CHAPTER 3

Advanced Concepts

Photo credit: NASA.

If you have gotten this far, then congratulations are in order. You know the basic concepts of computer programming, and you have everything you need to write useful MATLAB programs. You are ready to launch a career in MATLAB programming! This chapter builds on this foundation to introduce advanced concepts that can take you even further.

Linear Algebra

Objectives

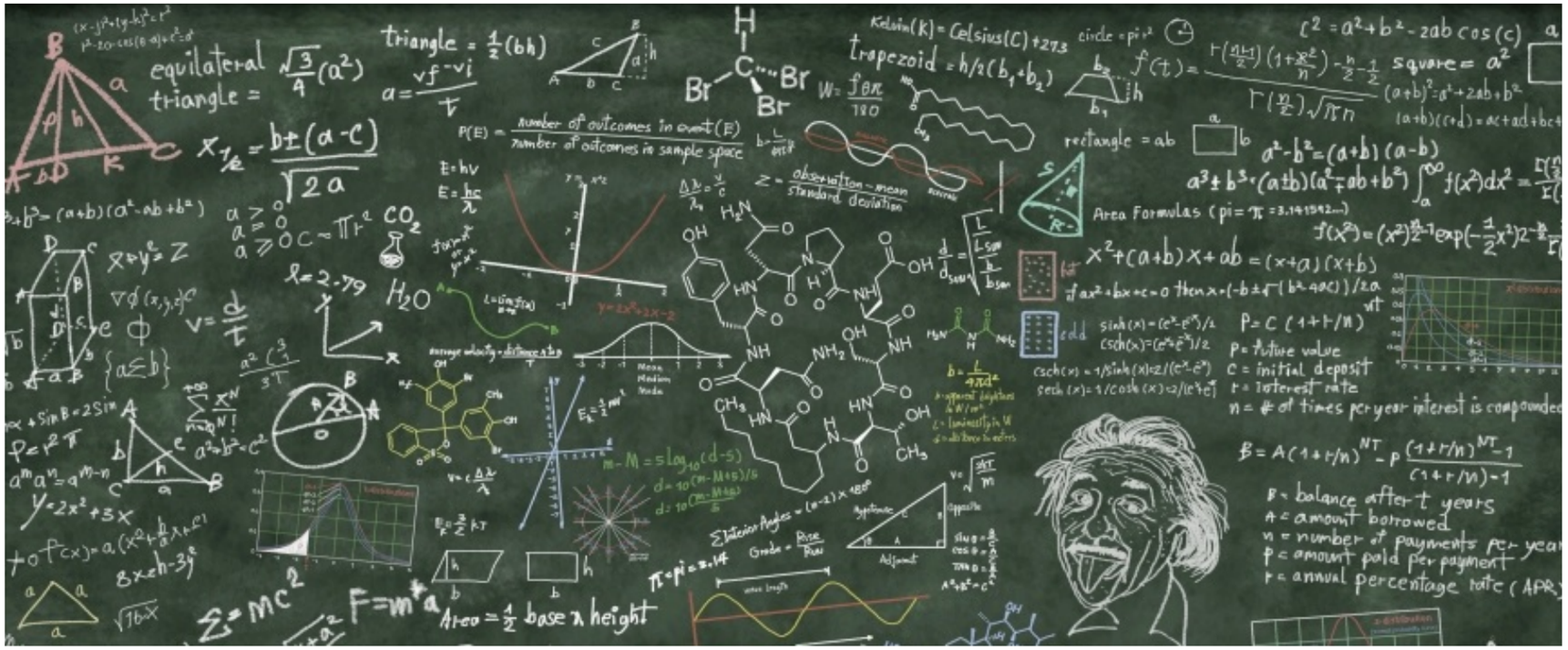
This section deals with concepts in both mathematics and MATLAB.

(1) We will learn how to use MATLAB's backslash operator (\) to solve in one simple command an entire set of linear algebraic equations.

(2) We will consider inconsistent equations, underdetermined equations, and overdetermined equations.

(3) We will study the errors encountered in solving these equations.

(4) We will introduce the concept of ill-conditioned equations and will show how to use MATLAB's **cond** command to detect them.



Problems involving multiple equations in multiple unknowns may seem overwhelming, but in many cases they can be solved easily using MATLAB's operators and built-in functions.

Mathematical problems involving equations of the form $Ax=b$, where A is a matrix and x and b are both column vectors are problems in **linear algebra**, sometimes called **matrix algebra**. Engineering and scientific disciplines are rife with such problems. Thus, an understanding of their set-up and solution is a crucial part of the education of many engineers and scientists. As we will see in this brief introduction, MATLAB provides a powerful and convenient means for finding solutions to these problems, but, as we will also see, the solution is not always what it appears to be.

Solution Difficulties

The most common problems of importance are those in which the unknowns are the elements of x or the elements of b. The latter problems are the simpler ones. The former are fraught with many difficulties. In some cases, the "solution" x may in fact not be a solution at all! This situation surfaces when there is conflicting information hidden in the inputs (i.e., A and b), and the non-solution obtained is a necessary compromise among them. That compromise may be a perfectly acceptable result, even though it is not in fact a solution to the equations. In other cases, MATLAB may give a solution, but, because of small errors in A or b, it may be completely different from the solution that would be obtained were A and b exact. This situation can have a serious impact on the application at hand. The elements of A and b are typically measurements made in the field, and even though much time, effort, and expense go into reducing the error of those measurements, the calculated x may have an error that is so large as to render those measurements useless. This situation occurs when the solution x is highly sensitive to small changes in A and b. This sensitivity issue is just as important, or in many cases more important, than the problem of the non-solution.

Neither of these problems is MATLAB's "fault", nor would it be the fault of any other automatic solver that you might employ. It is the nature of linear algebra. The person who relies on a solution to $Ax = b$ provided by MATLAB, or by any other solver, without investigating the sensitivity of the solution to errors in the input, is asking for trouble, and will often get it. When accuracy is critical (and it usually is), the reliability of the solution must be investigated, and that investigation requires some understanding of linear algebra. The study of linear algebra can be life-long, and there are many books and courses, both undergraduate and graduate, on the subject. While it is not possible for every engineer and scientist to master the subject, it is possible for them to be aware of the nature of the difficulties and to learn how to recognize them in their discipline. The goals of this section are to give you some insight into the nature of the difficulties associated with the solution of $Ax=b$ when x is the unknown and to show you how to use MATLAB to find the solution and to investigate its reliability.

Simultaneous Equations

Because of the definition of matrix multiplication, the matrix equation $Ax = b$ corresponds to a set of **simultaneous linear algebraic equations**:

$$\begin{array}{ccccccccc} A_{11}x_1 & + & A_{12}x_2 & + & \ldots & + & A_{1N}x_N & = & b_1 \\ A_{21}x_1 & + & A_{22}x_2 & + & \ldots & + & A_{2N}x_N & = & b_2 \\ \vdots & & \vdots & & & & \vdots & & \vdots \\ A_{M1}x_1 & + & A_{M2}x_2 & + & \ldots & + & A_{MN}x_N & = & b_M \end{array} ,$$

where the A_{ij} are the elements of the M-by-N matrix A, the x_j are the elements of the N-by-1 column vector x, and the b_i are the elements of the M-by-1 column vector b. This set of equations is sometimes written in tableau form to indicate more clearly the relationship between the matrix form and the equation forms and to show the shapes of the arrays:

$$\begin{bmatrix} A_{11} & A_{12} & \cdots & A_{1N} \\ A_{21} & A_{22} & \cdots & A_{2N} \\ & \vdots & & \\ & \vdots & & \\ & \vdots & & \\ A_{M1} & A_{M2} & \cdots & A_{MN} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_N \end{bmatrix} = \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ \vdots \\ \vdots \\ b_{\mathrm{M}} \end{bmatrix} .$$

Note that, despite the row-like arrangement of the x_j in the non-tableau format, x is in fact a column vector, as made clear in the tableau form. Note also that the length of x is N, which, as we have learned from the rules of matrix multiplication, must equal the width (not the height) of A. The length of b is

M, which is always equal to the height of A. We have formed this tableau to indicate that in this case, there are more equations than unknowns. In other words, $M > N$. While it is more common to have $M = N$, in which case the height of the x vector is the same as those of A and b, in many situations, M is in fact larger than N. The third situation, in which $N < M$, is far less common.

Typically, the A_{ij} and b_i are given, and the x_j are unknown. The equations are "simultaneous" because one set of x_j must satisfy all M equations simultaneously. They are "linear" because none of the x_j is raised to any power other than 1 (or 0) and there are no squares or products $x_{j1}x_{j2}$ of unknowns. When A and x are given, the problem is relatively easy, because finding b requires only matrix multiplication. The number of unknowns (i.e., the length of b) is always equal to the number of equations (otherwise Ax cannot equal b), and typically the elements of b are not highly sensitive to small errors in A or x. More often, however, the problem confronting us is to find x when A and b are given. This problem is much more difficult and is the primary subject of this section of the book.

MATLAB's Solutions

Most programming languages (e.g., C, C++, Fortran, Java) provide no commands at all for solving either of these linear algebra problems. Instead, the user of the languages must find a function written by other programmers or write his own functions to solve the problem. Writing a function to find x is in fact no simple matter, and the likelihood that a programmer unskilled in linear algebra will produce a reliable one is negligible. By contrast, MATLAB provides one simple operation to solve each of the two problems: When `A` and `x` are given, the MATLAB solution is `b = A*x;` when `A` and `b` are given, the MATLAB solution is `x = A\b`. There are amazingly few restrictions on `A`, `x`, and `b`. As explained previously, in order for the matrix multiplication `A*x` to make sense, the number of columns of `A` must equal the number rows of `x`. There is different rule for matrix division: in order for the division `A\b` to make sense, the number of rows of `A` must equal the number of rows of `b`. That's it. That's the end of rules for `*` and `\` in MATLAB!

Suppose, for example, we need to solve the equations:

$$4x_1 + 5x_2 = 6$$
$$3x_1 - 2x_2 = 14$$

Here we have $Ax = b$, where the matrix A and the vector b are as follows:

$$A = \begin{bmatrix} 4 & 5 \\ 3 & -2 \end{bmatrix}, \quad b = \begin{bmatrix} 6 \\ 14 \end{bmatrix},$$

and we are solving for x. Here $M = 2$ and $N = 2$. We can set up this problem and solve it in MATLAB as follows:

```
>> A = [4 5; 3 -2]

A =
     4     5
     3    -2

>> b = [6; 14]

b =
     6
    14

>> x = A\b

x =
    3.5652
   -1.6522
```

Note that, as always, the number of rows of **A** is equal to the number of rows of **b**. (In this example, the number of columns of **A** is also equal to the number of rows of **b**, but that equality is not necessary).

We can check the result to make sure that it solves the equation:

```
>> A*x

ans =
     6
    14
```

Since the elements are the same as those of the vector **b**, we can see that the equations have both been solved simultaneously by the two values $x_1 = 3.5652$ and $x_2 = -1.6522$. Another way to check is to look at the difference between **A*x** and **b**:

```
>> A*x - b

ans =
     0
     0
```

Since the difference between each element of **A*x** and the corresponding element of **b** is zero, we see again that MATLAB has given us the solution.

As we mentioned at the beginning of this section, we will soon see that there are some situations in which the result given by **A\b** does not solve the equations or, while it solves the equations, the solution is not trustworthy. In those cases, MATLAB may warn you, but not always. Such situations can cause serious trouble, if the user does not know what is going on. The trouble is not the fault of MATLAB. It is the fault of the user who is solving problems in linear algebra without knowing enough about linear algebra, or perhaps the fault of the instructor who failed to teach the user enough about linear algebra! We will learn below when this will happen and why it happens, and we will learn the meaning of the result that MATLAB gives us in these situations.

In the problem above, there were two equations in two unknowns. Because there were two equations, there were also two rows in both **A** and **b**. Because there were two unknowns, the number of columns of **A** was also 2 and the resultant answer **x** had a length of two. Problems such as this one, in which there are N equations and also N unknowns, are the most common linear algebraic problems that occur in all of engineering and science. They are solved in MATLAB by setting up an **N**-by-**N** matrix **A**, and a column vector **b** of length **N**.

Inconsistent Equations

In most cases for which $M = N$, a solution exists, but occasionally there is no solution. The simplest example is given by two equations in which the left sides are the same but the right sides are different, as for example,

$$4x_1 + 5x_2 = 6$$
$$4x_1 + 5x_2 = 12$$

Clearly this pair of equations is **inconsistent**. A plot of the lines represented by each of the equations will reveal that they are parallel. Thus, they never intersect, or, to put it another way, they intersect at infinity. If we try to solve this pair of equations using MATLAB, it will issue a warning that the matrix is "singular" and will give as its answer **Inf** for each element of **x**. As we learned in Chapter 2 in Data Types, **Inf** is a special value that means "infinity". Here it is simply a sign that no solution exists in our finite world.

Underdetermined Problems

Problems for which $M = N$ are not the only problems of importance. In the general case, the number of equations M need not be equal to the number of unknowns N. For $M < N$, the problem is **underdetermined**, meaning that

there is not enough information provided by the equations to determine the solution completely. The simplest example is a single equation in two unknowns:

$$4x_1 + 5x_2 = 6$$

Here $M = 1$ and $N = 2$. If we solve this equation for x_2 , we get $x_2 = -(4x_1 - 6)/5$. This relationship shows us that we can choose an infinite set of values of x_1 from $-\infty$ to $+\infty$, as long as we choose the right x_2 to go with it. If MATLAB is asked to solve for **x** in this situation, it will find one, and only one of this infinity of solutions, preferring one that includes as many zeros as possible (one in this case):

```
>> A = [4 5]; b = 6;
>> x = A\b

x =
         0
    1.2000
```

Inadequate information can sometimes occur when $M = N$ and even when $M>N$. This happens when the information in some of the equations is redundant. For example, it happens for these three equations

$$\begin{aligned} 4x_1 &+ 5x_2 &= 6 \\ 8x_1 &+ 10x_2 &= 12 \\ -2x_1 &- \tfrac{5}{2}x_2 &= -3 \end{aligned}$$

There is no more information in these three equations than in the first alone. The reason is that each of the second and third equations can be obtained from the first by multiplying all terms by the same number (2 for the second equation and -1/2 for the second). Thus, any two of these equations are redundant relative to the third. The set of solutions to these three equations is the same infinite set of solutions that solve the previous example.

Overdetermined Problems

A more common situation when $M > N$, is that the problem is **overdetermined**, meaning that there is too much information provided by the equations. The equations represented by $Ax = b$ in this case cannot all be satisfied simultaneously by any value of the vector x. The MATLAB command **`x = A\b`** will still result in a value being calculated for **x**, even though that value does not satisfy the equality **`A*x = b`**. Why does MATLAB give a so-called "solution", when no solution exists? Isn't that a bad thing to do? Well, it is actually a good thing to do, but to see that we need to understand better the meaning of the "solution" that MATLAB gives when the equations are overdetermined. We will begin by looking at the following equations again:

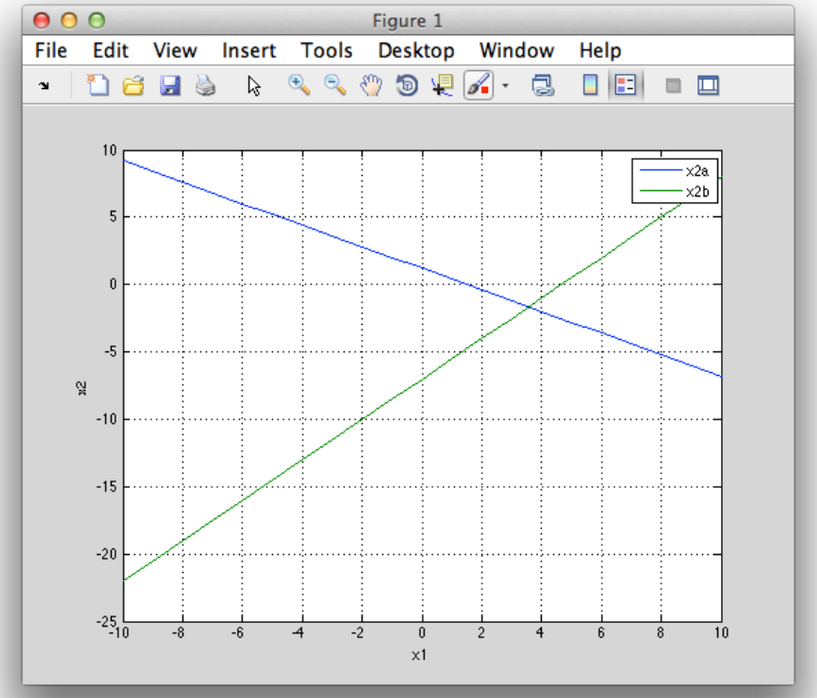
$$\begin{aligned} 4x_1 + 5x_2 &= 6 \\ 3x_1 - 2x_2 &= 14 \end{aligned}$$

which are not over determined. It is helpful to plot the lines defined by the equations. We begin by solving the first equation for x_2 and get $x_2 = -(4x_1 - 6)/5$. To plot x_2 versus x_1, we set up a MATLAB variable called **`x1`** that contains an assortment of values. We pick **`-10`**, **`-9`**, **`-8`**, . . ., **`9`**, **`10`**. We then set up a variable called **`x2a`** (the **`a`** indicates that we are plotting the first equation), which we set equal to **`-(4*x1 - 6)/5`**. We do a similar thing for the second equation, using **`x2b`** this time, and we plot both lines:

```
>> x1 = -10:10;
>> x2a = (-4*x1 + 6)/5;
>> x2b = (3*x1-14)/2;
>> plot(x1,x2a,x1,x2b)
>> grid on
>> xlabel('x1')
>> ylabel('x2')
>> legend('x2a','x2b')
```

The resulting plot is shown in Figure 3.1. It can be seen that the two lines cross at approximately the point (3.6, −1.7). Their intersection is the lone point at which both equations are satisfied. In other words, the pair of values, 3.6 and −1.7 are the approximate solution of the simultaneous equations. We solved these equations when we encountered them before by using **`x = A\b`**. We found that **`x(1)`**, which corresponds to x_1, equals 3.5652, and **`x(2)`**, which corresponds to x_2, equals −1.6522. These more exact values can be seen to agree with the intersection point in the plot.

Figure 3.1 *Solution to a system of two linear equations*



Thus, for the case of two equations in two unknowns, MATLAB gives us the exact (to within some small round-off error) solution. We now study an over-determined problem, in which there are more equations than unknowns, by considering the following set of equations,

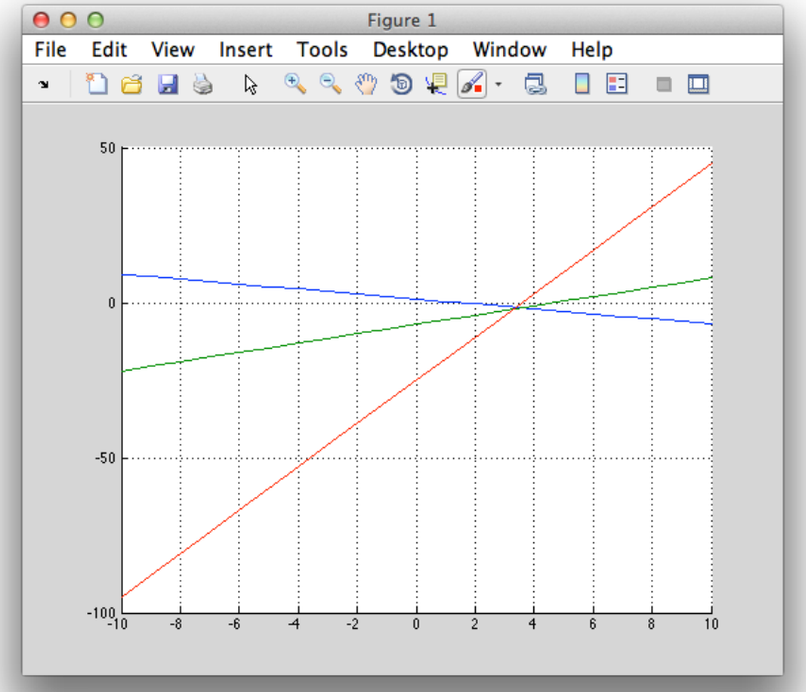
$$\begin{aligned} 4x_1 + 5x_2 &= 6 \\ 3x_1 - 2x_2 &= 14 \\ 7x_1 - x_2 &= 25 \end{aligned} ,$$

which are the same as the previous example with one new equation added. We can plot these three equations using the commands above plus these additional commands:

```
>> x2c = 7*x1 -25;
>> hold on
>> plot(x1,x2a,x1,x2b,x1,x2c)
```

The resulting plot is shown in Figure 3.2. The new line is plotted in red. The blue and green lines are the same plots as before, but their slopes appear to be different because the dimensions of the graph are different. MATLAB has chosen a larger vertical range to accommodate the third line.

Figure 3.2 *Solution to a system of three linear equations*

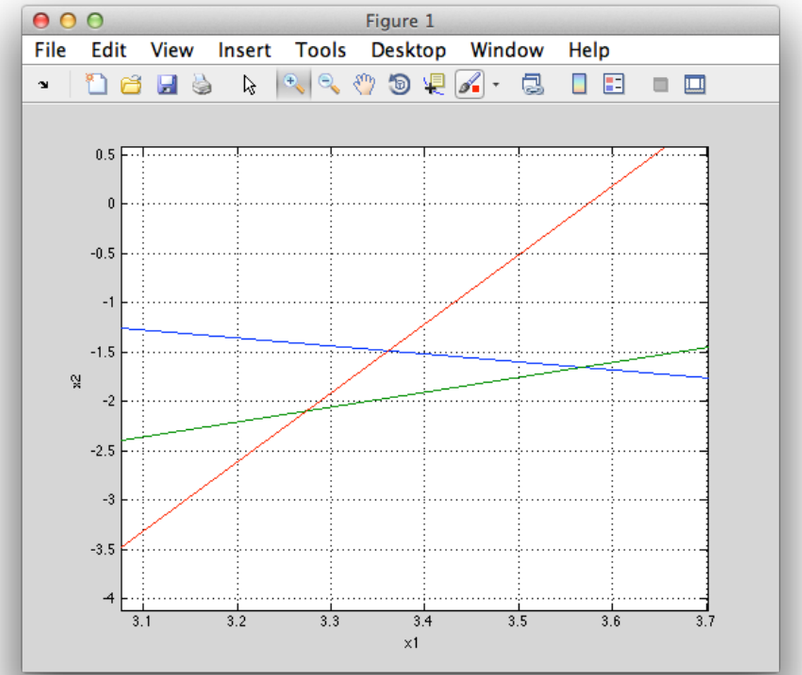


While it appears that the three lines cross at a point, in fact, they do not. To get a better look, we can click on the magnifying-glass icon with the "plus" sign in the middle of the circular "glass" part (just below the words "Insert" and "Tools" at the top of the figure window) and then click a few times near the area where the lines appear to cross. The result is shown in the magnified plot in Figure 3.3. At this level of magnification, we clearly see that the lines do not all cross at the same point, which means that there is no solution to these equations.

Nevertheless, we can ask MATLAB to try to "solve" them as follows:

```
>> A = [4 5; 3 -2; 7 -1]
A =
     4     5
     3    -2
     7    -1
```

Figure 3.3 *There is no exact solution to the equations*



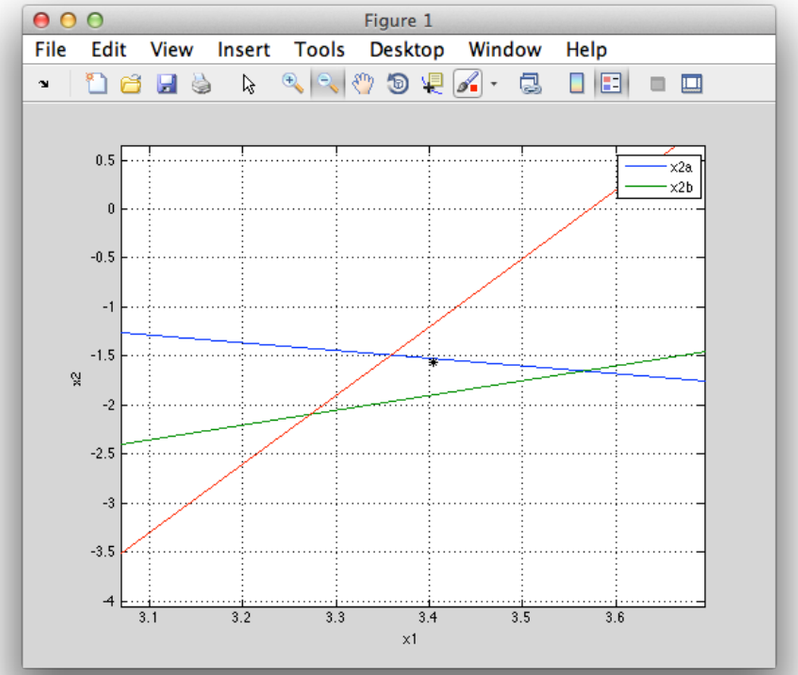

```
>> b = [6; 14; 25]
b =
     6
    14
    25

>> x = A\b
x =
    3.4044
   -1.5610
```

The resulting solution point can be plotted on the same graph by means of these commands:

```
>> hold on
>> plot(x(1),x(2),'k*')
```

The resulting plot (after magnification) is shown in Figure 3.4 and reveals that MATLAB has chosen a point that comes close to all three lines.

Figure 3.4 *Approximate solution provided by MATLAB*



Of course, this "solution" is only an approximation, and since there is no exact solution, there must be some errors in it. The errors in the solution can be seen by comparing **A*x** to **b**:

```
>> error = A*x-b

error =
   -0.1875
   -0.6647
    0.3920
```

If the solution were exact, all three elements of **error** would be zero. If an exact solution did exist (i.e., if all three lines crossed at the same point), MATLAB would have given us that solution. In all other cases, MATLAB gives that solution for which the sum of the squares of the elements of **error** is as small as possible. This value can be calculated using MATLAB's built-in function **norm** which gives the square root of the sum of the squares of the elements of the input vector:

$$\mathbf{norm(x)} = \sqrt{\sum_i x_i^2} \ .$$

There are other "norms", each of which measures the size of the vector in some sense (try **help norm**), but this norm is the most common and most useful one. In this case, we find that **norm(error)** gives the value **0.7941**. We can try other values for **x**. Each of them will produce a nonzero error, and the **norm** of the error will always be greater than **0.7941**. A solution for which **norm(error)** is as small as possible is called "the optimum solution in the least-squares sense".

Ill-conditioned Problems

We have seen that MATLAB will solve problems of the form $Ax = b$ when they have exact solutions, and it will give an optimum solution in the least-squares sense to problems that are overdetermined. In either of these situations, an important aspect of the problem is its stability to input error. While there are several definitions of stability, we will concentrate on the most common one: the effect, for a given matrix A and a given vector b, that changes in b have on x. This problem is confronted by engineers and scientists when (a) a problem of the form $Ax=b$ must be solved for x, (b) the values of A are calculated or known with high accuracy, and (c) the values of the elements of the vector b are measured with limited accuracy. There is always some level of error in the measurements, and if x changes drastically when small errors are made in b, then the problem is said to be **ill-conditioned**. The ill conditioning depends on both A and b, but it is possible to have a matrix A for which the problem is well behaved for any b.

To study the phenomenon of ill conditioning, we will consider the following two equations:

$$24x - 32y = 40$$
$$31x - 41y = 53$$

We calculate a solution to these equations using **x = A\b** as usual and we find that **x** equals 7 and **y** equals 4. If we perturb the values of **b** slightly to 40 and 53.1, we find that **x** equals 7.4 and **y** equals 4.3. While these may not seem to be significant changes, it is interesting to make some relative comparisons. Let's define the following convenient quantities:

bp: perturbed version of **b**
db: change in **b**: **db = bp - b**
xp: perturbed version of **x**: **xp = A\bp**
dx: change in **x**: **dx = xp - x**

We will use the norm as a measure of the size of some of these vectors, and we will examine the ratio of some of them. In particular, we find that

```
relative_input_error = norm(db)/norm(b) = 0.0015
```

and

```
relative_output_error = norm(dx)/norm(x) = 0.0620.
```

These two quantities represent the fractional change, respectively, in the sizes of **b** and **x** resulting from the addition of **db** to **b**. This addition represents a possible imprecision or error in the measurement. Because **b** is given, we call it the input (**A** is input as well, but, as we mentioned above, we are here treating the case in which its elements are measured with high accuracy, so we are not considering the effects of errors in its measurement.) Because **x** is calculated, we call it the output. This is the situation that would arise if errors in the measured values of the elements of **b** were to change by the amounts in **db**. These errors would result in the changes in **x** represented by **dx**. The fractional changes represent the seriousness of the changes in comparison to the quantities that are changing. In this case a 0.15% change in **b** results in a 6.2% change in **x**. Neither of these changes seems impressive, but their ratio is. The relative change in **x** is 6.2/0.15, or 41 times larger than the change in **b**. Thus, the error is increased by over four thousand percent! By comparison, if we perturb **b** for the first set of equations that we studied in this section from 6 and 14 to 6 and 14.1 and do a similar calculation, we find that the percent error is increase by a factor of only 1.1. This second situation is far more stable than the former one.

MATLAB provides a function to help you detect ill-conditioned problems. The function is called **cond**. To use it, one simply calls **cond** on the matrix **A**. If **cond(A)** returns a large value, then **A** is ill-conditioned; otherwise, it is not. The number returned by **cond** is called the **condition number** of **A**. It is the maximum possible percent change in the output for a one percent change in the input. Its precise definition is as follows: For any input vector **b** and any perturbed input vector **bp**:

```
relative_output_change ≤
            cond(A) × relative_input_change
```

Returning to the two examples above, we find that **cond([4 5; 3 -2])** returns 1.7888, while **cond([24-32;31-41])** returns 530.2481. The inequality above is clearly satisfied for both cases, and it is clear that the former is far more stable than the latter.

It is well to remember these examples when confronting any problem of the form $Ax=b$ when the unknown is x. If the condition number of A is large, the problem probably needs to be reformulated with more and/or different measurements taken to avoid having large errors in the output, even when the inputs are measured with relatively high accuracy.

Concepts From This Section

Computer Science and Mathematics:

linear algebra, matrix algebra
simultaneous linear algebraic equations
under determined
over determined
norm of a vector
ill-conditioning, ill-conditioned
relative input error and relative output error
condition number of a matrix

MATLAB:

```
b = A*x
x = A\b
norm(x)
cond(A)
```

Searching and Sorting

Objectives

Computer scientists expend a great deal of effort designing efficient algorithms and analyzing their behavior. This section will use two classic problems to introduce algorithm design and analysis.

(1) We will study the problem of searching a list for a specific value and the problem of sorting a list of values.

(2) We will learn two searching algorithms: sequential search and binary search.

(3) We will learn three sorting algorithms: selection sort, quicksort, and binary sort.

(4) We will learn how to analyze algorithmic behavior through the concepts of worst case, time complexity, dominance, and big-O notation.

You probably use a search algorithm almost every day without thinking much about how Google can find your needle in the world-wide-web haystack in a fraction of a second. Here we'll introduce simple techniques for searching and sorting.

In this section, we will look at two closely related problems: searching and sorting. Both problems appear in applications involving databases, and without efficient solutions to these problems, databases would be virtually useless. They would be useless because databases that are large enough to hold the information we need would be too large to use. They would be too large to use because it would take too long to find what we are looking for inside them. There is a large variety of specific applications of searching and sorting, but the most basic ones involve locating an element in a vector whose value is equal to a target value. We will use that simple problem as our driving example throughout the section.

Searching

Searching is important in many areas of computer science. Many large computer systems include databases that must be searched when an inquiry is made. For example, when you phone a credit-card company to inquire about your bill, you will be asked for your card number. In a few seconds, the company representative will be looking at your data. Your data and that of tens of millions of other customers are all stored in a database, and your information is found by searching among all those card numbers. The target of the search is your unique number. Furthermore, every day, millions of transactions, each involving one of those unique numbers, take place, and each one requires that your card number be located in a database of all credit cards. In each of these examples and in many other situations, such as looking up account balances, finding plane reservations, retrieving on-line orders, checking license plates, and many, many others, the first task is to find an entry in a database that is indexed by a target number, and it is always accomplished by a search algorithm.

Sequential search

The simplest search algorithm is **sequential search**, also known as **linear search**. In this method, the target number that is being sought is compared with the first member of the database, then with the second, etc., until either the number is found, or the end of the database is reached, whichever comes first. Instead of dealing with a real database, let's work on the simplest version of the search problem. We are given a vector of numbers and a target value. Our task is to search for that target value in the vector. If we find it, then the answer we give is the index of the element that we found.

If we do not find it, we return an impossible index, say, −1, as a flag that indicates that the search failed. That flag is crucial to the search algorithm because there must be some means of informing the searcher that the target is not there. As we learned in Chapter 2 in Loops, in computer science the term "flag" means a value indicating a special condition. In this case, the condition is "target not found", or "search failure".)

The function, **`sequential_search`**, below carries out the sequential search:

```
function index = sequential_search ...
                     (vector,target,first,last)
%SEQUENTIAL_SEARCH
%   SEQUENTIAL_SEARCH(VECTOR,TARGET,FIRST,LAST) returns
%   smallest index for which TARGET == VECTOR(index) or
%    -1, if TARGET not found within VECTOR(FIRST:LAST).

found = false; % Assume the target is not in vector
for n = first:last
   if target == vector(n)
      found = true;  % We found it...
      break;         % so we quit looking for it!
   end
end
index = n;
if ~found
   index = -1;
end
```

First we note that we have used the line continuation operator (`...`) to continue a statement on to the next line. Such continuation has nothing to do with functionality. Next we note that that there are four arguments. Argument 1 is the vector to be searched; Argument 2 is the target that we are searching for; Arguments 3 and 4 give the beginning and the end, respectively, of the range of indices over which we wish to search. Most of the time, when we call the function, we will set **`first`** to 1 and **`last`** to the full length of **`vector`**, but sometimes we may wish to search only a limited range. Arguments 3 and 4 are there to provide this option. So for example, if we wish to search all of a vector, named **`accounts`**, for the number stored in **`smith`**, we would give the call,

```
>> sequential_search(accounts, smith, 1, length(accounts))
```

but, if we wish to search only within, say, **`accounts(450:5600)`**, we would give the call,

```
>> sequential_search(accounts, smith, 450, 5600)
```

As promised, this function returns –1, as a flag indicating failure, where failure means that the target value is not equal to any of the elements in the vector. To assist in setting the flag, we have used an internal flag called **`found`** and the built-in functions **`false`** and **`true`**, which were introduced in Chapter 2 in section Loops.

It is customary in computer science to refer to a vector of numbers as a "list", and, if one of the elements of the list has a value equal to the target, it is customary to say that the target is "on the list" or the target is "in the list". If the target is on the list, then **`sequential_search`** returns the first position at which it finds the target. So, for example, suppose the function is applied to the list **`[45 23 17 17 -2 100 34]`**. In that case, the number 17 would be found at index 3 after the target had been compared with three numbers: 45, 23, and 17 (the first 17, that is). When searching for a target that is not on the list, 82, for example, the function will have to compare the target with every number on the list, requiring 7 comparisons. These few comparisons will not take long, but if there are a million numbers on the list, time can become a significant factor. If the target is not on the list, or if it is last on the list, a million comparisons are required. Numbers close to the beginning will be found quickly, numbers toward the end will take longer, but if all elements are equally probable, the mean number of comparisons required will be about half the length of the list: in this case, half a million. Faster searching methods are available, but only if the list has been sorted before the search is undertaken. For an unsorted list, the sequential search is the only possible choice.

Binary search

If a list has been sorted before the search is undertaken, searches can be carried out much more quickly. The search method that should be used on a sorted list is the **binary search**. It is more complicated than the sequential search method, but the complication is well worth it: A binary search requires no more than 40 comparisons to find a number in a list of one million numbers!

Recursive implementation of the binary search

If you have ever looked up a number in a phone book, the recursive approach to the binary search is one that you already know about. You don't look at every name sequentially, as in the sequential search algorithm. Instead, you look at a page near the middle of the phone book. You look at one name on that page and note whether the target name that you are searching for comes before or after it. If the target name comes before the name on that page, then you have eliminated about half the names in the book. Then, you go to a second place in the phone book closer to the beginning and repeat the process.

The "process" that you are repeating is the process of finding a name in a list by looking near the middle and eliminating about half the names in the list. When you repeat the process at the second place in the book, you are applying the process to a smaller list (one half of the book), but otherwise the process is the same. Since the process includes the application of itself (to a smaller list), it is recursive. Near the end of the recursive process, you will probably abandon the approach of eliminating sections of the book and just scan through the names on a page or two, but the major work was done when you eliminated large chunks of the book at the beginning with the first recursive steps.

When you use this approach you are carrying out your own recursive approximation of the binary search, and that is the version of binary search that we will consider first. Here is a recursive function that implements the binary search algorithm, providing the same interface (arguments) as that used by **`sequential_search`**:

```
1.  function index = binary_search_recursive ...
2.                       (vector,target,first,last)
3.  %BINARY_SEARCH_RECURSIVE
4.  %   BINARY_SEARCH_RECURSIVE(VECTOR,TARGET,FIRST,LAST)
5.  %   returns an index for which TARGET == VECTOR(index)
6.  %   or -1, if TARGET not found in VECTOR(FIRST:LAST).
7.
8.  mid = fix( (first + last)/2 );
9.  if ~(first <= last) % If first and last out of order..
10.   index = -1;       % ..then target not on the list!
11. elseif target == vector(mid)
12.    index = mid;     % found it!
13. elseif target < vector(mid)
14.    index = binary_search_recursive ...
15.              (vector,target,first, mid-1);
16. else
17.    index = binary_search_recursive ...
18.              (vector,target,mid+1, last);
19. end
```

The function works as follows:

Line 8: The middle of the range is determined. If **first** is odd and **last** is even, or vice versa, then that average will not be an integer. Since **mid** must be an integer in order to be an index, the **fix()** function is used to discard the fractional part (which is always 0.5), if there is one.

Lines 9-10: If **first** and **last** are out of order, then the range is empty, so **target** cannot possibly be in the range. Therefore, failure is flagged by returning –1.

Lines 11-12: The value of **target** is compared to the element at the middle of the range. If the **target** and the element are equal, then the search is successful, and the middle index is returned.

Line 13: If **target** is less than that middle number, then the middle element and all the elements that come after the middle element can be eliminated. The search can now be confined to the range of elements that come before the middle one.

Lines 14-15: A recursive call is made to search only the range of elements that come before the middle. Whatever index is received from this recursive call is returned.

Line 16: If this statement is reached, then **target** must be greater than the middle element. That means that the middle element and all the elements that come before the middle element are eliminated. The search can now be confined to the range of elements that come after the middle one.

Lines 17-18: A recursive call is made to search only the range of elements that come after the middle. Whatever index is received from this recursive call is returned.

It is important to note that during the first call of this function about half the numbers are eliminated before the first recursive call is made. This search algorithm is in fact named "binary" search because the list is repeatedly divided into two parts. (Actually, it is divided into two big parts and one very small one. The very small part is the single element in the middle.) The two big parts are subranges of the list. Subsequent searching in one of those subranges is accomplished by specifying the desired subrange via the third and fourth arguments and applying the binary search to it. The elimination of about half of the remaining elements happens again on the second recursive call, and on the third, and on each successive recursive call. Thus, each call cuts the size of the problem roughly in half.

This is a powerful approach to searching because relatively few divisions are required to trim a large list down to one number. If, for example, we search a list of one million elements for a target that is equal to the last number on the list, which can be shown to be the worst case, then the successive numbers of elements to be searched with this algorithm are 1000000, 500000, 250000, 125000, 62500, 31250, 15625, 7812, 3906, 1953, 976, 488, 244, 122, 61, 30, 15, 7, 3, 1. Thus, if the number is in the list, even in the worst case there will be only 20 calls of **binary_search_recursive** required to find a target number

within in a list of one million numbers. It takes no more than 21 calls to hit failure when the target is not there. In either case (i.e., the target is on the list or not on the list), during those 20 or 21 calls, no more than 40 comparisons will be made between target and the elements of vector. This is a very small number compared to the million comparisons required by the sequential search for the worst case: a *factor* of 25,000 fewer, and it is a factor of 12,500 smaller than the average case for sequential search, when the numbers are all equally probable, as they usually are.

The strategy of dividing a problem into smaller problems and then attacking the smaller problems recursively is a standard approach in the field of **algorithms**, which is the field of computer science that deals with the efficiency of algorithms. This strategy is called **divide-and-conquer**. We will look more closely at the idea of efficiency later in this section.

It is informative to follow the algorithm as it works its way through a few specific searches. Suppose we are searching for targets in a sorted vector named **A** that contains 11 elements:

```
A = [2  17  17  18  24  43  74  77  80  88  97] .
```

In our first search let's look for the target number 88:

```
>> binary_search_recursive(A,88,1,length(A))

ans =
    10
```

This search involves three calls of the function (the second two being recursive calls). The following is a summary of the search progression showing the search ranges for each successive function call along with the element **A(mid)** in the middle of the range that was compared to the target:

```
Range= A( 1)=  2 to A(11)= 97, mid element A( 6)= 43
Range= A( 7)= 74 to A(11)= 97, mid element A( 9)= 80
Range= A(10)= 88 to A(11)= 97, mid element A(10)= 88
```

Thus, the three numbers, **43**, **80**, and **88**, were compared with the target.

Now let's search for 17:

```
>> binary_search_recursive(A,17,1,length(A))

ans =
     3
```

Here is the summary of this search:

```
Range= A( 1)=  2 to A(11)= 97, mid element A( 6)= 43
Range= A( 1)=  2 to A( 5)= 24, mid element A( 3)= 17
```

Here, only the two numbers, 43 and 17, were compared with the target.

It should be noted that when there are duplicates on the list, the binary search algorithm, unlike the sequential search algorithm, *does not necessarily find the first element on the list that is equal to the target*. In this case the first element that is equal to the target value **17** is **A(2)**, but the binary search happened to encounter it first at **A(3)**. So it returned **3**.

It is interesting to note that in every case the first number compared with the target for this list is 43. This is always the first number for this list, because, when we ask the binary-search algorithm to search the entire list, it always starts at the middle of the list, *regardless of the value of the target*.

Finally, let's try one example in which the target is not on the list:

```
>> binary_search_recursive(A,90,1,length(A))

ans =
    -1
```

The summary of this search looks like this:

```
Range= A( 1)=  2 to A(11)= 97, mid element A( 6)= 43
Range= A( 7)= 74 to A(11)= 97, mid element A( 9)= 80
Range= A(10)= 88 to A(11)= 97, mid element A(10)= 88
Range= A(11)= 97 to A(11)= 97, mid element A(11)= 97
Range= A(11)= 97 to A(10)= 88
```

This time, the numbers 43, 80, 88, and 97 were compared with the target. In the last range, the first and last elements, 97 and 98, are out of order, so the algorithm treats the range as empty. No element is compared with the target. The algorithm halts and returns **-1** to indicate that the target is not on the list.

Iterative implementation of the binary search

As we learned in the previous chapter in the subsection entitled From Recursion to Iteration in section Functions Reloaded, every recursive function can be replaced by an iterative function that produces the same result. Here is an iterative version of binary search:

```
function index = binary_search_iterative ...
                          (vector,target,first,last)
%BINARY_SEARCH_ITERATIVE
%   BINARY_SEARCH_ITERATIVE(VECTOR,TARGET,FIRST,LAST)
%   returns an index for which TARGET == VECTOR(index)
%   or -1, if TARGET not found in VECTOR(FIRST:LAST).

found = false;
while first <= last && ~found
    mid = fix( (first + last) /2 );
    if target < vector(mid)
        last = mid - 1;
    elseif target > vector(mid)
        first = mid + 1 ;
    else
        found = true;
    end
end

if found
    index = mid;
else
    index = -1;
end
```

As in the recursive version, the list is repeatedly divided into smaller and smaller parts, but instead of executing a recursive call after each division, another iteration of the while-loop is executed. As with the recursive version, the variables **first** and **last** are set to be equal to the first and last indices, respectively, of the range being searched. The dividing of the list into smaller and smaller pieces is accomplished by changing, one at a time, either of these two variables, **first** or **last**, either moving **first** closer to the end or moving **last** closer to the beginning. As with the recursive version, the index of the element in the middle of the list is assigned to **mid** by taking the average of **first** and **last** and using **fix** to remove any fractional part.

All this work is done in a while-loop. That loop will end when the target is found because, in that case, the value of **found** , which is initialized to 0 with the built-in function **false**, is changed to 1 using the built-in function **true**. If the target value is not in the list, then the loop will still halt when the values of **first** and **last** become out of order (**first>last**), just as in the recursive version because, once again, having **first** greater than **last** means that the range is empty so the target cannot possibly be within it. Then the number **-1** is returned as a flag to mean that the number was not found. Let's apply this algorithm to the first example given above for the recursive version:

```
>> binary_search_iterative(A,88,1,length(A))

ans =
    10
```

The search ranges and middle elements are exactly the same as for the recursive version. In fact, the summaries given above for the recursive version apply to the iterative version as well as they do to the recursive version.

Sorting

Thanks to the advantage provided by the binary search algorithm over sequential search, it should be obvious that large lists that are going to be searched repeatedly should first be sorted in order to make subsequent searches more efficient. Sorting is another important task in computer applications, and it is carefully studied by students of that field. It is important because of the large savings in search time possible when using the binary search instead of the sequential search. Many sorting methods have been developed, some of which are quite complicated (far more complicated than the binary search algorithm given above, for example). The benefit in reduced sorting time is, however, worth the programming effort. Students who choose computer science or computer engineering as their major spend a good deal of time mastering the concepts involved in efficient sorting algorithms. The simplest form of the problem is to sort a vector of numbers into ascending order. We will confine our attention to that problem.

Selection sort

A sorting algorithm that is not very efficient, but is simple enough to be readily understood is the **selection sort**. Here is a function that implements the selection-sort algorithm:

```
1. function v = selection_sort(v)
2.
3. %SELECTION_SORT sort in ascending order
4. %   V = SELECTION_SORT(V) sorts vector V into
5. %   ascending order. The method used is
6. %   selection sort.
7. for m = 1:length(v)-1
8.     m_min = m;
9.     for n = m+1:length(v)
10.         if v(n) < v(m_min)
11.            m_min = n;
12.         end
13.    end
14.    if m_min ~= m
15.        temp = v(m);
16.        v(m) = v(m_min);
17.        v(m_min) = temp;
18.    end
19.end
```

It is used as follows,

```
>> vs = selection_sort(vu);
```

where **vu** is the vector to be sorted and **vs** is a vector containing the same elements as **vs** but arranged in ascending order.

To learn how the selection sort works, we will follow an example all the way through (laboriously!) step-by-step. Suppose **selection_sort** is given the following list:

```
v = [28    27    86    15    28    64]
```

Step 1: Find (i.e., "select", hence the name "selection sort") the smallest element, which in this case is **v(4)** and equals **15**, and swap it with the first element, **v(1)**, which equals **28**:

```
v = [28    27    86    15    28    64]
```

The result is

```
v = [15    27    86    28    28    64]
```

Now we have taken care of `v(1)`, and all that remains is to sort the items in `v(2:end)`.

`Step 2:` Find the smallest element in `v(2:end)` and swap it with `v(2)`. As it happens in this case, the smallest element in `v(2:end)` is already in `v(2)`. So no swap is necessary. The list is still

```
v = [15    27    86    28    28    64]
```

Now we have taken care of `v(1:2)`, and all that remains is to sort the items in `v(3:end)`.

`Step 3`: Find the smallest element in `v(3:end)` and swap it with `v(3)`. As it happens there are two smallest elements: both `v(4)` and `v(5)` have the smallest value, **`28`**. We choose to use the first of these, swapping with `v(3)`:

```
v = [15    27    86    28    28    64]
```

The result is

```
v = [15    27    28    86    28    64]
```

Now we have taken care of `v(1:3)`, and all that remains is to sort the items in `v(4:end)`.

`Steps 4 and 5:` We will spare you the details. Suffice it to say that 28 is swapped with 86, and then 64 is swapped with 86. At that point the list is sorted:

```
v = [15    27    28    28    64    86].
```

Each of these five steps is carried out by one iteration of the outer for-loop in **lines 7-19** of **`selection_sort`**. To sort a list of length six (like this one) requires exactly five iterations (one less than the length of the list), during which `m` has taken on the values 1, 2, 3, 4, and 5. Each iteration of that loop involves finding the smallest element in `v(m:end)` and swapping it with `v(m)`, as we have been doing above. When iteration `m` is over, `v(1:m)` has the correct elements and `v(m+1:end)` remains to be sorted.

The work of finding the smallest element is done by **lines 8-13**. (We could use MATLAB's **`min`** function to find it, but that would hide some of the complexity of the algorithm.) When those lines are completed, the variable `m_min` holds the index of the smallest element of those in `v(m:end)`. Those lines begin with the "guess" on **line 8** that the smallest element is at the beginning of `v(m:end)`. Thus, `m_min` is set equal to `m`. The inner loop in **lines 9-13** then compares that guess against the first element after `m`, namely, `v(m+1)`. If it is smaller, then `m_min` is set equal to `m+1`, which is the new guess. The loop index `n` holds the index of each element to be compared with the latest guess. When `n` has reached the end of the vector, and element `v(end)` has been checked, the guess has become a certainty. At this point, we know that `v(m_min)` is the smallest element in the `v(m:end)` vector.

Now it is time to swap `v(m)` and `v(m_min)`. That work is handled by **lines 14-18**. The if-statement checks to see whether a swap is necessary at all. It is unnecessary if the smallest element is already at `v(m)`, so the swapping is avoided if `m_min` is equal to `m`. If a swap is required, **lines 15-17** do the job. These three lines are very common in computer programming. In order to swap two numbers in MATLAB, it is necessary to hold one of them in a temporary variable, often called **`temp`**, as in this example. The swap always begins (**line 15**) by copying into `temp` the first one of the values to be swapped, in this case `v(m)`. The next step (**line 16**) copies the second of the values to be swapped, in this case `v(m_min)` into the first variable. If it were not for the fact that we have saved a copy of the original value of first variable in `temp`, this copying (i.e., **line 16**) would cause it to be lost. The final step (**line 17**) re-trieves that saved value and copies it into the second variable.

The selection sort continues in this way steadily whittling away at the list with its outer loop incrementing `m` and working on a smaller and smaller part of the list, `v(m:end)`, each time using the inner loop to find the smallest ele-

ment in **v(m:end)**, and using **lines 14-18** to swap it with element **v(m)**. The outer loop stops one position short of the end of the vector when the index **m = length(v)-1**. It can stop there because the previous step has established that the last element is not smaller than the next-to-last element. Therefore, the sort is complete. That is why our little six-element list **v** required only five steps to sort.

Quicksort

A sorting method that is much more efficient for most lists than any implementation of the selection sort and is fairly easy to understand, is called "quicksort". Here is a function that implements the quicksort algorithm:

```
1.  function v = quicksort(v)
2.  % QUICKSORT sort in ascending order
3.  %    Uses vectorized operations and
4.  %    logical variables. The method
5.  %    used is quicksort.
6.
7.  % Base case (empty or singleton)
8.  if length(v)<=1, return, end
9.
10. % Recursive case
11. less = v<v(1);
12. smaller = v(less);
13. remainder = v(~less);
14. remainder = remainder(2:end);
15. left = quicksort(smaller);
16. right = quicksort(remainder);
17. v = [left v(1) right];
```

Quicksort employs the divide-and-conquer strategy. The central idea is to divide the vector **v** that is to be sorted into three parts:

- **v(1)**
- **smaller**, a list that consists of all the elements in **v** that are smaller than **v(1)**
- **remainder**, a list that holds the remainder of the elements in **v**., i.e., those elements in **v[2:end]** that were not copied into **smaller**.

Then, we apply **quicksort** recursively to **smaller** to produce a sorted list, which we will call **left**, and then we apply **quicksort** recursively to **remainder** to produce a second sorted list, which we call **right**. Finally, we set

```
v = [left, v(1), right]
```

which shows why we named those sorted parts **left** and **right**. The resulting list is now sorted.

As an example, let's start again with the list we sorted above with the selection sort—before it was sorted:

```
v = [28    27    86    15    28    64]
```

The first (non-recursive) call,

```
>> quicksort(v)
```

begins by dividing **v** into three parts, as described above,

- **v(1)**, which equals **28**
- **smaller**, which equals **[27 15]**
- **remainder**, which equals **[86 28 64]**

Creating the list **smaller** is accomplished by means of **lines 11-12**. Together these two lines provide an example of MATLAB's "logical indexing" feature, which was introduced in subsection Logical Indexing of the Loops section, but that was long ago and far away, so we will refresh your memory by explaining each of the aspects of this feature as they are encountered here. As explained in that section, logical indexing is the use of a logical array to select the desired elements of an array, instead of the normal method of listing the indices of the desired elements of the array.

The right side of **line 11** produces a vector whose elements each equal 1 or 0, with 1 meaning it is true that the element is less than `v(1)` and 0 meaning false. The resulting vector is then assigned to `less`. The true/false value of each element `less(ii)` now indicates whether or not `v(ii)<v(1)`, for `ii` equal to 1, 2, ... The result is called a "logical array", meaning that the type of `less` is "logical" and can therefore be used in operations like the one on the right side of **line 12**. The value that is assigned to `less` in this example is `[0 1 0 1 0 0]`, since `v(2)` and `v(4)` are the (only) elements that are less than `v(1)`.

The right side of **line 12** produces a vector consisting of those elements of `v` that are located at the same positions as the `1`s in `less`. They are kept in the same order too. Thus, the vector assigned to `smaller` is `[27 15]`, since

`v(2) = 27` and `v(4) = 15`.

The total effect of **lines 11-12** is to remove the elements of `v` that do not satisfy the rule, `v(ii) < v(1)`.

Creating the list `remainder` is accomplished by **lines 13-14**. **Line 13** is similar to **line 12**, but the logical array `less` has been replaced by `~less`, whose value is "opposite" to that of `less`. Thus, it is equal to `[1 0 1 0 1 1]`. Thanks to the "not" operator `~`, the vector assigned to `smaller` has `1`s at (only) the positions for which the elements `v(ii)` is not less than `v(1)` (there are zeros elsewhere). These elements are `v(1) = 28`, `v(3) = 86`, `v(5) = 28`, and `v(6) = 64`. Logical indexing causes each of these elements to be copied into `remainder`, which now equals `[28 86 28 64]`. Note that `v(1)` is on the list. While it is true that `v(1)` is not smaller than `v(1)`, we do not want it on this list.

Line 14 is responsible for removing `v(1)` from `remainder`. Because the logical indexing operation in **line 13** does not change the order of the elements, `v(1)` is still in the first position. Since `remainder(2:end)` contains everything in `remainder` except its first element, we can be assured that `v(1)` has been left off.

Now we carry out the recursive call on **line 15**:

```
left = quicksort(smaller)
```

which returns `left` equal to the sorted list, `[15 27]`. Our second recursive call, **line 16**,

```
right = quicksort(remainder);
```

sets `right` equal to `[28 64 86]`. Finally, we combine the three parts into one sorted list:

```
v = [left, v(1), right],
```

which equals `[15 27 28 28 64 86]`.

The algorithm is completed and `v` has been sorted into ascending order.

Every recursive function must have a base case, and, as can be seen in **line 8**, the base case here is determined by the length of the list. If the length is one, then there is no work to do. The list is returned without change. The same holds for an empty list (length equal to zero). For all other lengths, recursion is employed.

The quicksort algorithm is much faster than selection sort for most lists. As an example, on one rather standard laptop the selection sort took 19 times as long as quicksort to sort one hundred thousand numbers.

Merge sort

While quicksort is much better than selection sort, it is not the best sorting algorithm available. It has a weakness that keeps it from being best. The weakness stems from the fact that when the list that it is given to sort happens to be almost sorted, its divide-and-conquer strategy no longer works

well. In that case, the list **`smaller`** is almost always empty, so the recursive call on **line 15** often does nothing. The recursive call, **`right = quicksort(remainder)`** is in that case given a list that is shorter than the original list by only one element. As a result there is one recursive call required for nearly every element on the list. For all but very small lists (e.g., less than 500 elements) the stack grows too large and MATLAB gives an error message. This stack-overflow problem can be solved by using a non-recursive version of quicksort (remember that there is always a non-recursive version of a recursive algorithm), but, because the division of the list is so one-sided, the search is in this case no more efficient than selection sort.

Fortunately for the modern world in which huge lists abound, there are sorts that do not suffer from this problem. One of them is the ***merge sort***. Like quicksort, it uses divide-and-conquer, but it is guaranteed to divide the list into two approximately equal parts. So there is none of this one-sided search business, even for almost-sorted lists, and there is no realistic chance of stack overflow. Here is a recursive implementation of merge sort:

```
1.  function v = merge_sort(v)
2.  % MERGE_SORT sort in ascending order
3.  %   A = MERGE_SORT(V) puts the elements
4.  %   of the row vector V into ascending
5.  %   order in the row vector A. The
6.  %   algorithm used is the merge sort.
7.
8.  N = length(v);
9.  if N == 1, return; % already sorted
10. else
11.     mid = fix(N/2);
12.     v1 = merge_sort(v(1:mid));
13.     v2 = merge_sort(v(mid+1:end));
14.     v  = merge_sorted_lists(v1,v2);
15. end
```

The algorithm is fairly simple, but this implementation hides a bit of the complexity because of the call to to the function **`merge_sorted_lists`**. The base case is determined by the length of the list, just as it is in quicksort: Lists with 0 or 1 elements are already sorted, so the function simply returns without changing the list. **Line 11** begins the interesting part of the algorithm. The midpoint is determined in **line 11**, and it is this line that guarantees that this algorithm, unlike quicksort, will always divide the list into approximately equal halves, regardless of the values of the elements in it.

In **line 12** the elements from the first up to this midpoint are sorted. In **line 13** the rest of the numbers are sorted. **Line 14** does the rest of the work by calling **`merge_sorted_lists`**. That function combines the two sorted lists into one sorted list. We leave the writing of that function as an exercise for the reader (because this is the Advanced-Concepts chapter!).

There are many other sorting algorithms available, and computer scientists spend a good deal of time studying them. MATLAB provides an excellent sorter called (what else?) **`sort`**. It uses varying strategies according to the size of the list and the size of the available memory. A study of these strategies and of sorting in general is beyond the scope of this textbook. But note that the function **`sort`** employs a very fast algorithm.

Algorithmic Complexity

When a computer scientist says that an algorithm is fast, she means something very specific. A "fast" algorithm is not simply an algorithm that runs fast. One problem with that definition is that the speed at which an algorithm runs depends on the hardware that it runs on. Since computers are different, and since computers improve every year, it is meaningless to measure the speed of an algorithm by timing it. Furthermore, the relative speeds of two algorithms that are running on the same machine may depend on the size of the data set that they are working on. For example, for a list of 10 numbers, the sequential search may take less time than the binary search, while for larger lists the binary search will beat the sequential search (and beat it

soundly). Instead of evaluating algorithms by observing their behavior, computer scientists predict their behavior by analyzing their description. The analysis inevitably shows that the behavior depends on the characteristics of the input, and since there is an almost infinite variety of inputs, it is necessary to make predictions for classes of inputs. The most important class is the worst-case input.

Worst-case analysis

A more important aspect of an algorithm than the time it takes on a particular dataset on a particular computer is the way in which the number of operations that it must perform grows as the size of the dataset grows. That number may depend on the data involved. An example of that data dependence is provided by the sequential search. The best case for the sequential search is that in which the first item on the list is the one we are looking for. In this trivial case, only one comparison is required. Its worst case happens when the item is at the end of the list or is missing from it altogether. Similarly, the quicksort works well most of the time, but fails when the list is almost sorted. In designing algorithms, the worst case is usually of most concern. This makes sense in the example of the search for credit-card information above, for example, where we would want to minimize longest time that a customer would have to wait.

Therefore, most algorithms are subjected to **worst-case analysis**, which is evaluation of an algorithm based solely on inputs for which the algorithm exhibits its worst behavior. For example, an analysis of the sequential search algorithm above reveals that there are two worst cases (i.e., they are equally bad and are worse than all other cases): (a) the target is not on the list at all or (b) it occurs only once and is the last item on the list. In each of these cases, the number of comparisons is equal to the number (`N`) of items on the list. While comparisons are not the only work done by the algorithm, the number of comparisons is a good measure of the algorithm's work because, in the general form of the algorithm, comparing a target to an item on the list is the most time-consuming step and also because the number of other required operations is approximately proportional to the number of comparisons.

For the binary search, there are also two worst cases: (a) the target is larger than all the items on the list and (b) it occurs only once and is the last item on the list. In each of these cases the number of comparisons is equal to `2*ceil(log2(N+1))`, where `log2(x)` is the power to which 2 must be raised to get `x`. Determining formulas like this is the sort of thing computer scientists do, but it is fairly easy to see that this is likely to be the correct formula by considering some specific worst-case examples.

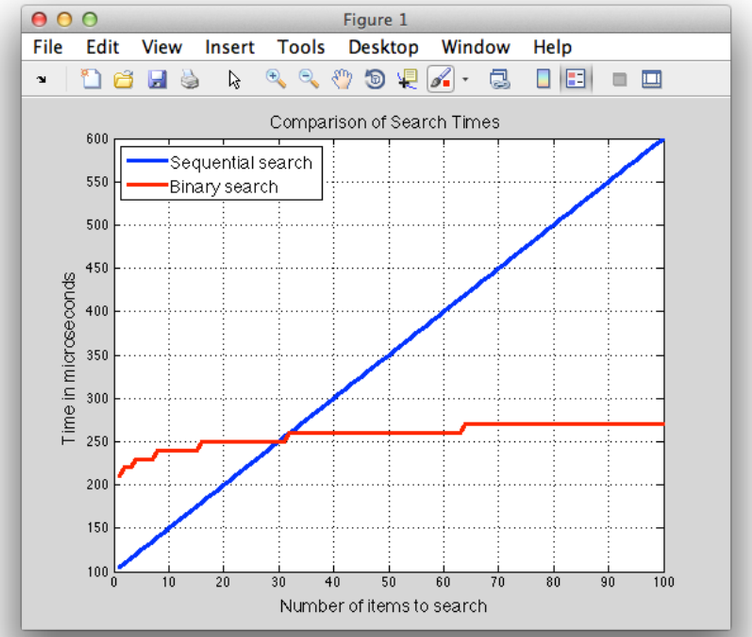
As mentioned above, the time required to complete the search is approximately proportional to the number of comparisons required. Thus, the worst-case time necessary to complete these algorithms on a list of length `N` are related to `N` as follows:

sequential: `a + b*N`

binary: `a + b*ceil(log2(N+1))`,

where the `a` and `b` represent constants that will be different for the two algorithms (i.e., two different values for `a` and two different values for `b`). The value of `a` equals the start-up time for the algorithm, including the time required to load the code (e.g., MATLAB's M-file), to put a new stack frame on the stack, and to assign initial values to some local variables. The value of `b` equals the amount of time required per comparison in the sequential search and equals twice the amount of time required per comparison in the binary search. The plot in Figure 3.5 shows a sample behavior for these two functions when `a=100` and `b=5` for the sequential search and `a=200` and `b=10` for the binary search. These plots are for fictitious implementations but they are representative of the algorithms' behaviors.

Figure 3.5 *Time complexity of the sequential and binary searches*



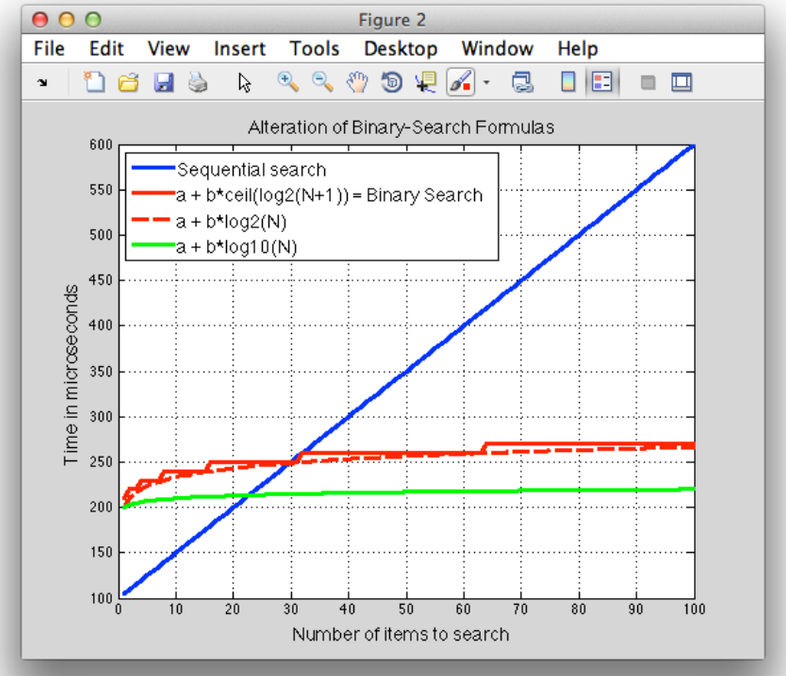
Note that our vaunted binary search actually requires more time then the sequential search for small lists (fewer than about 30 items), but after the size of the list has grown to 100, it takes less than half the time. The advantage of binary search grows as `N` grows. For `N = 1000`, the advantage is 8 to 1, and for `N` = one million the advantage is over 6000 to one.

Order Notation

The important difference between the two plots above lies in the difference between their shapes. Both plots increase monotonically with the number of items, but the red plot (the binary search) has a different shape from the blue one (the sequential search). The red plot curves downward. Because of that downward curve, it is inevitable that the straight-line blue plot (the sequential search) will eventually rise above it, and we can see that it does that just beyond `N` = 30.

There are three differences between the two formulas above. Each formula has different values of `a` and of `b` and one has `ceil(log2(N+1)` where the other has simply `N`. Of these, the difference that determines the shape of the curve is the last one—the dependence on `N`. That is so because, for any values of `a` and `b` in either or both formulas, the second formula will produce a plot that curves downward and hence, will cross the straight line for some sufficiently large `N`. That means that for a big enough problem the binary search will win, even if its `a` and `b` are bigger than the `a` and `b` for the sequential search (as they are in the example above). Furthermore, as Figure 3.6 shows, there is little importance to either the `ceil()` or the `+1` in `ceil(log2(N+1))`. When we omit them, we are left with `log2(N)`, and the resultant plot (the dashed red line) is almost the same. Finally, the importance of `log2(N)` is the fact that we are taking the logarithm of `N`, regardless of what the base is. It can be seen in Figure 3.6 that when we replace `log2`

Figure 3.6 *Illustration of the order notation*



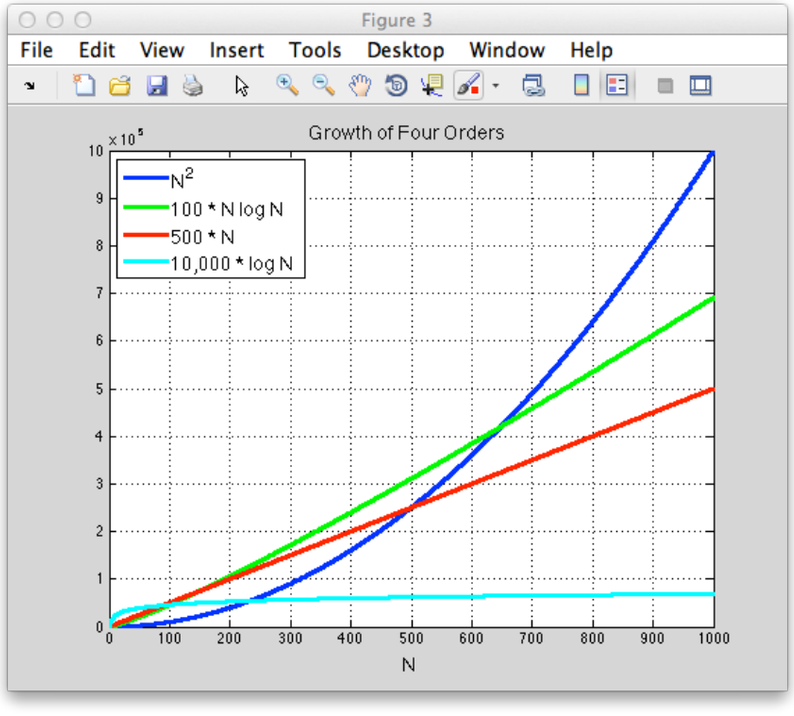
with **log10** (log base 10), the general shape of the plot (green line) remains the same. It is curved downward, so it will cross the plot of the sequential search somewhere. It crosses at a different place, but if we are comparing one algorithm to another the important thing is not where it crosses, but the fact that it crosses somewhere.

Thus, the important aspects of the worst-time behavior of the binary search can be described simply by saying that the plot of the search time versus **N** has the same shape as **log(N)** for any base. Likewise the important aspects of the worst-time behavior of the sequential search are captured by saying that the plot of the search time versus **N** has the same shape as **N**. These statements are said more formally as follows: The worst-case behavior of the sequential sort is "order N". The worst-case behavior of the binary sort is "order log N". The phrase "order N" is written in mathematical notation as O(N). The phrase "order log(N)" is written O(log N). Usually a computer scientist will simply write, "the binary search is O(log N)", leaving out the word "behavior", which is to be understood. Sometimes we are interested in the increase in memory required by an algorithm as N increases, as opposed to the increase in time. In that case, we determine a formula for the number of cells of memory required as a function of N. In either case, the notation is called **Order notation**, **Big-O notation**, or sometimes, **O notation**.

The selection sort is $O(N^2)$. The quicksort algorithm is also $O(N^2)$, because of its difficulties with its worst case, which is an almost sorted list. Merge sort, and all the best sorting algorithms, MATLAB's **sort** for example, are O(N log N).

We have introduced four worst-case behavior classes: O(log N), O(N), O(N log N), and $O(N^2)$. As the plots in Figure 3.7 show, they are listed here from fastest to slowest. The relative sizes are so different that in order to show them all on the same plot, we have multiplied the first three by 10,000, 500, and 100, respectively.

Figure 3.7 *Comparison of four different complexities*



Dominance

It is clear from the discussion above that the value of **a** in the formulas is unimportant. That is because the second term will **dominate** **a** for very large values of **N**. A similar thing happens for formulas like this: **a*log(N) + b*N**. Here, the second term is the only important one because **N** dominates **log(N)**. The dominance of **N** over **log(N)** can be stated this way: the ratio, **N/log(N)** can get as large as desired, provided **N** is made large enough. In terms of algorithms, that means that if an algorithm's time-behavior has the form **a*log(N) + b*N**, then the algorithm is O(N), not O(log N + N). The dominant term determines the order of the algorithm because the other terms become insignificant for very large **N**.

Complexity definitions

We are now (finally) ready to give a definition for the term **algorithmic complexity**, which is the name of this section. It is the shape of the plot of the al-

gorithm's resource requirements as a function of the size of its input. The resource of interest is typically time of execution, in which case the measure is **time complexity**, but in some cases the resource of interest is size of required memory, in which case the measure is **space complexity**. The shape is given by means of order notation.

Algorithmic complexity is an essential concept in the branch of computer science known as "Algorithms". Practitioners of that field develop better and better algorithms for harder and harder problems. When they evaluate an algorithm, there is no concern for how complicated the algorithm is or how hard it is to program, and no experiments are undertaken to measure speed. This is a theoretical field, the only measure of quality is complexity, and the quality is expressed using order notation. This very short introduction to that concept provides only the barest glimpse of what is involved in the field of Algorithms, but at the least it shows that it is possible to put the analysis of algorithms on a firm footing.

Concepts From This Section

Computer Science and Mathematics:

sequential search, also called linear search
flag
binary search
algorithms (i.e., the field of study)
divide and conquer
selection sort
quicksort
merge sort
algorithmic complexity
worst-case analysis
time complexity
order notation
big-O
O(log N), O(N), O(N log N), O(N2)
dominance

MATLAB:

[No new concepts specific to MATLAB]

Object-Oriented Programming

Objectives

Object-Oriented Programming, or OOP, is a robust approach employed by professional programmers for writing large programs that is the subject of entire books. This section introduces OOP via an extended tutorial example.

We will learn:

(1) basic concepts to get started with OOP in academic studies or in work;

(2) the three most prominent features of OOP: inheritance, encapsulation, and polymorphism;

(3) object references, which are known as "handles" in MATLAB terminology;

(4) the "linked list" and its implementation via handles.

Objects are everywhere in real life. It turns out that they are a very useful abstraction in computer programming also.

What we have learned so far falls within the category of procedural programming. We have organized our code into functions, typically creating a main function that utilizes helper functions to compute whatever we need and then returns the results to the Command Window. The data that our programs have worked on have been passed around as function arguments and have been stored in local variables and sometimes (rarely) in global ones. This has worked just fine for us, but one reason that it has worked is that the sample programs and exercises in this book are all necessarily small and relatively easy to comprehend. But what happens when we have to write large, complex programs? Does this approach scale well?

The answer is sadly no. Procedural programming focuses on the operations a program needs to do. But as the size and complexity of the program grow, it becomes more and more difficult to manage the data that the program works on. Having only global and local variables and function arguments at our disposal makes it harder to keep track of the data, to make sure that we do not make unintended changes to it, and to understand and modify existing programs. **Object-Oriented Programming** (OOP) helps with all of these problems.

Object-Oriented Programming, unlike procedural programming, is centered on data as opposed to functions. The **object** in Object-Oriented Programming consists of one or more data fields similarly to a struct. But unlike structs, an object also contains functions that operate on the data. A cool feature of an object is that you can control which data fields (and which functions) are accessible outside of the object. Let's say, for example, that you need a sorted-vector data type. Using procedural programming, you would create a regular vector `v` and write functions to insert a new element into it, to remove an element from it, and to find and element within it. However, the vector itself will be accessible from other functions as well, and it can be inadvertently modified. There is, for example, nothing stopping you or somebody else who uses and modifies your code from writing:

```
v(end+1) = 3;
```

If the variable `v` is in the current scope, MATLAB will happily increase the size of `v` by one and assign `3` to the last element. In all likelihood, `v` is not going to be sorted after that. With OOP, on the other hand, you can create an object with this vector as a data field and make it inaccessible from the outside. The only functions that can modify the vector are the insert, remove, and find functions that are part of the object. If you try to modify the vector as we did above, MATLAB will generate an error message instead of executing the assignment. Hence, once you have made sure that these functions work correctly, you never need to worry about this sorted vector again.

The OOP feature that we have just highlighted is called **encapsulation** or information hiding. You have fine control over how an object can be used by separating its implementation, which determines how an object works, from its interface, which specifies how the object can be used. You can hide (or encapsulate) the implementation details from the user of the object, who will be utilizing only a limited set of functions that you have provided to access the object. This hiding of implementation detail has the added advantage that you can freely modify the implementation of the object as long as the interface, which is the set of functions that users of the object can utilize, does not change. Many times the designer and the user of the object is one and the same person, you, but it is still a good idea to use these OOP features when writing large programs. It will make your program less error prone, easier to debug, and easier to maintain.

It is time to introduce some OOP notation. The definition of an object is called a **class**. An object is an **instance** of a class. You can create as many instances of a class as you want. A data field of a class is analogous to a field of a struct, and like the struct field, is one of the variables specific to that class. In OOP, it is called a **data member**. In MATLAB, it is called a **property**. A function within a class is a **member function** or, using MATLAB notation, a **method**. For the rest of the book, we will stick with MATLAB conventions.

You might have noticed that defining a class is equivalent to introducing a new data type. You specify what data it consists of using properties and what operations are supported using methods. OOP even allows you to determine which operators and built-in functions can work on the new datatype and how. The general concept of specifying that a function or operator can process a new data type is called **overloading**, and when an operator is overloaded it is specifically called **operator overloading**. This is an extension of the polymorphism that we defined in Section, Programmer's Tool-

box: In addition to having a function behave differently depending on how many and what kind of input and output arguments are used when calling it, now we can have operators redefined to support user-defined data types. This redefinition of operators is a very powerful feature that has the potential to make complex programs very intuitive and easy to understand.

Finally, one of the most useful OOP features is called **inheritance**. What it means is that a class can build on another class and extend it to provide more specialized functionality. For example, consider the problem of writing banking software that manages accounts. There are different kinds of bank accounts, such as checking, savings, and credit card. Instead of making three unrelated classes for these types of account, it makes sense to capture their common traits and behavior in a single class. Let's call this class `BankAccount`. It has properties for the account number, owner, institution, balance, etc. Now, we can create a subclass of the `BankAccount` class that inherits all the features of `BankAccount` and in addition implements the extra functionality associated with the new kind of account. And we can create multiple subclasses. For example, the `CreditCardAccount` class might have a credit-limit property, while the `CheckingAccount` class has a list-of-checks property. We say that the "subclasses" (`CreditCardAccount`, `CheckingAccount`) "extend" the "base class" or "superclass" (`BankAccount`). From this example it should be clear that a **subclass** is a class that has been defined via the mechanism of inheritance to be an extension of another class., and a **base class** or **superclass** (yes, they are seemingly contradictory terms for the same thing!) is a class whose properties have been inherited by another class. A very useful attribute of inheritance is that a subclass can behave like a superclass. For example, `CheckingAccount` has an account number automatically. If a function is expecting a `BankAccount` instance, we can supply instead a `CreditCardAccount` instance and the program will work as expected. This is another form of polymorphism supported by OOP.

MATLAB

MATLAB is primarily a procedural language. That's why we have been able to cover all the material so far without OOP concepts. However, the language designers at MathWorks decided to extend the original language with OOP features a few years ago. Since they had to work with the existing language and had to make sure that it remains backward compatible, that is, that all existing programs remain operational, they had to make some compromises on which OOP features to support and how to do it. Overall, they did an excellent job of supporting most OOP concepts in an intuitive and elegant manner.

An Extended Example

It is our experience that to appreciate the ingenuity of OOP, you need to see it in action. You can read all about encapsulation and inheritance, but until you see OOP actually being done and see through an example what it provides you, you will not understand what all the hoopla is about. So, let's consider a longer than usual example that illustrates many nice OOP concepts.

Let's a create a program that manages our contacts, which are people and their phone numbers. The first thing to do is to define a class for a single contact. For simplicity, it will have only the name of the person and a single phone number. Here is our first class definition in MATLAB:

```
classdef contact
  properties
    firstname
    middlename
    lastname
    phonenumber
  end
  methods
    function obj = contact(lname, mname, ...
                                      fname, phone)
      if nargin > 0 obj.lastname    = lname;  end
      if nargin > 1 obj.middlename  = mname;  end
      if nargin > 2 obj.firstname   = fname;  end
      if nargin > 3 obj.phonenumber = phone;  end
    end
    function disp(obj)
      if isscalar(obj)
        fprintf('Name: %s, %s %s\n',...
                                  obj.lastname,...
                                  obj.firstname,...
                                  obj.middlename);
        fprintf('Tel:  %s\n\n',...
                                  obj.phonenumber);
      else
        fprintf('array of contacts\n');
      end
    end
  end
end
```

In MATLAB, the definition of a class needs to reside in its own M-file. The class name and the file name must be the same. The class definition starts with the **classdef** keyword followed by the name of the class. The rest of the class definition consists of two sections: one introduced by the keyword **properties**, for the properties and one introduced by the keyword **methods** for the methods. For our contact class, we defined four properties, three for the different parts of one's name and one for the phone number. We intend to store these as strings, but since MATLAB does not require type specification, this decision is not obvious from the code so far.

The first method bears the name of the class, **contact**. As you might have guessed, this is a special function, called the **constructor**. MATLAB calls this function when it needs to create a new **contact** object, that is, a new instance of the **contact** class. You do not need to specify a constructor, although there are exceptions, but if you do, it must return a single output argument, a valid instance of the class. It can have zero or more input arguments. In our case, we have four input arguments serving as initial values for our properties. As you can see, the syntax to access the properties of a class calls for a period between the object name, i.e. the name of the variable holding the object, and the property name, just as a period separates a struct from a field name. For example, the line

```
obj.lastname = lname;
```

assigns the value of the input argument **lname** to the property **lastname**.

The second function of our class is also kind of special. MATLAB calls the **disp** function whenever it needs to display the value of a variable. For example, when we type in the name of a variable in the Command Window, the **disp** function is called to print out the value, whether it be a scalar or a matrix or anything else. Here we use overloading with the **disp** function provided by MATLAB (we "overload **disp**") to print out a **contact** object the way we want to. Notice how we handle the case when an entire array of **contact**s are passed to **disp**. Instead of printing out every value, we simply print the type of data.

Let's test drive our shiny new class. Assuming we saved the **contact.m** file in the current folder or that the folder is on the path, typing the following lines in the Command Windows should get you the results below:

```
>> a = contact('Smith','K','John','123 555 1234')

a =
  Name:     Smith, John K
  Tel:      123 555 1234
```

MATLAB realized that there was a **contact** class defined, so it called the **contact** constructor, which created a new instance of the **contact** class and, i.e., a new **contact** object, initialized all its properties and returned it in

the variable **a**. Since we did not use a semicolon when assigning a new contact object to the variable **a**, MATLAB prints its value by calling the **disp** function (as it has all along for non-OOP objects). If we had not overloaded **disp** with our own function definition, this is what MATLAB would have printed:

```
a =
  contact

  Properties:
      firstname: 'John'
     middlename: 'K'
       lastname: 'Smith'
    phonenumber: '123 555 1234'

  Methods
```

Note that we have written our constructor so that it can be called with zero to four arguments (with all those statements that start with `if nargin`). For example,

```
>> b = contact('Schwarzenegger')

b =
  Name:     Schwarzenegger,
  Tel:
```

Notice that only the **lastname** property has a value. In that case the other three are each assigned the empty matrix. Let's try this:

```
>> b.firstname = 'Arnold'

b =
  Name:     Schwarzenegger, Arnold
  Tel:
```

The **firstname** property is directly accessible from the outside of the object. Many times, that is exactly what we want. How about:

```
b.phonenumber = 5551234567

b =
  Name:     Schwarzenegger,  Arnold
  Tel:      5.551235e+09
```

What happened there? We intended the **phonenumber** property to be a string, but there is nothing in our code that enforces that. Users of our class may easily assume that a phone number is a number and get the unexpected behavior above. By the way, the choice of the string data type makes sense here because phone numbers can be formatted in a number of different ways with dashes, spaces, an parentheses. In fact, it would make a lot of sense to design a separate class for phone numbers with properties for country code, area code, number and extension. Then the **phonenumber** property in the contact class would be an instance of that new class. But that would make our example too long for a textbook, so we'll stick with a string here.

How can we enforce the rule that a phone number in our class must be a string? We need to start with the constructor, modifying it as shown below:

```
function obj = contact(lname,mname,fname,phone)
  if nargin > 0
    obj.lastname = lname;
  end
  if nargin > 1
    obj.middlename = mname;
  end
  if nargin > 2
     obj.firstname = fname;
  end
  if nargin > 3
    if ~ischar(phone)
      error('string expected for phone number');
    end
      obj.phonenumber = phone;
  end
end
```

This takes care of the problem when the constructor is called with a number for the phone number. If we wanted to write a bulletproof class, we should

make the same check for the other three properties as well. However, it is less likely that the user of the class will provide anything but a string for the name properties, so we shall leave the constructor as it is.

Fixing the constructor is not enough. We need to make sure that the **phonenumber** field of an existing object cannot be changed to anything but a string. At first this seems hard, since we would need to do something about the assignment operator. Fortunately, MATLAB provides a way to change how operators work on instances of a class. Basically, we can provide a method for our class that will be called when a given operator is encountered by MATLAB.

For the assignment operator, we need to write a function called **set.propertyname** where **propertyname** is the name of the property we want to set. This is what it looks like:

```
function obj = set.phonenumber(obj,value)
  if ~isempty(value) && ~ischar(value)
    error('string expected for phone number');
  end
  obj.phonenumber = value;
end
```

The first argument must be the object whose property is involved and the second argument is the value that is to be assigned to the property. The function has no output argument. Let's try the same assignment again but this time relying on our new overloaded assignment operator:

```
>> b.phonenumber = 5551234567
Error using contact/set.phonenumber (line 52)
string expected for phone number
```

We see an error message, but that is what we wanted to see, because it means that our overloaded operator is protecting the **phonenumber** property from getting assigned a value of the wrong type. By now, you might be seeing just how powerful **classdef** is and how cool OOP is. We have just changed MATLAB's assignment operator! Let's cross our fingers and try assigning the proper type to the **phonenumber** property:

```
>> a.phonenumber = '123 555 9876'
a =
Name: Smith, John K
Tel:  123 555 9876
```

This thing is working!

Inheritance

At this point we have a nice class that is able to handle personal contacts. But many of our contacts in real life are companies such as the car shop, the dry cleaner's, or the doctor's office. Our contact class is not prepared to handle a business contact well. Such a contact needs a business name, a phone number and optionally a contact person. Instead of creating a new class from scratch, it would be good to reuse our exiting **contact** class. Since it already has the properties and methods to handle a personal contact, we can utilize it to store that information and the phone number and only add the new features required for a business contact. Fortunately, inheritance in OOP was devised for exactly these kinds of situations. It allows us to define a new class as an extension of an existing class. The new class is a subclass of the existing class, and the existing class is the baseclass (also called the "superclass") of the subclass.

We are going to create a new subclass of the **contact** class called **businessContact**. Here is its definition:

```
classdef businessContact < contact
  properties
    companyname
    fax
  end
  methods
    function obj = businessContact(cname, contactobj, f)
      if nargin > 0
        obj.companyname = cname;
      end
      if nargin > 1 && ~isempty(contactobj)
        obj.lastname = contactobj.lastname;
        obj.firstname = contactobj.firstname;
        obj.middlename = contactobj.middlename;
        obj.phonenumber = contactobj.phonenumber;
      end
      if nargin > 2
        if ~ischar(f)
          error('string expected for fax number');
        end
        obj.fax = f;
      end
    end
    function obj = set.fax(obj,value)
      if ~isempty(value) && ~ischar(value)
          error('string expected for fax number');
      end
      obj.fax = value;
    end
    function disp(obj)
      if isscalar(obj)
        fprintf('  Company: %s\n',obj.companyname);
        if(~isempty(obj.lastname))
           fprintf('  Contact: %s, %s %s\n', ...
                           obj.lastname, ...
                           obj.firstname, ...
                           obj.middlename);
        end
        fprintf('  Tel:     %s\n',obj.phonenumber);
        if ~isempty(obj.fax)
           fprintf('  Fax:     %s\n',obj.fax);
        end
      else
        fprintf('array of businessContacts\n');
      end
    end
  end
end
```

The first thing to notice is how the baseclass-subclass relationship is specified. No new keyword is required. Instead, simply using the **<** operator between the name of the new class and its baseclass does the trick!

The rest of the class definition looks familiar. In addition to the **companyname** property, we have added a property to hold a fax number. The subclass constructor takes three input arguments.

An important behind-the-scenes action of the constructor for any subclasses is that MATLAB calls the constructor of its baseclass with no arguments right before executing the first line of the subclass constructor. You can override this behavior by calling the baseclass constructor yourself with arguments. The required syntax is

```
obj = obj@SuperClass(ArgumentList);
```

where **obj** is the object to be created and **SuperClass** is the name of the baseclass. Note that the superclass constructor must be called before the object is ever used (e.g., before any of its properties are accessed). Also, the call cannot be conditional (cannot be inside an if-statement).

The rest of our **businessContact** class consists of a **set** function to overload the assignment operator for the **fax** property and a new **disp** function. Similarly to the **phonenumber** property of the **contact** class, we want to protect the **fax** property from an assignment of anything other than a string. The **disp** function we have written for **businessContact** implements a different policy than the one we wrote for **contact**. It considers the company name as the most important property, so it prints it first.

This simple example illustrates the basic features of inheritance. All the properties of the baseclass exist in the subclass. You can safely overload existing methods by creating a new function with the same name as it exists in the baseclass. MATLAB will always call the correct method. For example, **businessContact** object will use their own **disp** function. On the other hand, if

`disp` were not defined in the `businessContact` class, MATLAB would simply call the one defined in the baseclass, that is, the `contact` class.

It's time for another test drive:

```
>> b = contact('Schwarzenegger','','Arnold','555 123-
4567')

b =
  Name:     Schwarzenegger, Arnold
  Tel:      555 123-4567

>> c = businessContact('Terminator Inc.',b,'555 987-
6543')

c =
  Company: Terminator Inc.
  Contact: Schwarzenegger, Arnold
  Tel:      555 123-4567
  Fax:      555 987-6543
```

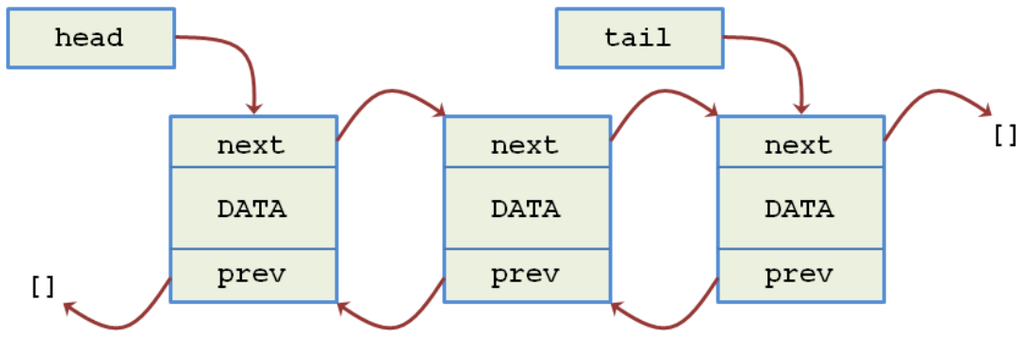
Flawless! Impressed? Well you should be impressed by Object-Oriented Programming, but before you get too awestruck by our programming abilities, you should know that we made a few errors while writing these class definitions (OK, maybe a lot of errors). We found them and correct them by using the debugger—the same debugger that we introduced in Chapter 2, in section Programmer's Toolbox under Debugging. This is very good news, because you don't have to master a new debugger when you employ Object-Oriented Programming. You can use the same debugger to set break points inside methods, that you have been using for normal (non-class) functions. You can watch what happens by executing the code step-by-step, while looking at the workspace and generally do all the things that you can do for any other function.

Now that we have these two nice classes to store contact information, we would like to organize our contacts into something like a phone book. What data structure would be a good fit? A simple vector would not be the best choice because we want to keep the contacts ordered alphabetically. While it is possible to accomplish that with a vector, inserting a new element somewhere in the middle or removing one would mean moving a lot elements of the vector which is inefficient. A very nice and flexible data structure that fits the bill is called the **linked list**.

Linked list

A linked list consists of set of individual elements linked into a chain. Each element stores some kind of data, for example, a `contact` object in a data field, and it has in addition a field typically called **next** that points to the next element in the list. The next field of the last element of the list is set to 0 in most languages. In MATLAB it is an empty array. The next field makes it easy to traverse a list. You start at the beginning (the **head** of the list using computer science terminology) and follow the next field to move from element to element until you encounter the element with a null next field. This is the last element of the list called the **tail**.

A variant of the linked list is the **doubly linked list** whose elements have an additional field typically called **prev** that points to the previous element of the list. Here is a visual depiction of a doubly linked list of three elements:



The labels `next`, `DATA`, and `pref` depict fields of individual elements on the list. Each box with these three labels depicts one element of the list. The element at the left is the head of the list. The element at the right is the tail. The boxes labeled `head` and `tail` are *not* the head and tail of the list. In fact they

are not part of the list at all but instead are merely pointers to the head and tail of the list.

The **next** field has a special data type. In most programming languages, for instance in C++, **next** is a pointer. The value of a pointer is an address in memory that specifies where the object pointed to by the pointer is located. The MATLAB concept closest to a pointer is a cell as explained before. While one might be tempted to try to implement a linked list with cells, any such attempt would be dismal failure. The fundamental problem is that, as we pointed out here in Chapter 2, in the section entitled Data Types, MATLAB does not allow more than one cell to point at the same object.

Fortunately, while cells won't work, MATLAB provides a special built-in class called **handle** that works very well for our purposes. A **handle** object represents an **object reference**, which holds information about the object including the address of the object, also known as a pointer. If you copy a handle object, for example,

```
handle2 = handle1;
```

the actual object is not copied and both variables now refer to the same object. This is a very important distinction between handles and regular variables. Recall that, when a function is called, all arguments are passed by value, meaning that a copy of the value to be passed to the object is created and pushed on the call stack. The argument inside the function will behave just like any other local variable. When the function returns, all local variables and input arguments are removed. This is true not just for scalars or arrays, but for objects as well. You cannot modify an object by passing it to a function through an input argument because the function works only on a copy of the object and not the original one. The only approach you can use to modify an object by using a function is to send a copy of the object to the function through an input argument, and then copy a value from an output argument into the object.

Handles, on the other hand, work differently. When you pass a handle to an object as an input argument, the copy that is created on the stack and that is stored in a local variable inside the function still refers to the original object. Hence, any modification to the object inside the function will be visible from the outside as well.

Note that the **handle** class is an **abstract class**. An abstract class is a class for which you are not allowed to create instances of it, i.e., you cannot created **handle** objects. However you can define subclasses of the class **handle**, and you can create instances of those subclasses. Let's use this idea to create a doubly linked list where the elements are kept continually in order, meaning that their DATA fields increase (or stay the same) as we go from one link to the next. Here is a partial definition of the class that corresponds to the elements of the list:

```
classdef orderedNode < handle

  properties
    id
  end
  properties (SetAccess = private)
    prev
    next
  end
  methods
    function node = orderedNode(n)
      node.id = 0;
      if nargin > 0 node.id = n; end
    end

    function disp(node)
      if isscalar(node)
         if ~isempty(node.id) disp(node.id); end
            disp(node.next);
         else
            disp('Array of ordered list nodes');
         end
      end
    end
  end
```

First of all, notice that our **orderedNode** class is derived from the **handle** class. We have three properties: **id**, **next** and **prev**. Notice how the latter two are grouped together and have their **SetAccess** attribute set to **private**. What does that mean?

The **prev** and **next** properties—you guessed it—will be used to organize our ordered doubly linked list. We do not want the user of our class to modify those properties. We will enforce the correct behavior of our class, and to do that these two properties are of crucial importance. Setting the access to a property to be **private** means that it can be modified only from within the class. So the methods of the class can freely manipulate them, but no code outside of this class can modify them. They are still visible from the outside and can be read. If we wanted to avoid having them be visible from outside, we could have set the **GetAccess** attribute to private as well. Note that these two attributes default to **public**, meaning that properties can be accessed and modified from outside the class at will unless we change their attributes in the **classdef** file as shown above.

What is the role of the **id** property? We intend this class to be reusable by inheritance. In fact, right now it does not have a placeholder to store data other than the properties for organizing the list, so it would not be very useful. The **id** property serves two purposes: first, it will be the basis for the default ordering behavior of the list and second, it will be used in the **disp** function to provide something to print on the screen. Both the ordering and the display functionality are expected to be overridden by subclasses.

The two methods of the class so far are a constructor that does nothing but set the **id**, and the **disp** method that prints the **id** and calls **disp** recursively on the next element of the list. In effect, it displays the entire list starting from the current element all the way to the last. (Note that MATLAB limits the maximum number of recursions to 500 by default, so **disp** can only handle lists of 500 elements or less. If that is a problem, we can either increase the recursion limit or simply change **disp** to use a while-loop instead.)

The most important functionality provided by the list, i.e., the **orderedNode** class, is to create and maintain a list of elements in an ordered fashion. For that we need two functions, the **insert** and **remove** methods:

```
classdef orderedNode < handle
...
  methods
    function insert(node, head)
      if isempty(head)
         error('no list provided');
      end
      cur = head;
      last = [];
      while ~isempty(cur) && node > cur
         last = cur;
         cur = cur.next;
      end
      if node == cur
         error('Node already in the list');
      end
      if isempty(last)
         node.next = head;
         node.prev = [];
         head.prev = node;
      else
         last.next = node;
         node.prev = last;
         node.next = cur;
         if ~isempty(cur)
            cur.prev = node;
         end
      end
    end
  end
...
end
```

The **insert** function takes a node and inserts it into the list whose first element is **head**. Both of these arguments are assumed to be of **orderedNode** type. After checking that there was a list provided as an input argument, it is the task of the while-loop to find where in the list the new node needs to be inserted. This is somewhat tricky, since we need to prepare for three different

cases: the insertion place is at the beginning of the list, at the end of the list or anywhere else. The while-loop tracks two variables **last** and **cur**. The last variable designates the node in the list that the new node needs to follow, while **cur** is what the new node needs to precede. The loop ends when we reach the end of the list or when the **cur** node becomes smaller than or equal to the new node. The latter condition is checked by the following statement:

```
node > cur
```

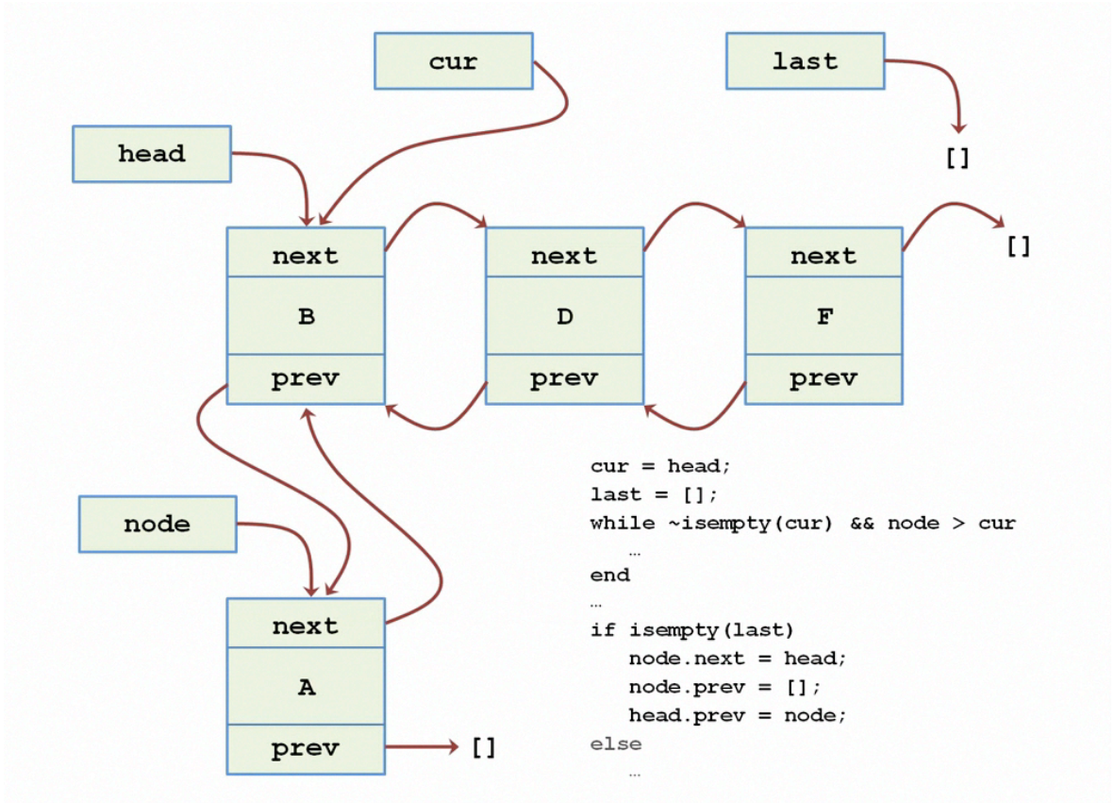
How can MATLAB compare two **orderedNode** objects? Remember that **orderedNode** is a subclass of the **handle** class. MATLAB does support the **isequal** function or **==** operator for handles. They return 1, meaning true, if the handles refer to the same object and 0, meaning false otherwise. However, the result of the **>** or **<** operators on handles is arbitrary and does nor depend on the actual objects. Fortunately, we can change that behavior using operator overloading. Here is the method we need to add to our **orderedNode** class:

```
function t = gt(a,b)
    t = (a.id > b.id);
end
```

The **gt** function implements the **>** operator for our class. As we said previously, we will use the **id** property simply for comparison (ordering) by default, but we expect that subclasses will override this behavior.

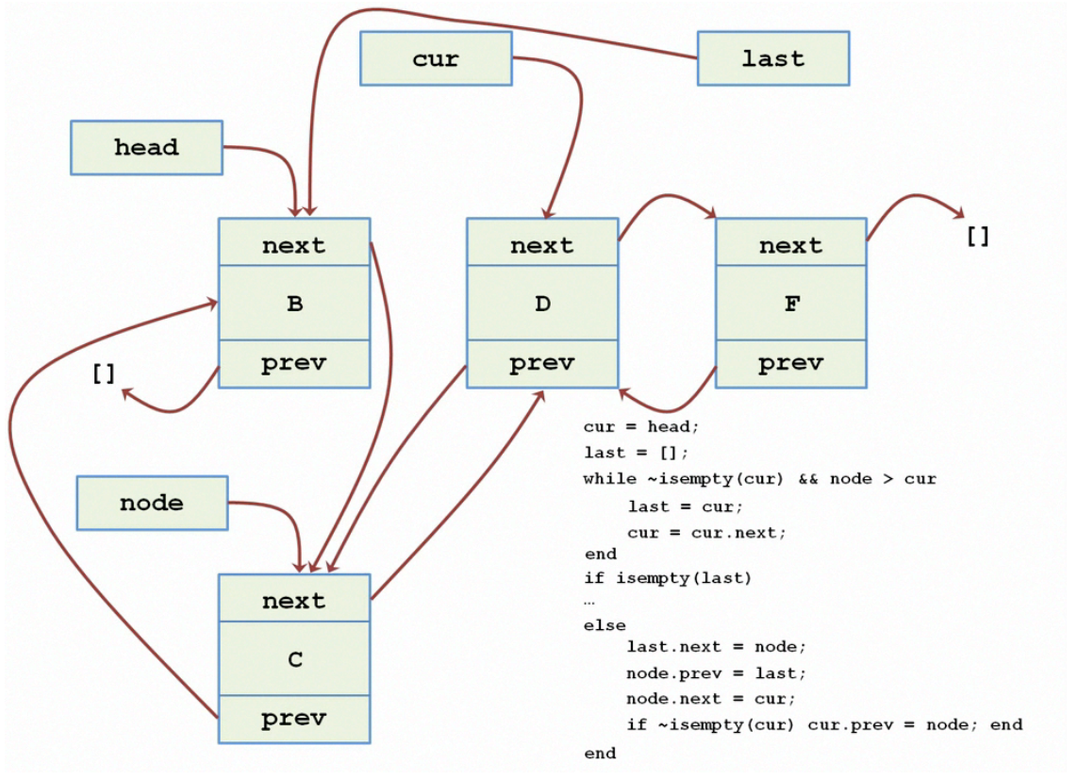
Now, let's get back to the insert method. At the end of the **while**-loop, we know where to insert the new node in the list. First, we need to check whether the node is already in the list. If it is, inserting it one more time would cause problems, so we do not allow it. Now, we are ready to make the insertion. If the **last** variable does not refer to an object, that means that the new node will go at the beginning of the list. We set up its **next** variable to refer to the **head**, the previous first item of the list. Its **prev** will refer to nothing because it is the new first element. Finally, the **prev** property of the **head** object needs to refer to the new node. Check out Movie 3.1.

Movie 3.1 ***Inserting at the beginning of a doubly linked list***



On the other hand, if the **last** variable does refer to an object, we need to insert the new object right after it. So, we set the **next** property of **last** to the new node, and the **prev** property of the new node to **last**. Now we need to check whether we are at the end of the list or not. If the **cur** variable does not refer to anything, we reached the end and we are done. Otherwise, we need to set the **prev** property of **cur** to the new node. See Movie 3.2 for an illustration.

Movie 3.2 ***Inserting in the middle of a doubly linked list***



Now, we are ready to try our new class:

```
>> a = orderedNode(12)
a =
    12
>> b = orderedNode(4)
b =
     4
>> c = orderedNode(6)
c =
     6
>> d = orderedNode(1)
d =
     1
>> a > d
ans =
     1
>> b > c
ans =
     0
```

We created four nodes with various `id`-s. We spot tested that the comparison operator indeed works. Now, let's try to create a list of these objects. We'll start by making the object `b` as the head of the list.

```
>> insert(a,b)
>> b

b =
     4
    12
```

We inserted object `a` into the list that starts at `b`. As you can see, the list `b` has now two elements in ascending order. This tested the case when the new object is added at the very end of the list. Let's add another object:

```
>> insert(c,b)
>> b

b =
     4
     6
    12
```

We inserted `c` into the `b` list. This tested the case when the new object goes somewhere in the middle of the list. Finally, let's insert another object:

```
>> insert(d,b)
>> b

b =
     4
     6
    12
```

What has just happened??! Where is `d`? Well, `d` is the smallest of the four nodes, so it got inserted at the beginning of the list. So `b` is no longer the head of the list; `d` is. Let's see:

```
>> d

d =
     1
     4
     6
    12
```

You may be wondering why we did not change **b** to refer to the new beginning of the list. Why don't we just update **head** in the insert method like this:

```
head = node;
```

We could not. The variable **head** is a local variable of the **insert** function. Therefore, any change to the variable itself will remain inside the function and will be lost upon return from it. There is a very important distinction between the object that **head** refers to and the **head** variable. We can use the local variable **head** to update properties of the object it refers to (since it is a handle object). But the **head** variable itself is a local variable, so the only way to change it outside the function is to return its new value as an output argument. Then the caller of the function can decide to update the variable that was passed in as **head** to refer to the new beginning of the list. Here is the updated **insert** function:

```
classdef orderedNode < handle
...
  methods
    function head = insert(node, head)
      if isempty(head)
         error('no list provided');
      end
      cur = head;
      last = [];
      while ~isempty(cur) && node > cur
         last = cur;
         cur = cur.next;
      end
      if node == cur
         error('Node already in the list');
      end
      if isempty(last)
         node.next = head;
         node.prev = [];
         head.prev = node;
         head = node;
      else
         last.next = node;
         node.prev = last;
         node.next = cur;
         if ~isempty(cur)
            cur.prev = node;
         end
      end
    end
...
end
```

The **insert** method now returns **head**. It is changed only if the new node is inserted at the beginning of the list. Otherwise, the method simply returns the **head** it got as an input argument. Let's try the new version:

```
>> a = orderedNode(12)

a =
    12

>> b = orderedNode(4)

b =
     4

>> c = orderedNode(6)

c =
     6

>> d = orderedNode(1)

d =
     1

>> b = insert(a,b)

b =
     4
    12

>> b = insert(c,b)

b =
     4
     6
    12

>> b = insert(d,b)

b =
     1
     4
     6
    12
```

Note that **b** changed only after the last insertion since that was the only time when the new node was inserted at the beginning of the list.

Now let's see how we can remove a node from the list.

```
classdef orderedNode < handle
...
  methods
    function remove(node)
      if isempty(node)
         error('invalid node handle');
      end
      if ~isempty(node.prev)
         node.prev.next = node.next;
      end
      if ~isempty(node.next)
         node.next.prev = node.prev;
      end
      node.next = [];
      node.prev = [];
    end
  end
...
end
```

As you can see, removal is much simpler than insertion. First, we check that the node to be removed does indeed exist. Then if the node is not the first element of the list, we make the previous node's (**node.prev**) **next** property jump over the node, that is, we make it refer to **node.next**. If the node is not the last element, we make the next element's (**node.next**) **prev** property jump over, that is, we make it refer to **node.prev**. Finally, we reset the node's own properties so that they do not refer to anything.

Let's try it out, shall we?

```
>> b

b =
     1
     4
     6
    12

>> remove(a)

>> b

b =

     1
     4
     6

>> remove(d)
>> b

b =

     1

     1
```

Oops, what happened there? It does not look like it, but the **remove** method actually worked correctly. Remember, **d** was the first element of the list and that is what **b** referred to since **b** is head of the list. So, when the **remove** method got rid of **d** from the head of the list, the object that **b** also refers to was orphaned. So while the object was removed correctly, our variable representing the list, got messed up as a side effect. The list is still OK, but it is hard to get to:

```
>> c

c =
     6

>> c.prev

ans =
     4
     6

>> c.prev.prev

ans =
     []
```

The original **b** (with **id** 6) is no longer directly accessible because **b** was reassigned to refer to the head of the list. The object that **c** refers to is the last element, and the one before it is the first element of the list at this point. So, the list is indeed fine, but we do not have a direct reference (handle) to its beginning.

How to fix this problem with the **remove** methods is a question of taste. Clearly, the method needs to return the potentially new head handle as an output argument just as **insert** does. But **remove** does not take the head handle as an input argument (it does not need it). We either change the method to take the head as an argument, or we follow the list backwards to the head and return it that way. We choose the former to make **remove** symmetrical with **insert**. Here is the new and improved remove method:

```
classdef orderedNode < handle
...
  methods
    function head = remove(node,head)
      if isempty(node)
         error('invalid node handle');
      end
      if ~isempty(node.prev)
         node.prev.next = node.next;
      end
      if ~isempty(node.next)
         node.next.prev = node.prev;
      else
         head = node.next;
      end
      node.next = [];
      node.prev = [];
    end
  end
...
end
```

Looks like we have successfully fixed that problem:

```
>> b = remove(a,b)

b =
     1
     4
     6

>> b = remove(d,b)

b =
     4
     6
```

We are almost done with the **orderedNode** class. However, there is one more danger lurking out there. Just as an object can be created, it can also be destroyed. The **delete** method is reserved just for that. When you no longer need an object, you can get rid of it, so it does not waste memory. Consider this:

```
>> c

c =
     6

>> c.prev

ans =
     4
     6

>> b

b =
     4
     6

>> delete(b)
>> b

b =

Invalid or deleted object.

Error in orderedNode/disp (line 70)
                if ~isempty(node.id)

>> c

c =
     6

>> c.prev

Invalid or deleted object.

Error in orderedNode/disp (line 70)
                if ~isempty(node.id)
```

The first error message after we deleted **b** and tried to access it is fine. We are not supposed to use **b** once we deleted it. In fact, it is a good practice to eliminate the variable as well:

```
>> clear b
```

The second error message, however, is our fault. Deleting a node left our list in a bad shape. How can we prevent it? Somehow we need to intercept that **delete** function call. Fortunately, MATLAB allows us to do that. Just as we can create our own constructor, which gets called when a new instance of a class is to be created, we can write our own **destructor** that will be called when an object is about to be deleted. Here is what we need to add to our class definition:

```
classdef orderedNode < handle
...
  methods
    function delete(node)
      remove(node);
    end
  end
...
end
```

A destructor function must be named **delete**, and it takes exactly one input argument, which is the object to be deleted, and it returns no output arguments. In our case, we simply call the **remove** method to make sure that our list remains intact after the given object is deleted. Notice that we do not bother with passing the head of the list to the **remove** method. We do not need to since we are not using what the methods returns anyways. Note that calling **remove** in the destructor in this case can be considered error handling. The user is not supposed to delete objects that are still part of a list. The correct usage is removing the object first and then deleting it. The difference is subtle, but important. The function **remove** deletes an object from our linked list; the function **delete** deletes is from the memory. If the user deletes the first element of the list without removing it first, we have no possible way of providing a variable that refers to the new head of the list since the destructor cannot return an argument.

Contact list

Now that we have the **contact**, **businessContact**, and **orderedNode** classes, we are ready to create our phone book, which is the contact list that we set out to make. Get ready to be impressed! The only change we need to make to the **contact** class is the very first line:

```
classdef contact < orderedNode
```

We simply add the **orderedNode** class as a base class of the **contact** class. Check this out:

```
>> a = contact('Simpson','Bart','','555-1234')

a =
  Name:     Simpson,  Bart
  Tel:      555-1234

>> b = businessContact('Apple Inc.',[],'555-4321')

b =
  Company: Apple Inc.
  Tel:
  Fax:      555-4321

>> c = contact('Wolverine')

c =
  Name:     Wolverine,
  Tel:

>> contacts = a;

>> contacts = insert(b,contacts);

>> contacts = insert(c,contacts);
```

```
>> contacts

contacts =
  Name:    Wolverine,
  Tel:

>> contacts.next

ans =
  Company: Apple Inc.
  Tel:
  Fax:     555-4321

>> contacts.next.next

ans =
  Name:    Simpson,  Bart
  Tel:     555-1234
```

Voila. We have the list of contacts. Notice that the **insert** function and the **next** property (as well as everything else from the **orderedNode** class) are now automatically available in both the **contact** and **businessContact** object. This is the beauty of inheritance. It took an effort to create a nice linked-list data structure, but now it is freely reusable in other classes as we wish.

Since the **disp** function of the contact class prints out only the current contact information, not the entire list (which is probably what we want to do since the list can get very long), let's us write a simple method for the contact class that prints out the list itself:

```
function dispList(node)
   while ~isempty(node)
      disp(node);
      node = node.next;
      if ~isempty(node)
         fprintf('|------------------------|\n');
      end
   end
end
```

We simply call the **disp** method for the current node and continue down the list inside the while-loop until the end of the list. Now let's see our list:

```
>> dispList(contacts)
  Name:    Wolverine,
  Tel:

|------------------------|
  Company: Apple Inc.
  Tel:
  Fax:     555-4321
|------------------------|
  Name:    Simpson,  Bart
  Tel:     555-1234
```

Unfortunately, we are not quite done yet. Can you see the problem with our contact list? It is not ordered alphabetically. The insert function put each and every new object at the beginning of the list. Why? It is because the **orderedNode** list only implements a default ordering done by the **id** property. Since the **id** defaults to 0, every contact in our list has the same **id**, hence, new nodes will go to the beginning of the list. It is the responsibility of the subclass to provide the information about ordering.

It is actually quite simple to do; the subclass needs to overload the **>** operator. In our case, it does get a bit complicated because string comparison in MATLAB has somewhat limited built-in support and we have to combine several properties in the decision (**lastname**, **firstname**, **middlename**, **companyname**). So, we have a little bit of work to do. Let's start by writing our own string comparison function. The built-in MATLAB **strcmp** function only tells whether two strings are equal or not. It does not specify which one is "bigger," that is, which one comes later in alphabetical order. So, here is our own function to compare two strings:

```
function a = strgt(s1,s2)
% a == 1 when s1 > s2
% a == 0 otherwise

    a = 0;
    s1 = upper(s1);
    s2 = upper(s2);

    for ii = 1 : min(length(s1),length(s2))
        if s1(ii) > s2(ii)
            a = 1;
            return;
        elseif s1(ii) < s2(ii)
            return;
        end
    end
    if length(s1) > length(s2)
        a = 1;
    end
end
```

Our **strgt** function takes two string inputs and returns **1** if the first one is greater than the second and **0** otherwise. We set the default answer to be false. Then we convert the strings to upper case, since we want our comparison to be case insensitive. The **upper** function comes with MATLAB.

The for-loop will run for the length of the shorter of the two strings. We compare the corresponding characters in the two strings one by one. The very first character that is not the same in the two strings decide the outcome of the function, so we can immediately return. If we exit our the loop and we are still in the function means that one of the strings starts with the other. Then we simply say that the longer of the string is greater than the other. If they are of equal length, that means they are the same string, so we return 0.

Let's play with the function a bit:

```
>> strgt('abc','xyz')
ans =
     0

>> strgt('xyz','abc')
ans =
     1

>> strgt('ABC','def')
ans =
     0

>> strgt('abc','DEF')
ans =
     0

>> strgt('abc','abc')
ans =
     0

>> strgt('abcd','abc')
ans =
     1
```

While this was not an exhaustive test, it seems that our function works as expected. The question is how to "package" this function. Our **contact** class will rely on it for ordering objects, but it cannot really be a regular method of the class since it does not use objects at all. If we keep it in a separate file, then we have to remember to keep it together with the class definition for the **contact** class. It is usually a good practice to try to keep code that a class uses together with the class even it is not a regular member method. Fortunately, most object oriented languages including MATLAB provide an elegant solution. Class definition can include so called *class methods* that are functions that belong to the class and not to individual objects. The syntax to define such a beast is as follows:

```
classdef contact < orderedNode
...
  methods (Static)
    function a = strgt(s1,s2)
    % a == 1 when s1 > s2
    % a == 0 otherwise

      a = 0;
      s1 = upper(s1);
      s2 = upper(s2);

      for ii = 1 : min(length(s1),length(s2))
        if s1(ii) > s2(ii)
           a = 1;
           return;
        elseif s1(ii) < s2(ii)
           return;
        end
      end
      if length(s1) > length(s2)
         a = 1;
      end
    end
  end
...
end
```

As you can see, the only thing that needs to be done is to include a separate methods section with the **`Static`** designation. The syntax to call a class method either inside or outside the class is **`classname.methodname(arguments)`**.

Now, we are ready to modify our **`contact`** and **`businessContact`** classes to support ordered lists. First, we need to decide how to order the various contacts. It seems logical to include the following properties in the following order:

`lastname`, **`firstname`**, **`middlename`** for **`contacts`**, and

`companyname`, **`lastname`**, **`firstname`**, **`middlename`** for **`businessContacts`**.

To handle both cases transparently, we'll add a method to the **`contact`** class called **`nameToCompare`** that constructs a single string according to the rule we specified above. Then we override the method in the **`businessContact`** class. That way if we have an instance of either class, we can simply call this method and the correct one will be called automatically. Here they are:

```
classdef contact < orderedNode
...
  methods
    function name = nameToCompare(obj)
      name = [obj.lastname ' ' obj.firstname ' ' ...
                                          obj.middlename];
    end
  end
...
end

classdef businessContact < coontact
...
  methods
    function name = nameToCompare(obj)
      name = [obj.companyname ' ' obj.lastname ' ' ...
                    obj.firstname ' ' obj.middlename];
    end
  end
...
end
```

Notice how we include a space between the various properties. This is an important point. Consider two persons: Joana Co and Ana Cojo. If we did not include a space between the first and last names, both would be turned into "COJOANA" by the **`nameToCompare`** and **`strgt`** methods. In turn, these would be considered equal even though the last name Co comes before Cojo in alphabetical order. The spaces between the various name properties solve this problem.

The only thing left now is to overload the **>** operator in the **`contact`** class. For symmetry, we also overload the **<** operator using the **`lt`** method:

```
classdef contact < orderedNode
...
  methods
    function a = gt(o1,o2)
      a = contact.strgt(o1.nameToCompare(), ...
                        o2.nameToCompare());
    end

    function a = lt(o1,o2)
      a = contact.strgt(o2.nameToCompare(), ...
                        o1.nameToCompare());
    end
  end
...
end
```

We don't even need to touch the **businessContact** class since the **nameToCompare** methods take care of the differences between the contact and **businessContact** classes. Let's try our mini phonebook:

```
>> clear
>> a = contact('Simpson','Bart','','555-1234')

a =
  Name:     Simpson,  Bart
  Tel:      555-1234

>> b = businessContact('Apple Inc.',[],'555-4321')

b =
  Company: Apple Inc.
  Tel:
  Fax:      555-4321

>> c = contact('Wolverine')

c =
  Name:     Wolverine,
  Tel:

>> contacts = a;

>> contacts = insert(b,contacts);

>> contacts = insert(c,contacts);

>> dispList(contacts)

  Company: Apple Inc.
  Tel:
  Fax:      555-4321
|--------------------------|
  Name:     Simpson,  Bart
  Tel:      555-1234

|--------------------------|
  Name:     Wolverine,
  Tel:

>> contacts = remove(b,contacts);

>> dispList(contacts)
  Name:     Simpson,  Bart
  Tel:      555-1234

|--------------------------|
  Name:     Wolverine,
  Tel:
```

Concepts From This Section

Computer Science and Mathematics:

Object Oriented Programming (OOP)
class
instance
object
inheritance
polymorphism
encapsulation
constructor
destructor
operator overloading
object reference
linked list

MATLAB:

`classdef`
property
`private`
`public`
method
class method
`handle`

Graphical User Interfaces

Objectives

A program's user interaction via pictures, mouse clicks, and screen touches is called its Graphical User Interface (GUI). Programming languages rarely have built-in GUI support. Instead, GUIs are programmed by calls to functions provided by the programming environment for that purpose. This section will introduce GUI concepts in an extended tutorial example.

(1) We will study the event-based programming model.

(2) We will learn how to use GUIDE, MATLAB's versatile tool for building an event-based GUI.

(3) We will study the callback function model.

(4) We will learn how to use callbacks to respond graphically to events such as mouse clicks and key presses.

The most widely used programs have excellent Graphical User Interfaces (GUIs) to let the user interact with the software in a natural way. The impression that the GUI makes on the user, known as its "look and feel", is very important. If its GUI is hard to use or visually unappealing, it does not matter how great a service your program provides. Few people will use it.

Widely distributed programs typically interact with their users not only through characters on the keyboard and the screen but also through pictures, mouse clicks, and screen touches. The scheme of interaction that such a program presents to its user is called its **Graphical User Interface** (**GUI**). A GUI is often perceived by the user as the most important aspect of distributed software because it acts as the gatekeeper through which the utility of the program is made available. As you hold this electronic book, you are actually interacting with a GUI that makes the content of the book available to you in a visually pleasing and informative manner. This GUI, just like all other iPad GUIs, is based on multitouch, which is a generic term meaning that input is accepted through

multiple, simultaneous screen touches. More traditional GUIs running on PCs and Macs utilize mouse and keyboard input to let you interact with buttons, menus and other GUI features. Such a feature, which provides output to the user or allows the user to perform input with the mouse or through natural gestures with a touch screen that mimic the use of a mechanical device, is called a **GUI control** or **widget**. The focus of this section is to teach you how to build such traditional GUIs.

Event-based User Interfaces

GUI development has been traditionally operating-system dependent. This is one of the reasons that programming languages typically do not have built-in support for GUI development. Instead, the programmer utilizes an **Integrated Development Environment** (**IDE**), which is a program with its own GUI, function library and set of tools, such as the debugger, that serves as a development environment to create programs (including GUIs) using the given programming language and operating system. A representative example is the Microsoft Visual Studio IDE or the MATLAB environment itself. While many GUIs are operating system dependent, a more recent trend is to try to support GUI development independently of the operating system. For example, Java comes with the Abstract Window Toolkit (AWT) library that works on all operating systems that Java runs on. MATLAB has its own built-in support for GUI development, as we shall see shortly. It also works on all platforms that support MATLAB.

One shared characteristic of these example GUI libraries is the so called **event-based** programming model. During normal operation, the GUI sits idle waiting for events. An event is caused by a user action, such as clicking on a button, pressing a key on the keyboard or even moving the mouse. Each of these actions cause an event that the GUI will react to. How? Each event has an **event handler**, a special function that gets called once the event occurs. The GUI developer can implement her own event handler, also called a **callback**, and register it with the system. When the given event occurs, the user-defined function gets called. In turn, the function can react to this event in any way it pleases. For example, the callback function handling a button press (i.e., a mouse click when the cursor is inside a picture of a button on the screen) may want to update something on the screen, play a sound, or perform a short computation. Lengthy processing tasks should not be performed in callback functions, because the GUI will not handle additional events while a callback function is running and hence, your program may be become irresponsive.

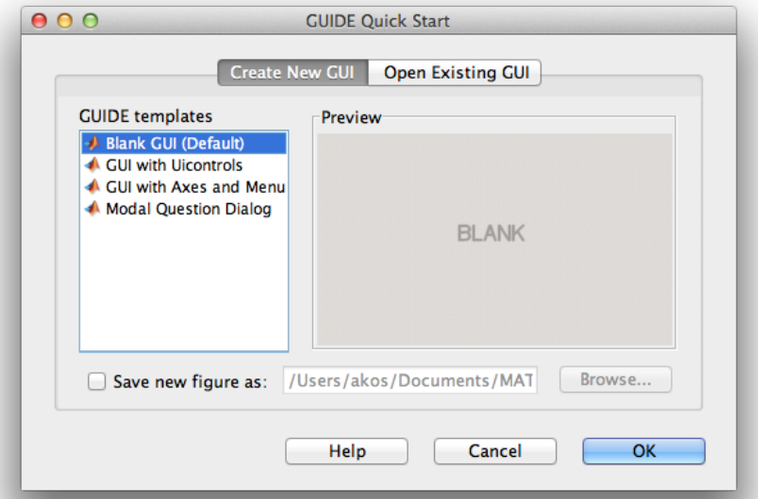
Most GUI development environments and libraries have the same kind of widgets supported as well as similar events. On the other hand, how to create the widgets and how to implement the event handlers are quite different in different environments. The rest of the section will focus on MATLAB GUI development exclusively.

The MATLAB GUIDE

GUIDE is a clever acronym for Graphical User Interface Development Environment. Ironically, it is a MATLAB GUI for designing GUIs in MATLAB. GUIDE lets you create a user interface, place various widgets on it, adjust the properties of them and save your design. It creates two files. The first file is a so-called Fig-file, which is a file whose name has the extension "**.fig**" and which contains the graphical aspects of your GUI: what widgets you have, where they are located within the GUI window, what properties they have, etc. The second file is an M-file. That M-file contains the MATLAB code that initializes the GUI, and it also has placeholders for the callback functions. You can edit this second file by providing the bodies for these callback functions. By writing code in those bodies, you implement the specific actions you want your GUI to take as the user interacts with your software.

Let's create our first MATLAB GUI then, shall we? Open MATLAB and type **`guide`** in the Command Window. You should see the window in Figure 3.8 pop up.

Figure 3.8 *Starting GUIDE*

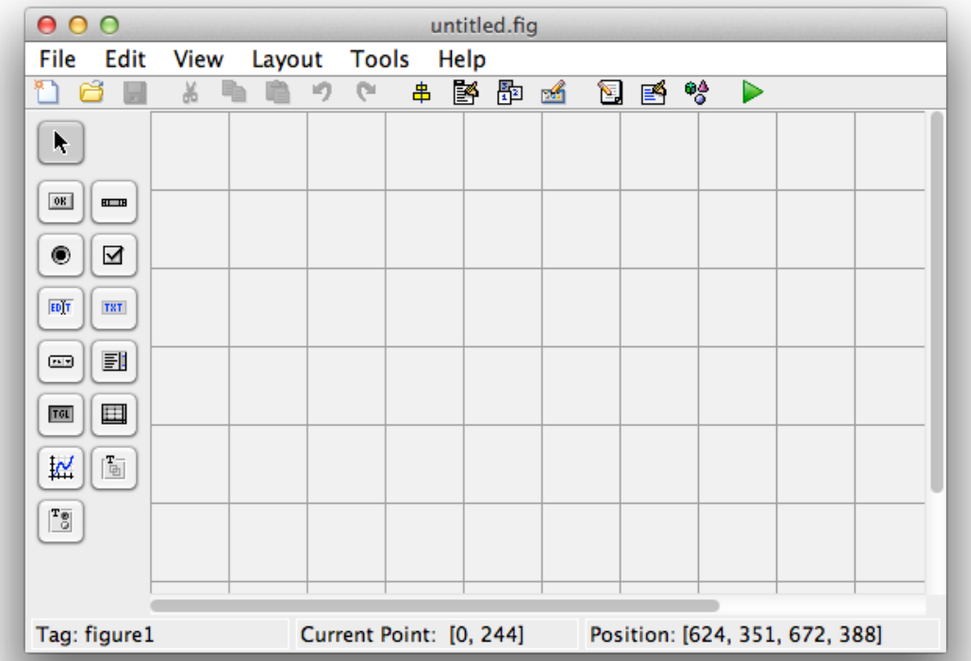


Let's select the Blank GUI option and press OK. Figure 3.9 shows what we get:

This is the main GUIDE window, in which you can design your GUI. On the top you have the usual menu and toolbars. The big gridded area on the right is the "canvas"; this is where you, as the "artist", create your GUI layout using the small buttons shown on the left hand side. These buttons represent the various widgets that you can add to your canvas. Each has a name, but you cannot see those names in this view. Let's make the names visible.

Open the preferences dialog in the File menu and select "Show names in component palette." You now get a better indication of which button corresponds to what kind of widget, as shown in Figure 3.10.

Figure 3.9 *Blank GUI design in GUIDE*

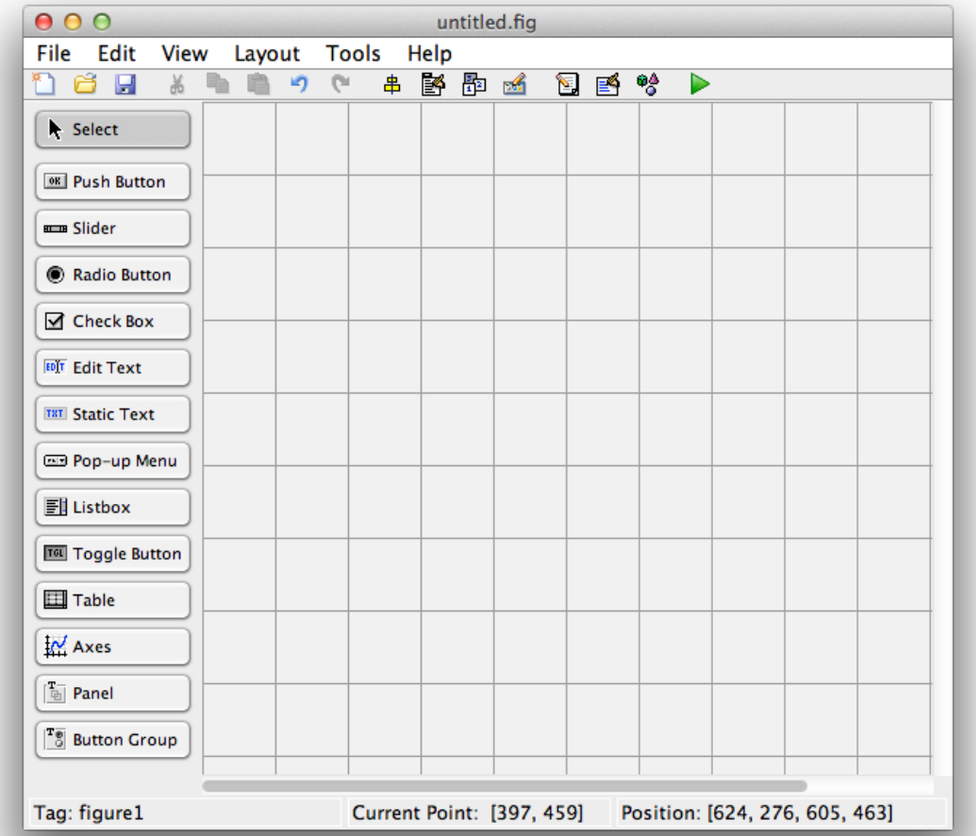


Let's describe the widgets you can put into the canvas on the right when you push the buttons on the left. From top to bottom:

Push Button. It is a standard button with a raised border. Clicking the button calls the callback function. You have used push buttons many times: Whenever a program asks you a question in a dialog box, you typically answer by pressing the OK or the Cancel push button.

Slider. A slider looks like a scrollbar. Its typical use in MATLAB is to change a numerical value in a continuous fashion. You can grab the pad within the slider and drag it, or you can click on the slider's background to cause the pad to jump by a predefined amount. Any change to the location of the pad generates a call to the callback function.

Figure 3.10 Blank GUI design in GUIDE with widget names displayed.



Radio Button. A radio button has two states: on or off. Clicking it toggles the state and calls the callback function. The on state is represented by a solid circle within a larger hollow circle, while in the off state only the hollow circle is shown. Radio buttons are typically grouped together (see Button group below) and if so, there can be only one button in the "on" state at a time. (They are called "radio" buttons because old-fashioned car radios used to have a set of mechanical buttons for selecting a station, and when you pushed one in, the previously pushed button popped out.) When a button in an "off" state is clicked, it is turned on and the current "on" button is turned off simultaneously. With grouped radio buttons, individual callback functions are not used; instead, a single callback function for the button group should be used.

Check Box. The check box is self-explanatory. Like a radio button it has two states, on and off, and the callback function is called whenever it is clicked. Unlike the radio buttons, clicking one checkbox does not automatically uncheck another checkbox.

Edit Text Box. This control allows the user to provide text input using the keyboard. It displays a string and allows you to modify it. When the Enter key is hit, or the user clicks anywhere else on the GUI, the callback function gets called.

Static text. A better name for the static text control would be "label" because that is what it is. It is the way to put textual information on the GUI that never changes. Titles, labels, or any other text that you want on the GUI should be displayed using static text. It has no callback function.

Pop-up Menu. The pop-up menu displays a list of choices when clicked. The user can navigate the list using the arrow keys to change the current selection. Pressing the Enter key or clicking on one of the items picks the given item, hides the menu, shows only the new selection, and calls the callback function. Visually the pop-up menu looks similar to a button, but it has an arrow on the right hand side.

Listbox. The listbox is similar to a pop-up menu in that it also allows you to select an item from a list of choices. The main difference is that multiple choices are visible all the time, so it takes up more room on the GUI. Making a selection can be done by clicking or using the arrow keys and then hitting Enter. The callback function is called when the selection changes.

Toggle Button. The toggle button is similar to a checkbox in that it also has two states: on and off. Whenever the user clicks on it, it changes its state and

causes the callback function to be invoked. The main difference between a check box and a toggle button is their look.

Table. The table control makes it possible to put a spreadsheet into your GUI. The table displays a set of rows and columns of data. You can make the table editable, in which case a callback gets called whenever the user modifies any of the fields.

Axes. Axes are utilized to display images, charts or plots in your GUI. They do not have callback functions.

Panel. Panels are used to group together other GUI controls. A panel is basically a box with a border and an optional title that you can draw around a set of GUI components to indicate that they belong together. Panels are passive components with no callbacks.

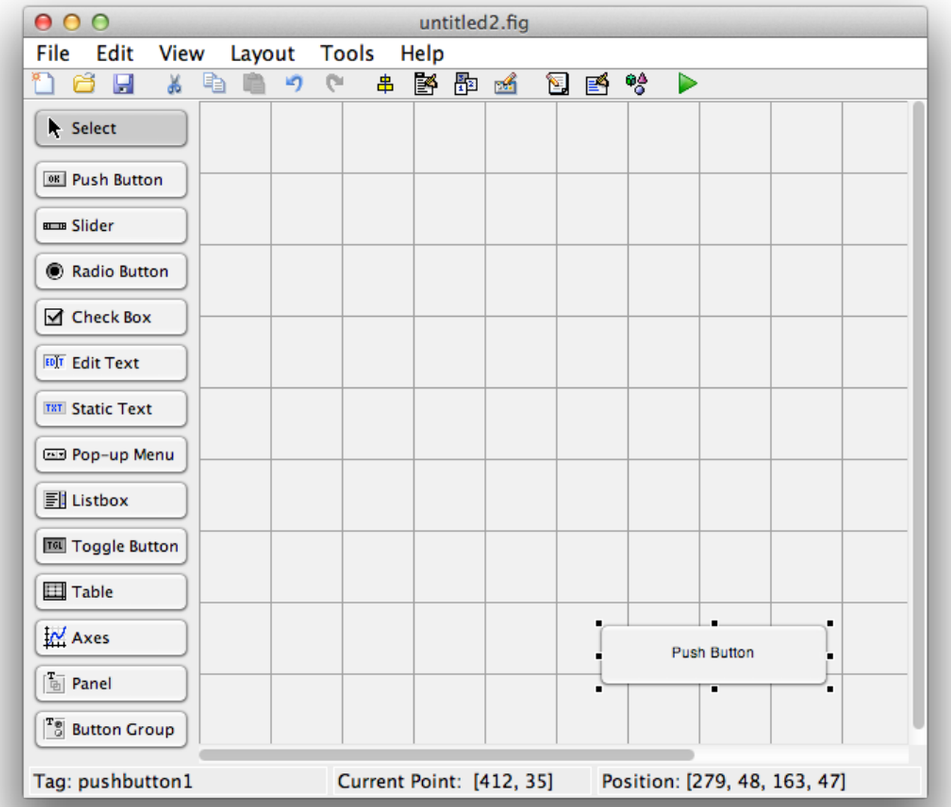
Button Group. A button group is used to group together radio buttons. It is similar in appearance to a panel. However, if you add radio buttons to a button group, those buttons will behave in a synchronized manner. Only one radio button can be on at any one time. To handle the changing state of the radio buttons, the callback function of the button group must be used. It is important to create the button group *first* and then add the radio buttons to it and not the other way around.

This nice variety of GUI components makes it possible to create quite complex and nice looking GUIs with MATLAB. Let's create one then, shall we?

Simple Example GUI

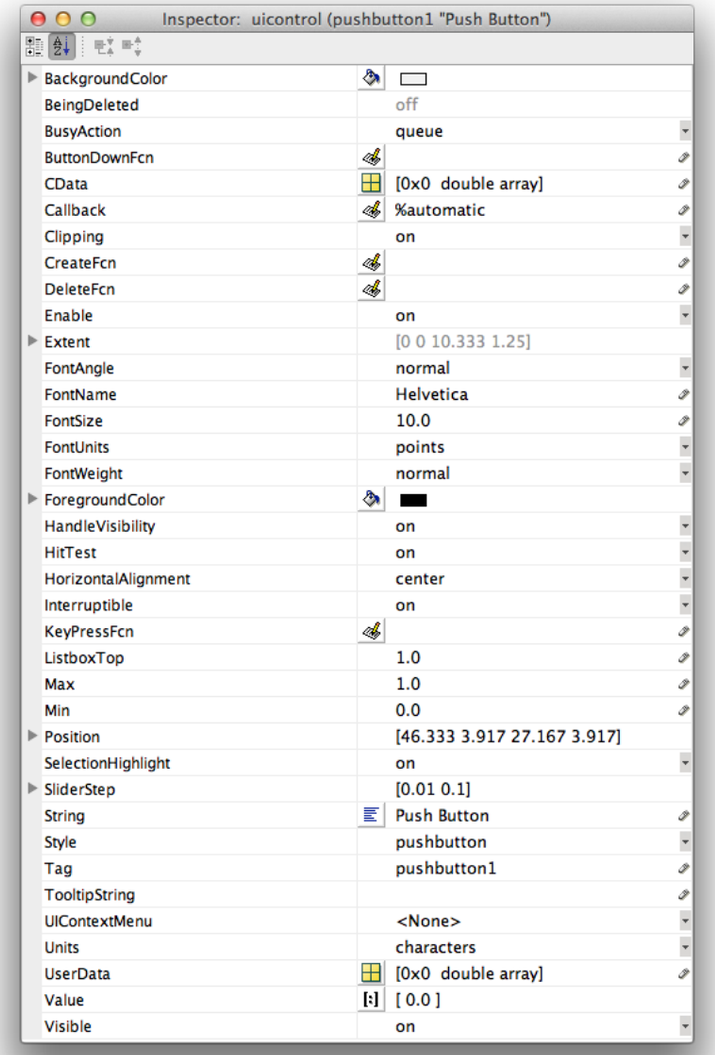
Open GUIDE and create a new blank GUI as described previously. Click on the Push Button control. Now click and drag a rectangle in the canvas. Figure 3.11 shows what you should see.

Figure 3.11 *Adding a push button to our GUI design*



Now, double click on the new Push Button (or alternatively, right click and select Property Inspector, from the context menu). A new window comes up that looks like the in Figure 3.12. What we are seeing is the properties of our newly created Push Button.

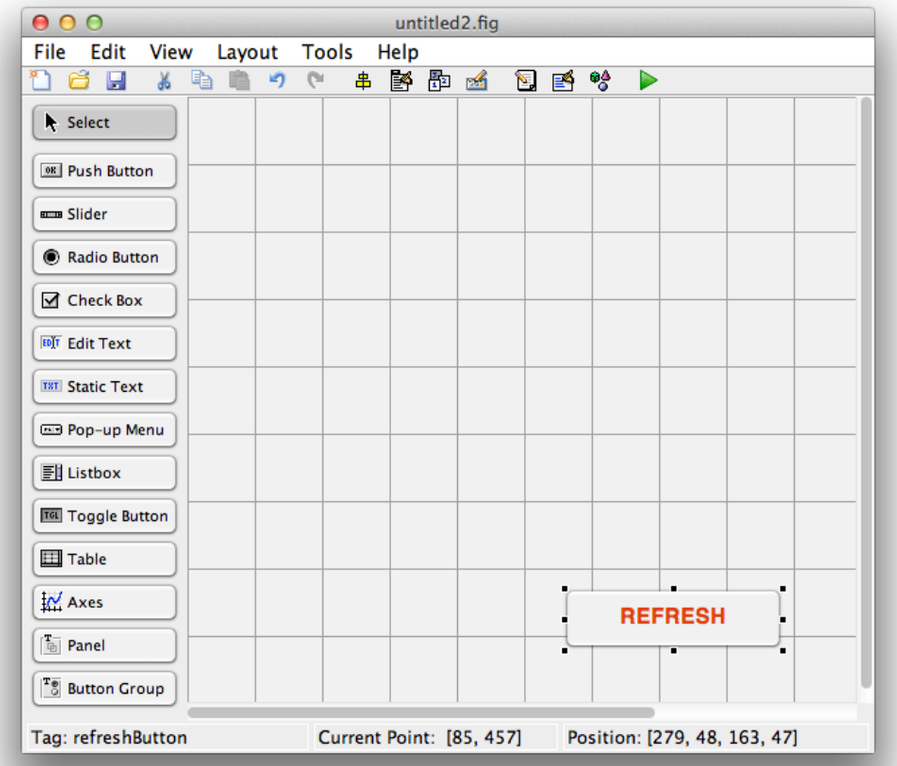
Figure 3.12 *Property inspector*



The terms objects and properties should be familiar to the reader by now. Indeed, GUI controls are similar to objects in the OOP sense. They have properties and are typically accessed through handles from MATLAB code. Unfortunately, the MATLAB GUI infrastructure is not implemented using MATLAB's own OOP concepts, so these similarities end at the terminology. This can be somewhat confusing at times.

Let's modify a few properties. Change the **`String`** property to REFRESH, the **`FontSize`** to 18.0, the **`FontWeight`** to bold, the **`ForegroundColor`** to red, and the **`Tag`** to "refreshButton". Once all this is done, Figure 3.13 shows what we get:

Figure 3.13 *Push button with modified properties*



The only property out of the ones that we changed that is not self explanatory is the **`Tag`**. The **`Tag`** is the name that we can use to access this push button object in our MATLAB code, as we shall see momentarily. Let's click the save button on the toolbar of GUIDE or select the Save command from the File menu. In the dialog box provide "smooth" for the filename (leaving the extension to remain as .fig), pick the folder in which you want MATLAB to be when you use test the GUI, and save it there. When you click OK, MATLAB

generates two files: **smooth.fig** and **smooth.m**. MATLAB opens up the M-file immediately.

The fig-file contains all the necessary information of our GUI including the kinds of GUI controls we have, their properties including where they are on the canvas, how big they are etc. Most of the M-file is generic: it would look the same for any GUI. The only specific information it has is the name **smooth** that appears in a few places and the very last function:

```
function refreshButton_Callback(hObject, eventdata, handles)
```

You guessed it: it is the callback function that will be invoked when somebody clicks the button in our GUI. Let's add this line to the body of `refreshButton_Callback`:

```
fprintf('You have just pressed the REFRESH button!\n');
```

In the Command Window type in **smooth** (or click the green arrow in the GUIDE toolbar) and watch the GUI popup as shown in Figure 3.14.

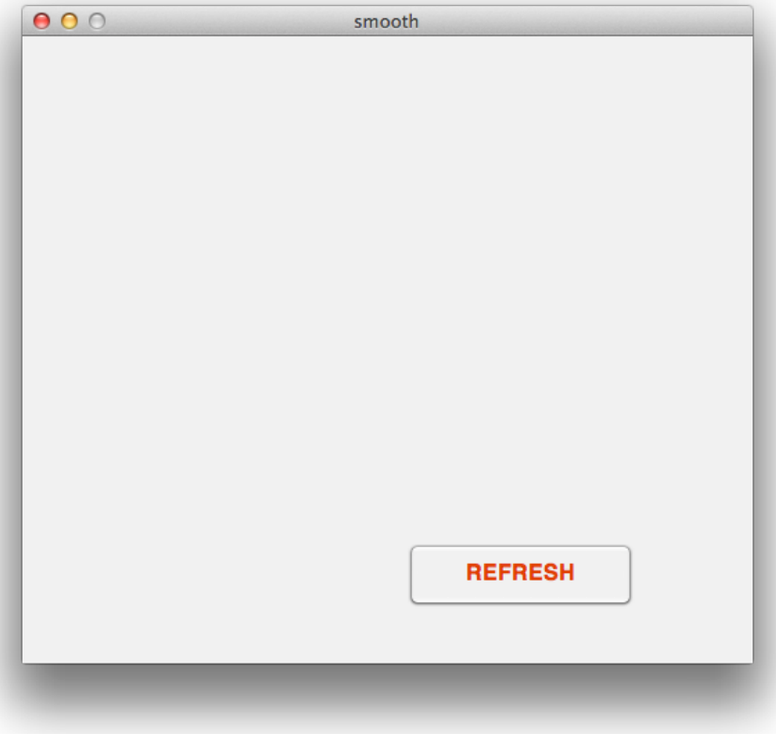
Now, click the REFRESH button. This is what you should see in the Command Window:

```
>> smooth

You have just pressed the REFRESH button!
```

Congratulations! You have just created your very first operational GUI in MATLAB.

You may be wondering what all that cryptic code is in the smooth.M-file. It is the implementation of the function called **smooth** that is the main function of our GUI-based program. The good news is that we do not need to understand most of what is in that file. MATLAB has lots of code that works behind the scenes to make GUIs work. Only a tiny fraction of it needs to be in the M-file where the callback functions reside, i.e., in **smooth.m** in this case. As we revise this file to build more and more complex versions of this program, we will explain the parts that you *do* need to understand in order to create more complex and more useful GUIs.

Figure 3.14 *Our first running program with a GUI*

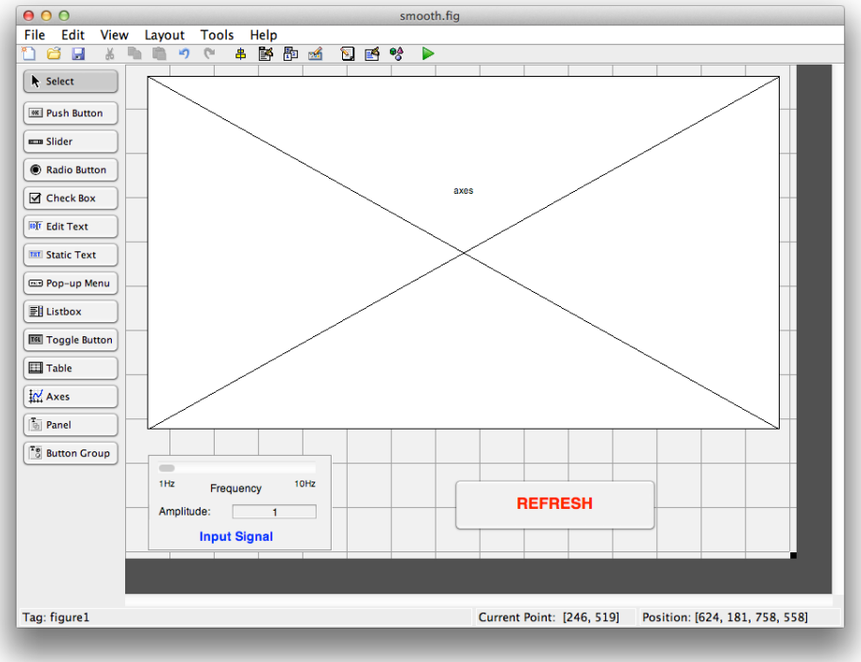


Simple Plotting Example

You may have been wondering about why we named our GUI "smooth" and why we labeled the button "REFRESH." What we are building here is a nice plotting program with a fancy GUI. It is called **smooth** because it will demonstrate how to generate a smoother version of a ragged (noisy) signal. And the REFRESH button will be used to redisplay the plot once the user has changed its parameters. So, let's continue.

The most important part of our GUI will be the plot area. Let's add an Axes widget and resize it so that it takes up the top two thirds of the window.

Figure 3.15 *Multiple widgets in the GUI design*



Then let's add a Slider, an Edit Text Box, a Panel, and a few Static Text Boxes. After we have laid them out where we want them, we fire up the Property Inspector and change the Tag property of each of these objects as follows: Axes: "axes", Slider: "freqSlider" and Text Edit Box: "amplEdit". Let's make the layout of the objects in our GUI look similar to the layout in Figure 3.15. As you can see, we have also used the Inspector to change font sizes, styles and colors as well to type in the strings.

Saving this Fig-file regenerates the corresponding M-file **`smooth.m`**. Note that MATLAB does not simply overwrite the file; our changes to callback functions are preserved, so we do not need to worry about losing our work. Let's open the M-file to see what changed. We have two new callback functions, one each for the Slider and the Text Edit Box as well as two new "create" functions, one each for the same controls. The create function of an object is called right after the object has been created and its properties have been set, but before it is displayed on the screen. Create functions can be used if additional initialization is needed for an object beyond simply setting the properties specified in the Property Inspector. We will not use create functions in this introductory section on GUIs.

We want our GUI to support this interaction: the user can specify the frequency and amplitude of a sine wave using the slider and the edit box. Then, hitting the REFRESH button will display the signal in the plot area of the window.

How can we accomplish this? The heavy lifting will be done in the callback function of the REFRESH button. What it needs to do is get the current frequency setting from the Slider and the current amplitude value from the Text Box and then plot the sine wave accordingly. This means that we do not even need to use the callback function of the Slider. The callback of the Edit Box is still needed because we need to do some error checking. The error checking has to do with the string that the user types in. It must represent a number so that we can use it to set the amplitude, so we'll implement some validation to make sure that we have a number there.

Let's see the callback of the Text Edit Box then:

```
function amplEdit_Callback(hObject, eventdata, handles)
% hObject    handle to amplEdit (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)

% Hints: get(hObject,'String') returns contents of amplEdit as text
%        str2double(get(hObject,'String')) returns contents of amplEdit as
a double
amp = str2double(get(hObject,'String'));
if isnan(amp) || ~isreal(amp)
    % isdouble returns NaN for non-numbers and we need a real number
    % Disable the REFRESH button
    set(hObject,'String','Error: NaN')
    set(handles.refreshButton,'Enable','off')
else
    set(handles.refreshButton,'Enable','on')
end
```

Consider the input arguments to our function. The comments generated by MATLAB actually do a good job of explaining them. **hObject** is a handle to the GUI widget that generated the event and caused the callback function to be invoked. In this case, it is a handle referring to Text Edit Box. Remember, we changed the Tag property of the edit box to **amplEdit**, so that's why the name of the function is **amplEdit_Callback**. The second argument, **eventdata**, is not used, but the third and last argument, **handles**, is very important. It is a **struct** that contains many fields including one for each object handle present in our GUI. It is through the **handles** structure that we can access all other GUI components in a callback function. In this callback then

```
hObject == handles.amplEdit
```

would be **true**.

The first line of our callback function gets the value the user has just typed in the Edit Text Box:

```
amp = str2double(get(hObject,'String'));
```

It calls the **get** function to get the **String** property of the **hObject** handle and then calls **str2double** to (attempt to) convert the string value returned by get into a **double**. Instead of calling the **get** function, why can we not get it directly from the object this way: **hObject.String**? We cannot do that because, as we mentioned earlier, GUIs do not use the MATLAB OOP infrastructure. The value stored in the variable **hObject** is called a "handle", but it is not an object of a subclass of the **handle** class, which can have properties that can be accessed via the dot operator. The value stored in **hObject** is simply a unique number that identifies to the GUI control system the widget that caused the callback to be called. The **get** function then finds the data structure with all the properties of the given GUI object identified by the **hObject** and returns the value of requested property.

The important point to learn here is that, when we need to access properties of an object of our GUI from within a callback function, we must use the **get** function to read its properties and the **set** function to modify them.

Returning to the callback function, we check whether **amp** is a real number. If not, we change the text it shows from whatever the user typed to "Error:NaN" and disable the REFRESH button. Notice the syntax of the **set** function. The first argument is the handle of the GUI object, the second is a string that specifies the name of the property and the third argument is the new value for that property. In this case, it is also a string.

Disabling the REFRESH button is simply setting its "Enable" property to off using the **set** function. The handle is obtained using the **handles** struct. Its **refreshButton** field is the correct handle, because we set the Tag property of the REFRESH button to **refreshButton** previously.

If the **amp** is indeed a correct number (**else** branch), then we enable the REFRESH button. This means that refreshing the plot will be possible only if the amplitude specification is valid. Notice that **amp** is a local variable, that is, it is not used outside this callback function. It is not needed outside because the Text Edit Box contains the value we need, so we can use it elsewhere in our program through the **handles** structure.

Notice that we do not need a similar error checking for the slider, since the user cannot set it in an incorrect manner. Its value will always be correct.

The most involved part of the program is the callback function of the REFRESH button, since it is here where we actually generate the data and plot it on the GUI:

```
function refreshButton_Callback(hObject, eventdata, handles)
% hObject    handle to refreshButton (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)

minFreq = 1;
maxFreq = 10;
t = 0:0.001:1;

% Get parameters from GUI
fs = get(handles.freqSlider,'Value');
freq = minFreq + fs * (maxFreq - minFreq);
amp = str2double(get(handles.amplEdit,'String'));

% Calculate data
x = amp * sin(2*pi*freq*t);

% Create time plot in proper axes
plot(handles.axes,t,x)
set(handles.axes,'XMinorTick','on')
grid on
```

First, we set up three helper variables. The minimum and maximum frequencies are set to 1 and 10 respectively. They will be used to translate the sliders value to the desired frequency. The **`t`** vector will be used to generate the corresponding sine values and help in plotting them.

Next we get the Slider's value. We use the **`get`** function passing in the handle of the Slider from the **`handles struct`** and access the **`Value`** property. The value of property is a **`double`** and its default range is between 0 and 1. We simply covert this value from the 0-1 range to the desired 1-10 range for our frequency. (Note that we could have set the range in the Property Inspector and save the conversion code.) Then we get the value of the amplitude from the Text Edit Box. We do not need to check whether it is valid or not, since its own callback function guarantees correctness.

Next we generate the data series for our sine wave using the specified frequency and amplitude values. Finally, we plot the sine wave specifying the **`handles.axes`** as the target plot area. Let's test it. See Figure 3.16.

You can drag the slider or click on its white background to change its value, but you will not notice any changes in the plot until you hit the REFRESH button. We can play with different frequencies and amplitudes. Let's test the error checking code we implemented for the amplitude. If we type in some non numerical text, Figure 3.17 shows what we see. Notice that the REFRESH button is "greyed out". We cannot press it until there is a valid number in the amplitude Text Box.

Figure 3.16 *The GUI in action*

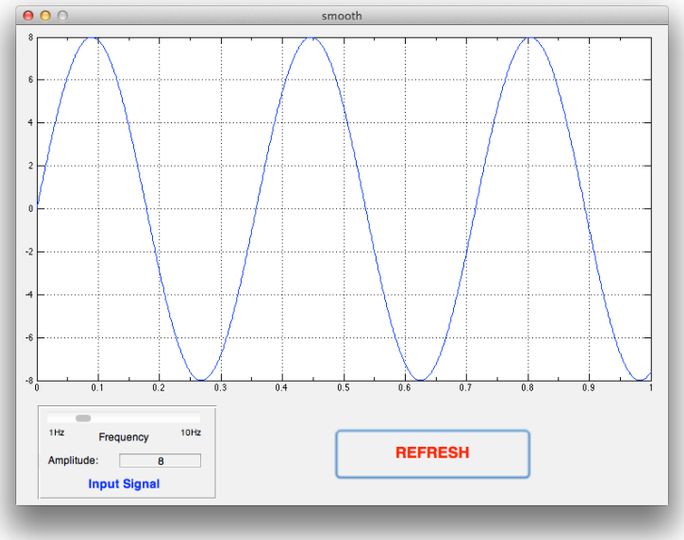
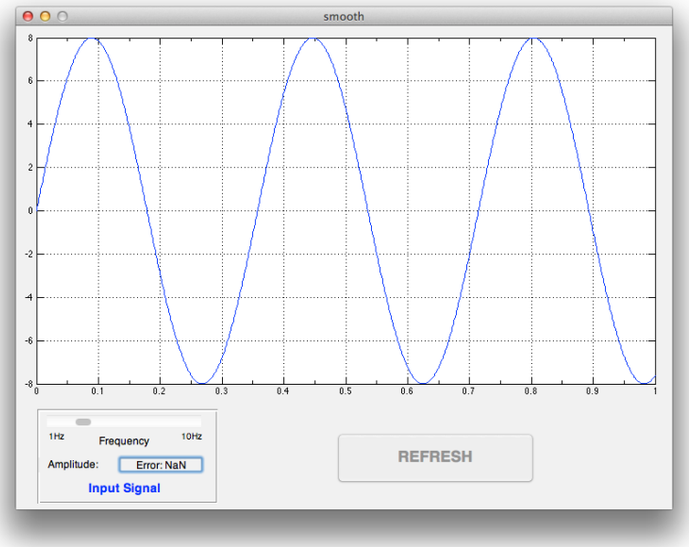


Figure 3.17 *Disabled button in the GUI*



Sharing data

While it is question of taste, most people would probably say that having to press the REFRESH button is unnecessary; it would be better to replot the data as soon as any of the parameters changed. This is doable, but the program will be a bit more involved. Removing the REFRESH button eliminates its callback function too. Therefore, we will need to implement the plotting functionality of our GUI elsewhere. In fact, the callback functions of both the Slider and the Edit Text Box will need to implement plotting.

While not strictly necessary in this case, an elegant approach to this problem is to share the data that is needed for plotting among the various callback functions. As we discussed before, global variables should be avoided whenever possible. So, how can we share data among various functions and preserve it in-between function calls? There are various ways of doing this, but the most common method with MATLAB GUIs is using the **`handles`** structure.

Recall that you can add new fields to structures dynamically during runtime. This is exactly what we shall do. You might have noticed that there was an extra function in the generated smooth.m-file called **`smooth_OpeningFcn`**. This method is called just before the GUI is shown on the screen. This is a perfect place to add additional fields to the **`handles`** structure and initialize it with the data that we need for plotting:

```
% --- Executes just before smooth is made visible.
function smooth_OpeningFcn(hObject, eventdata, handles, varargin)
% This function has no output args, see OutputFcn.
% hObject    handle to figure
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)
% varargin   command line arguments to smooth (see VARARGIN)

% Choose default command line output for smooth
handles.output = hObject;

% add our own data fields
handles.minFreq = 1;
handles.maxFreq = 10;
handles.amp = 1;
handles.freq = 1;
handles.t = 0:0.001:1;
handles.x = handles.amp * sin(2*pi*handles.freq*handles.t);
plot(handles.axes,handles.t,handles.x);
set(handles.axes,'XMinorTick','on');
grid on

% Update handles structure
guidata(hObject, handles);
```

The **`hObject`** argument in this case is the handle of the GUI window. The last argument, **`varargin`** is used if the program handles command line arguments. Let us not worry about that.

The portion that we added to the code lies after the "add our own data fields comment." Basically we store all relevant data items that are needed for plotting and set their default values. We also plot the data, so the GUI will not be blank at startup as it was before; instead it will immediately show a plot.

The last line of the function

```
guidata(hObject, handles);
```

is very important. The **`handles`** structure is an input argument to the function. Like all arguments in all functions, it is a local variable. Therefore, any modification we make to it inside the function will be lost upon returning from the function. The **`guidata`** function supplied by MATLAB takes these

changes to the **handles** structure and stores them somewhere. (In a global variable maybe? We'll just apply the "Don't ask, Don't tell" policy here...)

Let's delete the REFRESH button using GUIDE and update the callback functions:

```
function freqSlider_Callback(hObject, eventdata, handles)
% hObject    handle to freqSlider (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)

% Hints: get(hObject,'Value') returns position of slider
%        get(hObject,'Min') and get(hObject,'Max') to determine range

fs = get(handles.freqSlider,'Value');
handles.freq = handles.minFreq + fs * (handles.maxFreq - handles.minFreq);
handles.x = handles.amp * sin(2*pi*handles.freq*handles.t);
plot(handles.axes,handles.t,handles.x);
set(handles.axes,'XMinorTick','on');
grid on

% Update handles structure
guidata(hObject, handles);
```

First, we obtain the current Slider value, compute the corresponding frequency and recompute the sine wave. Then we display the new plot. Finally, we call **guidata** again, since we made changes to the **handles** structure.

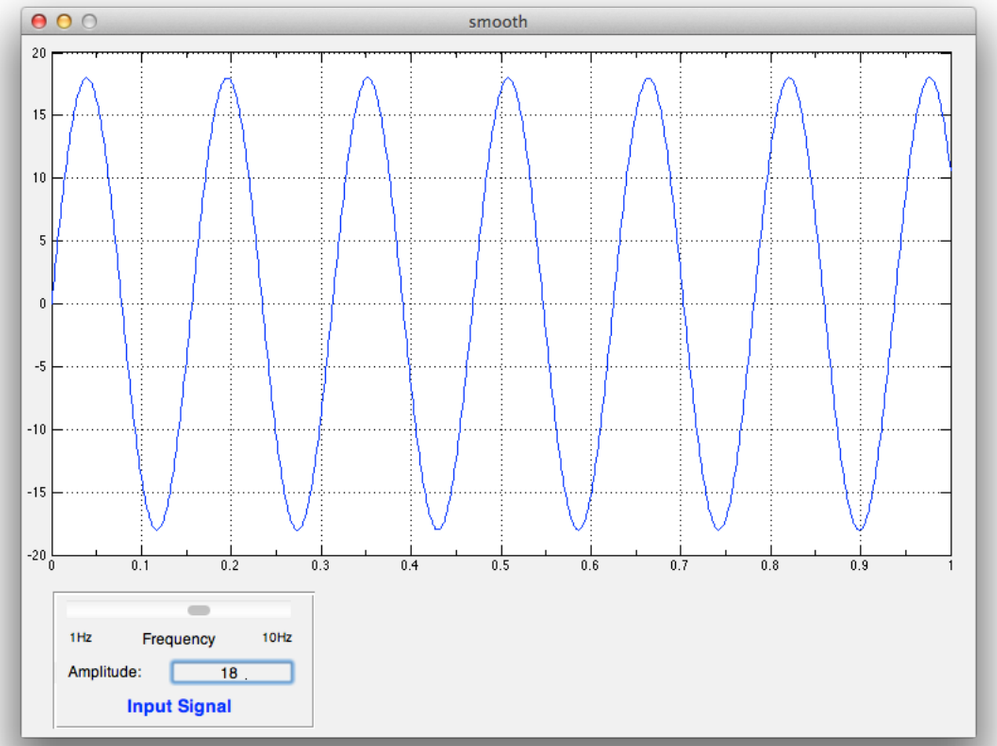
The callback function of the Edit Text Box needs to be adjusted too. First of all, there is no REFRESH button to disable in case the input is invalid. Therefore, we simply disregard any invalid values, display the previous (correct) value and do not change the plot.

```
function amplEdit_Callback(hObject, eventdata, handles)
% hObject    handle to amplEdit (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)

% Hints: get(hObject,'String') returns contents of amplEdit as text
%        str2double(get(hObject,'String')) returns contents of amplEdit as
a double
amp = str2double(get(hObject,'String'));
if isnan(amp) || ~isreal(amp)
    % isdouble returns NaN for non-numbers and f1 cannot be complex
    % Disable the Plot button and change its string to say why
    set(hObject,'String',num2str(handles.amp));
else
    if abs(amp) > 100
        amp = 100;
        set(hObject,'String',num2str(amp));
    end
    handles.amp = amp;
    handles.x = handles.amp * sin(2*pi*handles.freq*handles.t);
    plot(handles.axes,handles.t,handles.x);
    set(handles.axes,'XMinorTick','on');
    grid on
    guidata(hObject, handles);
end
```

Notice that we also limit the amplitude to 100 or less. In this case, we need to set the value of the Edit Box, so that it displays the correct value. Finally, we call **guidata** here also.

Now we save and run the program. Figure 3.18 shows the result. We can put breakpoints in the various callback functions to see when they are actually invoked. We find that the Edit Text Box calls its callback whenever the Return key is pressed or the user clicks outside the box. But first we need to click inside the box for it to get the "focus." There is always one GUI control that has the "focus," that is, one selected widget that is to receive the keyboard input. If you think about it, it has to be this way because, if there were multiple Edit Text Boxes, the GUI would need to know which one is to receive the input. Once the Edit Box has the focus (caused by clicking on it) it handles the Return key and calls the callback. It also calls it when it loses the focus when the user clicks outside the box (or presses the Tab key which cycles through all GUI components passing the focus around).

Figure 3.18 The GUI without the Refresh button

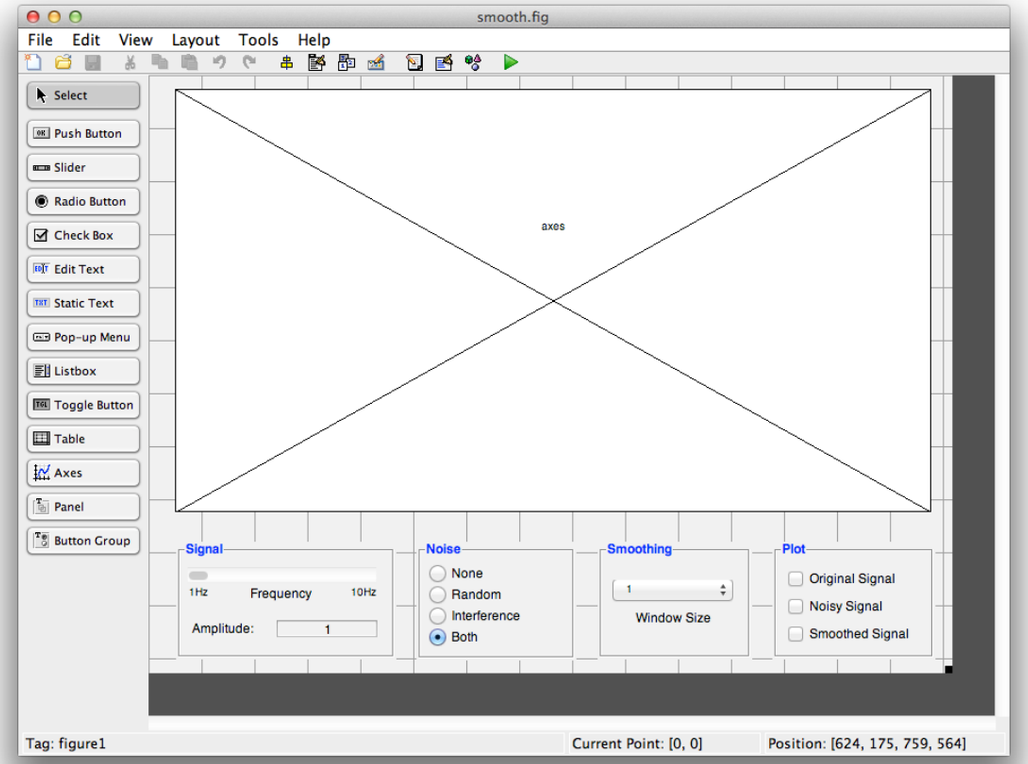


The Slider calls its callback if you click on the white background or drag and release the tab. However, during dragging, the callback is not called.

Advanced Plotting Example

You might have noticed all that unused space in the bottom right corner of our GUI. Well, we are about to use every last pixel there. We are going to extend our example, and here is what our extended plotting GUI will do. We will make it possible to add two different kinds of noise to the nice sine wave: white noise obtained by adding random numbers with normal distribution to the data and/or an interference signal, that is, a sine wave with a fixed

Figure 3.19 Final GUI design in GUIDE



60 Hz frequency simulating interference from a power line. We will also generate a smoothed version of this noisy data by using simple averaging. The GUI will allow the user to change the window size, that is the number of data points to be averaged. Finally, the user will have the option to plot the original sine wave, the noisy signal, and/or the smoothed signal on a single plot. Figure 3.19 shows our final GUI design as it appears in GUIDE.

There are quite a few controls in our final design that we have not used before. The rectangle labelled "Noise" is a Button Group. As we pointed out above, it is important that you create it first and then add the individual Radio Buttons. This way, it will allow only one active choice at a time out of "None," "Random," "Interference," and "Both." The other rectangles ("Sig-

nal," "Smoothing," and "Plot") are simple Panels with no active role. They only help in visually organizing the GUI. Notice that we have removed the Static Text "Input Signal" and have added the **`Title`** property to the Panel instead.

The "Smoothing" Panel contains a single Pop-up Menu where we can set the "Window Size". Finally, the Plot Panel has three Checkboxes. They can be on or off independently from each other, in contrast with Radio Buttons.

In the updated **`smooth_OpeningFcn`** method, we create and set new fields of the handles structure:

```
% add our own data fields
handles.minFreq = 1;
handles.maxFreq = 10;
handles.amp = 8;
handles.freq = 2.5;
handles.t = 0:0.001:1;
handles.x = handles.amp * sin(2*pi*handles.freq*handles.t);
handles.noise = randn(1,length(handles.t));
handles.interference = sin(2*pi*60*handles.t);
handles.window = NaN;
handles.windowIndex = 6;
handles.addRandom = true;
handles.addInterference = false;
handles.plotOriginal = false;
handles.plotNoisy = true;
handles.plotSmoothed = true;
```

Notice how we generate random noise by create a vector of random numbers using the **`randn`** function. Recall that **`rand`** generates uniformly distributed numbers, while we need a Gaussian distribution here. That is exactly what **`randn`** does. (If you have not learned about various distributions, do not worry, you can safely ignore this.) The interference signal is nothing but a 60 Hz sine wave with amplitude 1.

The **`window`** and **`windowIndex`** fields are both necessary for handling the pop-up menu. The menu contains a list of strings in its **`String`** property stored as a cell array of strings. Its **`value`** property is the index of the currently selected item. It is useful to store both in the **`handles struct`**, but initialize only the index at first. We'll see shortly how we obtain the actual value.

The **`addRandom`** and **`addInterference`** logical values store the current selection of additional noise sources to be added to the sine wave. Finally, the **`plotOriginal`**, **`plotNoisy`** and **`plotSmoothed`** fields specify the choices for plotting. We have used the built-in functions **`false`** and **`true`**, which return the values 0 and 1, to emphasize that these are truth values.

So far, we have initialized only our own internal data. The state of the various GUI controls are not set yet, that is, not set to correspond to the default values we selected for our variables. We need to do that carefully, so that they are in complete sync. The rest of the **`smooth_OpeningFcn`** function looks like this:

```
set(handles.amplEdit,'String',num2str(handles.amp));
fs = (handles.freq - handles.minFreq) / ...
                    (handles.maxFreq - handles.minFreq);
set(handles.freqSlider,'Value',fs);
set(handles.originalBox,'value',handles.plotOriginal);
set(handles.noisyBox,'value',handles.plotNoisy);
set(handles.smoothedBox,'value',handles.plotSmoothed);
set(handles.windowMenu,'Value',handles.windowIndex);
str = get(handles.windowMenu, 'String');
handles.window = str2double(str{handles.windowIndex});
set(handles.radiobutton2,'Value',1);

plotit(handles);

% Update handles structure
guidata(hObject, handles);
```

Setting the Edit Text Box and Slider values as well as the plot Checkboxes is quite straightforward. The Pop-up Menu is a bit trickier. First we set its value according to the default we set in **`handles.windowIndex`** previously. Then we get its **`String`** property back and using the index, we obtain the actual value specifying the window size. Why are we doing it in this convoluted

manner? The actual value for the choices in the menu were set using the Property Inspector in GUIDE. These are non-consecutive,somewhat arbitrary numbers that seem to be good choices for the window size. If we set the actual window property in the M-file and we accidentally chose a number that is not a valid choice in our menu, we would be in trouble. We would have to make sure that one of the values in the Property Inspector and the M-file match. It would be error prone. It is always a good programming practice to make sure that if you change something in one place in your code, you do not have to change it somewhere else also. By setting the index only and then obtaining the actual value from the Pop-up Menu, we ensure that any change in either the Property Inspector or the M-file will not cause any trouble in the other place. The only thing that we need to guarantee is that the Menu needs to have at least six items, because the default index we chose was six.

Setting the Radio Button is also easy. Note that setting one Button to 1, will automatically turn the others to 0 because they are part of a Button Group. Note also that here we do not check programmatically whether the handles fields **addRandom** and **addInterference** match the setting of the Radio Buttons. The reason is that the corresponding code is in the same function and selecting the correct radio button based on the two fields would have involved a somewhat long if-elseif-else statement. Omitting it seems to be a reasonable compromise. If we make a mistake here, it is easy to find and correct.

Once we are ready with all the initialization, we can plot the data according to the default values. Since we will need to plot from multiple callback functions, we moved the code to a separate function called **plotit**:

```
function plotit(handles)
    noisy = handles.x;
    if handles.addRandom
        noisy = noisy + handles.noise;
    end
    if handles.addInterference
        noisy = noisy + handles.interference;
    end
    smoothed = smoothit(noisy,handles.window);

    if ~(handles.plotOriginal ||
           handles.plotNoisy ||
           handles.plotSmoothed)
        cla(handles.axes,'reset');
    else
        if handles.plotOriginal
            plot(handles.axes,handles.t,handles.x,'k');
            hold(handles.axes,'on');
        end
        if handles.plotNoisy
            plot(handles.axes,handles.t, noisy,'r');
            hold(handles.axes,'on');
        end
        if handles.plotSmoothed
            plot(handles.axes,handles.t,smoothed,'b');
        end
        set(handles.axes,'XMinorTick','on');
        grid on
        hold(handles.axes,'off');
    end
```

The first part of the function computes the various data series. Depending on what options are selected on the user interface (stored in the **handles** structure as the **addRandom** and **addInterference** fields), we add the appropriate data to the sine wave stored in the **x** field. The smoothed vector is computed by the **smoothit** function that we'll discuss soon.

Next we plot the chosen data. If none of the options are selected (stored in **handles** in the **plotOriginal**, **plotNoisy** and **plotInterference** fields), then we reset the plot to erase the previous plotted figure. The **cla** built-in MATLAB method clears the axes by deleting all graphics objects from it.

In the else-branch, that is, if at least one of the plots is selected, we plot each that is desired with a different color: the original sine wave with black, the noisy signal with red and the smoothed signal with blue. Note how a separate if-statement (not nested!) is used for each plot since the conditions are independent of each other. Notice how we use the **hold** command to include all plots and not replace the previously drawn plot. Without the **hold**, only the last signal plotted would be visible.

Here is the **smoothit** function:

```
function b = smoothit(a,n)
% a - input vector
% n - number of samples to average

    k = floor(n/2);
    for ii = 1:length(a)
        b(ii) = mean(a(max(1,ii-k) :  ...
                       min(length(a),ii+k)));
    end
```

It is pretty simple really. We take the average of **n** values: the current sample and the **k** samples that precede and follow it, where **k** is about the half the specified window size. We have to be careful at the beginning and end of the vector where there are fewer samples available before or after the current sample. That's why the argument to **mean** seems a bit complicated at first. Indexing **a** this way simply keeps track of how many actual samples are to be included in the window (the minimum index is 1 and the maximum is the length of vector **a**). Note that we cannot compute the new values in place in **a**, since we need the original values to compute the subsequent smoothed values. That's why we use a separate vector **b** for the output.

Now, let's see what the callback functions look like for the various controls we have not seen before. The callbacks for the frequency Slider and the amplitude Edit Text Box changed very little. Instead of the various plot commands that were used in the previous versions, now they simply call the **plotit** function.

Here is how the plot Checkboxes are handled:

```
function originalBox_Callback(hObject, eventdata, handles)
% hObject    handle to originalBox (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)

% Hint: get(hObject,'Value') returns toggle state of originalBox
handles.plotOriginal = get(hObject,'value');
plotit(handles);
guidata(hObject, handles);
```

It simply gets the value property of the Checkbox, sets the corresponding field in the **handles** structure, calls **plotit** and makes sure to update the **handles** struct by calling **guidata**. The other callback functions are equivalent.

Handling the Pop-up Menu is somewhat more involved:

```
function windowMenu_Callback(hObject, eventdata, handles)
% hObject    handle to windowMenu (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)

% Hints: contents = cellstr(get(hObject,'String')) returns windowMenu con-
tents as cell array
%        contents{get(hObject,'Value')} returns selected item from window-
Menu
sel = get(hObject,'String');
handles.window = str2double(sel{get(hObject,'Value')});
plotit(handles);
guidata(hObject, handles);
```

The first line gets the **String** property from the control, namely, the cell array of strings containing all menu items, and assigns it to the variable **sel**. The second line gets the **Value** property, namely, the index of the current selection. It uses it to index into **sel** to get the **string** value of the current selection. In turn, it converts it into a double and assigns it to the **window** field of the **handles struct**. We do not need to do error checking once we make sure that all the possible menu items we set up in GUIDE are indeed numbers. Finally, we call **plotit** and update **handles** with **guidata**.

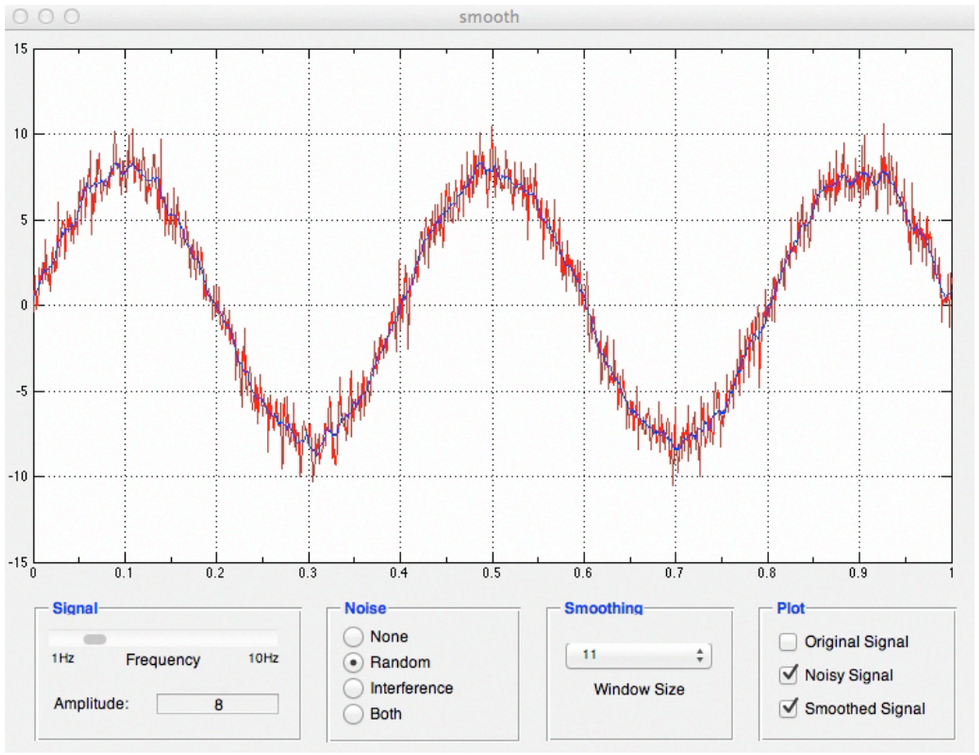
As we mentioned before, if we use Radio Buttons inside a Button Group then it is not the callback method of the Radio Buttons that we need to use, but that of the Button Group. The function is called "**SelectionChangeFcn**." Here is what we need to do to handle the selection concerning adding noise to our sine wave:

```
function signalControl_SelectionChangeFcn(hObject,eventdata,handles)
switch get(eventdata.NewValue,'Tag') % Get Tag of selected object.
    case 'radiobutton1'
        handles.addRandom = false;
        handles.addInterference = false;
    case 'radiobutton2'
        handles.addRandom = true;
        handles.addInterference = false;
    case 'radiobutton3'
        handles.addRandom = false;
        handles.addInterference = true;
    case 'radiobutton4'
        handles.addRandom = true;
        handles.addInterference = true;
end
plotit(handles);
guidata(hObject, handles);
```

This looks a bit different from the callback functions we have seen before. The important information is contained by the **eventdata** argument. It is a structure whose **OldValue** field contains the handle of the Radio Button that was selected before and **NewValue** that is the handle of the newly selected button. The first line of the function with the **switch**-statement gets the **Tag** property of the newly selected button using the **NewValue** field of the **eventdata**. The various **case** branches then simply set the two fields of handles **addRandom** and **addInterference** according to the users' selection. We finish up by plotting and updating **handles**.

That is it. We have created a fairly complex graphical user interface. Let's check it out, shall we? We can do that either by typing **smooth** in the Command Window or by pressing the green arrow in the toolbar of GUIDE. The result is shown in Movie 3.3.

Movie 3.3 *Our signal plotting GUI in action*



There are a few interesting points worth discussing about this application. Sometimes when changing something on the user interface, the plot seems to jump. This is because MATLAB does autoscaling, that is, it automatically adjusts the axis to fit the plot to the window size. Watch the video or try the program and see how the scale changes when such a jump occurs.

When using a large window size, the beginning and end of the smoothed signal looks different from the rest. This is because these parts of the signal are smoothed using a smaller, asymmetrical window. For example, there is no data before the very first data sample, so it is averaged only with samples that follow it.

Concepts From This Section

Computer Science and Mathematics:

Graphical User Interface (GUI)
Event-based programming
Callback function
GUI control

MATLAB:

GUIDE
widget
property
Property Editor
Edit Text Box
Static Text Box
Slider
Push Button
Radio Button
Toggle Button
Button Group
Checkbox
Axes
Pop-up Menu
Panel
Table
List Box
handle

*NOTE TO READER: The entries below comprise a combined index and glossary of the most important technical terms in this book. While it is easy to search for all uses of any term in any electronic document via the "Find" functionality (e.g., with Ctrl-f), this index provides the number of the page on which its definitive introduction appears. On that page in the text, you will find that term highlighted in **blue**. In addition, a definition is included below for each term plus a list of closely related terms to which the reader can refer in this same index+glossary for additional context.*

Abstract class, 319
: a class for which objects cannot be created in Object-Oriented Programming.
Related Terms: Class, Object, Object-Oriented Programming

Activation record, 269
: [a synonym for stack frame].
Related Terms: Stack frame

Algorithm, 26
: step-by-step procedure for solving a problem.
Related Terms: Algorithms, Divide-and-conquer

Algorithmic complexity, 309
: the shape of the plot of an algorithm's resource requirements as a function of the size of its input.
Related Terms: O notation, Order annotation, Space complexity, Time complexity, Worst-case analysis

Algorithms, 300
: a branch of computer science that deals with the efficiency of algorithms.
Related Terms: Algorithm, Divide-and-conquer

Apostrophe operator, 47
: [synonym for transposition operator]
Related Terms: Transposition operator

Argument, 34
: input to, or output from, a function.
Related Terms: Input argument, Output argument, Polymorphic, Polymorphism

Array, 34
: a multi-dimensional, rectangular arrangement of numbers.
Related Terms: Column-major order, Elementary type, End, Matrix, Pre-allocation, Scalar, Subarray operation, Vector, Vector command

ASCII, 202
: acronym for American Standard Code for Information Interchange, which is a scheme for encoding characters into bit patterns involving seven bits. It was designed in the 1960s to encode the so-called "Latin alphabet", the 10 decimal digits, and punctuation. Newer encoding schemes provide thousands of codes, whereas ASCII is limited to only 128 codes, but for backward compatibility each new code includes ASCII as a subset. It is common to refer to a text file as "an ASCII file", even if the particular character-encoding scheme used is unknown or is known to be different from ASCII.
Related Terms: Text file

Assignment statement, 21
: a statement of the form <variable> = <expression> that means "assign the value of the expression on the right side of the equal sign to the variable that is on the left side".
Related Terms: Variable

Associativity, 60
: the order in which operators of equal precedence are applied within an expression. The options are right-to-left and left-to-right. MATLAB uses right-to-left associativity.
Related Terms: Operator, Precedence

Base case, 270
: **1.** a part of a recursive definition that does not involve the concept being defined. **2.** a part of a recursive function that does not involve a recursive function call.
Related Terms: Recursive case, Recursive definition, Recursive function

Base class, 313
: a class in Object-Oriented Programming whose properties have been inherited by a new class. The new class is called a subclass of the base class. [synonym is superclass]
Related Terms: Class, Inheritance, Object-Oriented Programming, Subclass, Superclass

Big-O notation, 309
: [synonym for Order notation]
Related Terms: O notation, Order notation

Binary file, 247
a file that contains a stream of bits that, as opposed to the stream of bits of a text file, is not encoded into characters.
Related Terms: Bit, Text file

Binary operator, 47
an operator, such as * (multiplication), that operates on two operands.
Related Terms: Operator, Unary operator

Binary search, 298
a search method that is appropriate only for a sorted list, in which the target value that is being sought is compared with an element in the middle of the list first. If the target is less than the middle element, then the method is applied recursively to the first half of the list, otherwise it is applied to the last half of the list until a middle element is found that is equal to the target value or the list to be searched is empty, whichever comes first.
Related Terms: Sequential search

Bit, 19
the smallest unit of memory on a computer. The value of a bit is either 0 or 1.
Related Terms: Binary file, Byte

Block, 115
a set of contiguous statements.
Related Terms: Control statement

Branching, 114
[a synonym for "selection"]
Related Terms: Selection

break, 158
a MATLAB keyword that makes up the break-statement.
Related Terms: Break-statement, Keyword

Break-statement, 158
a control statement that causes a loop to stop. Execution continues at the next statement following the loop.
Related Terms: Break, Continue-statement, Loop

Bugs, 27, 103
errors in programs that cause them to behave incorrectly.
Related Terms: Debugger, Debugging, Software

Byte, 200
a group of eight bits.
Related Terms: Bit

Callback, 334
a function that executes in response to a user-generated event, such as a mouse motion or click, a screen touch, or a key press. Callbacks are essential for the implementation of a Graphical User Interface. [synonym: event handler]
Related Terms: Event handler, Event-based, Graphical User Interface

case, 120
a MATLAB keyword that introduces the beginning of a block of statements in a switch-statement.
Related Terms: Keyword, Switch-statement

Cell, 214
the type of a pointer variable in MATLAB.
Related Terms: Pointer variable

Class, 312
the definition of an object in Object-Oriented Programming. It is introduced by the keyword **classdef**. (also the name of a function in MATLAB that returns the name of a built-in type)
Related Terms: Abstract class, Base class, Classdef, Constructor, Data member, Destructor, Inheritance, Instance, Member function, Methods, Object, Object-Oriented Programming, Operator overloading, Overloading, Private, Public, Subclass

classdef, 314
a MATLAB keyword that introduces a class definition in Object-Oriented-Programming.
Related Terms: Class, Keyword, Object-Oriented Programming, Properties

Code, 26
part or all of a program.
Related Terms: Executable code, Program, Source code

Coding, 26
writing a program
Related Terms: Computer programming, Program, Programmer, Programming

Colon operator, 38
an operator symbolized by a colon (**:**) whose output is a row vector comprising a regularly spaced list of numbers determined by two or three operands. The operands are an initial number, an optional difference between successive numbers (default equals 1), and a final number.
Related Terms: Row vector

Column, 40
position in the second dimension of a matrix or array
Related Terms: Page, Row

Column vector, 35
a vector whose elements are arranged vertically.
Related Terms: Linear algebra, Linear indexing, Vector

Column-major order, 35
an order of processing elements in an array, in which all the elements of one column are processed before the elements of the next column.
Related Terms: Array, Row-major order

Comment, 28
text that is included in a program but is ignored by a system that is interpreting the program, such as MATLAB, or a system that is compiling the program, such as a C++ compiler. Such text is intended to explain the operation of the code to a human reader.
Related Terms: Compiling, Interpreting

Compiling, 21
translating. Applies when a program is being translated from a language in which a programmer writes it (e.g., C++) into a language that the computer hardware uses.
Related Terms: Comment, Executable code, Interpreting

Complex conjugate, 48
a complex number in which the real part is the same as that of another complex number and the imaginary part has the same magnitude, but opposite sign. The two numbers are complex conjugates of each other.
Related Terms: Complex number

Complex number, 37
a number that includes the square root of -1, which is imaginary and is symbolized by the letter *i* in mathematics. In MATLAB, the imaginary part of a complex number is indicated by the suffix **i** or the suffix **j**.
Related Terms: Complex conjugate

Computer program, 26
a program that describes an algorithm to be executed on a computer.
Related Terms: Program, Software

Computer programming, 26
1. writing a program to run on a computer. **2.** the subject of this book.
Related Terms: Coding, Programming

Condition number, 294
a number that gives the maximum possible percent change in output relative to a one percent change in input.
Related Terms:

Conditional, 119
an expression that determines whether or not a block within an if-statement or a while-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.
Related Terms: Else-clause, If-else-statement, If-elseif-else-statement, If-elseif-statement, If-statement, While-loop

Constructor, 314
a special class function in Object-Oriented Programming. It has the same name as the class, and when it is called, it creates a new object as an instance of the class.
Related Terms: Class, Destructor, Instance, Object, Object-Oriented Programming

continue, 160
a MATLAB keyword that comprises the continue-statement.
Related Terms: Continue-statement, Keyword

Continue-statement, 160
a control statement that causes a loop to begin execution of the next iteration without completing the current one.
Related Terms: Break-statement, Continue, Loop

Control construct, 114
a particular method by which the interpreter selects the next statement to be executed after the execution of the current statement has concluded.
Related Terms: End, Interpreter, Loop, Selection, Sequential control

Control statement, 114
a statement that controls the execution of another statement or a set of statements.
Related Terms: Implicit loop

Conversion function, 199
a function that takes an input argument of one type and returns an output argument of a different type with a value as close as possible, but not necessarily equal to, that of the input argument.
Related Terms: Type
Data member, 312
in Object-Oriented Programming, an encapsulated variable of the class.
Related Terms: Class, Encapsulation, Object-Oriented Programming, Property
Data type, 197
a set of values and a set of operations that can be performed on those values.
Related Terms: Type
Debugger, 104
a tool to assist in finding and removing bugs.
Related Terms: Bug, Debugging
Debugging, 103
the procedure of finding and correcting programming errors, which are also known as bugs.
Related Terms: Bug, Debugger
Delimiter, 244
a single character or a string that allows a program to determine where one part of a character sequence ends and the next part begins without knowing the expected lengths of any of the parts.
Related Terms:
Destructor, 327
a special class function in Object-Oriented Programming. When it is called for a given object, it removes the object, possibly performing further processing necessary to maintain consistency among the remaining objects of the class.
Related Terms: Class, Constructor, Instance, Object, Object-Oriented Programming
Diagonal, 86
noun. the set of elements of a matrix whose indices are equal to each other. **adjective**. having non-zero elements only on the diagonal of a matrix.
Related Terms: Matrix
Divide-and-conquer, 300
a strategy in which a problem is solved efficiently by applying an algorithm to parts of the problem individually and achieving a solution to the complete problem by combining the results of the partial solutions.
Related Terms: Algorithm, Algorithms
Dominate, 309
get larger faster than. Example: *N*-cubed dominates *N*-squared as *N* increases.
Related Terms:
Dot-apostrophe operator, 48
a MATLAB operator that produces that transposes a matrix without taking the complex conjugate of its elements. It is a unary postfix operator. Its symbol is a dot and an apostrophe (.').
Related Terms: Transposition operator
Doubly linked list, 318
a linked list each of whose elements has both a pointer to its next element, if there is one, and a pointer to the previous element, if there is one.
Related Terms: Head, Linked list, Next, Prev, Tail
Element, 29
a single item in a vector, matrix, or array.
Related Terms:
Elementary type, 197
the type of one element of an array. (All elements of a given array must be of the same type.)
Related Terms: Array
else, 115
a MATLAB keyword used to introduce an else-clause.
Related Terms: Else-clause, If-else-statement, If-elseif-else-statement, If-statement, Keyword
Else-clause, 119
an alternative with no conditional within an if-statement. It is executed if the conditional in the if-statement is false (or none of the conditionals in the if-elseif-statement is true).
Related Terms: Conditional, Else, If-else-statement, If-elseif-else-statement, If-statement

elseif, 116
a MATLAB keyword used to introduce an elseif-clause.
Related Terms: Elseif-clause, If-else-statement, If-elseif-else-statement, If-elseif-statement, Keyword
Elseif-clause, 119
an alternative with a conditional within an if-statement. It is executed if its conditional is true and none of the previous conditionals in the if-statement is true.
Related Terms: Elseif, If-elseif-else-statement, If-elseif-statement
Empty matrix, 29
a matrix with no elements.
Related Terms: Matrix
Encapsulation, 312
prohibition of access of an object in Object-Oriented Programming by all but a designated set of functions associated with that object. Those designated function are known as "member functions" or "methods".
Related Terms: Data member, Member function, Method
end, 42
a MATLAB keyword that (a) stands for the last index in one dimension of an array, (b) terminates a control construct and (c) is optionally used to terminate a function definition.
Related Terms: Array, Control construct, Function, Keyword
Escape character, 96
a character signifying that the character or characters that follow it in a format string have a special meaning.
Related Terms: Escape sequence, Format specifier, Format string
Escape sequence, 96
a meaningful sequence of characters that begins with an escape character. For example, %**5.2f**, which begins with the escape character % or **\n**, which begins with the escape character \.
Related Terms: Escape character, Format specifier, Format string
Event handler, 334
[synonym for callback]
Related Terms: Callback
Event-based, 334
having the ability to initiate function calls in response to a user-generated event, such as a mouse motion or click, a screen touch, or a key press. A function called in response to an event is termed a "callback". Event-based programs are used to implement Graphical User Interfaces, for example.
Related Terms: Callback, Graphical User Interface
Executable code, 26
part or all of a program that is written in a language that a computer can execute directly. It is typically produced by compiling source code.
Related Terms: Code, Compiling, Program, Source, Source code
Field, 208
an element of a struct. It is designated by means of its field name.
Related Terms: Field name, Struct
Field name, 208
a name used as an index into a struct.
Related Terms: Field, Struct
File, 228
an area in permanent memory, 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, accessed by other programs, and managed by the operating system.
Related Terms: Text file
Flag, 159
a value indicating a special condition or a variable that hold the value.
Related Terms:
for, 143
a MATLAB keyword that introduces the control-statement of a for-loop.
Related Terms: For-loop, Keyword
For-loop, 143
a control construct that causes a block of statements to be executed repeatedly: once for every one of a set of values. One value is assigned to a "loop index" at the beginning of each iteration.
Related Terms: For, Implicit loop, Loop, Loop index, While-loop
Format specifier, 96
a character specifying the format in which an object is to be printed or text is to be converted into an object.
Related Terms: Escape character, Escape sequence, Format string

Format string, 95
a string that specifies the way in which either, (a) 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., or (b) reading is to be done, such as words that must be read, spacing, the number of decimal places to be included when reading numbers, etc.
Related Terms: Escape character, Escape sequence, Format specifier
Frame, 269
[synonym for stack frame]
Related Terms: Stack frame
function, 34
1.an operation that is invoked by giving its name. This computer-science definition is in contrast to the mathematical definition, which is any operation that produces a result that depends only on its input. **2**. a MATLAB keyword that introduces the function declaration in the function's M-file
Related Terms: End, Keyword, Libraries, M-file, Return-statement
Functional decomposition, 77
process of dividing up a program into functions.
Related Terms:
global, 76
a MATLAB keyword used to give a variable global scope.
Related Terms: Global scope, Keyword
Global scope, 76
visibility in more than one function and/or the Command Window.
Related Terms: Global, Local scope, Local variable, Scope
Graphical User Interface, 333
a program's scheme for interacting with the user through pictures, mouse clicks, and screen touches as an adjunct to the input and output of characters via the keyboard and the screen. GUI is its commonly used acronym.
Related Terms: Callback, Event-based, GUI, Integrated Development Environment, Widget
GUI, 333
acronym for Graphical User Interface.
Related Terms: Graphical User Interface
GUI control, 334
[synonym for widget]
Related Terms: Widget
Head, 318
the first element of a linear linked list, i.e., an element that is pointed at by no other element on the list unless the list is a doubly linked list.
Related Terms: Doubly linked list, Linked list, Tail
I/O, 229
input/output, i.e., input to a program and/or output from a program.
Related Terms:
IDE, 334
the commonly used acronym for Integrated Development Environment.
Related Terms: Integrated Development Environment
if, 114
a MATLAB keyword that introduces the control-statement of an if-statement.
Related Terms: If-statement, Keyword
If-else-statement, 115
a selection construct. It causes the interpreter to choose which of two statements or blocks of statements to execute on the basis of the truth of a conditional.
Related Terms: Conditional, Else, Else-clause, Elseif, If-elseif-else-statement, If-elseif-statement, If-statement, Selection
If-elseif-else-statement, 116
a selection construct. It causes the interpreter to choose which of three or more statements or blocks of statements to execute on the basis of the truth of two or more conditionals.
Related Terms: Conditional, Else, Else-clause, Elseif, Elseif-clause, If-else-statement, If-elseif-statement, If-statement, Selection
If-elseif-statement, 117
a selection construct. It causes the interpreter to choose whether one or neither of two or blocks of statements to execute on the basis of the truth of two conditionals.
Related Terms: Conditional, Elseif, Elseif-clause, If-else-statement, If-elseif-else-statement, If-statement, Selection

If-statement, 114
the simplest selection construct. It causes the interpreter to choose whether or not to execute a statement or block of statements on the basis of the truth of a conditional. The term "if-statement" also refers generically to if-else-statements, if-elseif statements, and if-elseif-else-statements.
Related Terms: Conditional, Else, Else-clause, If, If-else-statement, If-elseif-else-statement, If-elseif-statement, Selection

Ill-conditioned, 293
having the property that small changes in the inputs produce very large changes in the outputs.
Related Terms:

Image processing, 149
the generation of a new image from one or more existing images.
Related Terms: Libraries

Implicit loop, 161
a loop that is invoked without the use of either a for-loop control statement or a while-loop control statement.
Related Terms: Control statement, For-loop, Loop, While-loop

Inconsistent, 289
having constraints that are no more numerous than the number of unknowns and yet cannot all be satisfied simultaneously. Applies, for example, to a set of simultaneous linear algebraic equations represented by $Ax = b$ where A has no more rows than columns and yet no solution vector x exists.
Related Terms: Linear algebra, Overdetermined, Underdetermined

Index, 29
1. a non-negative integer that enumerates the elements of a vector, a matrix, or an array. In MATLAB the integer must be positive (i.e., 0 is not allowed). Typically multiple indices are used—one for each dimension of the array, but see also linear indexing, in which case the term “subscript” is often used instead of “index”. In linear indexing, only one index is used regardless of the number of dimensions. **2.** a field name in a struct.
Related Terms: Field name, Linear indexing, Loop index, Struct, Subscript

Infinite loop, 157
a loop that continues iterating without any possibility of stopping.
Related Terms:

Infix, 47
description of an operator whose symbol for its operation comes between its operands.
Related Terms: Postfix, Prefix

Inheritance, 313
the ability in Object-Oriented Programming to define a class as an extension of an existing class.
Related Terms: Base class, Class, Object-Oriented Programming, Subclass

Input argument, 67
a local variable that receives a value that is input into the function.
Related Terms: Argument, Local variable, Output argument

Instance, 312
an object in Object-Oriented Programming created by invoking its class.
Related Terms: Class, Constructor, Destructor, Object-Oriented Programming

Integrated Development Environment, 334
a program designed for a given programming language that is used to develop programs in that language. It typically includes multiple tools such as a text editor, debugger and others. IDE is its commonly used acronym. The MATLAB environment itself is an IDE.
Related Terms: Graphical User Interface, IDE

Interpreter, 113
a program that executes statements. The simplest interpreter is the central processing unit (CPU), which is a program implemented in hardware. Interpreters, like that provided as part of the MATLAB programming environment execute statements in far more complex languages. It reads the statements in a program 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.
Related Terms: Control construct, Interpreting

Interpreting, 20
executing. Applies to a command or statement executed by a computing environment such as MATLAB.
Related Terms: Comment, Compiling, Interpreter, P-code

Iterate, 142
repeatedly execute a block of statements. execute a loop.
Related Terms: Iteration, Iterative, Loop

Iteration, 142
1. the repeated execution of a block of statements. **2**. one execution of the body of a loop. 3. the act of executing a loop.
Related Terms: Iterate, Iterative, Loop

Iterative, 142
involving the repeated execution of a block of statements.
Related Terms: Iterate, Iteration, Loop

Keyword, 42
a word that is defined by a programming language to have special meaning. The keywords for the language MATLAB are as follows (**`catch`**, **`parfor`**, **`spmd`** and **`try`** are not covered in this book):

`break`	`enumeration`	`parfor`
`case`	`events`	`persistent`
`catch`	`for`	`properties`
`classdef`	`function`	`return`
`continue`	`global`	`spmd`
`else`	`if`	`switch`
`elseif`	`methods`	`try`
`end`	`otherwise`	`while`

Except for `enumeration`, `events`, `methods`, and `properties`, all of these keywords are reserved words (see Reserved word).
Related Terms: Reserved word

Libraries, 85
sets of ready-to-use functions for frequently used operations. A given library is typically targeted at a specific class of operations, e.g., an image-processing library or a statistics library.
Related Terms: Function, Image processing

Linear algebra, 286
the manipulation and solution of equations of the form $Ax=b$, where A is a matrix and x and b are both column vectors.
Related Terms: Column vector, Inconsistent, Matrix, Matrix algebra, Overdetermined, Simultaneous linear algebraic equations, Underdetermined

Linear indexing, 58
the specification of just one index for an array, regardless of its number of dimensions. The meaning is that the array is treated as a column vector and the indexing is in column-major order.
Related Terms: Column vector, Index, Subscript

Linear search, 297
[synonym for sequential search]
Related Terms: Sequential search

Linked list, 318
set of individual elements linked into a chain, meaning that every element is linked to two other elements, except that two of the elements (the head and tail) may be linked to only one other element. If there is a head and tail, then the list is a linear linked list. If there is no head and tail, the list is a circularly linked list. A link is typically implemented by means of a pointer stored in the element that points at another element. Each element other than a tail element points at the "next" element. If the list is a doubly linked list, then each element other than a head element also points at the "previous" element.
Related Terms: Doubly linked list, Head, Next, Prev, Tail

Local scope, 76
accessibility by statements in only one function or only in the Command Window.
Related Terms: Global scope, Local variable, Scope

Local variable, 65
a variable that is accessible only by statements inside one function. A local variable exists only during the function call.
Related Terms: Global scope, Input argument, Local scope, Output argument, Scope

Logical, 164
a type that includes only two values: true and false, where true in arithmetic expressions is treated as 1 and false is treated as 0.
Related Terms: Logical array, Logical indexing

Logical array, 164
an array of logical type.
Related Terms: Logical

Logical indexing, 164
the use of logical values as indices into an array. Only those elements for which the logical index has the value true are selected.
Related Terms: Logical

Logical operator, 126
an operator that produces a value that depends on the truth of its two operands.
Related Terms: Short circuiting

Loop, 140
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.
Related Terms: Break-statement, Continue-statement, Control construct, For-loop, Implicit loop, Iterate, Iteration, Iterative, Loop index, Sequential control, Vectorization, While-loop

Loop index, 142
a variable that is changed by the control statement of a for-loop. In MATLAB, C, C++, and Java, a loop index, unlike an array index, need not be an integer.
Related Terms: For-loop

M-file, 25
a file whose name has the extension **.m**. It contains either a MATLAB script or a MATLAB function. Also called a "Dot-M-file", it is where MATLAB looks to find a function or script. If the function **foo** is called, MATLAB runs the code found in a file named **foo.m**. An M-file may contain any number of functions. The first function in a given file is the main function and the rest are subfunctions.
Related Terms: Function, Main function, MAT-file, Script, Subfunction

Main function, 75
the first function in an M-file. It is the only function that can be called from outside the file. An M-file may contain more than one function. Functions in the file other than the main function are called subfunctions. A file can contain any number of subfunctions.
Related Terms: M-file, Subfunction

MAT-file, 20
a file format used by MATLAB to save a workplace via the **load** command. Its extension is **.mat**.
Related Terms: M-file

MATLAB, vii
a programming language and programming environment designed for writing programs that solve numerical problems. MATLAB stands for "Matrix Laboratory". It is sold by The MathWorks, Inc., Natick, Massachusetts.
Related Terms:

Matrix, 34
a two-dimensional, rectangular arrangement of numbers, also known as a two-dimensional array.
Related Terms: Array, Diagonal, Empty matrix, Linear algebra, Square matrix, Subarray operation, Vector

Matrix algebra, 286
[synonym for linear algebra]
Related Terms: Linear algebra, Simultaneous linear algebraic equations

Member function, 312
a function within a class in Object-Oriented Programming. [synonym: method].
Related Terms: Class, Encapsulation, Method, Methods, Object-Oriented Programming

Method, 312
[MATLAB synonym for member function]
Related Terms: Encapsulation, Member function, Methods, Object-Oriented Programming, Private, Public

methods, 314
a MATLAB keyword that introduces the methods section of a class definition in Object-Oriented-Programming.
Related Terms: Class, Member function, Method, Object-Oriented Programming, Properties

Mixed-mode arithmetic, 200
arithmetic operations involving two operands of different types.
Related Terms: Operand, Operation, Type

Nesting, 131
the inclusion of one control construct inside another control construct.
Related Terms: Control construct

Next, 318
: a common name for the field of an element of a linked list that contains a pointer to the next element on the list.
Related Terms: Linked list

O notation, 309
: [synonym for Order notation]
Related Terms: Algorithmic complexity, Big-O notation, Order notation

Object, 312
: in Object-Oriented-Programming, a set of variables and a set of functions that can operate on those variables. An object is an instance of a class.
Related Terms: Abstract class, Class, Constructor, Destructor, Instance, Object-Oriented Programming, Operator overloading

Object reference, 319
: an object that holds information about another object including the address of the other object.
Related Terms: Pointer

Object-Oriented Programming, 312
: an approach to programming that is centered on data as opposed to functions.
Related Terms: Abstract class, Base class, Class, Classdef, Constructor, Data member, Destructor, Inheritance, Instance, Member function, Method, Methods, Object, Operator overloading, Private, Properties, Property, Public, Subclass, Superclass

Operand, 38
: an input argument to an operator.
Related Terms: Mixed-mode arithmetic, Operation, Operator

Operation, 38
: the action of an operator on its operand(s).
Related Terms: Mixed-mode arithmetic, Operand, Operator

Operator, 38
: a function that is invoked by a symbol, the most familiar examples being `+`, `-`, `*`, and `/`.
Related Terms: Associativity, Binary operator, Operand, Operation, Operator overloading, Precedence, Prefix

Operator overloading, 312
: in Object-Oriented Programming, the defining of new functionality for an operator inside a class definition that allows it to operate on an instance of a user-defined class.
Related Terms: Class, Object, Object-Oriented Programming, Operator, Overloading, Polymorphism

Order notation, 309
: a method for categorizing the shape of growth curves used to specify algorithmic complexity. synonyms: O notation, Big-O notation.
Related Terms: Algorithmic complexity, Big-O notation, O notation

otherwise, 120
: a MATLAB keyword that introduces an optional block of statements in a switch-statement. That block is executed if and only if none of the case expressions matches the expression of the control-statement.
Related Terms: Keyword, Switch-statement

Output argument, 66
: a local variable that holds a value that is output by the function.
Related Terms: Argument, Input argument, Local variable

Overdetermined, 290
: having constraints that are more numerous than the number of unknowns and that cannot all be satisfied simultaneously. Applies, for example, to a set of simultaneous linear algebraic equations represented by $Ax = b$ where A has more rows than columns and no solution vector x exists.
Related Terms: Inconsistent, Linear algebra, Underdetermined

Overloading, 312
: in Object-Oriented Programming, the defining of new functionality for a function or an operator inside a class definition that allows the function or operator to process an instance of a user-defined class.
Related Terms: Class, Operator overloading

P-code, 27
: an abbreviation for "portable code", which is a program written in a language that can be interpreted by programs on different types of computers. Each interpreter must be tailored to run on a specific type of computer, but a program written in p-code can be the same for all computers and so is portable from one type of computer to another. MATLAB uses a form of p-code.
Related Terms: Interpreting

Page, 44
Position in the third dimension of an array
Related Terms: Column, Row
persistent, 267
a MATLAB keyword used to declare persistent variables.
Related Terms: Keyword, Persistent variable
Persistent variable, 267
a local variable within a function whose value persists across function calls.
Related Terms: Persistent
Pixel, 30, 151
one square piece of a mosaic of colors that make up a digital image (the word "pixel" is a shortening and alteration of the phrase "Picture Element").
Related Terms:
Pointer, 214
the address of an object in memory.
Related Terms: Object reference, Pointer variable
Pointer variable, 214
a variable that holds an address.
Related Terms: Cell, Pointer
Polymorphic, 87
a term applied to a function. A polymorphic function allows the number and/or type of its input and/or output arguments to vary from one call of the function to the next. In many cases, the behavior of a polymorphic function depends not only on the values of its input arguments but also on the number and types of its arguments (both input and output).
Related Terms: Argument, Polymorphism
Polymorphism, 87
the ability of a function to allow the number and/or type of its input and/or output arguments to vary from one call of the function to the next. Polymorphism allows for the behavior of a function to depend not only on the values of its input arguments but also on the number and types of its arguments (both input and output).
Related Terms: Argument, Operator overloading, Polymorphic
Postfix, 47
description of an operator whose symbol for its operation comes after its operand(s).
Related Terms: Infix, Prefix
Pre-allocation, 180
the designation and reservation of space for an entire array before calculating the values to put into the array.
Related Terms: Array
Precedence, 59
an operator's ranking relative to other operators which determines the order in which they are applied within an expression. A lower ranking means that an operator has higher precedence and so is applied earlier.
Related Terms: Associativity, Operator, Precedence table
Precedence table, 130
a listing of operators along with a number indicating the precedence of each operator. A lower number indicates a higher precedence.
Related Terms: Precedence
Prefix, 47
description of an operator whose symbol for its operation comes before its operand(s).
Related Terms: Infix, Operator, Postfix
Prev, 318
a common name for the field of an element of a doubly inked list that contains a pointer to the previous element on the list.
Related Terms: Doubly linked list, Linked list, Next
Price is Right, 38
a television game in which a prize is won by guessing the price that comes closest to its actual retail value without exceeding it, recognizable primarily by the way its contestants jump up and down and scream.
Related Terms:
private, 320
in Object-Oriented Programming, visible and/or modifiable only by methods within the class in which it is defined. Pertains to a property of the class.
Related Terms: Class, Method, Object-Oriented Programming, Property, Public

Program, 26
a sequence of symbols that describes an algorithm.
Related Terms: Code, Coding, Computer program, Executable code, Programmer, Programming

Programmer, 26
a person who writes a program.
Related Terms: Coding, Program, Programming

Programming, 26
writing a program.
Related Terms: Coding, Computer programming, Program, Programmer

Prompt, 15
a symbol, or symbols, printed by a program (MATLAB is an example) to indicate that it is ready for input from the user of the program. MATLAB uses the prompt **>>**
Related Terms:

properties, 314
a MATLAB keyword that introduces the properties section of a class definition in Object-Oriented Programming.
Related Terms: Classdef, Methods, Object-Oriented Programming

Property, 312
[synonym for data member]
Related Terms: Data member, Object-Oriented Programming, Private, Public

public, 320
in Object-Oriented Programming, visible and/or modifiable not only by methods within a class in which it is defined but also by functions outside. Pertains to a property of the class. It is the default.
Related Terms: Class, Method, Object-Oriented Programming, Private, Property

Recursion, 268
1. the use of a concept in the definition of the concept. **2**. the calling of a function by the function itself (e.g., **f** calls **f**). **3**. the chaining of two or more function calls that ends with the function that made the first call (e.g., **f** calls **g**, **g** calls **h**, **h** calls **f**).
Related Terms: Recursive definition, Recursive function call

Recursive case, 270
1. part of a recursive definition that involves the concept being defined. **2**. part of a recursive function that involves a recursive function call.
Related Terms: Base case, Recursive definition

Recursive definition, 268
a definition that defines a concept in terms of the concept itself.
Related Terms: Base case, Recursion, Recursive case

Recursive function, 268
a function that makes a recursive function call.
Related Terms: Base case, Recursive function call

Recursive function call, 268
a call of a function by the function itself (e.g., **f** calls **f**). 2. a function call that begins a chain that ends with the function that made the first call (e.g., **f** calls **g**, **g** calls **h**, **h** calls **f**).
Related Terms: Recursion, Recursive function

Relational operator, 123
an operator that produces a value that depends on the relation between the values of its two operands. An example is "greater than".
Related Terms:

Reserved word, 64
a keyword (see Keyword) that cannot be used as a variable name or function name. It can however be used as a field name in a struct (see Struct). A list of reserved words can be obtained with the command **`iskeyword`**, which despite its name lists only *reserved* keywords.
Related Terms: Keyword

return, 118
a MATLAB keyword that comprises the return-statement.
Related Terms: Keyword, Return-statement

Return-statement, 118
a statement that causes the function in which it appears to be halted and control passed to the caller, which may be another function or the Command Window. If the function has output arguments, then the most recent value assigned to each requested output argument will be passed to the caller, just as it would be if the function had ended after executing its last statement.
Related Terms: Function, Return

Row, 40
position in the first dimension of a matrix or array.
Related Terms: Colum, Page

Row vector, 35
a vector whose elements are arranged horizontally.
Related Terms: Colon operator, Vector

Row-major order, 35
an order of processing elements in an array, in which all the elements of one row are processed before the elements of the next row.
Related Terms: Column-major order

Scalar, 34
a single number. A scalar is treated by MATLAB as a 1-by-1 array.
Related Terms: Array

Scientific notation, 17
a technique for expressing a number. The number is written in the form $x \times 10n$, where x may be signed (+ or −) and is expressed as a decimal number and n is an integer (which may be positive or negative). In MATLAB, $x \times 10n$ is expressed in the form x**e**n.
Related Terms:

Scope, 76
the set of statements that can access a variable.
Related Terms: Global scope, Local scope, Local variable

Script, 78
a collection of MATLAB commands in an M-file that does not contain a function.
Related Terms: M-file

Selection, 114
a control construct in which the interpreter decides which statement or block of statements is to be executed next on the basis of the value of an expression.
Related Terms: Branching, Control construct, If-else-statement, If-elseif-else-statement, If-elseif-statement, If-statement, Sequential control, Switch-statement

Semantics, 21
the meaning of a statement or set of statements.
Related Terms: Syntax

Sequential control, 114
a control construct in which the interpreter executes the statements in the order that they were written by the programmer.
Related Terms: Control construct, Loop, Selection

Sequential search, 297
a search method in which the target value that is being sought is compared with the first member of the list, then with the second, etc., until either the number is found, or the end of the list is reached, whichever comes first.
Related Terms: Binary search, Linear search

Short circuiting, 126
the skipping of the evaluation of a second operand by a logical operator because its value will have no effect on the result.
Related Terms: Logical operator

Simultaneous linear algebraic equations, 287
equations of the form $Ax=b$, where A is a matrix and x and b are both column vectors.
Related Terms: Linear algebra, Matrix algebra

Software, 27
a set of files containing source code, executable code, or both that describes a single computer program or a set of programs. It is called software to distinguish it from the hardware that makes up the physical part of a computer on which the programs run.
Related Terms: Bugs, Computer program

Source, 26
shorthand for source code, which is part or all of a computer program, written by a programmer that will be translated into executable code, which is written in a language that a computer can execute directly.
Related Terms: Executable code, Source code

Source code, 26
part or all of a program, written by a programmer that will be translated into executable code, which is written in a language that a computer can execute directly.
Related Terms: Code, Executable code, Source

Space complexity, 310
the shape of the plot of the amount of memory space required by an algorithm as a function of the size of its input.
Related Terms: Algorithmic complexity, Time complexity

Square matrix, 34
a matrix whose number of rows equals its number of columns.
Related Terms: Matrix

Stack, 269
an area of the computer's memory organized so that only one element, known as the element "on top of the stack", is accessible. Variables are placed on the top of the stack or taken from the top, and no variables are taken from the middle of the stack.
Related Terms: Stack frame

Stack frame, 269
an area in the computer's stack in which all the arguments and other local variables of a function are stored during the time from its call to its return. One stack frame is placed on the stack for each active function. [synonyms: frame, activation record]
Related Terms: Activation record, Frame, Stack

String, 74
a sequence of characters. In MATLAB, a string may be entered explicitly by typing a single quote, followed by some number of characters followed by a second single quote. A string is stored in MATLAB as a row vector of numbers of type **char**, each number being a code that represents one character.
Related Terms: Char

Struct, 208
a vector that is heterogeneous (elements may be of different types) and is indexed by field names instead of numerical indices (see index).
Related Terms: Field, Field name

Subarray operation, 41
an operation that accesses a rectangular subarray within a matrix or array. It is invoked by specifying a vector of integers for each index. The result is that elements from multiple rows and/or multiple columns (and for higher dimensional arrays, multiple pages, etc.) can be read or written.
Related Terms: Array, Matrix

Subclass, 313
in Object-Oriented Programming, a class that has been defined via the mechanism of inheritance to be an extension of some other class. The other class is called the "base class" of the subclass.
Related Terms: Base class, Class, Inheritance, Object-Oriented Programming

Subfunction, 75
a function in an M-file other than the first function in the file. It cannot be called from outside the file, but it may be called from inside the file. An M-file can contain any number of subfunctions.
Related Terms: M-file, Main function

Subscript, 40
a non-negative integer that enumerates the elements of a vector, a matrix, or an array. In MATLAB the integer must be positive (i.e., 0 is not allowed). One subscript is given for each dimension of the array, but see also linear indexing, in which only one integer is used.
Related Terms: Index, Linear indexing

Superclass, 313
[synonym for base class in Object-Oriented Programming]
Related Terms: Base class, Object-Oriented Programming

switch, 120
a MATLAB keyword that introduces the control-statement of a switch-statement.
Related Terms: Keyword, Switch-statement

Switch-statement, 120
a selection construct in which (a) the control-statement has an expression that governs the selection, (b) a set of statements is partitioned into blocks called "cases", each of which has its own associated case expression, (c) the first block whose case expression matches the value of the control-statement's expression is selected for execution, and (d) if there is no such match, then, if there is an optional block of statements introduced by the keyword `otherwise`, that block is executed, and, if there is no optional block, none of the statements in the set is executed. Alternative forms of the switch-statement are described in the text.
Related Terms: Case, Otherwise, Selection, Switch

Symbol table, 212
a table maintained automatically during the execution of code in some programming languages, including MATLAB, in which are kept the names (or "symbols") and addresses of variables that exist and are accessible during the execution and possibly other information about those variables, such as their types. In MATLAB, a collection of variables in a symbol table is called a "workspace".
Related Terms: Type, Variable, Workspace

Syntax, 21
the form of a statement (as opposed to its meaning, which is its semantics).
Related Terms: Semantics

Tail, 318
The last element of a linked list, i.e., an element that points at no other element in the list unless the list is a doubly linked list.
Related Terms: Doubly linked list, Head, Linked list

Text file, 238
a file that contains characters (i.e., letters, numbers, and punctuation marks) encoded in a standard format, such as ASCII, Unicode, or UTF-8.
Related Terms: ASCII, Binary file, File

Time complexity, 310
the shape of the plot of an algorithm's execution time as a function of the size of its input.
Related Terms: Algorithmic complexity, Space complexity

Transpose, 46
1. a matrix operation that interchanges all the elements of a matrix **X** so that **X(m,n)** is replaced by **X(n,m)**. **2**. the result of the transpose operation.
Related Terms: Transposing, Transposition operator

Transpose operator, 47
[synonym for transposition operator]
Related Terms: Transposition operator

Transposing, 46
performing the matrix operation that interchanges all the elements of a matrix **X** so that **X(m,n)** is replaced by **X(n,m)**.
Related Terms: Transpose

Transposition operator, 47
a MATLAB operator that produces that transposes of a matrix and takes the complex conjugate of its elements. It is a unary postfix operator. Its symbol is the apostrophe, or single quote, ('). synonyms: transpose operator and apostrophe operator
Related Terms: Apostrophe operator, Dot-apostrophe operator, Transpose, Transpose operator

Type, 197
a set values and a set of operations that can be performed on those values [a synonym for "data type"].
Related Terms: Conversion function, Data type, Mixed-mode arithmetic, Struct, Symbol table

Unary operator, 47
an operator, such as the transposition operator ('), that takes only one operand.
Related Terms: Binary operator

Underdetermined, 289
having fewer constraints than unknowns with the result that there are an infinite number of solutions. Applies, for example, to a set of simultaneous linear algebraic equations represented by $Ax = b$ where A has fewer rows than columns with the result that the number of solution vectors x is infinite.
Related Terms: Inconsistent, Linear algebra, Overdetermined

Variable, 16
a named location in memory into which values can be stored (computer science definition, as opposed to mathematics definition)
Related Terms: Assignment statement, Symbol table

Vector, 29
an ordered list of numbers. In MATLAB, it is also a matrix or array with exactly one column or exactly one row.
Related Terms: Array, Column vector, Matrix, Row vector, Vector command

Vector command, 169
a command that operates on an entire vector or an entire array.
Related Terms: Array, Vector, Vectorization

Vectorization, 169
the translation of code from a version that uses an explicit loop into one that uses a vector command.
Related Terms: Loop, Vector command
while, 155
a MATLAB keyword that introduces the control-statement of a while-loop.
Related Terms: Keyword, While-loop
While-loop, 154
a control construct that causes a block of statements to be executed repeatedly as long as a conditional in the control-statement remains true.
Related Terms: Conditional, For-loop, Implicit loop, Loop, While
Widget, 334
an element of a Graphical User Interface that provides output to the user or allows the user to perform input with the mouse or through natural gestures with a touch screen that mimic the use of a mechanical device. synonym: graphical control.
Related Terms: Graphical User Interface, GUI control
Workspace, 19
the set of variables that exist and are accessible during the execution of statements in the Command Window or the set of variables that exist and are accessible during the execution of the statements in a function.
Related Terms: Symbol table
Worst-case analysis 307
evaluation of an algorithm based solely on inputs for which the algorithm exhibits its worst algorithmic complexity.
Related Terms: Algorithmic complexity