER	DESCRIPTION OF THE VALUE PRODUCED
c	single character
d	decimal notation drops fractional part
e, E, f, g, G	exponential -or- fixed-point notation: for upper or lower case e
o	unsigned octal notation
s	string: reads characters up to (not including) first non-white-space
x, X	unsigned hexadecimal notation

For example, the percent sign in **'%s:'** means that the next character—in this case the **s**—signifies the format of the data. In **fprintf**, the format specifies the manner in which the data is printed, while in **sscanf**, the format specifies the manner in which the data is read (scanned). Here, **'%s:'** means that **sscanf** is looking for a string followed by white space. This pattern matches the input string up to and including its colon, so we used a colon here, but we could have used any character. For example, **'%sg'** would have the same effect! The matched pattern excluding the white space but including the colon is assigned to **Mo_1{ii}**. The white space is very important here. It serves as a delimiter that separates the string we want (the month abbreviation plus colon) from the first number. A **delimiter** is a character or string that allows a program to know where one part of a sequence ends and the next part begins without knowing the expected lengths of any of the parts. There are additional options available for the format string, including field-width specifiers, number digits to the right of the decimal, and an asterisk to indicate that reading is to be done but no value is to be output—all similar to the options available with **fprintf**.

If there were no character after the **s** in the format string and no white space in the input string, then **sscanf** would use **%s** to devour the digits, decimals, and everything else in its path until it reached the end of **oneline{ii+4}**. However, when it is followed by an ordinary, non-white-space character, which excludes spaces, tabs, and newlines, it does a look-ahead on each input character, and when it sees that the next character is white, it stops shoving characters into **Mo_1{ii}**. Let's look at what happens when there is no character after the **%s** in the format string. Here is an example, showing what happens:

```
>> str = sscanf('  April:  12  ', '%s')
str =
April:12
```

The leading white space before **April** , the white space between **April:** and **12**, and the trailing white space after **12** in the string read by **sscanf** have all been discarded. The resulting output string is, therefore, **April:12**. The behavior is different when **%s** is followed by a (non-white) character and the input contains white space. Here are three examples, using the letter **m** as a delimiter:

```
>> str = sscanf('  April:  12  ', '%sm')
str =
April:
>> str = sscanf('  Aprilm  12  ', '%sm')
str =
Aprilm
>> str = sscanf('  Aprilm12  ', '%sm')
str =
Aprilm12
>> length(str)
ans =
     8
```

In each case the leading space is skipped, but after any non-white space has been read, the first subsequent white space causes reading to be suspended, and whether or not there is a character in the input that matches the character that follows **%s** (**m** in these examples) has no effect on the result. The length of **str**, which is 8, shows that the trailing white space is excluded.

The function **sscanf** actually returns four output arguments (they are explained in the help system), but here we are using the tilde (**~**) to skip the sec-

ond and third output arguments because we are interested only in the first, which is the argument that receives the value that matches the format string, and the fourth, which is the argument that receives a number that equals the index of the next character in the string after the part that matched the format string. In this case the next character is the first character after the colon. We use the variable **n** to receive this index.

Line 23: `Data_1(ii,1:3) = sscanf(oneline{ii+4}(n:end),'%f,');`

In this line, we make use of **n**, whose value was set in **line 22**, in order to give **sscanf** only the characters in **oneline{ii+4}** from its **n**th character to the end. This is the part of the line that contains the three numbers separated by commas. To capture these numbers, we use as the format string `'%f,'`. Let's analyze what happens when **ii** = 1, for which,

- **oneline{ii+4}** contains the string, **Jan: 46.00, 28.00, 3.98**
- **n** equals **7** (as it does for every value of **ii**), and
- **oneline{ii+4}(n:end)** contains the string, **46.00, 28.00, 3.98**

When **sscanf** applies `'%f,'` to the string being parsed, namely **46.00, 28.00, 3.98**, it finds that the **%f** in the format string matches the **46.00** in the parsed string and the comma in the format string matches the comma after the **46.00** in the parsed string. It converts **46.00** into a double-precision number and copies it into **Data_1(ii,1)**. The next thing that **sscanf** does is very interesting (well, at least to people interested in this sort of thing). It has not reached the end of the string, and yet it *has* reached the end of its format string. Instead of issuing an error, it "recycles" its format string. It continues from where it left off in the string being parsed, which means that it is now parsing the remainder of the string, which is **' 28.00, 3.98'**, and it applies its format string, **'%f,'** again! This second application of the format string results in 28 being assigned to the output argument's next element, **Data_1(ii,2)**. This recycling is continued until **sscanf** has reached the end of the string being parsed, which in our example is just one more time. The result is that **Data_1(ii,1:3)** now equals **[46.00, 28.00, 3.98]**. Note that if we had used **Data_1(ii,1:2)** or **Data_1(ii,1:4)** as the output argument, MATLAB would have declared an error and written a note like this:

```
In an assignment  A(I) = B, the number of elements in B
and I must be the same.
```

It does this because **sscanf** has produced a three-element vector as its output, and we have specified a different number of elements for the output argument. This is not a new rule. It is MATLAB's standard response for any assignment statement in which the number of elements on the two sides of the equal sign do not agree.

Recycling of the format string allows us to force **sscanf** to place all the characters in a string into its output argument by means of the format specifier **c**, which has the meaning "read one character", as shown in this example:

```
>> str = sscanf('  April  12  ','%c')
str =
  April  12
>> length(str)
ans =
    13
```

Because the reading of one character does not bring **sscanf** to the end of the string **' April 12 '**, it recycles **'c'**, again and again, putting the successive single characters into successive elements of **str**. Comparing this example with the earlier example in which **%s** was used to read a string with space reveals the important difference that all the spaces are included in the output with **%c**.

There is more to the recycling feature. If **sscanf** reaches the end of the format string AND the end of of the string being parsed, while there are elements of the output argument that have not been given values, **sccanf** recy-

cles both strings until it has reached the end of the argument. Here is an example of this behavior:

```
>> x(1:2) = sscanf('12.1','%f')
x =
        12.1          12.1
```

Thus, **sscanf**, not wishing to waste anything, performs a dual recycling: both strings are used repeatedly.

This recycling of the format string, both single and dual, is a special feature of MATLAB's version of **sscanf**. The name "sscanf" and most of the functionality of **sscanf** was borrowed by MATLAB from the language C, but that profligate C does not recycle.

Before we leave **sscanf,** let's examine one more feature, one that can be very confusing if it is encountered before it is understood. Let's suppose that we had omitted the comma from the format string. Thus, we would have this command:

```
Data_1(ii,1:3) = sscanf(oneline{ii+4}(n:end),'%f');
```

This small change would cause the second and third elements to be read incorrectly with the result that **Data_1(ii,1:3)** would be assigned **[46.00, 46.00, 46.00]**. Why is 46 assigned repeatedly despite the fact that there are two additional numbers in **oneline{ii+4}(n:end)**? The answer reveals the new feature. Since commas are not accepted by the **'%f'** format, when **sscanf** encounters a comma in **46.00, 28.00, 3.98**, it stops converting characters and assigns 46 to **Data_1(ii,1)**. Since there remain elements in the output argument that have not yet been assigned values, **sscanf** then recycles its format string as before, so that it can produce a value for the next element, which is **Data_1(ii,2)**. So far the behavior is the same as when there was comma in the format string. However, this time **sscanf** is faced with parsing the rest of the string, which is **',28.00, 3.98'**. Because the first character in this string is a comma and since the format string **'%f'** does not support commas, it is impossible to process the rest of the string. What **sscanf** does in this situation is even more interesting. In addition to recycling the format string, it recycles *just the initial part* of the string being parsed! It returns to the beginning of **46.00, 28.00, 3.98** and re-applies its format string, which causes 46 to be assigned to **Data_1(ii,2)**. It continues this dual, partial recycling until all elements have been assigned values, which in our example is just one more time for **Data_1(ii,3)**.

It might have been expected (and some might prefer) that instead of risking the assignment of incorrect values to some of its output arguments, **sscanf** would issue an error message and halt execution in this last situation. The choices instead to resort to dual recycling and to the recycling of part of a string, are examples of MATLAB's tendency to continue processing whenever possible so that at least some usable results might be produced. This approach makes sense when a very long process might otherwise be halted just before it does its most important work, and it also makes sense during debugging, when it might be helpful to see the results of operations that take place after an operation that would in other systems be halted. However, this behavior sometimes makes it possible for very wrong results to be produced. Thus, it makes even more sense that the programmer should understand thoroughly the semantics of **sscanf**, or of any function, before relying on its results.

Binary Files

The phrase, **binary file** originally meant a file encoded using binary notation, but since nowadays all files are encoded in binary notation, the name now simply means "not a text file". A binary file contains a stream of bits that can be decoded directly into numbers the way the computer reads and writes them, as opposed to being decoded into characters that represent digits in the way that humans read and write them. Thus, for example, the string '37' consists of two characters, a '3' and a '7', that represent digits which together mean thirty plus seven, or thirty seven to us humans. In a text file, each character is encoded as eight bits. The character '3' is encoded as 00110011 and the character '7' is encoded as 00110111. Thus the text-file encoding of the string '37' would occupy two bytes, or sixteen bits, and looks like this: 0011001100110111. This is not, however, the way the number 37 is stored in computer memory when it is assigned to a variable. For example, **`x = 37`** would not put this pattern of zeros and ones into the memory location of **`x`**, instead it would use an encoding scheme called "double-precision", which consists of 64 bits and looks like this:

0100000001000010100000000000000000000000000000000000000000000000

We have never worried about either of these encodings—character or double precision—or the encoding of any of the other types, and fortunately we won't have to because MATLAB takes care of those details for us, as do other programming languages. The important thing to know here is that with binary files it is possible to copy the actual memory bit patterns directly into a file, and it is possible to read those bit patterns from the file into variables, which is a much more efficient way to store and retrieve numbers than using text files. Fortunately, MATLAB uses standard encoding for all its types, double precision being one example, which means that many other programs can read binary files that MATLAB writes and that MATLAB can likewise read binary files that the other programs write.

As with text files, binary files are best understood via examples. Our first example is a function that writes the values from an array into a file, using double-precision encoding:

```
function write_array_bin(A,filename)
fid = fopen(filename,'w+');
if fid < 0
   fprintf('error opening file\n');
   return;
end
fwrite(fid,A,'double');
fclose(fid);
```

Opening a binary file

A binary file must be opened before it can be written or read. The opening process is almost identical to that of the text file. As can be seen from our example, the same function is used to open a binary file as that used to open a text file—**`fopen`**. The only difference is that for a binary file the second argument **`'w+'`** does not have a **`'t'`** at the end, which, as will be recalled means text file. Other than the fact that the file is to be treated as a binary file instead of a text file, the meaning is the same: write to the file (**`w`**), and if the file does not exist, create it (**`+`**). For binary files, as for text files, **`fopen`** returns a file identifier, and once again it has a negative value if the file cannot be opened with the requested permission. Table 2.23 shows six important file permissions that can be requested for binary files with **`fopen`**.

Writing to a binary file

As for text files, the file identifier is used as an argument in any function that accesses the data in the file. This time the accessing function is **`fwrite`**, which is designed to write to a binary file. The file identifier is always its first argument, and its second argument must be the array containing the data to be written. Only one type of data can be written in a single call to **`fwrite`**, and that data type is given by the third argument. We have chosen to write the data in double precision, which we indicate by using the MATLAB string

Table 2.23 Permissions for opening binary files indicated by the second argument to **`fopen`**

2ND ARGUMENT	PERMISSION
`'r'`	open binary file for reading
`'w'`	open binary file for writing; discard existing contents
`'a'`	open or create binary file for writing; append data to end of file
`'r+'`	open (do not create) binary file for reading and writing
`'w+'`	open or create binary file for reading and writing; discard existing contents
`'a+'`	open or create binary file for reading and writing; append data to end of file

`'double'`. While the type we have specified is the same as the type of the array whose values we are writing into the file, it is not necessary for them to be the same. We could, for example, have written the data into the file using **`'single'`** or **`'int16'`** or any other of the many available types (see Date Types). MATLAB will convert the values in the array to the type we specify before writing to the file. If no third argument is given, the default type **`'uint8'`** is used. (See the help entry for **`fopen`** for other arguments and options.)

A question occurs when the array is not a vector: What is the order in which the values are written into the file? The answer to the question about the order of array elements is always the same in MATLAB: column-major order. We demonstrate below that this order is used when writing binary files.

Closing a binary file

Once we are through writing or reading a binary file, we must "close" it, just as with a text file. As before, it is closed with the function **`fclose`**. It requires one argument—the file identifier.

Reading data from a binary file into variables

When reading data from a binary file, we face the same problem that we faced when reading data from a text file: We need to know the format of the file. However, the problem is usually simpler with binary files, because usually the file consists of variables that are all of the same type, so all we need to know is that one type, with no concern about how values of varying types are organized on a line-by-line basis and where the text ends and the numerical values begin. If there is more than one type, then we need to know the list of types in the order they appear and the number of each. We will handle that situation in the next subsection, but first we will show to handle the simplest, and most common case. Here is a function that will read any binary file that consists of just one type:

```
function A = read_array_bin(filename,data_type)

fid = fopen(filename,'r');
if fid < 0
   fprintf('error opening file\n');
   return;
end
A = fread(fid,inf,data_type);
fclose(fid);
```

The hard work is done for us by the function **`fread`**, which is designed for binary files. As usual, its first argument is the file identifier. The second argument determines the maximum number of elements to be read from the file. We have used **`inf`**, which is the default (this argument can be omitted regardless of whether there are additional arguments) and means that the maximum is infinity. In that case, reading continues to the end of file. The third argument to **`fread`** specifies the data type in the file. As with **`fwrite`**, the default type is **`uint8`**, but for this function, rather than force the type to be **`'double'`**, as in **`write_array_bin`** , we have allowed the caller of our function to specify the type via the input argument **`data_type`**. Here is where the required knowledge of the format of the file being read is used. After the data is read into **`A`**, the file is closed via **`fclose`**.

Here is an example of the use of our two functions for accessing a binary file, one to write it and one to read it (we have set **`format short`**):

```
>> Data_1
Data_1 =
   46.0000   28.0000    3.9800
   51.0000   31.0000    3.7000
   61.0000   39.0000    4.8800
   70.0000   47.0000    3.9400
   78.0000   57.0000    5.0800
   85.0000   65.0000    4.0900
>> write_array_bin(Data_1,'data.bin')
>> B = read_array_bin('data.bin','double')
B =
   46.0000
   51.0000
   61.0000
   70.0000
   78.0000
   85.0000
   28.0000
   31.0000
   39.0000
   47.0000
   57.0000
   65.0000
    3.9800
    3.7000
    4.8800
    3.9400
    5.0800
    4.0900
```

It might be noticed that **`Data_1`** contains the same data that it did in our climate example above in the section on text files. The function call, **`write_array_bin(Data_1,'data.bin')`**, specifies **`Data_1`** as the array to be written to a file and **`'data.bin'`** as the file name. The function call, **`B = read_array_bin('data.bin','double')`**, specifies **`'data.bin'`** as the file to read and **`'double'`** as the format to use. The values returned by **`read_array_bin`** are put into **`B`**, and when we look at **`B`** we find that it is a column vector. An inspection of its values reveals that, as promised, MATLAB used column-major order to write the elements of **`Data_1`** into the file.

Additional options for binary files

In some cases as new data is generated, it might make more sense to add the data to an existing file than to write it to a new one. This might be the situation, for example, if we received the weather data in **`Data_2`** after we had written **`Data_1`** into **`'data.bin'`**. Nothing could be simpler! We simply open **`'data.bin'`** a second time and write the new data into it, BUT, when we open it, we use a different second argument in **`fopen`**. Instead of **`'w+'`** we use **`'a'`** (for **a**ppend). Here is a function that does just that:

```
function append_array_bin(A,filename,data_type)

fid = fopen(filename,'a');
if fid < 0
   fprintf('error opening file\n');
   return;
end
fwrite(fid,A,data_type);
fclose(fid);
```

and here is the function being called:

```
>> append_array_bin(Data_2,'data.bin','double')
```

Now, when we read the file, we see both the data that was already there, which we have already stored in **`B`** above plus the new data appended to it:

```
>> C = read_array_bin('data.bin','double')
C =
   46.0000
   51.0000
   61.0000
   70.0000
   78.0000
   85.0000
   28.0000
   31.0000
   39.0000
   47.0000
   57.0000
   65.0000
    3.9800
    3.7000
    4.8800
    3.9400
    5.0800
    4.0900
   89.0000
   88.0000
   82.0000
   71.0000
   59.0000
   49.0000
   69.0000
   68.0000
   61.0000
   49.0000
   40.0000
   31.0000
    3.7800
    3.2700
    3.5800
    2.8700
    4.4500
    4.5300
```

Sometimes a binary file contains more than one type of data. It might have **doubles** and **ints** or **chars** and **singles**, for example. To read or write such files, it is necessary to know how many of each type occur and their order. As an example, suppose we want to write the following types into a file:

- three **int16**s
- some **char**s
- some **single**s
- some **int32**s
- some **single**s

in that order.

Suppose further that

- the first **int16** tells us how many **char**s there are
- the second **int16** tells us how many **single**s there are
- the third **int16** tells us how many **int32**s there are.

We don't need to know how many **single**s there are because they come last, so we can simply read them until we get to the end of the file. Here is a custom-made function to write data in that format, followed by a custom-made function to read it in that same format:

```
function custom_write_bin(d1,d2,d3,d4,filename)
fid = fopen(filename,'w+');
if fid < 0
   fprintf('error opening file\n');
   return;
end
n1 = length(d1(:));
n2 = length(d2(:));
n3 = length(d3(:));
fwrite(fid,[n1,n2,n3],'int16');
fwrite(fid,d1,'char');
fwrite(fid,d2,'single');
fwrite(fid,d3,'int32');
fwrite(fid,d4,'single');
fclose(fid);
```

The first four arguments **d1**, **d2**, **d3**, **d4**, are vectors containing data to be written. Note that the types of these arguments are not specified. They can be any type, but when they are written to the file, they will be converted, respec-

tively, into **char**s, **single**s, **int32**s, and **single**s. The conversion process may cause some values to be changed if the new type does not include the values stored in the arguments. For example, if **d1** contains a negative number, which is outside the range of a **char**, it will be converted to 0; if **d1** contains a non-integer, which cannot be stored as a **char**, its fractional part (i.e., the part to the right of the decimal) will be dropped; if **d1** contains a number greater than 65,535, which it too large to be stored in a **char**, then it will be converted to 65,535.

The lines:

```
n1 = length(d1(:));
n2 = length(d2(:));
n3 = length(d3(:));
```

determine the number of **char**s, **single**s, and **int32**s that need to be written, and the function call

```
fwrite(fid,[n1,n2,n3],'int16')
```

writes these three numbers as three separate values of type **int16**. They become the first three numbers in the binary file.

Once we have written our file, we may want to read it. Even if the file is designed to be read by another application, we will want to read it in MATLAB as a test to make certain that it is correct. Here is a function that will read a file in the same format that is written by **custom_write_bin**:

```
function [o1,o2,o3,o4] = custom_read_bin(filename)
fid = fopen(filename,'r');
if fid < 0
   fprintf('error opening file\n');
   return;
end
nums = fread(fid,3,'int16');
o1 = char(fread(fid,nums(1),'char'))';
o2 = fread(fid,nums(2),'single');
o3 = fread(fid,nums(3),'int32');
o4 = fread(fid,'single');
fclose(fid);
```

A new feature of the function **fread** appears here: a numeric (non-**char**) argument after the file-identifier argument. This argument tells the function the number of values it must read in. In the first instance,

```
nums = fread(fid,3,'int16');
```

it is told to read in three values. Since the next argument tells **fread** that these values are encoded in the file using type **int16**, 48 bytes will be read (3 x 16) in three groups of 16 bytes, each of which is decoded as one **int16**. The resulting three values are returned as a vector, which we have chosen to store in the variable **nums**.

It may be surprising to see the conversion functions in the command,

```
o1 = char(fread(fid,nums(1),'char'))';
```

where we have used the **char()** function to convert the value returned by **fread** to the type **char**. The reader might have expected that, since **fread** was told to decode the bytes in the file as a **char**, that **fread** would return a value of type **char**. Instead, **fread** by default returns values of type **double**. We have used this default and then converted the returned value to **char**. However, **fread** can be told to return values of any type by altering the precision argument to include both the input type and the output type separated by the string **=>** (equals, greater-than). For example, we could have written this statement as follows with the same result:

```
o1 = fread(fid,nums(1),'char=>char'))';
```

It is not necessary that the input and output values specified in this way be the same. If, for example, we had wished to return the last values as **int64**s, we would have given this command:

```
o4 = fread(fid,'single=>int64');
```

Now it is time to see these functions in action. First we make up some data:

```
>> header = 'Data requested from 4/17/2011';
>> Vega = [8, 7, -145];
>> VLA = [1000, 2000, 700, 0, 48];
>> W9GFO = [1.45e8, 34e6, 4e7, -1e8];
```

Then, we feed the data to **custom_write_bin**, which writes the data into a file named **'Arecibo.dat'**:

```
>> custom_write_bin(header, Vega, VLA, W9GFO,
'Arecibo.dat');
```

At this point, if we use our operating system to look in the current directory, we will see **Arecibo.dat** listed there.

Finally, we call **custom_read_bin**, which reads the data from **'Arecibo.dat'** into the four variables **o1**, **o2**, **o3**, and **o4**:

```
>> [o1,o2,o3,o4] = custom_read_bin('Arecibo.dat')

o1 =
Data requested from 4/17/2011

o2 =
     8
     7
  -145

o3 =
        1000
        2000
         700
           0
          48

o4 =
   145000000
    34000000
    40000000
  -100000000
```

It is apparent that the type of **o1** is **char** because, it was printed in character form in the Command Window. To make certain of that fact we can use the class function:

```
>> class(o1)
ans =
char
```

The other variables are, as expected, of class **double**:

```
>> class(o2),class(o3),class(o4)
ans =
double
ans =
double
ans =
double
```

There are other options available for reading and writing, and other functions for dealing with files, as Table 2.24 shows, but the methods we have shown in the four examples,

`read_array_bin`
`write_array_bin`
`read_customn_bin`
`write_custom_bin`

should handle almost any task required when it comes to reading and writing binary files with MATLAB.

Table 2.24 *Functions for file I/O*

FUNCTION	FUNCTIONALITY
`fclose`	close a text file or binary file
`feof`	detect the end of a text file or binary file
`ftell`	determine the current position in a text file or binary file
`type`	display the contents of a text file in the Command Window
`fseek`	move to a position in a text file or binary file
`frewind`	move to the beginning of a text file or binary file
`fopen`	open a text file or binary file
`fread`	read from a binary file into variables
`importdata`	read from a text file into variables
`textscan`	read from a text file into variables
`fscanf`	read from a text file into variables
`ferror`	return the error string from the most recent file I/O operation
`fwrite`	write from variables to a binary file
`fprintf`	write from variables to a text file (or the Command Window)

Additional Online Resources

- Video lectures by the authors:
  - Lesson 8.1 Introduction to File Input/Output (15:00)
  - Lesson 8.2 Excel Files (9:12)
  - Lesson 8.3 Text Files (12:17)
  - Lesson 8.4 Binary Files (25:23)

Concepts From This Section

Computer Science and Mathematics:

file
folder and directory
text files
binary files
opening and closing files

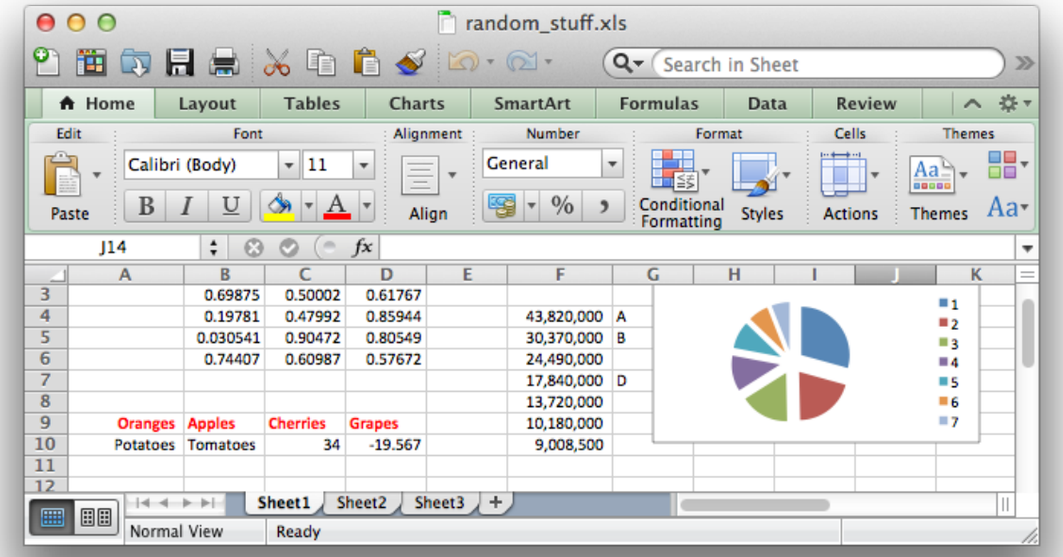
MATLAB:

navigating folder-directories
`cd`
`pwd`
`ls`
reading and writing MAT-files
`save`
`load`
reading and writing Excel files
opening and closing files:
`fopen`
permissions
`fclose`
reading text or binary files
`fread`
`fwrite`

Practice Problems

Problem 1. Write a function named **`xls_numbers_read`** that takes one string as an input argument (the function does not have to check the format of the input) and returns one array as an output argument. If it is called like this: **`A = xls_numbers_read(filename)`**, then it reads an Excel file whose filename is stored in the input argument and copies only the numeric data into the array **`A`**. For example, suppose the Excel file named **`random_stuff.xls`** looks like this when opened in Excel:

Figure 2.54 Problem 1



	A	B	C	D	E	F	G
3		0.69875	0.50002	0.61767			
4		0.19781	0.47992	0.85944		43,820,000	A
5		0.030541	0.90472	0.80549		30,370,000	B
6		0.74407	0.60987	0.57672		24,490,000	
7						17,840,000	D
8						13,720,000	
9	Oranges	Apples	Cherries	Grapes		10,180,000	
10	Potatoes	Tomatoes	34	-19.567		9,008,500	
11							
12							

Here is **`xls_numbers_read`** operating on this file:

```
>> Numbers = xls_numbers_read('random_stuff')
Numbers =
      NaN      NaN      NaN   NaN          NaN
      NaN      NaN      NaN   NaN          NaN
  0.69875  0.50002  0.61767   NaN          NaN
  0.19781  0.47992  0.85944   NaN    4.382e+07
 0.030541  0.90472  0.80549   NaN    3.037e+07
  0.74407  0.60987  0.57672   NaN    2.449e+07
      NaN      NaN      NaN   NaN    1.784e+07
      NaN      NaN      NaN   NaN    1.372e+07
      NaN      NaN      NaN   NaN    1.018e+07
      NaN       34  -19.567   NaN  9.0085e+06
```

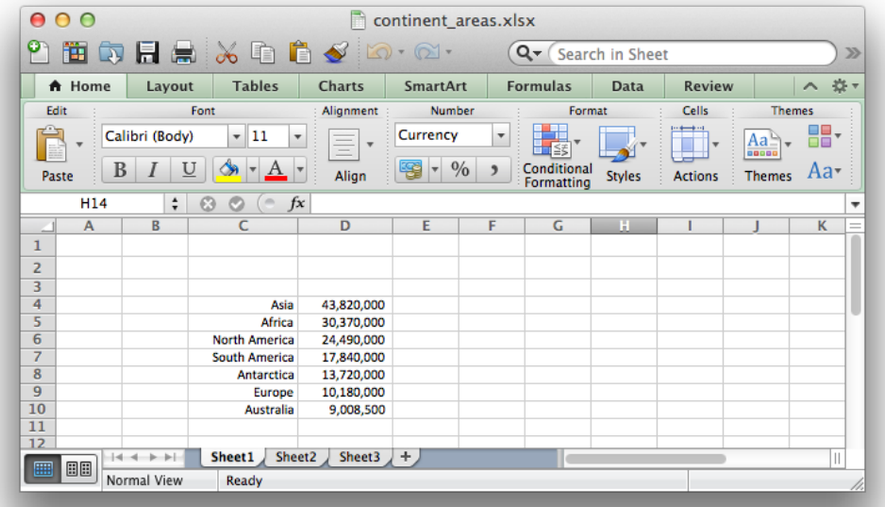
?

Problem 2. Write a function named **`xls_strings_read`** that takes one string as an input argument (the function does not have to check the format of the input) and returns one array as an output argument. If it is called like this: **`C = xls_strings_read(filename)`**, then it reads an Excel file whose filename is stored in the input argument and copies into the cell array **`C`** only the contents of the cells that contain strings instead of numbers. For example, here is the function operating on the Excel file, **`random_stuff.xls`**, which is shown in the previous problem:

```
>> Strings = xls_strings_read('random_stuff')
Strings =
''          ''          ''          ''          ''  ''  ''
''          'First'     'Second'    'Third'     ''  ''  ''
''          ''          ''          ''          ''  ''  ''
''          ''          ''          ''          ''  ''  'A'
''          ''          ''          ''          ''  ''  'B'
''          ''          ''          ''          ''  ''  ''
''          ''          ''          ''          ''  ''  'D'
''          ''          ''          ''          ''  ''  ''
'Oranges'   'Apples'    'Cherries'  'Grapes'    ''  ''  ''
'Potatoes'  'Tomatoes'  ''          ''          ''  ''  ''
```

Problem 3. Write a function named **`xls_areas_read`** that takes one string as an input argument (the function does not have to check the format of the input) and returns one array as an output argument. If it is called like this: **`[names,areas] = xls_areas_read (filename)`**, then it reads an Excel file whose filename is stored in filename and copies the names an areas of land masses listed in the file, into **`names`** and **`areas`**. The output argument **`names`** is a column vector of **`cells`** with each element pointing to one string; the output argument **`areas`** is a column vector of **`doubles`** with the value of each element equaling one area. The format of the Excel files that it must read is illustrated by the file **`continent_areas.xlsx`**.

Figure 2.55 Problem 3



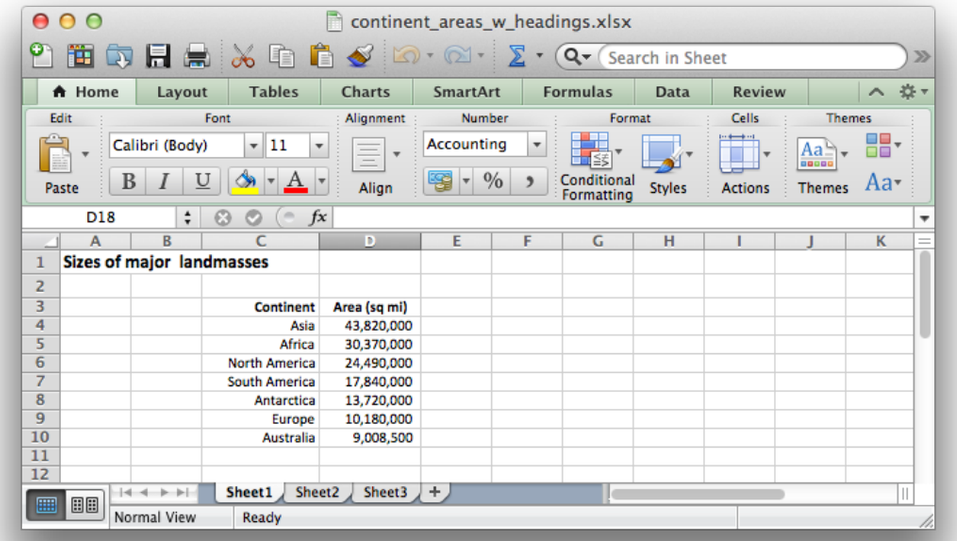
The names of landmasses are contiguous in a column, and each has its area beside it in the neighboring column to the right. Note, that the exact location in the spreadsheet is not part of the format, nor is the number of areas. Thus, the function must work properly even if first name is not necessarily in cell C4 and even if there are more than seven areas or fewer. The use of the function with **`continent_areas.xlsx`** looks like this:

```
>> [names,areas] = xls_read_areas('continent_areas')
names =
    'Asia'
    'Africa'
    'North America'
    'South America'
    'Antarctica'
    'Europe'
    'Australia'

areas =
    43820000
    30370000
    24490000
    17840000
    13720000
    10180000
     9008500
```

?

Problem 4. Write a function named **`xls_areas_read_w_headings`** that has the same input an output arguments as **`xls_areas_read`**, and reads files of areas with the same format that it reads, but it its input file may contain headings as well as names and areas. The only numeric data in the file are the areas. HINT: Use the function **`isa`** to determine where the numeric cells are an **`isnan`** to identify values equal to **`NaN`**. An example is given in Figure 2.56 of an input file displayed in Excel, followed by the application of the function to that same file:

Figure 2.56 *Problem 4*




```
>> [names,areas] =
xls_read_areas_w_headings('continent_areas_w_headings');
```

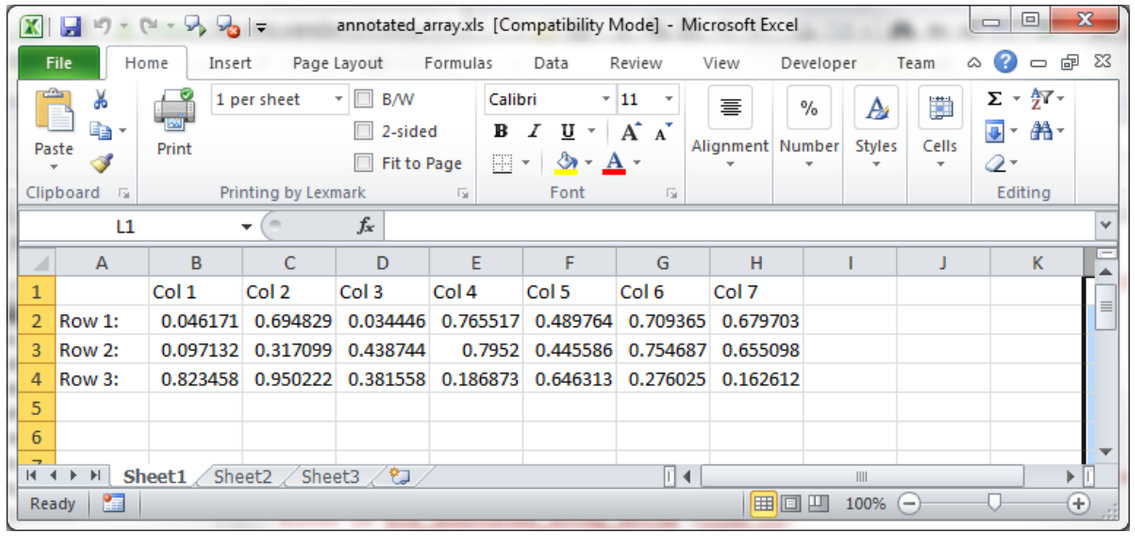
This command has exactly the same output as that shown in the previous problem.

Problem 5. Write a function named **`xls_annotated_array_write`** that takes one string and one numeric array as input arguments (the function does not have to check the format of the input) and returns no output argument. If it is called like this: **`xls_annotated_array_write(filename,A)`**, then it writes **`A`** into the an Excel file whose file name is given by the first argument, and it labels each row and each column. Here is an example call of the function on a Windows computer (This function will not work on an Apple computer, nor will any function that writes strings into an Excel spreadsheet).

```
>> rng(0);
>> xls_numbers_write('annotated_array.xls',rand(3,7))
```

Under Windows the resulting file when opened in Excel is shown in the Figure.

Figure 2.57 *Problem 5*



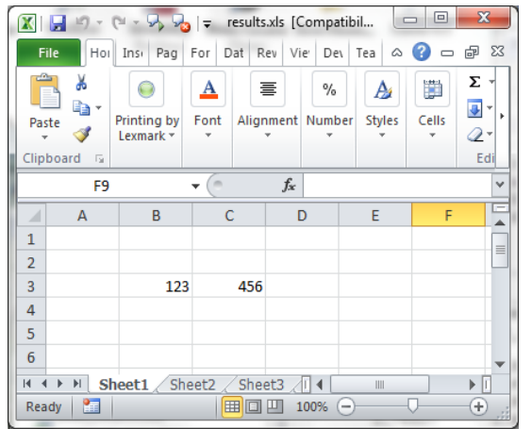
The required format of the labels can be seen from this example. HINT: The function **`num2str`** may be helpful. It returns a string in which a number given it as an input argument is written out.

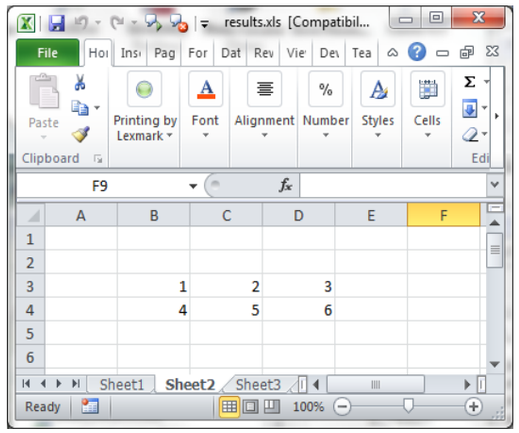
?

Problem 6. Write a function named **`xls_2_sheets_write`** that takes one string and two numeric arrays as input arguments (the function does not have to check the format of the input) and returns no output argument. If it is called like this: **`xls_2_sheets_write(filename,A,B)`**, then it writes **`A`** into the first sheet and **`B`** into the second sheet of an Excel file whose file name is given by the first argument. **`A`** and **`B`** must each begin on the third row and second column of its spreadsheet. Here is an example call of the function on a Windows computer:

```
>> xls_2_sheets_write('results.xls',[123,456],
[1 2 3; 4 5 6])
```

On the Windows computer an Excel file will be produced, whose first and second sheets are shown below as displayed by Excel:

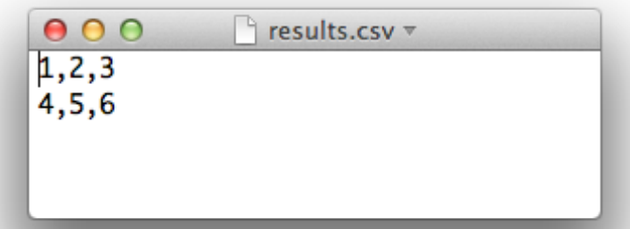




Here is an example call of the function on an Apple Macbook:

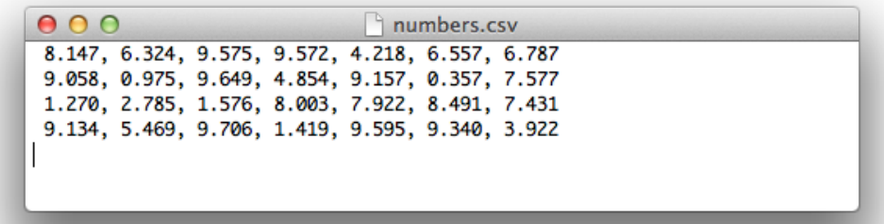
```
>> xls_2_sheets_write('results.xlsx',[123,456],
[1 2 3;4 5 6])
Warning: Could not start Excel server for export.
XLSWRITE will attempt to write file in CSV format.
> In xlswrite at 175
  In xls_2_sheets_write at 2
Warning: Could not start Excel server for export.
XLSWRITE will attempt to write file in CSV format.
> In xlswrite at 175
  In xls_2_sheets_write at 3
```

On the MacBook, no Excel file is produced. Instead a text file named **`results.csv`** will be produced. The extension **`csv`** stands for "comma-separated values". The meaning of comma-separated can be seen from the file as displayed here by the MacBook's text editor. Note that only the values in second input argument made it into the file:



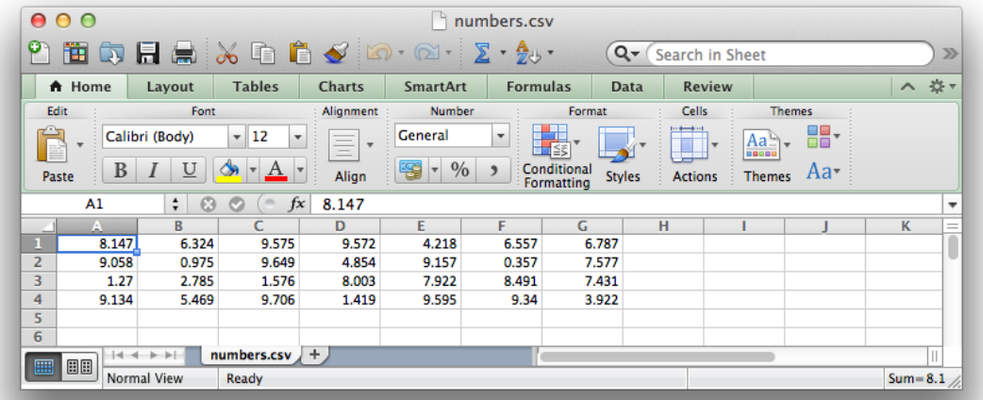
Problem 7. Write a function named **`txt_numbers_write`** that takes one string and one numerical array as an input argument (the function does not have to check the format of the input) and returns no output argument. If it is called like this, **`txt_numbers_write(filename,A)`**, then it writes the elements of **`A`** into a text file whose name is stored in **`filename`** using the format called Comma-Separated Values (CSV), in which numbers on the same row in **`A`** appear on the same line of the file, and one number is separated from the next by a comma and a space. Each number in **`A`** is written in fixed-point decimal representation with at least three digits to the right of the decimal point. If the text file does not exist, then **`txt_numbers_write`** must cause the file to be created. Here is an example call with the resulting output file displayed by the MacBook text editor:

```
>> rng(0)
>> txt_numbers_write('numbers.csv',10*rand(4,7))
```



Files in CSV format can also be read by Excel, as shown in the Figure.

Figure 2.58 *Problem 7*



Problem 8. Write a function named **`txt_annotated_array_write`** that takes one string and one numeric array as input arguments (the function does not have to check the format of the input) and returns no output argument. If it is called like this:

```
txt_annotated_array_write(filename,A),
```

then it writes **`A`** into a text file whose name is stored in **`filename`** using an elaborate format that includes labels for each row and each column, as in the previous problem, but also uses the characters, **`|`** and **`-`**, to outline the labels and the data. If the text file does not exist, then **`txt_annotated_array_write`** must cause it to be created. Each number in **`A`** is written in fixed-point decimal representation with at least three digits to the right of the decimal point. The rest of the required format is shown in the following example showing the function call and the resulting text file displayed by the MacBook text editor:

```
>> rng(0)
>> txt_annotated_array_write('annotated_array.txt', 10*rand(7,8))
```

Figure 2.59 Problem 8

annotated_array.txt

```
        ------------------------------------------------------------------
       ||  Col  1|  Col  2|  Col  3|  Col  4|  Col  5|  Col  6|  Col  7|  Col  8|
-------||--------|--------|--------|--------|--------|--------|--------|--------|
Row  1:||   8.147|   9.706|   8.491|   0.462|   1.869|   4.984|   5.472|   2.511|
Row  2:||   9.058|   9.572|   9.340|   0.971|   4.898|   9.597|   1.386|   6.160|
Row  3:||   1.270|   4.854|   6.787|   8.235|   4.456|   3.404|   1.493|   4.733|
Row  4:||   9.134|   8.003|   7.577|   6.948|   6.463|   5.853|   2.575|   3.517|
Row  5:||   6.324|   1.419|   7.431|   3.171|   7.094|   2.238|   8.407|   8.308|
Row  6:||   0.975|   4.218|   3.922|   9.502|   7.547|   7.513|   2.543|   5.853|
Row  7:||   2.785|   9.157|   6.555|   0.344|   2.760|   2.551|   8.143|   5.497|
Row  8:||   5.469|   7.922|   1.712|   4.387|   6.797|   5.060|   2.435|   9.172|
Row  9:||   9.575|   9.595|   7.060|   3.816|   6.551|   6.991|   9.293|   2.858|
Row 10:||   9.649|   6.557|   0.318|   7.655|   1.626|   8.909|   3.500|   7.572|
Row 11:||   1.576|   0.357|   2.769|   7.952|   1.190|   9.593|   1.966|   7.537|
-------||------------------------------------------------------------------
```

Problem 9. Write a function named **`txt_numbers_read`** that takes one string as an input argument (the function does not have to check the format of the input) and returns one array as an output argument. If it is called like this: **`A = xls_numbers_read(filename)`**, then it reads a text file whose filename is stored in the input argument and whose format is identical to that of the file written by **`txt_numbers_write`**, which is described in a problem above and writes an array in comma-separated-value (CSV) format. After reading the input file, **`txt_numbers_read`** copies the numeric data into the array **`A`**. Here is an example that uses the text file **`numbers.csv`** that is shown above in the same problem above.

```
>> A = txt_numbers_read('numbers.csv')
A =
  8.147   6.324   9.575   9.572   4.218   6.557   6.787
  9.058   0.975   9.649   4.854   9.157   0.357   7.577
  1.27    2.785   1.576   8.003   7.922   8.491   7.431
  9.134   5.469   9.706   1.419   9.595   9.34    3.922
```

?

Problem 10. Write a function named **`txt_annotated_array_read`** that takes one string as an input argument (the function does not have to check the format of the input) and returns one array as an output argument. If it is called like this: **`A = txt_annotated_array_read(filename)`**, then it reads a text file whose filename is stored in the input argument and whose format is identical to that of the file written by **`txt_annotated_array_write`**, which is described in an earlier problem above. Then **`txt_annotated_array_read`** copies all the numeric data that it finds in the file into the output array **`A`**. Here is an example that uses the text file **`annotated_array.txt`** that is shown above in the earlier problem:

```
>> A = txt_annotated_array_read('annotated_array.txt')

A =
 Columns 1 through 7
8.147   9.706   8.491   0.462   1.869   4.984   5.472
9.058   9.572   9.34    0.971   4.898   9.597   1.386
1.27    4.854   6.787   8.235   4.456   3.404   1.493
9.134   8.003   7.577   6.948   6.463   5.853   2.575
6.324   1.419   7.431   3.171   7.094   2.238   8.407
0.975   4.218   3.922   9.502   7.547   7.513   2.543
2.785   9.157   6.555   0.344   2.76    2.551   8.143
5.469   7.922   1.712   4.387   6.797   5.06    2.435
9.575   9.595   7.06    3.816   6.551   6.991   9.293
9.649   6.557   0.318   7.655   1.626   8.909   3.5
1.576   0.357   2.769   7.952   1.19    9.593   1.966

 Column 8
2.511
6.16
4.733
3.517
8.308
5.853
5.497
9.172
2.858
7.572
7.537
```

Problem 11. Write a function named **`bin_numbers_write`** that takes one string and one two-dimensional numerical array as an input argument (the function does not have to check the format of the input) and returns no output argument. If it is called like this, **`bin_numbers_write(filename,A)`**, then it uses binary format to write into a file whose name is stored in **`filename`**. It writes the number of rows of **`A`** followed by the number of columns using the type **`unint16`**, followed by the values of the elements of **`A`** in column-major order using the type **`double`**. If the file does not exist, then **`bin_numbers_write`** must cause the file to be created. Here is an example call:

```
>>  rng(0)

>> A = randi(99,8,6)

A =
    81    95    42    68    28    44
    90    96    91    76     5    38
    13    16    79    74    10    76
    91    97    95    39    82    79
    63    95    65    65    69    19
    10    49     4    17    32    49
    28    80    85    70    95    45
    55    15    93     4     4    64

>> bin_numbers_write('numbers.bin',A)
```

?

Problem 12. Write a function named **`bin_numbers_stats_write`** that takes one string and one two-dimensional numerical array as an input argument (the function does not have to check the format of the input) and returns no output argument. If it is called like this:

```
bin_numbers_stats_write(filename,A)
```

then it uses binary format to write into a file whose name is stored in **`filename`**. It writes the number of rows of **`A`** followed by the number of columns using the type **`unint16`**, followed by the means of the values in each column followed by the mean of the values in each row, followed by the values of the elements of **`A`** in column-major order with the means and the values all being written in the type **`double`**. If the file does not exist, then **`bin_numbers_write`** must cause the file to be created. Here is an example call using the same array **`A`** generated for **`bin_numbers_write`** in the immediately preceding problem:

```
>> bin_numbers_stats_write('numbers_stats.bin',A)
```

Problem 13. Write a function named **`bin_numbers_read`** that takes one string as an input argument (the function does not have to check the format of the input) and returns one array as an output argument. If it is called like this: **`A = bin_numbers_read(filename)`**, then it reads a text file whose filename is stored in the input argument and whose format is identical to that of the file written by **`bin_numbers_write`**, which is described in an earlier problem. The function reads the dimensions of the two-dimensional matrix stored in the file and the values of its elements and it returns that matrix in **`A`** such that it has the same dimensions. Here is an example call using the same file that was generated in the example shown in the earlier problem of **`bin_numbers_write`** being called:

```
>> A = bin_numbers_read('numbers.bin')

A =
    81    95    42    68    28    44
    90    96    91    76     5    38
    13    16    79    74    10    76
    91    97    95    39    82    79
    63    95    65    65    69    19
    10    49     4    17    32    49
    28    80    85    70    95    45
    55    15    93     4     4    64
```

?

Problem 14. Write a function named **bin_numbers_stats_read** that takes one string as an input argument (the function does not have to check the format of the input) and returns one array as an output argument. If it is called like this:

```
[col_means,row_means,A] = bin_numbers_stats_read(filename),
```

then it reads a text file whose filename is stored in the input argument and whose format is identical to that of the file written by **bin_numbers_stats_write**, which is described in an earlier problem. The function reads the dimensions of the two-dimensional matrix stored in the file, the means of its columns, the means of its rows, and the values of its elements and it returns a row vector of the column means in **col_means**, a column vector of the row means **row_means**, and the values of the matrix in **A** such that it has the same dimensions as those read from the file. Furthermore, if the third output argument is omitted from the call, the values of the matrix are not read from the file to save time. Here are two example calls using the same file that was generated in the example of shown above of **bin_numbers_write** being called:

```
>> [col_means,row_means,A] =
bin_numbers_stats_read('numbers_stats.bin')

col_means =
   53.875   67.875   69.25   51.625   40.625   51.75

row_means =
       59.667
           66
       44.667
         80.5
       62.667
       26.833
       67.167
       39.167
```

```
A =
    81    95    42    68    28    44
    90    96    91    76     5    38
    13    16    79    74    10    76
    91    97    95    39    82    79
    63    95    65    65    69    19
    10    49     4    17    32    49
    28    80    85    70    95    45
    55    15    93     4     4    64


>> [col_means,row_means] =
bin_numbers_stats_read('numbers_stats.bin')

col_means =
  53.875  67.875   69.25  51.625  40.625   51.75

row_means =
       59.667
           66
       44.667
         80.5
       62.667
       26.833
       67.167
       39.167
```

In the second call the values of the matrix are not read from the file.

Functions Reloaded

Objectives

(1) MATLAB is very flexible when it comes to the number of input and output arguments. We will learn how this feature can be utilized.

(2) We will learn how to use a powerful technique called "recursion" that makes it possible to solve difficult problems elegantly. We will learn that recursive functions call themselves!

(3) We will see classical recursive problems, such as the greatest common divisor or the factorial and will cover additional recursive problems, such as prime factorization, in the practice problems.

(4) MATLAB makes it easy to make dynamic plots. We will learn how to employ animation to create really powerful visualizations.

Functions were introduced before selection and loops. Hence, some important aspects had to be postponed. This section will give functions their full power.

In this section, we will cover a number of concepts related to functions that we could not cover in the section entitled Functions.

Variable Number Of Arguments

Robust programming requires that your functions be able to handle function calls with the numbers of input and/or output arguments varying from one call to another. Sometimes that simply means returning an error message to the user informing them that the function needs additional arguments. Other times, it might significantly change what your function does. MATLAB has a series of built-in functions that will help you determine how your function has been called. The three most important of these functions are:

- **nargin**: Returns the number of actual input arguments used to call the function.
- **nargout**: Returns the number of actual output arguments requested by the function call.
- **narginchk(min,max)**: Returns a standard error message if your function was called with fewer than **min** or more than **max** arguments.

These functions should be used to determine how many arguments have been passed to the function in the function call and how many arguments have been requested as output. The function you write should be able to accommodate the varying possibilities.

The following example problem illustrates the use of **nargin** and **nargout** to write robust functions. The task is to compute a multiplication table of size either **n**-by-**n** or **n**-by-**m**. Also, if an optional input argument **start** is supplied, we will start the multiplication table at that value as opposed to 1. The default output variable **table** will contain the actual table, while the optional second output argument will be assigned the sum of all the elements of the multiplication table. The only way to decide that some of the inputs and outputs are optional is to look at the code and see whether the function relies on the **nargin** and/or **nargout** to adjust its behavior. Basically, we implement polymorphism using **nargin** and **nargout**.

```
function [table summa] = multable(n, m, start)
%MULTABLE multiplication table.
%    T = MULTABLE(N) returns an N-by-N matrix
%    containing the multiplication table for
%    the integers 1 through N.
%    MULTABLE(N,M) returns an N-by-M matrix.
%    MULTABLE(N,M,S) returns a N-by-M matrix
%    containing the multiplication table for
%    the integers S through N+S-1 and M+S-1.
%    [T SM] = MULTABLE(...) returns the matrix
%    containing the multiplication table in T
%    and the sum of all its elements in SM.
%    All three input arguments must be positive
%    integers.

if nargin < 1
    error('must have at least one input argument');
end

if nargin < 3
    start = 1;
elseif ~isscalar(start) || start < 1 || ...
                                start ~= fix(start)
    error('start needs to be a positive integer');
end

if nargin < 2
    m = n;
elseif ~isscalar(m) || m < 1 || m ~= fix(m)
    error('m needs to be a positive integer');
end

if ~isscalar(n) || n < 1 || n ~= fix(n)
    error('n needs to be a positive integer');
end

table = (start : (n+start-1))' * ...
                      (start : (m+start-1));

if nargout == 2
    summa = sum(table(:));
end
```

Notice that in this relatively long function, the actual work is done in just two lines:

```
table = (start : (n+start-1))' * ...
                    (start : (m+start-1));
```

and

```
summa = sum(table(:));
```

The rest of the function are comments that provide information to the help utility and code that makes sure that the inputs and outputs are correct.

Notice how we use the **nargin** and **nargout** functions. If **nargin** is smaller than 3, that means that the input argument **start** was not supplied because, as expected, MATLAB fills in the input arguments from left to right. In this case, we initialize the variable **start** at 1. Otherwise, we make sure that **start** is a positive scalar integer. Similarly, if **nargin** is smaller than 2, then **m** was not provided either. In this case, we assign the value of **n** to **m**. The input **n** is required, and we made sure that it was supplied at the very beginning of the function, so at this point it is safe to use it to initialize **m**.

The variable number of output arguments is handled by checking whether **nargout** equals to 2 or not. If so, the user wants the function to return the sum of the elements of the multiplication table also, so we compute it.

Note that the value of **nargout** is always at least one, even if the user does not assign the return value to a variable. That happens because MATLAB assigns the value of the first output argument to **ans** by default.

Let's run the function a few different ways now:

```
>> multable
Error using multable (line 13)
must have at least one input argument

>> multable()
Error using multable (line 13)
must have at least one input argument
```

This second example shows that there are two ways to call a function without giving arguments. The effect is the same. The first way is customary in MATLAB. (It is illegal in many other languages!)

```
>> multable(4)
ans =
     1     2     3     4
     2     4     6     8
     3     6     9    12
     4     8    12    16

>> multable(4,6)
ans =
     1     2     3     4     5     6
     2     4     6     8    10    12
     3     6     9    12    15    18
     4     8    12    16    20    24

>> multable(4,6,3)
ans =
     9    12    15    18    21    24
    12    16    20    24    28    32
    15    20    25    30    35    40
    18    24    30    36    42    48

>> [x y] = multable(4,6,3)
x =
     9    12    15    18    21    24
    12    16    20    24    28    32
    15    20    25    30    35    40
    18    24    30    36    42    48
y =
   594

>> [x y] = multable(-3)
Error using multable (line 29)
input argument n needs to be a positive integer
```

If we run the help utility, this is what we see:

```
>> help multable
 multable multiplication table.
     T = multable(N) returns an N-by-N matrix
     containing the multiplication table for
     the integers 1 through N.
     multable(N,M) returns an N-by-M matrix.
     multable(N,M,S) returns a N-by-M matrix
     containing the multiplication table for
     the integers S through N+S-1 and M+S-1.
     [T SM] = multable(...) returns the matrix
     containing the multiplication table in T
     and the sum of all its elements in SM.
     All three input arguments must be positive
     integers.
```

Arbitrary number of input and/or output arguments

Sometimes it is desired to write a function that can take an arbitrary number of input arguments or output arguments. We know how to write a function that can take differing numbers of arguments, but not how to write one that can take any number of arguments. MATLAB provides a facility for arbitrary numbers of arguments through two special argument names: **varargin** (for "variable number of arguments in") and **varargout** (for "variable number of arguments out").

We will explain their uses through examples. Suppose we want a function that will take one or more numbers as input arguments and print them each on a separate line. We can write that function with the help of **varargin**, as follows:

```
function print_all_args(first,varargin)
fprintf('%d\n', first);
for ii = 1:nargin-1
    fprintf('%d\n', varargin{ii});
end
```

The special argument **varargin** must be the last argument in the list. It can also be the only argument. When the function is called with more input arguments than there are "normal" arguments (i.e., not **varargin**) in the function header, **varargin** receives the surplus. It is a **cell** vector, and each element points to one argument. Inside the function, it is treated as an ordinary **cell** array. Furthermore, the function **nargin**, which returns a count of the number of arguments used in the call, counts all arguments, both those that are captured by the normal arguments in the header and those that are captured by **varargin**. Here is this function in action,

```
>> print_all_args(14)
14
>> print_all_args(14,15,16,17,18)
14
15
16
17
18
```

Now suppose we want a function that will take an input vector and copy the value of each of its elements into separate output arguments. We can accomplish that with **varargout**. Here is a function that does it:

```
function [first,varargout] = distribute(v)
first = v(1);
for ii = 1:length(v)-1
    varargout{ii} = v(ii+1);
end
```

As is the rule with **varargin**, **varargout** must be the last output argument in the list. Also, in analogy to **varargin**, it holds any excess output arguments after the "normal" ones. Here is a function in the expected situation, namely, the length of the input vector is equal to the number of output arguments:

```
>> [a,b,c] = distribute([14,15,16])
a =
    14
b =
    15
c =
    16
```

If fewer output arguments are given, there is no error. The extra elements placed into **varargout** are simply ignored:

```
>> [a,b] = distribute([14,15,16])
a =
    14
b =
    15
```

On the other hand, if there are not enough elements in **varargout** to handle all the remaining output arguments, MATLAB complains and halts execution,

```
>> [a,b,c,d] = distribute_to_args([14,15,16])
??? Error using ==> distribute_to_args
Too many output arguments.
```

Persistent Variables

We have learned about local variables and global variables. In addition, there is another kind of variable in-between these two, but for its meaningful use, you need an if-statement that was not introduced until after the Functions section. This special kind of variable is called a **persistent variable**, because even though it is a local variable that is only accessible within the functions it is defined in, its value persists across function calls. Variables are declared to be persistent with the keyword **persistent**. Let us consider a quick example:

```
function total = accumulate(n)

persistent summa;

if isempty(summa)
    summa = n;
else
    summa = summa + n;
end
total = summa;
```

The function **accumulate** keeps adding up its input arguments as it is being called repeatedly. The function returns the cumulative total. The key to the function is the persistent variable **summa**, which is declared using the **persistent** keyword. When a function is saved in the editor, any persistent variables in it are initialized to the empty matrix. Hence, by means of the if-statement above, we can determine whether this is the first time the function has been called. If it is the first call, we set **summa** equal to the input argument **n**. Otherwise, we take the current value of **summa** and add **n** to it.

All persistent variables within a function can be re-initialized to the empty matrix in any of these three ways:

- by re-saving the function in the editor
- by clearing the function with **clear** (e.g., **clear accumulate**)
- by exiting and restarting MATLAB

It is an error to declare a variable persistent that already exists in the current workspace. That's why we could not use the output argument **total** to store the value inside **accumulate**; we needed to introduce the new variable, **summa**.

It is clear that we could not accomplish what accumulate does with regular local variables. The only other way would have been using a global variable. As pointed out before, however, globals should be used only as a last resort since they are error-prone because any other function and the workspace can

freely modify global variables. Persistent variables, on the other hand, are local variables and are visible only inside the function in which they are declared.

Recursion

It is time for you to learn a fundamental new method of programming called "recursion". Recursion is a powerful idea that makes it possible to solve some problems easily that would otherwise be quite difficult. Recursion cannot be used in all programming languages, but it is supported by most modern languages, including MATLAB.

The best way to show how recursion works is through an example. Let us write a function that computes the greatest common divisor (GCD) of two positive integers. A variant of Euclid's algorithm computes the GCD quite efficiently. The underlying idea is quite simple: the GCD of two numbers is the same as the GCD of the smaller number and the remainder of dividing the larger number by the smaller number. The mathematical definition looks like this:

$$\gcd(a,b) = \begin{cases} a & \text{if } b = 0 \\ \gcd(b, \text{remainder}(a,b)) & \text{if } a \geq b \text{ and } b > 0 \end{cases}$$

The interesting thing to notice is that the formula for computing GCD uses GCD itself! A definition of a concept that uses the concept itself is called a **recursive definition**, and the use of a concept in the definition of the concept is called **recursion**. At first, this might seem to be nonsense because the definition seems to require that the concept already be defined! Looking more closely at the formula, we see that it is only the second part that presents the potential problem because it is only in that part that the function gcd shows up in its own definition. It seems to be circular reasoning, but it is not. The crucial aspect of the second part that avoids circular reasoning is that $\gcd(a,b)$ is defined in terms of $\gcd(c,d)$ where it is guaranteed that $c \leq a$ and $d < b$ because $a \leq b$ and the remainder is always smaller than the divisor. Thus, the definition of $\gcd(a,b)$ requires only that gcd be previously defined for smaller numbers than a and b. Therefore, it is not circular reasoning. It is more like spiral reasoning because, instead of going round and round in a circle, we are spiraling down from a and b towards 0. Conveniently, the first part of the formula defines the value of $\gcd(a,b)$ when b is 0 without relying on gcd at all.

Here is the MATLAB function that implements GCD:

```
function d = rgcd(x,y)

a = max([x y]);
b = min([x y]);

if b == 0
    d = a;
else
    d = rgcd(b,rem(a,b));
end
```

As you can see, we named the function **rgcd** for "recursive gcd". First, we make sure that the variable **a** is the larger of the two arguments (or at least equal to **b**) and **b** is the smaller. Then we simply implement the recursive definition of the GCD given previously. If **b** is **0** then the **gcd** is **a**, so we assign **a** to the output argument **d**. Otherwise, we call the **rgcd** function with the arguments **b** and the remainder of **a** divided by **b** (relying on the built-in MATLAB function **rem**).

Wait a minute! How can we call the function **rgcd** from itself? Is that legal? What happens with variables? It turns out that it is completely legal. It is called a **recursive function call** and it can be a very useful programming tool. A function that makes a recursive function call is a **recursive function**.

To see what happens with the variables inside the function during a recursive function call, we need to take a small detour. Remember that the input and output arguments of a function and all its other variables have local scope, that is, they exist only inside the function and are not accessible from the outside. That makes it possible to use the same variable names in the workspaces of multiple functions without a clash. How does this work? MATLAB, just like most other programming languages, stores local variables and function arguments on the **stack**. The stack is an area of the computer's memory dedicated for this purpose. It is called a stack, because only its top is accessible: MATLAB puts variables on the top of the stack or takes them off from the top. (Think of a deck of cards.) When a function is called, a new area on the top of the stack is allocated for the function to store all of its arguments and local variables. This area is called a **stack frame.** A stack frame is also known as a **frame** and as an **activation record**. The function is free to use this memory area to access all these variables, modify their values, etc. When the function calls another function, a new stack frame is created for the use of this second function. This function can use its frame, but it cannot access the stack frame of the function that called it (or that of any other function). When the function returns, its stack frame is discarded and the caller will have access to the frame at the top of the stack which happens to be its own stack frame.

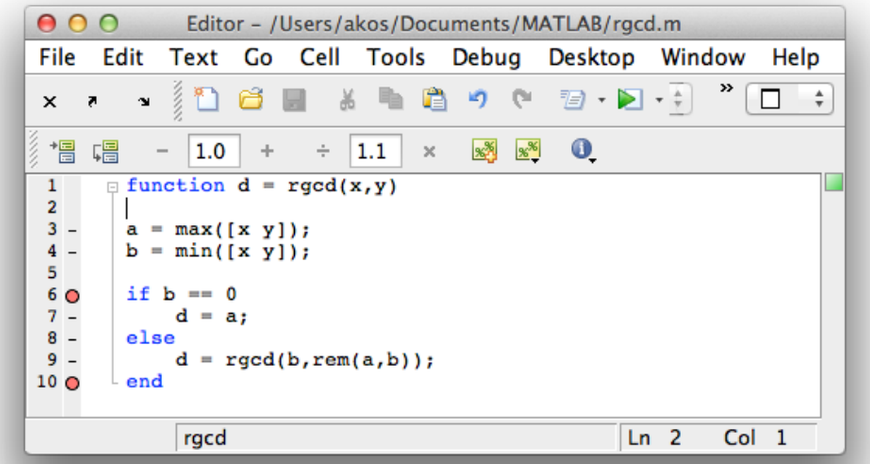
This is the reason that recursive functions work just fine: each call to a new instance of the recursive function will get its own stack frame, that is, it will have its very own set of arguments and local variables. These variable have no relation to the variables of other instances of the same function, because they sit in their very own stack frame.

Let's look at our **rgcd** function again, this time inside the debugger window (Figure 2.60).

Notice the breakpoints in lines 6 and 10. The first one is before the recursive function call and the second is right after it. If we run the function like this,

```
>> rgcd(12,32)
```

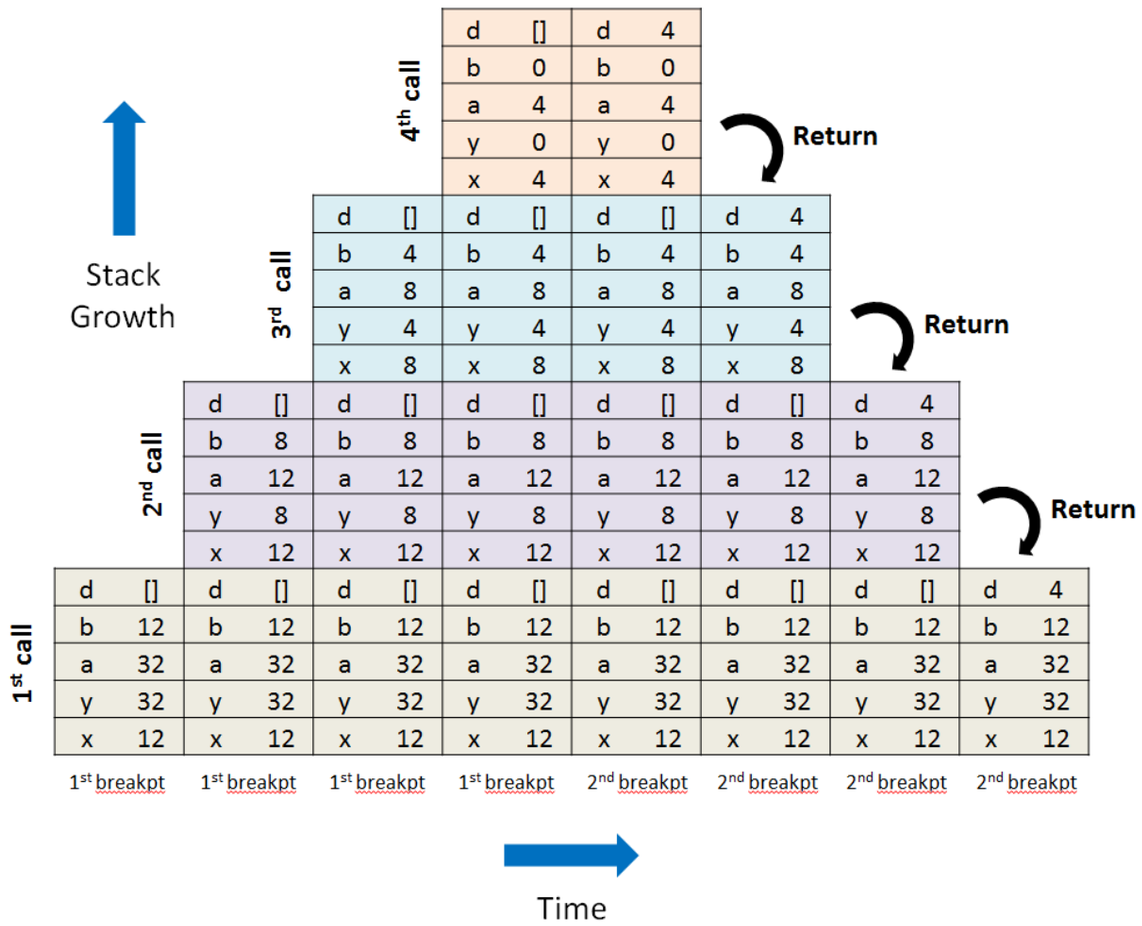
Figure 2.60 The ***rgcd*** *function inside the debugger with two breakpoints*



we shall hit these breakpoints repeatedly. Figure 2.61 shows how the stack will look as we go through the code.

The lower left corner shows the single stack frame on the stack after we call the function the first time just when we hit the first breakpoint. The input arguments **x** and **y** have the values **12** and **32** respectively, while the local variables **a** equals **32** and **b** equals **12**. The output argument **d** has not been initialized yet. The next time we hit a breakpoint may seem surprising since it is the first breakpoint again. This is because before we hit the second breakpoint, we call **rgcd** again and hit the first one again inside the second instance of the function. Notice that the stack now has a second stack frame on it, with the input arguments **x = 12** and **y = 8**. This is because we called **rgcd** with **b** (which is **12**) and the remainder of **a/b** (which is **rem(32,12) = 8**). Repeating this pattern, we call **rgcd** a third time with **8** and **4** as input arguments and hit the first breakpoint again and finally, we call **rgcd** a fourth time with **4** and **0** as inputs. This is where the recursion ends, because the if-statement in line 6 finally gets a true condition and assigns **a** (which is **4**) to the output argument **d**. This is when we hit the second

Figure 2.61 *Evolution of the stack as recursion occurs after the* ***rgcd(12,32)*** *call*



breakpoint for the very first time. Now the stack frame shows that **d** has finally been assigned a value (**4**).

As we return from the 4th call to **rgcd**, the stack frame is removed from the top of the stack and we hit the second breakpoint again. The output argument **d** gets assigned **4**, since that was the result of the 4th call to the **rgcd** function. As we keep returning from the recursive function calls, the top stack frame is removed and the result propagates to the caller. Finally, we return from the original call to **rgcd** to the Command Window with the result **4**.

```
>> rgcd(12,32)

ans =
     4
```

We are almost done with the **rgcd** function. However, consider this call:

```
>> rgcd(12,-32)

Maximum recursion limit of 500 reached. Use set(0,'Recur-
sionLimit',N)to change the limit.  Be aware that exceed-
ing your available stack space can crash MATLAB and/or
your computer.

Error in rgcd
```

What happened here? We ran into the problem called infinite recursion. In our **rgcd** function, similarly to the recursive definition of the GCD, there is a case when we do not rely on recursion any more. In this case, if **b** equals **0**, we do not need to call **rgcd** anymore, we simply return **a** as the result. This is true for all recursive functions: there must be at least one **base case**, which is a case in which there is no recursive call. A base case in a function always corresponds to a base case in the definition that the function implements, and there must also be a base case in any recursive definition for it to be valid. There must also be at least one **recursive case** in order for a function or definition to earn the label "recursive", which is a case that employs the function itself or the concept that is being defined itself. Reaching the base case stops the recursion, and without one, the chain of recursive function calls would never stop. This is what happened above. The recursive definition of gcd is valid for positive integers. Our function never checks whether the input arguments are valid. When we call the function with a negative number, in the chain of recursive function calls, **b** never becomes **0**, hence, the recursion never stops and stack frames keep getting added to the stack with none being removed. MATLAB prevents true infinite recursion by setting a limit of 500

calls to be legal. Once we reach this limit, MATLAB notices, quits the function and returns with the error message above.

We have to be very careful when writing recursive functions and need to make sure that we always reach the base case and do not get into infinite recursion. So, let us make sure that our **rgcd** function is prepared for all possible cases of bad input arguments:

```
function d = rgcd(x,y)
if (~isscalar(x) || ~isscalar(y) || ...
        x ~= fix(x) || y ~= fix(y) || ...
                            x < 0 || y < 0)
    error('x and y must be positive integers');
end

a = max([x y]);
b = min([x y]);

if b == 0
    d = a;
else
    d = rgcd(b,rem(a,b));
end
```

In the long if-statement at the beginning of the new version of the function, we make sure that the inputs are scalar and not arrays, that they are integers and also positive. This last characteristic is the one that guards against infinite recursion. If we try to run the function with a negative input, we get a different error message, one that we wrote expressly for this function–one that is more informative about the nature of the problem:

```
>> rgcd(12,-32)
Error using rgcd (line 4)
x and y must be positive integers
```

Now our recursive **rgcd** function is ready.

From recursion to iteration

Every recursive function has an equivalent non-recursive version. For simple problems, there is typically not much difference between the difficulty of writing the solution with recursion and the difficulty of writing it without recursion. For more complex problems, a recursive solution is sometimes far simpler than the equivalent non-recursive solution.

Consider, the non-recursive version of GCD below.

```
function d = igcd(x,y)
if (~isscalar(x) || ~isscalar(y) || ...
        x ~= fix(x) || y ~= fix(y) || ...
                            x < 0 || y < 0)
    error('x and y most be positive integers');
end

a = max([x y]);
b = min([x y]);

while b > 0
    d = b;
    b = rem(a,b);
    a = d;
end
```

The function is called **igcd** for "iterative gcd". As you can see, the recursion was replaced by iteration (the while-loop), hence the name. The idea is to follow the recursive definition as before, but instead of calling the function recursively, we simply change the values of the variables **a** and **b** according to the definition. Inside the loop, we use the output argument **d** to store the value of **b** because we need to change **b** and we need to replace the value of **a** with the value of **b**. So, we need to store the value of **b** temporarily. Using **d** for this purpose has the side effect of it containing the result already when we hit the stopping condition, that is, when **b** reaches **0**. At that point, the previous value of **b** is exactly what the GCD is.

In the case of the GCD problem, the recursive version is slightly easier to understand, since it follows the recursive definition to the letter. The iterative version needs a bit of thinking figuring out how to change the variables around.

Sometimes the iterative versions needs a *lot* of thinking, so it is not uncommon for experienced programmers who tackle an inherently recursive problem to write and debug a recursive solution first. Once that version is done, the programmer designs, writes, and debugs an iterative version. That iterative version will always involve one or more while-loops, whereas the recursive version will typically have no loops, and the iterative version will often require more local variables than the recursive version. The iterative version will typically be more difficult to understand than the recursive version as well (for those who understand recursion).

So why not stick with the recursive version, avoid writing two programs to solve one problem, and just move on? The answer is efficiency. A well written iterative solution will almost always be more efficient than a well written recursive version. The reason is the hidden overhead brought on by the need in recursive functions for additional function calls. For each call, that overhead includes (1) the creation of a new frame on the stack for every new instance of the function as a recursive call is made, (2) the copying of values from the local variables of one instance of the function to the input arguments of the new instance of the function, (3) the copying of values from the output arguments of an instance of the function to the local variables of the instance of the function that called it, and (4) the removal of the frame from the stack when the called function returns. This set of four actions takes time, and it can take place many thousands of times for some recursive functions. All those costly actions are eliminated when recursion is replaced by iteration. As with so many problems in programming (and in life), the easy solution is not always the best one, but it is often a very good first one. Before we leave recursion, we will look at a second and third recursive problem, and, as we did for GCD, we will give both recursive and iterative solutions for each of them.

The second one is an example of a problem whose recursive solution requires multiple recursive calling points in the function that solves it. We will perform a detailed time comparison between the recursive solution and the iterative solution for that one.

Factorial

Possibly the most frequently used function to illustrate recursion is the ***factorial***. The factorial, usually written in mathematics as $n!$, is defined as follows:

$$\text{fact}(n) = n! = 1 \times 2 \times 3 \times \ldots \times (n-2) \times (n-1) \times n$$

For example,

$$\text{fact}(5) = 5! = 1 \times 2 \times 3 \times 4 \times 5 = 120$$

Here is the recursive way to define the factorial function:

$$\text{fact}(n) = \begin{cases} 1 & \text{if } n = 1 \\ n \times \text{fact}(n-1) & \text{if } n > 1 \end{cases}$$

It is pretty straightforward to understand why these definitions are equivalent. The corresponding MATLAB code is also pretty simple:

```
function f = fact(n)
if ( ~isscalar(n) || n < 1 || n ~= fix(n))
    error('n must be a positive integer!');
end

if n == 1
    f = 1;
else
    f = n * fact(n-1);
end
```

The first if-statement makes sure that **n** is a positive integer. The second if-statement checks for the base case (when **n** equals 1), and if it is true, it returns 1. Otherwise, we call the function again, but using **(n-1)** as the argu-

ment this time. This makes sure that we are getting closer and closer to the base case and will eventually stop the recursion. The return value of the recursive call of **`fact`** is simply multiplied by **`n`** to get the correct result.

The key to this recursive function is again that each call to **`fact`** gets a new stack frame where the new instance of the function stores its variables, **`n`** and **`f`**. Therefore, the multiple active calls to the function do not interfere with each other at all.

While the factorial is frequently used to illustrate recursive functions, its iterative version, like that for the greatest common divisor, is equally simple:

```
function f = ifact(n)
if ( ~isscalar(n) || n < 1 || n ~= fix(n))
    error('n must be a positive integer!');
end

f = 1;
for ii = 2:n
    f = f * ii;
end
```

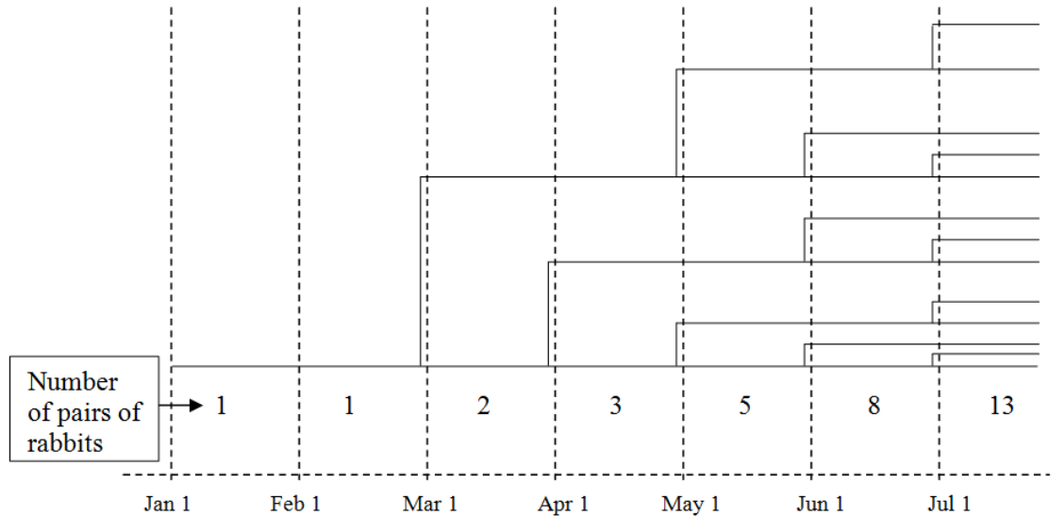
Multiple recursive calls

There is no reason why we could not make multiple recursive calls inside a function. Let's look at a famous example: *The Fibonacci Series*.

The Fibonacci Series is a famous series that can be generated by a function that makes two recursive calls per activation. The series is named after the 13th century mathematician who invented it. Fibonacci posed this problem:

Suppose a pair of rabbits, one male and one female, is born on January 1. Suppose further that this pair of rabbits gives birth to a new male and female on the last day of their second month of life, their third month of life, their fourth month, etc., forever, and that rabbits never die (mathematicians do not care about reality). Finally, suppose that all new pairs of rabbits do the same. How many pairs of rabbits would be alive on the first day of the Nth month (and who but a mathematician would care)?

Figure 2.62 *Fibonacci and rabbit reproduction*



To help us understand the problem, let's consider the first few months. Figure 2.62 is a schematic representation of the development of the rabbit population from January 1 through July 31. Each solid horizontal line is the life-line of a pair of rabbits, with the first pair having the longest line. Each vertical line represents the birth of a new pair.

The number of pairs of rabbits alive at the first day of a month is equal to the number of solid horizontal lines that intersect the vertical dotted line for that day. By counting the intersections we can see the following progression in the number of pairs:

```
1,  1,  2,  3,  5,  8,  13
```

These numbers are a consequence of the rules Fibonacci gave for the births. To see how these rules produce these numbers, we can look the changes, month by month:

- **January 1 and February 1**: The first two months are simple. Since the first pair requires two months to produce offspring, there is one pair on January 1 and still just one pair on February 1.

- **March 1**: On the last day of February, the first pair produces its first offspring, so there are two pairs on March 1.
- **April 1**: April is the first interesting month. It is interesting because some rabbits reproduce and some do not. Only the rabbits that are at least two months old reproduce, so only the pairs that were alive on February 1 will produce a new pair. There was only one pair alive then, so one new pair is added to the number alive March 1 to bring the total to three. Thus, the number of pairs on April 1 is equal to the number of pairs on February 1 plus the number of pairs on March 1.
- **May 1**: May is interesting as well for the same reason. Again only those rabbits that are at least two months old will reproduce, so only the pairs that were alive on March 1 will produce a new pair. There were two pairs alive then, so two new pairs are added to the number that were alive on April 1. Thus, the number of pairs on May 1 is equal to the number of pairs on March 1 plus the number of pairs on April 1.

By now, we can see a pattern: To determine the number of pairs alive at the beginning of month n, we add the number of pairs alive at the beginning of month $n-2$ to the number of pairs alive at the beginning of month $n-1$. Letting $F(n)$ stand for the number of pairs of rabbits alive on the first day of the n^{th} month, we have

$$F(n) = F(n-2) + F(n-1)$$

The corresponding recursive definition is

$$F(n) = \begin{cases} 1 & \text{if } n \leq 2 \\ F(n-2) + F(n-1) & \text{if } n > 2 \end{cases}$$

Here is a recursive MATLAB function that implements this definition:

```
function f = fibo(n)
if ( ~isscalar(n) || n < 1 || n ~= fix(n))
    error('n must be a positive integer!');
end

if n <= 2
    f = 1;
else
    f = fibo(n-2) + fibo(n-1);
end
```

Again, we first make sure that the input to our function is legal. The base case handles both **n** equal to 1 and **n** equal to 2. The recursive case is used whenever **n** is greater than 2. We simply add together the results of calling the function with arguments 2 and 1 less than **n**, respectively.

Let's test the function:

```
>> fibo(5)
ans =
     5
>> fibo(6)
ans =
     8
>> fibo(7)
ans =
    13
>> fibo(8)
ans =
    21
```

Everything works as expected. While the results are correct, the function is kind of wasteful. Consider the recursive case:

```
f = fibo(n-2) + fibo(n-1);
```

Let's say that **n** is equal to 8 at this point. We first compute **fibo(n-2)**, that is, **fibo(6)**, and then **fibo(n-1)**, that is **fibo(7)**. But computing **fibo(7)**, will cause computing **fibo(6)** again in this case because we call **fibo(n-1)** when **n** is equal to 7. So, we compute **fibo(6)** twice. But com-

puting **fibo(6)**, results in computing **fibo(4)** twice also. That means that **fibo(4)**, in turn, is computed 4 times! In fact, **fibo(7)** calls **fibo(5)** once, which will also call **fibo(4)**. So computing **fibo(8)** results in calling **fibo(4)** five times. And this goes on for almost every element until we reach the base case. Effectively, we are computing almost all of the elements of the series many times. How many times exactly? Here are the numbers for **fibo(8)**:

```
fibo(8):  1
fibo(7):  1
fibo(6):  2
fibo(5):  3
fibo(4):  5
fibo(3):  8
fibo(2): 13
```

These numbers should look familiar. Indeed, the number of times the function is called with the same argument form a Fibonacci series itself. The reason for that is left as an exercise for the reader. (Don't you just hate it when authors do that?) Note that we left calling **fibo(1)** out of the list because **fibo(2)** is one of the base cases already, and it does not call **fibo(1)**.

As you can see, our recursive function is very inefficient; it carries out a lot of unnecessary computation. The iterative version of the function does not do that:

```
function f = ifibo(n)
if ( ~isscalar(n) || n < 1 || n ~= fix(n))
    error('n must be a positive integer!');
end

fv = ones(1,n);
for ii = 3:n
    fv(ii) = fv(ii-2) + fv(ii-1);
end
f = fv(n);
```

It turns out that it is easier to compute the entire series in a vector and return the last element. The variable **fv** is used for this purpose. Initializing it with all ones takes care of the first two elements, so it is enough to start the loop at 3. The new element is computed by adding together the previous two elements at every iteration step. Finally, we simply assign the n^{th} element of **fv** to the output argument **f**.

We can use the same idea for a different implementation of the recursive version of the Fibonacci function. If we modify the requirement for the function such that, instead of returning the n^{th} element of the series, it returns the first n elements of the series, we can create a more efficient implementation:

```
function fv = fiboseries(n)
if ( ~isscalar(n) || n < 1 || n ~= fix(n))
    error('n must be a positive integer!');
end

if n <= 2
    fv = ones(1,n);
else
    fv = fiboseries(n-1);
    fv = [fv  fv(n-2)+fv(n-1)];
end
```

The base case simply creates a vector **fv** of either one or two ones. The recursive case simply calls itself with **n-1** as the input argument. Then we simply append a new element at the end of the vector **fv** that has the value of the sum of the last two elements of the current vector. Here is how it works:

```
>> fiboseries(1)
ans =
     1
>> fiboseries(2)
ans =
     1     1
>> fiboseries(3)
ans =
     1     1     2
```

```
>> fiboseries(4)
ans =
     1     1     2     3
>> fiboseries(5)
ans =
     1     1     2     3     5
>> fiboseries(6)
ans =
     1     1     2     3     5     8
```

On the other hand, if we do not want to modify the requirement and we need a function that returns only the n^{th} element of the Fibonacci series, we can still use the more efficient implementation by using the **fiboseries** function as a subfunction like the following:

```
function f = fibo2(n)
    if ( ~isscalar(n) || n < 1 || n ~= fix(n))
        error('n must be a positive integer!');
    end

    fv = fiboseries(n);
    f = fv(n);

function fv = fiboseries(n)

    if n <= 2
        fv = ones(1,n);
    else
        fv = fiboseries(n-1);
        fv = [fv fv(n-2) + fv(n-1)];
    end
```

Here the main function **fibo2** handles the checking of the input argument and then simply calls the subfunction **fiboseries**. The result is then the n^{th} element of the series returned by **fiboseries**. The subfunction does not even have to do the input argument checking, since we can only call it from **fibo2** and it already made sure that the input is correct. The function **fiboseries** computes the first **n** elements of the series in a recursive manner and passes it back to **fibo2**.

Let's run this:

```
>> fibo2(1)
ans =
     1
>> fibo2(2)
ans =
     1
>> fibo2(3)
ans =
     2
>> fibo2(4)
ans =
     3
>> fibo2(5)
ans =
     5
>> fibo2(6)
ans =
     8
```

It works like a charm. Now let's compare how long it takes to run the various versions of the Fibonacci function. (Note the use of the built-in functions **tic** and **toc**: **tic** starts/resets a clock, **toc** prints the elapsed time since the last call to **tic**.)

```
>> tic; fibo(10); toc
Elapsed time is 0.000999 seconds.
>> tic; fibo2(10); toc
Elapsed time is 0.000235 seconds.
>> tic; ifibo(10); toc
Elapsed time is 0.000076 seconds.

>> tic; fibo(20); toc
Elapsed time is 0.069333 seconds.
>> tic; fibo2(20); toc
Elapsed time is 0.000409 seconds.
>> tic; ifibo(20); toc
Elapsed time is 0.000089 seconds.

>> tic; fibo(30); toc
Elapsed time is 5.898723 seconds.
>> tic; fibo2(30); toc
Elapsed time is 0.000576 seconds.
>> tic; ifibo(30); toc
Elapsed time is 0.000077 seconds.
```

```
>> tic; fibo(40); toc
Elapsed time is 731.346860 seconds.
>> tic; fibo2(40); toc
Elapsed time is 0.000757 seconds.
>> tic; ifibo(40); toc
Elapsed time is 0.000075 seconds.
```

As you can see, the original recursive version takes exponentially longer time as **n** increases. It goes from one thousandth of a second to more than ***12 minutes*** as **n** goes from 10 to 40. The time required by the more efficient recursive implementation increases approximately linearly and it remains less than a thousand of a second even for **n** equal to 40. As always, the fastest is the iterative version. In fact, the running time hardly changes as the argument increases. This is because it is so fast computing the series that other factors dominate the time such as calling the function and allocating the memory for the vector. The actual computation hardly takes any time in this case.

With this last example completed we now leave recursion. Hungry for more? Well, you will be happy to know that there is more coming. It is coming in the next chapter, entitled Advanced Concepts. There we will encounter recursion again when we consider the problems of finding items in a database and of putting a database in order in the section entitled, Searching and Sorting.

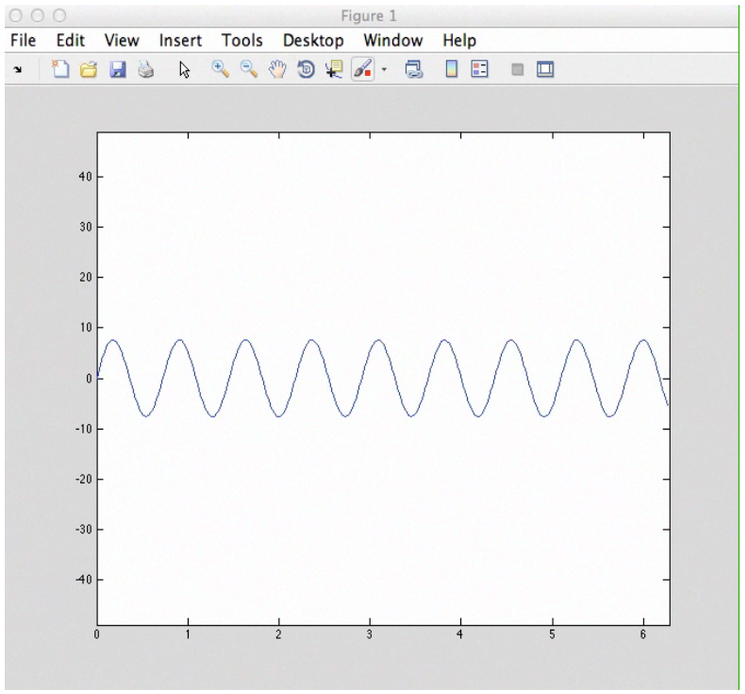
Animation

Previously we have seen how to create plots with MATLAB. It is also easy to create animated plots. The basic idea is to plot repeatedly in the same figure. The most important thing to remember is that we have to tell MATLAB when to update the display using the **drawnow** function. The reason behind this is that todays' computers can process information much faster than it can be displayed on the screen. So, as we tell repeatedly MATLAB to plot something very fast, it carries out all the instructions, but it only refreshes the screen occasionally. The function **drawnow** tells MATLAB to update the screen immediately. Consider the following example:

```
function anim(fmax)
t = 0:0.03: 2 * pi;
step = (fmax-1)/1000;
for ii = [(1:step:fmax) (fmax:-step:1)]
    ampl = ii - 1;
    v = ampl * sin(t * ii);
    plot(t,v);
    axis([0 2*pi -fmax+1 fmax-1]);
    drawnow;
    pause(0.005)
end
```

The function **anim** repeatedly generates a sine wave with increasing frequency from 1 to **fmax** and then back down to 1. The amplitude of the sine wave also increases as the frequency increases. The loop index **ii** goes from **1** to **fmax** and then back to **1**. In each iteration of the for-loop, a new sine wave is generated and plotted. Then **drawnow** is called to make sure that the

Movie 2.2 *Sine wave with continuously changing frequency*



new plot appears on the screen immediately. Finally, we make MATLAB wait a fraction of a second to slow down the animation using **pause**. On slower computers, this may not be necessary. The result can be seen in Movie 2.2.

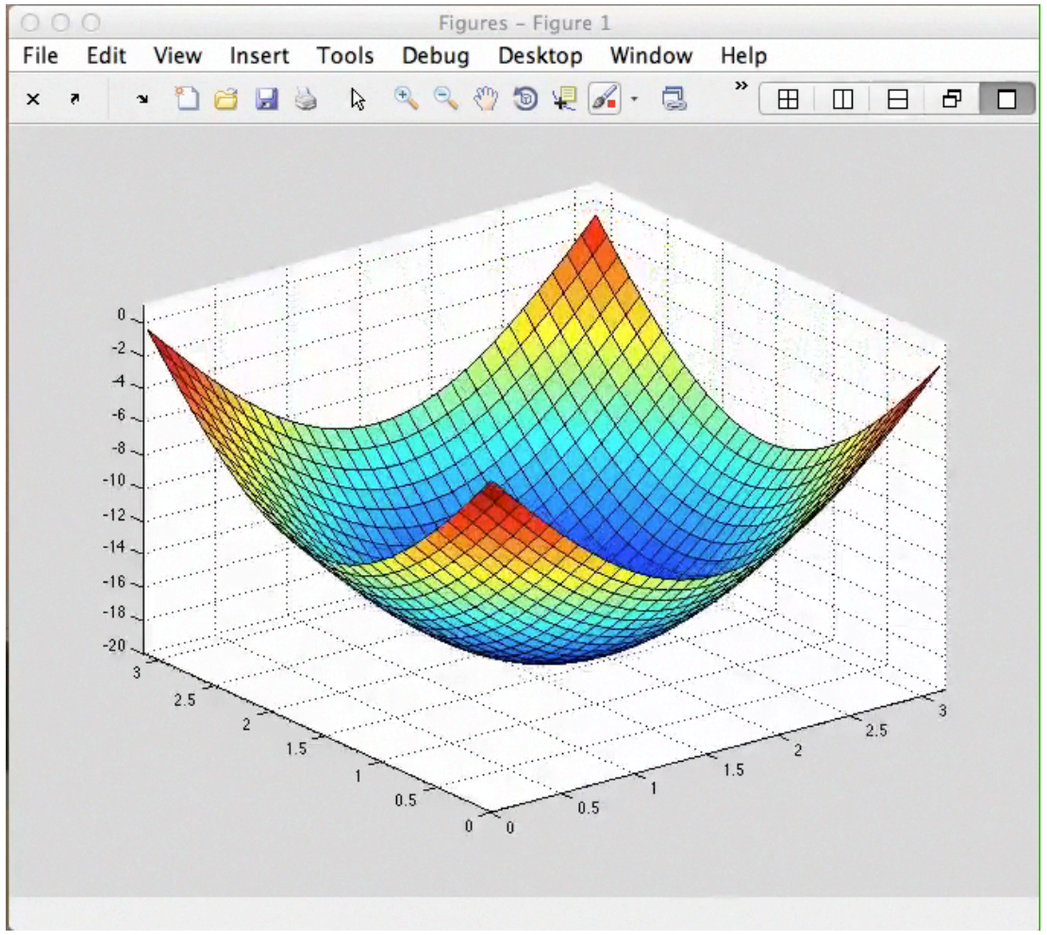
MATLAB also supports drawing surface plots with the built-in function called **surf**. Similarly to **plot**, **surf** can be called a number of different ways. The usual method is to call it with three matrices of the same size specifying the *x*, *y* and *z* coordinates of the points of the service surface. The following example uses this method also. Additionally, it animates the plot in a couple different ways:

```
function surf_anim

t = 0:0.1:pi;
[X, Y] = meshgrid(t);
s = 0.2;
for kk = [0:-s:-10  -10+s:s:0-s  0-s:-s:-10]
    surf(X,Y,kk *(sin(X) + sin(Y)));
    axis([0,pi,0,pi,-20,1]);
    drawnow;
end

for kk = -37.5:30
    view(kk,30);
    drawnow;
end
for kk = 30:-1:5
    view(30,kk);
    drawnow;
end
```

The function creates a surface that is the sum of a half period of a sine wave in the *x* dimension and a half period of a sine wave in the *y* dimension. The **meshgrid** built-in function creates a grid from the **t** vector and stores the *x* and *y* coordinates in the **X** and **Y** matrices to be used by **surf**. (Type **help meshgrid** to learn more.) The first for-loop, using index **kk**, animates the plot by changing the amplitude of the surface in each step. It goes from 0 to -10, back to 0 and to -10 again. Notice that we call **drawnow** after **surf** again.

The two for-loops at the end of the function do another kind of animation. They change the viewpoint from which we view the picture by calling the built-in **view** function. The default angles are -37.5 and 30 degrees of azimuth and elevation, respectively. In the first loop, we change the azimuth angle from the default to positive 30 degrees. Then we we change the elevation angle from 30 down to 5 by one degree at a time. Movie 2.3 shows what the result looks like.

Movie 2.3 *Surface plot animation*



Notice how MATLAB puts a grid on the surface that corresponds to the resolution of the data we provided. If you look at the **surf_anim** function again you can see that we had only 32 elements in the **x** and **y** vectors each. If we want a higher resolution, then the grid on the surface may cover the colors completely. Fortunately, you can turn off the grid by calling surf the following way:

```
surf(t,t,s,'EdgeColor','none');
```

Once we eliminate the grid, we can increase the resolution as well. See, how we change the vector **t** below:

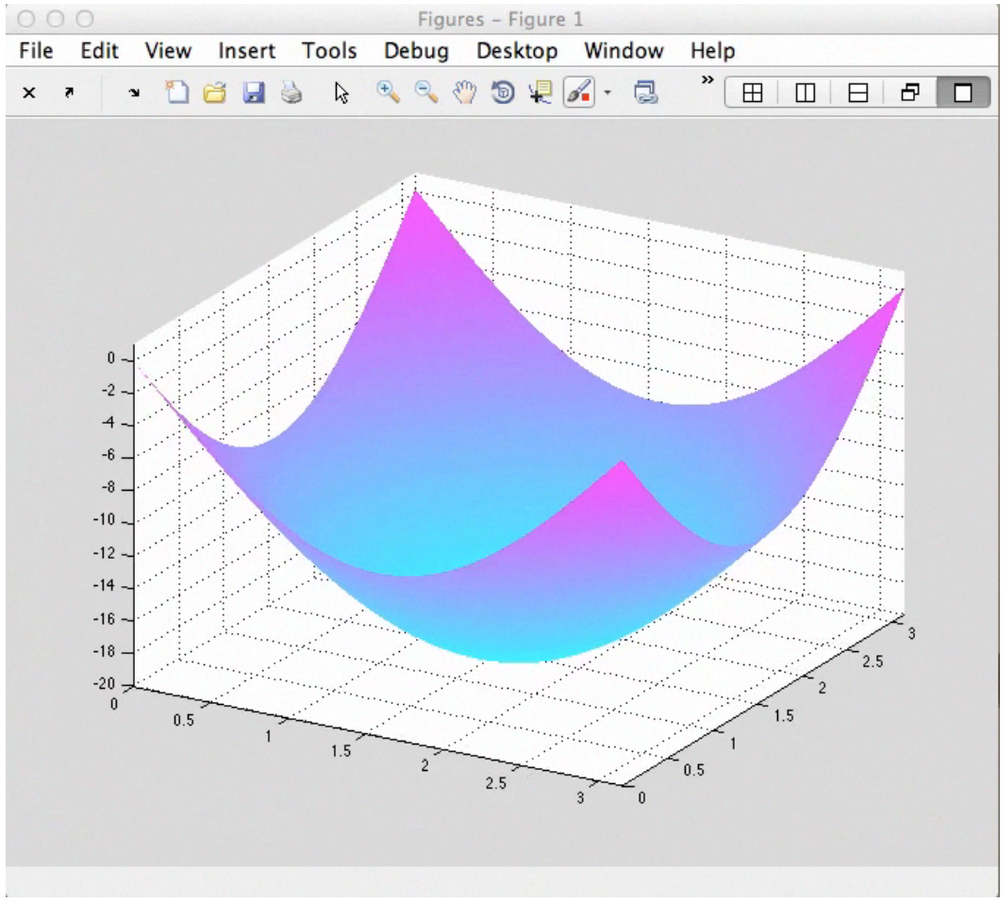
```
function surf_anim

t = 0:0.01:pi;
[X, Y] = meshgrid(t);
s = 0.2;
for kk = [0:-s:-10  -10+s:s:0-s  0-s:-s:-10]
    surf(t,t,s,'EdgeColor','none');
    axis([0,pi,0,pi,-20,1]);
    drawnow;
end
for kk = -37.5:30
    view(kk,30);
    drawnow;
end
for kk = 30:-1:5
    view(30,kk);
    drawnow;
end
```

You can also change what colors MATLAB uses to display the plot. The **colormap** function has a number of standard color palettes. For example, we can call it like this:

```
colormap('cool');
```

Movie 2.4 shows the result:

Movie 2.4 *High resolution surface plot with no grid lines*



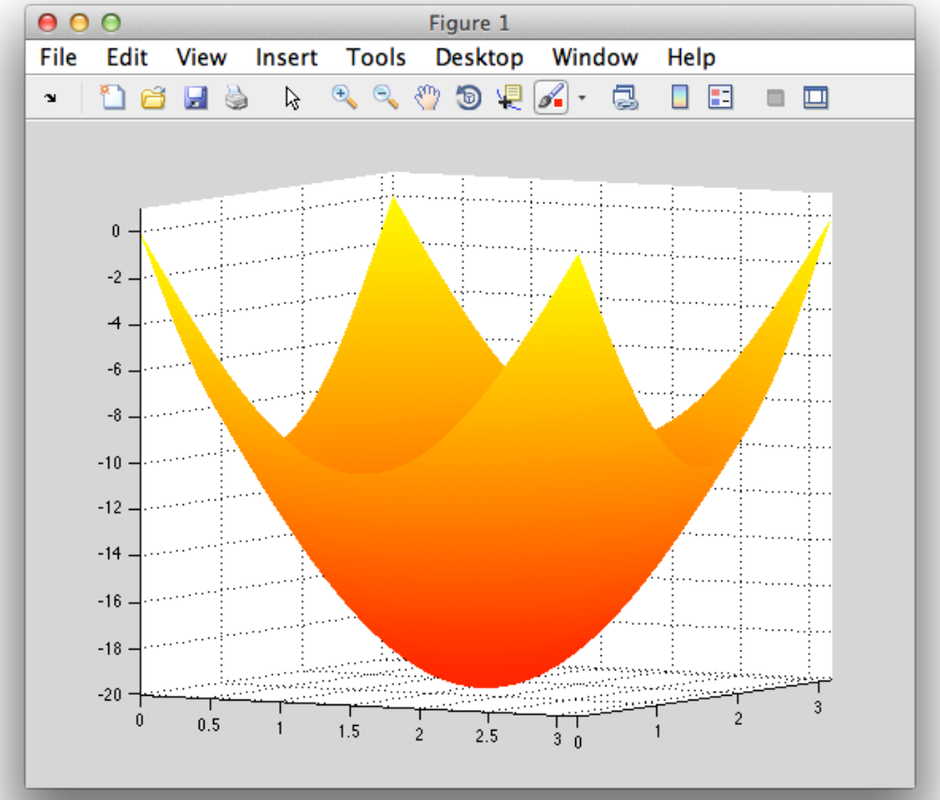
The **colormap** function can also be called with a matrix that has three columns. Each row of the matrix specifies one color. The columns correspond to the red, green and blue components, respectively. Use the **doc colormap** command in the Command Window of MATLAB for more information.

The **colormap** function can be called to change the colors on an existing figure as well. For example, once the previous animation ends, type the following command in the Command Window:

```
>> colormap('autumn')
```

Figure 2.63 shows what you will see.

Figure 2.63 *Surface plot using the autumn color map.*



Concepts From This Section

Computer Science and Mathematics:

stack
stack frame
recursive definition
base case
recursive case
recursive function
recursive function call
recursion
animation

MATLAB:

nargin
nargout
varargin
varargout
surf
colormap

Practice Problems

Variable number of arguments

Problem 1. Write a function called **`arithmetic`** that takes two scalar input arguments (the function does not have to check the format of the input) and returns up to four output arguments. The outputs are the sum, difference, product and ratio of the two inputs, respectively. If the second input argument is not provided, it is set to equal the first. Here are a few example runs:

```
>> arithmetic(3,2)

ans =
     5

>> [a b c] = arithmetic(3,2)

a =
     5
b =
     1
c =
     6

>> [a b c d] = arithmetic(3)

a =
     6
b =
     0
c =
     9
d =
     1
```

?

Problem 2. Write a function called **`random_test`** that takes three input arguments and three output arguments. The function needs to demonstrate that the built-in function **`randi`** and **`rand`** use the same random number generator. The function generates two arrays of equal size that contains random integers. The arrays must be identical, but the first must be generated using **`randi`** and the second using **`rand`**. The input arguments to **`arithmetic`** must behave like those of **`randi`**. That is, the first argument determines the range of the desired random integers. If it is scalar, then it specifies the upper limit (maximum) and in this case, the lower limit (minimum) is 1. If it is a vector, its first element specifies the lower limit, the second element sets the upper limit. The second input argument is the number of rows of the output arrays, while the third is the number of columns. If the third argument is missing, it is assumed to be the same as the second. If the second argument is missing, it is set to 1. The first argument must be provided by the caller. The first and second output argument are the two identical arrays. The third output argument is a logical value: true if the two arrays are really identical and false otherwise. (Note that this needs to be a single logical value and not an array.) The second and third output arguments are optional. Here is an example run:

```
>>> [R1 R2 c] = random_test([2 5], 3, 8)

R1 =
     3     3     2     4     2     4     2     5
     4     2     3     3     5     3     2     3
     2     2     3     4     2     4     5     4
R2 =
     3     3     2     4     2     4     2     5
     4     2     3     3     5     3     2     3
     2     2     3     4     2     4     5     4
c =
     1
```

Problem 3. Write a function called **`add_all`** that takes any number of scalar input arguments (the function does not need to check the format of the input) and returns the sum of all its inputs. Here are a couple of runs:

```
>> add_all(1, 2)
ans =
     3
>> add_all(1, 2, 3, 4, 5, 6, 7, 8, 9, 10)
ans =
    55
```

?

Problem 4. Write a function called **`print_all`** that takes any number of scalar input arguments (the function does not need to check the format of the input) and prints them out one by one as illustrated by a few runs:

```
>> print_all()
We received no numbers.
>> print_all(34)
We received the following number: 34.
>> print_all(34, 56, 78, 90)
We received the following numbers: 34, 56, 78, and 90.
```

Make sure that the function handles the three cases (no input, one input, more than one input) correctly, as illustrated above. The function has no output arguments.

Recursion

Problem 5. Write a recursive function called **`vectflip`** that takes a single vector input argument (the function does not need to check the format of the input) and returns a single vector input argument that contains the elements of the input vector in reverse order. Note that the MATLAB command **`v(end:-1:1)`** does exactly what is required, but you need to write a recursive function instead that flips the first element with the result of calling **`vect_flip`** with the original input vector less its first element. The function must not use a loop.

?

Problem 6. Write a recursive function called **`vectflip2`** that does exactly the same as **`vect_flip`** from the previous problem except that it flips the last element with the result of calling **`vect_flip`** with the original input vector less its last element. The function must not use a loop.

Problem 7. Write a recursive function **`digit_sum`** that takes a single scalar positive integer input argument (the function does not need to check the format of the input) and returns the sum of its digits. Here is a sample_run:

```
>> digit_sum(1234)

ans =
    10
```

The function must not use a loop.

?

Problem 8. Write a recursive function called **`flip`** that takes one scalar positive integer input argument (the function does not need to check the format of the input) and returns a single scalar output that is the input with its digits in the reverse order. For example, consider this run:

```
>> flip(1234)

ans =
        4321
```

Note that it is allowed to write a subfunction that is the actual recursive function as opposed to **`flip`** itself. It may take more input arguments also.

Problem 9. Write a recursive function called **`prime_fact`** that takes a single scalar positive integer input argument (the function does not need to check the format of the input) and returns a vector whose element are the prime factors of the input. Each call to **`prime_fact`** should compute a single prime factor and call **`prime_fact`** again. Here is an sample run:

```
>> prime_fact(60)

ans =
     2     2     3     5
```

?

Problem 10. Write a recursive function called **`connected`** that takes a square matrix called **`G`** as an input argument and returns a single logical output argument. **`G`** represents a graph, a mathematical construct consisting of nodes and edges between the nodes. You can think of it as a representation of a map where the nodes are cities and the edges are roads connecting cities directly together. Each row (and column) of **`G`** represents a city, its name is the index of the row (column). If there is a direct road from city **`N`** to city **`M`**, **`G(M,N)`** and **`G(N,M)`** are both **`1`**, otherwise, they are both **`0`**. Think of it as the roads being two-way: if there is a road from **`M`** to **`N`**, then there is a road from **`N`** to **`M`** too. The diagonal of **`G`** is **`0`** (there is no road from a city to itself). (The function does not need to check that the matrix **`G`** is indeed correct.) The function **`connected`** needs to decide whether we can get from any city to any other city. For example, if there are four cities and there is a road from **`1`** to **`2`**, from **`2`** to **`3`** and from **`3`** to **`4`**, then we can get from any city to any other city. However, if there is a road from **`1`** to **`2`**, and from **`3`** to **`4`**, then we cannot visit **`3`** or **`4`** from either **`1`** or **`2`** and vice versa. The name of the function comes from the mathematical term connected graph. The function needs to return 1, meaning true, if the graph is connected (there is a way to get from any city to any other) and 0, meaning false, otherwise. Note that it is allowed to write a subfunction that is the actual recursive function as opposed to **`connected`** itself. It may take more input arguments also.

Problem 11. There is a narrow bridge across a river. Four injured soldiers are trying to cross in pitch dark. They have a single flashlight and they need it not to fall in the river below. At most two soldiers can cross at the same time. Due to their injuries, they need a different amount of time to make it. The first soldier needs 1 minute, the rest 2, 5 and 10 minutes, respectively. When two of them cross together, they progress at the speed of the slower soldier. The bridge will be blown up in 18 minutes. Can they make it?

Write a recursive function called **`bridge`** that takes a single vector input argument whose elements are positive integers (you do not need to check the input format). The elements represent the time each soldier needs to cross the bridge. For example, using the numbers provided above, we would call the function like this: **`bridge([1 2 5 10])`**. The function needs to work for any number of soldiers and all possible crossing times. The function returns a single scalar output argument that is equal to the minimum possible cumulative time the group of soldiers need to cross the bridge.

Note that it is allowed to write a subfunction that is the actual recursive function as opposed to **`bridge`** itself. It may take more input arguments also. This is a difficult problem.

?

Problem 12. Write a recursive function called **`bridge2`**. It does the same thing as the function **`bridge`** in the previous problem, but it also returns the strategy the soldiers need to follow to achieve the minimum time. Figure out a suitable format for this second output argument.

Note that it is allowed to write a subfunction that is the actual recursive function as opposed to **`bridge2`** itself. It may take more input arguments also. This is an even more difficult problem than the previous one.