CHAPTER 14

Advanced Mathematics

KEY TERMS

curve-fitting
best fit
symbolic mathematics
mean
sorting
index vectors
searching
arithmetic mean
average
outlier
harmonic mean
geometric mean
standard deviation
variance
mode
median
set operations
polynomials
degree
order
discrete
continuous
data sampling
interpolation
extrapolation
complex number
real part
imaginary part
purely imaginary
complex conjugate
magnitude
complex plane
linear algebraic equation
square matrix
main diagonal
diagonal matrix
trace
identity matrix
banded matrix
tridiagonal matrix
lower triangular matrix
upper triangular matrix
symmetric matrix
matrix inverse
matrix augmentation
coefficients
unknowns
determinant
Gauss-Jordan method
reduced row echelon form
integration
differentiation

CONTENTS

In this chapter, selected advanced mathematical concepts and related built-in functions in the MATLAB® software are introduced. This chapter begins with coverage of some simple statistics, as well as set operations that can be performed on data sets.

In many applications data are sampled, which results in discrete data points. Fitting a curve to the data is often desired. ***Curve-fitting*** is finding the curve that ***best fits*** the data. The third section in this chapter first explores fitting simple polynomial curves to data.

MATLAB®. https://doi.org/10.1016/B978-0-12-815479-3.00014-3

Other topics include complex numbers and a brief introduction to differentiation and integration in calculus. ***Symbolic mathematics*** means doing mathematics on symbols. Some of the symbolic math functions, all of which are in Symbolic Math Toolbox™ in MATLAB, are also introduced. (Note that this is a Toolbox and, as a result, may not be available universally.)

Solutions to sets of linear algebraic equations are important in many applications. To solve systems of equations using MATLAB, there are basically two methods, both of which will be covered in this chapter: using a matrix representation and using the **solve** function (which is part of Symbolic Math Toolbox™).

14.1 STATISTICAL FUNCTIONS

There are a lot of statistical analyses that can be performed on data sets. In MATLAB, the statistical functions are in the data analysis help topic called **datafun**.

Statistics can be used to characterize properties of a data set. For example, consider a set of exam grades {33, 75, 77, 82, 83, 85, 85, 91, 100}. What is a "normal," "expected," or "average" exam grade? There are several ways by which this could be interpreted. Perhaps the most common is the ***mean*** grade, which is found by summing the grades and dividing by the number of them (the result of that would be 79). Another way of interpreting that would be the grade found most often, which would be 85. Also, the value in the middle of the sorted list, 83, could be used. Another property that is useful to know is how spread out the data values are within the data set.

MATLAB has built-in functions for many statistics; the simplest of which we have already seen (e.g., **min** and **max** to find the minimum or maximum value in a data set).

Both of these functions also return the index of the smallest or largest value; if there is more than one occurrence, it returns the first. For example, in the following data set 10 is the largest value; it is found in three elements in the vector, but the index returned is the first element in which it is found (which is 2):

```
>> x = [9  10  10  9  8  7  3  10  9  8  5  10];
>> [maxval, maxind] = max(x)
maxval =
    10
maxind =
     2
```

For matrices, the **min** and **max** functions operate columnwise by default:

```
>> mat = [9  10  17  5;  19  9  11  14]
mat =
     9    10    17    5
    19     9    11   14
```

```
>> [minval, minind] = min(mat)
minval =
     9     9    11     5

minind =
     1     2     2     1
```

These functions can also compare vectors or matrices (with the same dimensions) and return the minimum (or maximum) values from corresponding elements. For example, the following iterates through all elements in the two vectors, comparing corresponding elements and returning the minimum for each:

```
>> x = [3 5 8 2 11];
>> y = [2 6 4 5 10];
>> min(x,y)
ans =
     2     5     4     2    10
```

MATLAB also has functions **mink** and **maxk** (introduced in R2017b) that return the minimum and maximum k values in an array. The values that are returned are in sorted order.

```
>> vec = randi(100,1,10)
vec =
    77    80    19    49    45    65    71    76    28    68
>> mink(vec, 3)
ans =
    19    28    45
>> mat = randi(33, 3,5)
mat =
    20     9    30     5    28
     8    17    32     5     9
    25    24    19     9    27
>> maxk(mat,2)
ans =
    25    24    32     9    28
    20    17    30     5    27
```

Some of the other functions in the **datafun** help topic that have been described already include **sum**, **prod**, **cumsum**, **cumprod**, **cummin**, **cummax**, and **histogram**. Other statistical operations, and the functions that perform them in MATLAB, will be described in the rest of this section.

14.1.1 Mean

The ***arithmetic mean*** of a data set is what is usually called the ***average*** of the values or, in other words, the sum of the values divided by the number of values in the data set. Mathematically, we would write this as $\frac{\sum_{i=1}^{n} x_i}{n}$.

THE PROGRAMMING CONCEPT

Calculating a mean, or average, would normally be accomplished by looping through the elements of a vector, adding them together, and then dividing by the number of elements:

mymean.m

```
function outv = mymean(vec)
% mymean returns the mean of a vector
% Format: mymean(vector)

mysum = 0;
for i=1:length(vec)
    mysum = mysum + vec(i);
end
outv = mysum/length(vec);
end
```

```
>> x = [9  10  10  9  8  7  3  10  9  8  5  10];
>> mymean(x)
ans =
    8.1667
```

THE EFFICIENT METHOD

There is a built-in function, **mean**, in MATLAB to accomplish this:

```
>> mean(x)
ans =
    8.1667
```

For a matrix, the **mean** function operates columnwise. To find the mean of each row, the dimension of 2 is passed as the second argument to the function, as is the case with the functions **sum**, **prod**, **cumsum**, and **cumprod** (the [] as a middle argument is not necessary for these functions like it is for **min** and **max**).

```
>> mat = [8  9  3;  10  2  3;  6  10  9]
mat =
     8     9     3
    10     2     3
     6    10     9
>> mean(mat)
ans =
     8     7     5
>> mean(mat,2)
ans =
    6.6667
    5.0000
    8.3333
```

Sometimes a value that is much larger or smaller than the rest of the data (called an ***outlier***) can throw off the mean. For example, in the following all of the numbers in the data set are in the range from 3 to 10, with the exception of the 100 in the middle. Because of this outlier, the mean of the values in this vector is actually larger than any of the other values in the vector.

```
>> xwithbig = [9  10  10  9  8  100  7  3  10  9  8  5  10];
>> mean(xwithbig)
ans =
    15.2308
```

Typically, an outlier like this represents an error of some kind, perhaps in the data collection. In order to handle this, sometimes the minimum and maximum values from a data set are discarded before the mean is computed. In this example, a **logical** vector indicating which elements are neither the largest nor smallest value is used to index into the original data set, resulting in removing the minimum and the maximum.

```
>> xwithbig = [9  10  10  9  8  100  7  3  10  9  8  5  10];
>> newx = xwithbig(xwithbig ~= min(xwithbig)  &  ...
                   xwithbig ~= max(xwithbig))
newx =
 9   10   10   9   8   7   10   9   8   5   10
```

Instead of just removing the minimum and maximum values, sometimes the largest and smallest 1% or 2% of values are removed, especially if the data set is very large.

There are several other means that can be computed. The ***harmonic mean*** of the n values in a vector or data set x is defined as

$$\frac{n}{\frac{1}{x_1}+\frac{1}{x_2}+\frac{1}{x_3}+\ldots+\frac{1}{x_n}}$$

The ***geometric mean*** of the n values in a vector x is defined as the nth root of the product of the data set values.

$$\sqrt[n]{x_1 * x_2 * x_3 * \ldots * x_n}$$

Both of these could be implemented as anonymous functions:

```
>> x = [9  10  10  9  8  7  3  10  9  8  5  10];
>> harmhand = @ (x) length(x) / sum(1 ./ x);
>> harmhand(x)
ans =
    7.2310
>> geomhand = @ (x) nthroot(prod(x), length(x));
>> geomhand(x)
ans =
     7.7775
```

Note that Statistics and Machine Learning Toolbox™ has functions for these means, called **harmmean** and **geomean**, as well as a function **trimmean** which trims the highest and lowest n% of data values, where the percentage n is specified as an argument.

14.1.2 Variance and Standard Deviation

The ***standard deviation*** and ***variance*** are ways of determining the spread of the data. The variance is usually defined in terms of the arithmetic mean as:

$$\text{var} = \frac{\sum_{i=1}^{n} (x_i - mean)^2}{n-1}$$

Sometimes, the denominator is defined as n rather than $n-1$. The default definition in MATLAB uses $n-1$ for the denominator, so we will use that definition here.

For example, for the vector [8 7 5 4 6], there are $n = 5$ values so $n-1$ is 4. Also, the mean of this data set is 6. The variance would be

$$\begin{aligned}\text{var} &= \frac{(8-6)^2 + (7-6)^2 + (5-6)^2 + (4-6)^2 + (6-6)^2}{4} \\ &= \frac{4+1+1+4+0}{4} = 2.5\end{aligned}$$

The built-in function to calculate the variance is called **var**:

```
>> xvals = [8 7 5 4 6];
>> myvar = var(xvals)
yvar =
    2.5000
```

The standard deviation is the square root of the variance:

$$\text{sd} = \sqrt{\text{var}}$$

The built-in function in MATLAB for the standard deviation is called **std**; the standard deviation can be found either as the **sqrt** of the variance or using **std**:

```
>> shortx = [2 5 1 4];
>> myvar = var(shortx)
myvar =
    3.3333
>> sqrt(myvar)
ans =
    1.8257
>> std(shortx)
ans =
    1.8257
```

The less spread out the numbers are, the smaller the standard deviation will be, as it is a way of determining the spread of the data. Likewise, the more spread out the numbers are, the larger the standard deviation will be. For example, the following are two data sets that have the same number of values and also the same mean, but the standard deviations are quite different:

```
>> x1 = [9  10  9.4  9.6];
>> mean(x1)
ans =
    9.5000
>> std(x1)
ans =
    0.4163

>> x2 = [2  17  -1.5  20.5];
>> mean(x2)
ans =
    9.5000
>> std(x2)
ans =
   10.8704
```

14.1.3 Mode

The ***mode*** of a data set is the value that appears most frequently. The built-in function in MATLAB for this is called **mode**.

```
>> x = [9  10  10  9  8  7  3  10  9  8  5  10];
>> mode(x)
ans =
   10
```

If there is more than one value with the same (highest) frequency, the smaller value is the mode. In the following case, as 3 and 8 appear twice in the vector, the smaller value (3) is the mode:

```
>> x = [3  8  5  3  4  1  8];
>> mode(x)
ans =
     3
```

Therefore, if no value appears more frequently than any other, the mode of the vector will be the same as the minimum.

14.1.4 Median

The ***median*** is defined only for a data set that has been ***sorted*** first, meaning that the values are in order. The median of a sorted set of n data values is defined as the value in the middle, if n is odd, or the average of the two values in the

middle if *n* is even. For example, for the vector [1 4 5 9 12], the middle value is 5. The function in MATLAB is called **median**:

```
>> median([1  4  5  9  12 ])
ans =
     5
```

For the vector [1 4 5 9 12 33], the median is the average of the 5 and 9 in the middle:

```
>> median([1  4  5  9  12  33 ])
ans =
     7
```

If the vector is not in sorted order to begin with, the **median** function will still return the correct result (it will sort the vector automatically). For example, scrambling the order of the values in the first example will still result in a median value of 5.

```
>> median([9  4  1  5  12 ])
ans =
     5
```

PRACTICE 14.1

For the vector [2 4 8 3 8], find the following

- minimum
- maximum
- arithmetic mean
- variance
- mode
- median

In MATLAB, find the harmonic mean and the geometric mean for this vector (either using **harmmean** and **geomean** if you have Statistics and Machine Learning Toolbox, or by creating anonymous functions if you do not).

PRACTICE 14.2

For matrices, the statistical functions will operate on each column. Create a 5 x 4 matrix of random integers, each in the range from 1 to 30. Write an expression that will find the mode of all numbers in the matrix (not column-by-column).

14.2 SET OPERATIONS

MATLAB has several built-in functions that perform ***set operations*** on vectors. These include **union**, **intersect**, **unique**, **setdiff**, and **setxor**. All of these functions can be useful when working with data sets. By default, in earlier versions of MATLAB, all returned vectors were sorted from lowest to highest (ascending order). Beginning with MATLAB Version R2013a, however, these set functions provide the option of having the results in sorted order or in the original order. Additionally, there are two "is" functions that work on sets: **ismember** and **issorted**.

For example, given the following vectors:

```
>> v1 = 6:-1:2
    6     5     4     3     2
>> v2 = 1:2:7
v2 =
    1     3     5     7
```

The **union** function returns a vector that contains all of the values from the two input argument vectors, without repeating any.

```
>> union(v1,v2)
ans =
     1     2     3     4     5     6     7
```

By default, the result is in sorted order, so passing the arguments in the reverse order would not affect the result. This is the same as calling the function as:

```
>> union(v1,v2, 'sorted')
```

If, instead, the character vector 'stable' is passed to the function, the result would be in the original order; this means that the order of the arguments would affect the result.

```
>> union(v1,v2,'stable')
ans =
     6     5     4     3     2     1     7
>> union(v2,v1,'stable')
ans =
     1     3     5     7     6     4     2
```

The **intersect** function instead returns all of the values that can be found in both of the two input argument vectors.

```
>> intersect(v1,v2)
ans =
     3     5
```

The **setdiff** function receives two vectors as input arguments and returns a vector consisting of all of the values that are contained in the first vector argument but not the second. Therefore, the result that is returned (not just the order) will depend on the order of the two input arguments.

```
>> setdiff(v1,v2)
ans =
     2     4     6
>> setdiff(v2,v1)
ans =
     1     7
```

The function **setxor** receives two vectors as input arguments and returns a vector consisting of all of the values from the two vectors that are not in the intersection of these two vectors. In other words, it is the union of the two vectors obtained using **setdiff** when passing the vectors in different orders, as seen before.

```
>> setxor(v1,v2)
ans =
     1     2     4     6     7
>> union(setdiff(v1,v2), setdiff(v2,v1))
ans =
     1     2     4     6     7
```

The set function **unique** returns all of the unique values from a set argument:

```
>> v3 = [1:5 3:6]
v3 =
    1     2     3     4     5     3     4     5     6
>> unique(v3)
ans =
    1     2     3     4     5     6
```

All of these functions—**union**, **intersect**, **unique**, **setdiff**, and **setxor**—can be called with 'stable' to have the result returned in the order given by the original vector(s).

Many of the set functions return vectors that can be used to index into the original vectors as optional output arguments. For example, the two vectors *v1* and *v2* were defined previously as:

```
>> v1
v1 =
    6     5     4     3     2
>> v2
v2 =
    1     3     5     7
```

The **intersect** function returns, in addition to the vector containing the values in the intersection of *v1* and *v2*, an index vector into *v1*, and an index vector into *v2* such that *outvec* is the same as *v1(index1)* and also *v2(index2)*.

```
>> [outvec, index1, index2] = intersect(v1,v2)
outvec =
     3     5

index1 =
     4

     2

index2 =
     2

     3
```

Using these vectors to index into *v1* and *v2* will return the values from the intersection. For example, this expression returns the second and fourth elements of *v1* (it puts them in ascending order):

```
>> v1(index1)
ans =
     3     5
```

This returns the second and third elements of *v2*:

```
>> v2(index2)
ans =
     3     5
```

The function **ismember** receives two vectors as input arguments and returns a **logical** vector that is the same length as the first argument, containing **logical** 1 for **true** if the element in the first vector is also in the second, or **logical** 0 for **false** if not. The order of the arguments matters for this function.

```
>> v1
v1 =
     6     5     4     3     2
>> v2
v2 =
     1     3     5     7
>> ismember(v1,v2)
ans =
     0     1     0     1     0
>> ismember(v2,v1)
ans =
     0     1     1     0
```

Using the result from the **ismember** function as an index into the first vector argument will return the same values as the **intersect** function (although not necessarily sorted).

```
>> logv = ismember(v1,v2)
logv =
     0     1     0     1     0
>> v1(logv)
ans =
     5     3
>> logv = ismember(v2,v1)
logv =
     0     1     1     0
>> v2(logv)
ans =
     3     5
```

The **issorted** function will return **logical** 1 for **true** if the argument is sorted in ascending order, or **logical** 0 for **false** if not.

```
>> v3 = [1:5 3:6]
v3 =
     1     2     3     4     5     3     4     5     6
>> issorted(v3)
ans =
     0
>> issorted(v2)
ans =
     1
```

PRACTICE 14.3

Create two vector variables *vec1* and *vec2* that contain five random integers, each in the range from 1 to 20. Do each of the following operations by hand first and then check in MATLAB (if you have one of the latest versions, do this with both 'stable' and 'sorted'):

- union
- intersection
- setdiff
- setxor
- unique (for each)

14.3 FITTING CURVES TO DATA

MATLAB has several curve-fitting functions; Curve-Fitting Toolbox™ has many more of these functions. Some of the simplest curves are polynomials of different degrees, which are described next.

14.3.1 Polynomials

Simple curves are ***polynomials*** of different ***degrees*** or ***orders***. The degree is the integer of the highest exponent in the expression. For example:

- a straight line is a first-order (or degree 1) polynomial of the form `ax + b`, or, more explicitly, $ax^1 + b$
- a quadratic is a second-order (or degree 2) polynomial of the form $ax^2 + bx + c$
- a cubic (degree 3) is of the form $ax^3 + bx^2 + cx + d$

MATLAB represents a polynomial as a row vector of coefficients. For example, the polynomial $x^3 + 2x^2 - 4x + 3$ would be represented by the vector `[1 2 -4 3]`. The polynomial $2x^4 - x^2 + 5$ would be represented by `[2 0 -1 0 5]`; note the zero terms for x^3 and x^1.

The **roots** function in MATLAB can be used to find the roots of an equation represented by a polynomial. For example, for the mathematical function:

$f(x) = 4x^3 - 2x^2 - 8x + 3$

to solve the equation $f(x) = 0$:

```
>> roots([4 -2 -8 3])
ans =
   -1.3660
    1.5000
    0.3660
```

The function **polyval** will evaluate a polynomial *p* at *x*; the form is **polyval(p,x)**. For example, the polynomial $-2x^2 + x + 4$ is evaluated at `x = 3`, which yields

$-2 * 3^2 + 3 + 4$, or -11:

```
>> p = [-2  1  4];
>> polyval(p,3)
ans =
   -11
```

The argument *x* can be a vector:

```
>> polyval(p,1:3)
ans =
     3    -2   -11
```

14.3.2 Curve-Fitting

Data that we acquire to analyze can be either ***discrete*** (e.g., a set of object weights) or ***continuous***. In many applications, continuous properties are ***sampled***, such as:

- The temperature recorded every hour
- The speed of a car recorded every one tenth of a mile
- The mass of a radioactive material recorded every second as it decays
- Audio from a sound wave as it is converted to a digital audio file

Sampling provides data in the form of (x,y) points, which could then be plotted. For example, let us say the temperature was recorded every hour one afternoon from 2:00 pm to 6:00 pm; the vectors might be:

```
>> x = 2:6;
>> y = [65  67  72  71  63];
```

14.3.3 Interpolation and Extrapolation

In many cases, estimating values other than at the sampled data points is desired. For example, we might want to estimate what the temperature was at 2:30 pm, or at 1:00 pm. ***Interpolation*** means estimating the values in between recorded data points. ***Extrapolation*** is estimating outside of the bounds of the recorded data. One way to do this is to fit a curve to the data and use this for the estimations. Curve-fitting is finding the curve that "best fits" the data.

Simple curves are polynomials of different degrees, as described previously. Thus, curve-fitting involves finding the best polynomials to fit the data; for example, for a quadratic polynomial in the form $ax^2 + bx + c$, it means finding the values of a, b, and c that yield the best fit. Finding the best straight line that goes through data would mean finding the values of a and b in the equation $ax + b$.

MATLAB has a function to do this called **polyfit**. The function **polyfit** finds the coefficients of the polynomial of the specified degree that best fits the data using a least squares algorithm. There are three arguments passed to the function: x and y vectors that represent the data, and the degree of the desired polynomial. For example, to fit a straight line (degree 1) through the points representing temperatures, the call to the **polyfit** function would be

```
>> polyfit(x,y,1)
ans =
    0.0000    67.6000
```

which says that the best straight line is of the form $0x + 67.6$.

However, from the plot (shown in Figure 14.1), it looks like a quadratic would be a much better fit. The following would create the vectors and then fit a polynomial of degree 2 through the data points, storing the values in a vector called *coefs*.

```
>> x = 2:6;
>> y = [65  67  72  71  63];
>> coefs = polyfit(x,y,2)
coefs =
   -1.8571   14.8571   41.6000
```

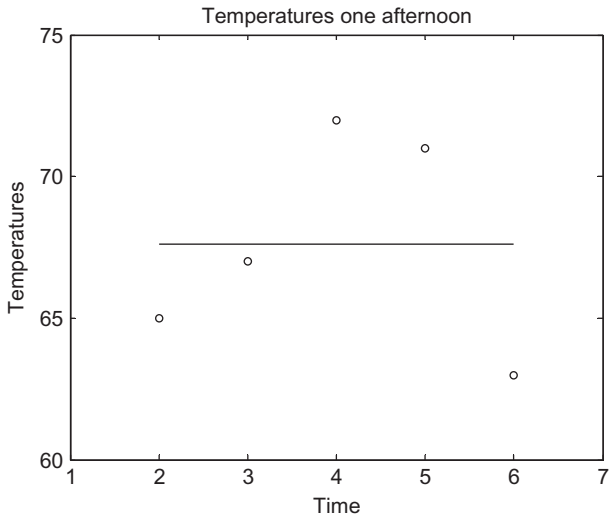


FIGURE 14.1
Sampled temperatures with straight line fit.

This says that the **polyfit** function has determined that the best quadratic that fits these data points is $-1.8571x^2 + 14.8571x + 41.6$. So, the variable *coefs* now stores a coefficient vector that represents this polynomial.

The function **polyval** can then be used to evaluate the polynomial at specified values. For example, we could evaluate at every value in the *x* vector:

```
>> curve = polyval(coefs,x)
curve =
   63.8857    69.4571    71.3153    69.4571    63.8857
```

This results in *y* values for each point in the *x* vector and stores them in a vector called *curve*. Putting all of this together, the following script called *polytemp* creates the *x* and *y* vectors, fits a second-order polynomial through these points, and plots both the points and the curve on the same figure. Calling this results in the plot seen in Figure 14.2. The curve doesn't look very smooth on this plot, but that is because there are only five points in the *x* vector.

polytemp.m

```
% Fits a quadratic curve to temperature data
x= 2:6;
y =[65   67   72   71   63];
coefs = polyfit(x,y,2);
curve = polyval(coefs,x);
plot(x,y,'ro',x,curve)
xlabel('Time')
ylabel('Temperatures')
title('Temperatures one afternoon')
axis([1 7 60 75])
```

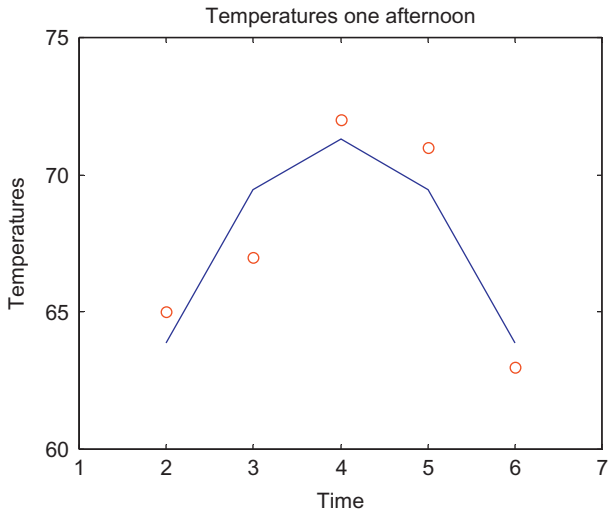


FIGURE 14.2
Sampled temperatures with quadratic curve.

PRACTICE 14.4

To make the curve smoother, modify the script *polytemp* to create a new x vector with more points for plotting the curve. Note that the original x vector for the data points must remain as is.

To estimate the temperature at different times, **polyval** can be used for discrete *x* points; it does not have to be used with the entire *x* vector. For example, to interpolate between the given data points and estimate what the temperature was at 2:30 pm, 2.5 would be used.

```
>> polyval(coefs,2.5)
ans =
    67.1357
```

Also, **polyval** can be used to extrapolate beyond the given data points. For example, to estimate the temperature at 1:00 pm:

```
>> polyval(coefs,1)
ans =
    54.6000
```

The better the curve fit, the more accurate these interpolated and extrapolated values will be.

Using the **subplot** function, we can loop to show the difference between fitting curves of degrees 1, 2, and 3 to some data. For example, the following script will

accomplish this for the temperature data. (Note that the variable *morex* stores 100 points so the graph will be smooth.)

polytempsubplot.m

```
% Fits curves of degrees 1-3 to temperature
%  data and plots in a subplot
x = 2:6;
y = [65  67  72  71  63];
morex = linspace(min(x),max(x));
for pd = 1:3
    coefs = polyfit(x,y,pd);
    curve = polyval(coefs,morex);
    subplot(1,3,pd)
    plot(x,y,'ro',morex,curve)
    xlabel('Time')
    ylabel('Temperatures')
    title(sprintf('Degree %d',pd))
    axis([1  7  60  75])
end
```

Executing the script

```
>> polytempsubplot
```

creates the Figure Window shown in Figure 14.3.

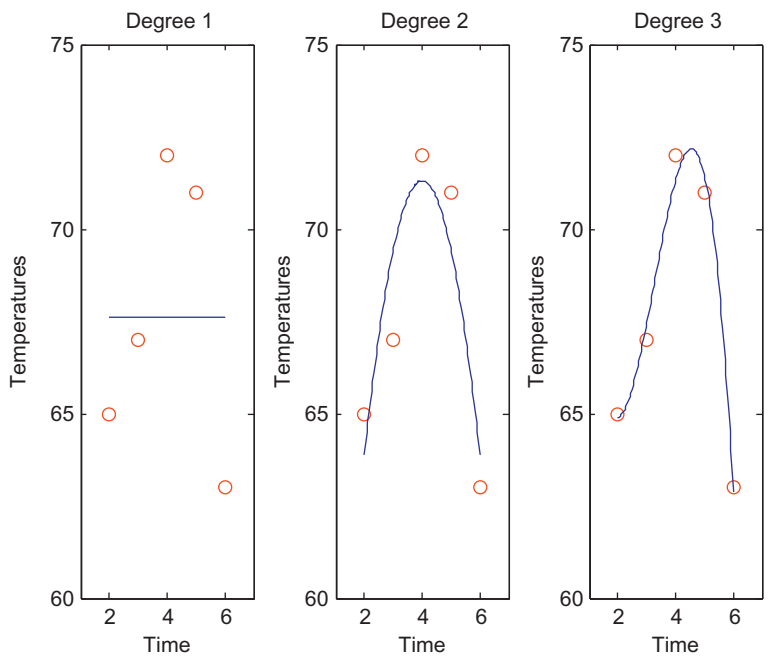


FIGURE 14.3
Subplot to show temperatures with curves of degrees 1, 2, and 3.

14.4 COMPLEX NUMBERS

A ***complex number*** is generally written in the form

```
z = a + bi
```

Note
This is the way mathematicians usually write a complex number; in engineering it is often written as $a+bj$, where j is $\sqrt{-1}$.

where a is called the ***real part*** of the number z, b is the ***imaginary part*** of z, and i is $\sqrt{-1}$. A complex number is ***purely imaginary*** if it is of the form z = bi (in other words if *a* is 0).

We have seen that in MATLAB both **i** and **j** are built-in functions that return $\sqrt{-1}$ (so, they can be thought of as built-in constants). Complex numbers can be created using **i** or **j**, such as "5 + 2i" or "3 − 4j". The multiplication operator is not required between the value of the imaginary part and the constant **i** or **j**.

QUICK QUESTION!

Is the value of the expression "3i" the same as "3*i"?

Answer: It depends on whether *i* has been used as a variable name or not. If *i* has been used as a variable (e.g., an iterator variable in a **for** loop), then the expression "3*i" will use the defined value for the variable and the result will not be a complex number. The expression "3i" will always be complex. Therefore, it is a good idea when working with complex numbers to use 1**i** or 1**j** rather than just **i** or **j**. The expressions 1**i** and 1**j** always result in a complex number, regardless of whether *i* and *j* have been used as variables. So, use "3*1i" or "3i".

```
>> i = 5;
>> i
i =
     5
>> 1i
ans =
     0 + 1.0000i
```

MATLAB also has a function **complex** that will return a complex number. It receives two numbers, the real and imaginary parts in that order, or just one number, which is the real part (in which case the imaginary part would be 0). Here are some examples of creating complex numbers in MATLAB:

```
>> z1 = 4 + 2i
z1 =
   4.0000 + 2.0000i

>> z2 = sqrt(-5)
z2 =
   0 + 2.2361i

>> z3 = complex(3,-3)
z3 =
   3.0000 - 3.0000i
```

```
>> z4 = 2 + 3j
z4 =
   2.0000 + 3.0000i

>> z5 = (-4) ^ (1/2)
ans =
    0.0000 + 2.0000i

>> myz = input('Enter a complex number: ')
Enter a complex number: 3 + 4i
myz =
    3.0000 + 4.0000i
```

Note that even when **j** is used in an expression, **i** is used in the result. MATLAB shows the type of the variables created here in the Workspace Window as **double (complex)**, or by using **whos** as **double** with the attribute complex. MATLAB has functions **real** and **imag** that return the real and imaginary parts of complex numbers.

```
>> real(z1)
ans =
   4

>> imag(z3)
ans =
   -3
```

In order to print an imaginary number, the **disp** function will display both parts automatically:

```
>> disp(z1)
   4.0000 + 2.0000i
```

The **fprintf** function will only print the real part unless both parts are printed separately:

```
>> fprintf('%f\n', z1)
4.000000

>> fprintf('%f + %fi\n', real(z1), imag(z1))
4.000000 + 2.000000i
```

The function **isreal** returns **logical** 1 for **true** if there is no imaginary part of the argument or **logical** 0 for **false** if the argument does have an imaginary part (even if it is 0). For example,

```
>> isreal(z1)
ans =
     0
```

```
>> z6 = complex(3)
z5 =
     3
>> isreal(z6)
ans =
     0

>> isreal(3.3)
ans =
     1
```

For the preceding variable *z6*, even though it shows the answer as 3, it is really stored as 3 + 0i, and that is how it is displayed in the Workspace Window. Therefore, **isreal** returns **logical false** as it is stored as a complex number.

14.4.1 Equality for Complex Numbers

Two complex numbers are equal to each other if both their real parts and imaginary parts are equal. In MATLAB, the equality operator can be used.

```
>> z1 == z2
ans =
     0

>> complex(0,4) == sqrt(-16)
ans =
     1
```

14.4.2 Adding and Subtracting Complex Numbers

For two complex numbers z1 = a + bi and z2 = c + di,

```
z1 + z2 = (a + c) + (b + d)i
z1 – z2 = (a – c) + (b – d)i
```

As an example, we will write a function in MATLAB to add two complex numbers together and return the resulting complex number.

THE PROGRAMMING CONCEPT

In most cases, to add two complex numbers together you would have to separate the real and imaginary parts, and add them to return your result.

addcomp.m

```
function outc = addcomp(z1, z2)
% addcomp adds two complex numbers z1 and z2 &
%    returns the result
% Adds the real and imaginary parts separately
% Format: addcomp(z1,z2)

realpart = real(z1) + real(z2);
imagpart = imag(z1) + imag(z2);
outc = realpart + imagpart * 1i;
end
```

```
>> addcomp(3+4i, 2-3i)
ans =
   5.0000 + 1.0000i
```

THE EFFICIENT METHOD

MATLAB does this automatically to add two complex numbers together (or subtract).

```
>> z1 = 3 + 4i;
>> z2 = 2 - 3i;
>> z1+z2
ans =
 5.0000 + 1.0000i
```

14.4.3 Multiplying Complex Numbers

For two complex numbers z1 = a + bi and z2 = c + di,

```
z1 * z2 = (a + bi) * (c + di)
     = a*c + a*di + c*bi + bi*di
     = a*c + a*di + c*bi – b*d
     = (a*c – b*d) + (a*d + c*b)i
```

For example, for the complex numbers

```
z1 = 3 + 4i
z2 = 1 - 2i
```

the result of the multiplication would be defined mathematically as

```
z1 * z2 = (3*1 - -8) + (3*-2 + 4*1)i = 11 -2i
```

This is, of course, automatic in MATLAB:

```
>> z1*z2
ans =
   11.0000 - 2.0000i
```

14.4.4 Complex Conjugate and Absolute Value

The ***complex conjugate*** of a complex number z = a + bi is $\bar{z} = a - bi$. The ***magnitude*** or absolute value of a complex number z is $|z| = \sqrt{a^2 + b^2}$. In MATLAB, there is a built-in function **conj** for the complex conjugate, and the **abs** function returns the absolute value.

```
>> z1 = 3 + 4i
z1 =
   3.0000 + 4.0000i

>> conj(z1)
ans =
   3.0000 - 4.0000i

>> abs(z1)
ans =
     5
```

14.4.5 Complex Equations Represented as Polynomials

We have seen that MATLAB represents a polynomial as a row vector of coefficients; this can be used when the expressions or equations involve complex numbers, also. For example, the polynomial $z^2 + z - 3 + 2i$ would be represented by the vector `[1  1  -3+2i]`. The **roots** function in MATLAB can be used to find the roots of an equation represented by a polynomial. For example, to solve the equation $z^2 + z - 3 + 2i = 0$:

```
>> roots([1  1  -3+2i])
ans =
  -2.3796 + 0.5320i
   1.3796 - 0.5320i
```

The **polyval** function can also be used with this polynomial, for example

```
>> cp = [1  1  -3+2i]
cp =
   1.0000        1.0000       -3.0000 + 2.0000i
>> polyval(cp,3)
ans =
   9.0000 + 2.0000i
```

14.4.6 Polar Form

Any complex number z = a + bi can be thought of as a point (a,b) or vector in a ***complex plane*** in which the horizontal axis is the real part of z, and the vertical axis is the imaginary part of z. So, *a* and *b* are the Cartesian or rectangular coordinates. As a vector can be represented by either its rectangular or polar coordinates, a complex number can also be given by its polar coordinates *r* and θ, where *r* is the magnitude of the vector and θ is an angle.

To convert from the polar coordinates to the rectangular coordinates:

```
a = r cos θ
b = r sin θ
```

To convert from the rectangular to polar coordinates:

$$r = |z| = \sqrt{a^2 + b^2}$$

$$\theta = \arctan\left(\frac{b}{a}\right)$$

So, a complex number z = a + bi can be written as `r cos θ + (r sin θ)i` or

```
z = r (cos θ + i sin θ)
```

As $e^{i\theta} = \cos\theta + i \sin\theta$, a complex number can also be written as $z = re^{i\theta}$. In MATLAB, r can be found using the **abs** function, while there is a built-in function called **angle** to find θ.

Note

Note that the function is **atan** in MATLAB.

```
>> z1 = 3 + 4i;
r = abs(z1)
r =
     5
>> theta = angle(z1)
theta =
     0.9273
>> r*exp(i*theta)
ans =
   3.0000 + 4.0000i
```

14.4.7 Plotting

Several methods are used commonly for plotting complex data:

- plot the real parts versus the imaginary parts using **plot**
- plot only the real parts using **plot**
- plot the real and the imaginary parts in one figure with a legend, using **plot**
- plot the magnitude and angle using **polarplot** (or, prior to R2016a, **polar**)

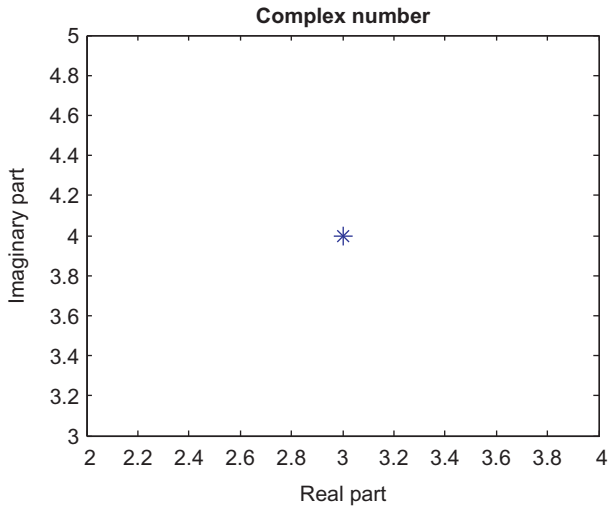


FIGURE 14.4
Plot of complex number.

Using the **plot** function with a single complex number or a vector of complex numbers will result in plotting the real parts versus the imaginary parts, for example, **plot(z)** is the same as **plot(real(z), imag(z))**. Thus, for the complex number `z1 = 3 + 4i`, this will plot the point (3,4) (using a large asterisk so we can see it!), as shown in Figure 14.4.

```
>> z1 = 3 + 4i;
>> plot(z1,'*', 'MarkerSize', 12)
>> xlabel('Real part')
>> ylabel('Imaginary part')
>> title('Complex number')
```

PRACTICE 14.5

Create the following complex variables

```
c1 = complex(0,2);
c2 = 3 + 2i;
c3 = sqrt(-4);
```

Then, carry out the following:

- Get the real and imaginary parts of *c2*.
- Print the value of *c1* using **disp**.
- Print the value of *c2* in the form 'a+bi'.
- Determine whether any of the variables are equal to each other.
- Subtract *c2* from *c1*.
- Multiply *c2* by *c3*.
- Get the complex conjugate and magnitude of *c2*.
- Put *c1* in polar form.
- Plot the real part versus the imaginary part for *c2*.

14.5 MATRIX SOLUTIONS TO SYSTEMS OF LINEAR ALGEBRAIC EQUATIONS

A ***linear algebraic equation*** is an equation of the form

$$a_1x_1 + a_2x_2 + a_3x_3 + \ldots + a_nx_n = b$$

Solutions to sets of equations in this form are important in many applications. In MATLAB, to solve systems of equations, there are basically two methods:

- using a matrix representation
- using the **solve** function (which is part of Symbolic Math Toolbox™)

In this section, we will first investigate some relevant matrix properties and then use these to solve linear algebraic equations. The use of ***symbolic mathematics*** including the **solve** function will be covered in the next section.

14.5.1 Matrix Properties

In Chapter 2, we saw several common operations on matrices. In this section, we will examine some properties that will be useful in solving equations using a matrix form.

14.5.1.1 Square Matrices

If a matrix has the same number of rows and columns (e.g., if $m == n$), the matrix is ***square***. The definitions that follow in this section only apply to square matrices.

The ***main diagonal*** of a square matrix (sometimes called just the ***diagonal***) is the set of terms a_{ii} for which the row and column indices are the same, so from the upper left element to the lower right. For example, for the following matrix, the diagonal consists of 1, 6, 11, and 16.

$$\begin{bmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ 9 & 10 & 11 & 12 \\ 13 & 14 & 15 & 16 \end{bmatrix}$$

A square matrix is a ***diagonal matrix*** if all values that are not on the diagonal are 0. The numbers on the diagonal, however, do not have to be all nonzero, although frequently they are. Mathematically, this is written as $a_{ij} = 0$ for $i \sim= j$. The following is an example of a diagonal matrix:

$$\begin{bmatrix} 4 & 0 & 0 \\ 0 & 9 & 0 \\ 0 & 0 & 5 \end{bmatrix}$$

MATLAB has a function **diag** that will return the diagonal of a matrix as a column vector; transposing will result in a row vector instead.

```
>> mymat = reshape(1:16,4,4)'
mymat =
     1     2     3     4
     5     6     7     8
     9    10    11    12
    13    14    15    16
>> diag(mymat)'
ans =
     1     6    11    16
```

The **diag** function can also be used to take a vector of length n and create an $n \times n$ square diagonal matrix with the values from the vector on the diagonal:

```
>> v=1:4;
>> diag(v)
ans =
     1     0     0     0
     0     2     0     0
     0     0     3     0
     0     0     0     4
```

So, the **diag** function can be used two ways: (i) pass a matrix and it returns a vector, or (ii) pass a vector and it returns a matrix!

The ***trace*** of a square matrix is the sum of all of the elements on the diagonal. For example, for the diagonal matrix created using v, it is 1 + 2 + 3 + 4, or 10.

QUICK QUESTION!

How could we calculate the trace of a square matrix?

Answer: See the following Programming Concept and Efficient Method.

THE PROGRAMMING CONCEPT

To calculate the trace of a square matrix, only one loop is necessary as the only elements in the matrix we are referring to have subscripts (i, i). So, once the size has been determined, the loop variable can iterate from 1 through the number of rows or from 1 through the number of columns (it doesn't matter which, as they have the same value!). The following function calculates and returns the trace of a square matrix or an empty vector if the matrix argument is not square.

THE PROGRAMMING CONCEPT—CONT'D

mytrace.m

```
function outsum = mytrace(mymat)
% mytrace calculates the trace of a square matrix
% or an empty vector if the matrix is not square
% Format: mytrace(matrix)

[r, c] = size(mymat);
if r ~= c
   outsum = [];
else
   outsum = 0;
   for i = 1:r
        outsum = outsum + mymat(i,i);
   end
end
end
```

```
>> mymat = reshape(1:16,4,4)'
mymat =
     1     2     3     4
     5     6     7     8
     9    10    11    12
    13    14    15    16
>> mytrace(mymat)
ans =
    34
```

THE EFFICIENT METHOD

In MATLAB, there is a built-in function **trace** to calculate the trace of a square matrix:

```
>> trace(mymat)
ans =
   34
```

A square matrix is an ***identity*** matrix called I if $a_{ij} = 1$ for $i == j$ and $a_{ij} = 0$ for $i \sim= j$. In other words, all of the numbers on the diagonal are 1 and all others are 0. The following is a 3×3 identity matrix:

$$\begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

Note that any identity matrix is a special case of a diagonal matrix.

Identity matrices are very important and useful. MATLAB has a built-in function **eye** that will create an $n \times n$ identity matrix, given the value of n:

```
>> eye(5)
ans =
     1     0     0     0     0
     0     1     0     0     0
     0     0     1     0     0
     0     0     0     1     0
     0     0     0     0     1
```

Note that **i** is built into MATLAB as the square root of -1, so another name is used for the function that creates an identity matrix: **eye**, which sounds like "i" (... get it?)

QUICK QUESTION!

What happens if a matrix M is multiplied by an identity matrix (of the appropriate size)?

Answer: For the size to be appropriate, the dimensions of the identity matrix would be the same as the number of columns of M. The result of the multiplication will always be the original matrix M (thus, it is similar to multiplying a scalar by 1).

```
>> M = [1 2 3 1; 4 5 1 2; 0 2 3 0]
M =
     1     2     3     1
     4     5     1     2
     0     2     3     0

>> [r, c] = size(M);
>> M * eye(c)
ans =
     1     2     3     1
     4     5     1     2
     0     2     3     0
```

Several special cases of matrices are related to diagonal matrices.

A ***banded matrix*** is a matrix of all 0s, with the exception of the main diagonal and other diagonals next to (above and below) the main. For example, the following matrix has 0s, except for the band of three diagonals; this is a particular kind of banded matrix called a ***tridiagonal matrix***.

$$\begin{bmatrix} 1 & 2 & 0 & 0 \\ 5 & 6 & 7 & 0 \\ 0 & 10 & 11 & 12 \\ 0 & 0 & 15 & 16 \end{bmatrix}$$

A ***lower triangular matrix*** has all 0s above the main diagonal. For example,

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 5 & 6 & 0 & 0 \\ 9 & 10 & 11 & 0 \\ 13 & 14 & 15 & 16 \end{bmatrix}$$

An ***upper triangular matrix*** has all 0s below the main diagonal. For example,

$$\begin{bmatrix} 1 & 2 & 3 & 4 \\ 0 & 6 & 7 & 8 \\ 0 & 0 & 11 & 12 \\ 0 & 0 & 0 & 16 \end{bmatrix}$$

It is possible for there to be 0s on the diagonal and in the lower part or upper part and still be a lower or upper triangular matrix, respectively.

MATLAB has functions **triu** and **tril** that will take a matrix and make it into an upper triangular or lower triangular matrix by replacing the appropriate elements with 0s. For example, the results from the **triu** function are shown:

```
>> mymat
mymat =
     1    2    3    4
     5    6    7    8
     9   10   11   12
    13   14   15   16
>> triu(mymat)
ans =
     1    2    3    4
     0    6    7    8
     0    0   11   12
     0    0    0   16
```

A square matrix is ***symmetric*** if $a_{ij} = a_{ji}$ for all i, j. In other words, all of the values opposite the diagonal from each other must be equal to each other. In this example, there are three pairs of values opposite the diagonals, all of which are equal (the 2s, the 9s, and the 4s).

$$\begin{bmatrix} 1 & 2 & 9 \\ 2 & 5 & 4 \\ 9 & 4 & 6 \end{bmatrix}$$

PRACTICE 14.6

For the following matrices:

$$A = \begin{bmatrix} 4 & 3 \\ 3 & 2 \end{bmatrix} \quad B = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix} \quad C = \begin{bmatrix} 1 & 0 & 0 \\ 4 & 6 & 0 \\ 3 & 1 & 3 \end{bmatrix}$$

Which are equal?

Which are square?

For all square matrices:

- Calculate the trace.
- Which are symmetric?
- Which are diagonal?
- Which are lower triangular?
- Which are upper triangular?

MATLAB has several "is" functions that determine whether or not matrices have some of the properties explained in this section, e.g., **isdiag**, **issymmetric**, **istril**, **istriu**, and **isbanded**; these functions were introduced in R2014b.

14.5.1.2 Matrix Operations

There are several common operations on matrices, some of which we have seen already. These include matrix transpose, matrix augmentation, and matrix inverse.

A matrix transpose interchanges the rows and columns of a matrix. For a matrix A, its transpose is written A^T in mathematics. For example, if

$$A = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix}$$

then

$$A^T = \begin{bmatrix} 1 & 4 \\ 2 & 5 \\ 3 & 6 \end{bmatrix}$$

In MATLAB, as we have seen, there is a built-in transpose operator, the apostrophe.

If the result of multiplying a matrix A by another matrix is the identity matrix I, then the second matrix is the ***inverse*** of matrix A. The inverse of a matrix A is written as A^{-1}, so

$$AA^{-1} = I$$

To actually compute the inverse A^{-1} of a matrix by hand is not so easy. MATLAB, however, has a function **inv** to compute a matrix inverse. For example, here a matrix is created, its inverse is found, and then multiplied by the original matrix to verify that the product is in fact the identity matrix:

```
>> a = [1 2; 2 2]
a =
     1     2
     2     2
>> ainv = inv(a)
ainv =
   -1.0000    1.0000
    1.0000   -0.5000
>> a*ainv
ans =
     1     0
     0     1
```

Matrix augmentation means adding column(s) to the original matrix. In MATLAB, matrix augmentation can be accomplished using square brackets to concatenate the two matrices. The square matrix *A* is concatenated with an identity matrix which has the same size as the matrix *A*:

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

>> [A eye(size(A))]
ans =
     1     3     7     1     0     0
     2     5     4     0     1     0
     9     8     6     0     0     1
```

14.5.2 Linear Algebraic Equations

A ***linear algebraic equation*** is an equation of the form

$$a_1x_1 + a_2x_2 + a_3x_3 + \ldots + a_nx_n = b$$

where the *a*s are constant ***coefficients***, the *x*s are the ***unknowns***, and *b* is a constant. A solution is a sequence of numbers that satisfies the equation. For example,

$$4x_1 + 5x_2 - 2x_3 = 16$$

is such an equation in which there are three unknowns: x_1, x_2, and x_3. One solution to this equation is $x_1 = 3$, $x_2 = 4$, and $x_3 = 8$, as 4*3 + 5*4 − 2*8 is equal to 16.

A system of linear algebraic equations is a set of equations of the form:

$$\begin{array}{ccccccccccc}
a_{11}x_1 & + & a_{12}x_2 & + & a_{13}x_3 & + & \ldots & + & a_{1n}x_n & = & b_1 \\
a_{21}x_1 & + & a_{22}x_2 & + & a_{23}x_3 & + & \ldots & + & a_{2n}x_n & = & b_2 \\
a_{31}x_1 & + & a_{32}x_2 & + & a_{33}x_3 & + & \ldots & + & a_{3n}x_n & = & b_3 \\
\vdots & & \vdots & & \vdots & & & & \vdots & & \vdots \\
a_{m1}x_1 & + & a_{m2}x_2 & + & a_{m3}x_3 & + & \ldots & + & a_{mn}x_n & = & b_m
\end{array}$$

This is called an $m \times n$ system of equations; there are m equations and n unknowns.

Because of the way that matrix multiplication works, these equations can be represented in matrix form as A x = b where A is a matrix of the coefficients, x is a column vector of the unknowns, and b is a column vector of the constants from the right side of the equations:

$$\begin{array}{ccc} \mathrm{A} & \mathrm{x} & \mathrm{b} \\ \begin{bmatrix} a_{11} & a_{12} & a_{13} & \ldots & a_{1n} \\ a_{21} & a_{22} & a_{23} & \ldots & a_{2n} \\ a_{31} & a_{32} & a_{33} & \ldots & a_{3n} \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ a_{m1} & a_{m2} & a_{m3} & \ldots & a_{mn} \end{bmatrix} & \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ \vdots \\ x_n \end{bmatrix} & = \begin{bmatrix} b_1 \\ b_2 \\ b_3 \\ \vdots \\ b_m \end{bmatrix} \end{array}$$

Once the system of equations has been written in matrix form, what we want is to solve the equation $Ax = b$ for the unknowns x. To do this, we need to isolate x on one side of the equation. If we were working with scalars, we would divide both sides of the equation by A. In fact, with MATLAB, we can use the **divided into** operator to do this. However, most languages cannot do this with matrices, so, instead, we multiply both sides of the equation by the inverse of the coefficient matrix A:

$$A^{-1} A x = A^{-1} b$$

Then, because multiplying a matrix by its inverse results in the identity matrix I, and because multiplying any matrix by I results in the original matrix, we have:

$$I x = A^{-1} b$$

or

$$x = A^{-1} b$$

For example, consider the following three equations with three unknowns—x_1, x_2, and x_3:

$$\begin{aligned}
4x_1 - 2x_2 + 1x_3 &= 7 \\
1x_1 + 1x_2 + 5x_3 &= 10 \\
-2x_1 + 3x_2 - 1x_3 &= 2
\end{aligned}$$

We write this in the form Ax = b, where A is a matrix of the coefficients, x is a column vector of the unknowns x_i, and b is a column vector of the values on the right side of the equations:

$$\underset{\text{A}}{\begin{bmatrix} 4 & -2 & 1 \\ 1 & 1 & 5 \\ -2 & 3 & -1 \end{bmatrix}} \underset{\text{x}}{\begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix}} = \underset{\text{b}}{\begin{bmatrix} 7 \\ 10 \\ 2 \end{bmatrix}}$$

The solution is then `x = A^-1 b`. In MATLAB, there are two simple ways to solve this. The built-in function **inv** can be used to get the inverse of A and then we multiply this by b, or we can use the divided into operator.

```
>> A = [4 -2 1; 1 1 5; -2 3 -1];
>> b = [7;10;2];
>> x = inv(A)*b
x =
    3.0244
    2.9512
    0.8049

>> x = A\b
x =
    3.0244
    2.9512
    0.8049
```

14.5.2.1 Solving 2 × 2 Systems of Equations

Although this may seem easy in MATLAB, in general finding solutions to systems of equations is not. However, *2 × 2* systems are fairly straightforward, and there are several methods of solution for these systems for which MATLAB has built-in functions.

Consider the following *2 × 2* system of equations:

$$x_1 + 2x_2 = 2$$
$$2x_1 + 2x_2 = 6$$

This system of equations in matrix form is:

$$\underset{\text{A}}{\begin{bmatrix} 1 & 2 \\ 2 & 2 \end{bmatrix}} \underset{\text{x}}{\begin{bmatrix} x_1 \\ x_2 \end{bmatrix}} = \underset{\text{b}}{\begin{bmatrix} 2 \\ 6 \end{bmatrix}}$$

We have already seen that the solution is `x = A^-1 b`, so we can solve this if we can find the inverse of A. One method of finding the inverse for a *2 × 2* matrix involves calculating the ***determinant*** D.

For a 2×2 matrix

$$A = \begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix}$$

the determinant D is defined as:

$$D = \begin{vmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{vmatrix} = a_{11}a_{22} - a_{12}a_{21}$$

It is written using vertical lines around the coefficients of the matrix and is defined as the product of the values on the diagonal minus the product of the other two numbers.

For a 2×2 matrix, the matrix inverse is defined in terms of D as

$$A^{-1} = \frac{1}{D}\begin{bmatrix} a_{22} & -a_{12} \\ -a_{21} & a_{11} \end{bmatrix}$$

The inverse is therefore the result of multiplying the scalar 1/D by every element in the matrix shown here. Note that this is not the matrix A, but is determined using the elements from A in the following manner: the values on the diagonal are reversed and the negation operator is used on the other two values.

Notice that if the determinant D is 0, it will not be possible to find the inverse of the matrix A.

For our coefficient matrix $A = \begin{bmatrix} 1 & 2 \\ 2 & 2 \end{bmatrix}$, $D = \begin{vmatrix} 1 & 2 \\ 2 & 2 \end{vmatrix} = 1*2 - 2*2$ or -2 so

$$A^{-1} = \frac{1}{1*2-2*2}\begin{bmatrix} 2 & -2 \\ -2 & 1 \end{bmatrix} = \frac{1}{-2}\begin{bmatrix} 2 & -2 \\ -2 & 1 \end{bmatrix} = \begin{bmatrix} -1 & 1 \\ 1 & -\frac{1}{2} \end{bmatrix}$$

and

$$\begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} -1 & 1 \\ 1 & -\frac{1}{2} \end{bmatrix}\begin{bmatrix} 2 \\ 6 \end{bmatrix}$$

The unknowns are found by performing this matrix multiplication. Consequently,

```
x1 = -1 * 2 + 1 * 6 = 4
x2 = 1 * 2 + (-1/2) * 6 = -1
```

To do this in MATLAB, we would first create the coefficient matrix variable *A* and column vector *b*.

```
>> A = [1 2; 2 2];
>> b = [2;6];
```

THE PROGRAMMING METHOD

For *2 × 2* matrices, the determinant and inverse are found using simple expressions.

```
>> deta = A(1,1)*A(2,2) - A(1,2)*A(2,1)
deta =
   -2

>> inva = (1/deta) * [A(2,2) -A(1,2); -A(2,1) A(1,1)]
inva =
   -1.0000    1.0000
    1.0000   -0.5000
```

THE EFFICIENT METHOD

We have already seen that MATLAB has a built-in function, **inv**, to find a matrix inverse. It also has a built-in function **det** to find a determinant:

```
>> det(A)
ans =
   -2

>> inv(A)
ans =
   -1.0000    1.0000
    1.0000   -0.5000
```

And then, the unknowns x are found:

```
>> x = inv(A) * b
x =
     4
    -1
```

PRACTICE 14.7

For the following *2 × 2* system of equations:

$$x_1 + 2x_2 = 4$$
$$-x_1 \quad = 3$$

Do the following on paper:

- Write the equations in matrix form Ax = b.
- Solve by finding the inverse A^{-1} and then $x = A^{-1} b$.

Next, get into MATLAB and check your answers.

14.5.2.2 Reduced Row Echelon Form

For 2×2 systems of equations, there are solving methods that are well-defined and simple. However, for larger systems of equations, finding solutions is frequently not as straightforward.

Several methods of solving are based on the observation that systems of equations are equivalent if they have the same solution set. Performing some simple operations on rows of the matrix form of a set of equations results in equivalent systems. The ***Gauss-Jordan method*** starts by augmenting the coefficient matrix A with the column vector b and performing operations until the square part of the matrix becomes diagonal.

Reduced Row Echelon Form takes this one step further to result in all 1s on the diagonal, or in other words, until the square part is the identity matrix. In this case, the column of b's is the solution. For example, for a 3×3 matrix.

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} & b_1 \\ a_{21} & a_{22} & a_{23} & b_2 \\ a_{31} & a_{32} & a_{33} & b_3 \end{bmatrix} \rightarrow \begin{bmatrix} 1 & 0 & 0 & b'_1 \\ 0 & 1 & 0 & b'_2 \\ 0 & 0 & 1 & b'_3 \end{bmatrix}$$

Note
The prime in b' indicates that the numbers may have changed, but the systems are equivalent.

In other words, we are reducing [A|b] to [I | b']. MATLAB has a built-in function to do this, called **rref**. For example:

```
>> a = [1 3 0; 2 1 3; 4 2 3];
>> b = [1 6 3]';
>> ab = [a b];
>> rref(ab)
ans =
     1     0     0    -2
     0     1     0     1
     0     0     1     3
```

The solution is found from the last column, so $x_1 = -2$, $x_2 = 1$, and $x_3 = 3$. To get this in a column vector in MATLAB:

```
>> x = ans(:,end)
x =
    -2
     1
     3
```

14.5.2.3 Finding a Matrix Inverse by Reducing an Augmented Matrix

For a system of equations larger than a 2×2 system, one method of finding the inverse of a matrix A mathematically involves augmenting the matrix with an identity matrix of the same size, and then reducing it. The algorithm is:

- Augment the matrix with I: [A | I].
- Reduce it to the form [I | X]; X will be A^{-1}.

For example, in MATLAB we can start with a matrix, augment it with an identity matrix, and then use the **rref** function to reduce it.

```
>> a = [1 3 0; 2 1 3; 4 2 3];
>> rref([a eye(size(a))])
ans =
   1.0000         0         0   -0.2000   -0.6000    0.6000
        0    1.0000         0    0.4000    0.2000   -0.2000
        0         0    1.0000         0    0.6667   -0.3333
```

In MATLAB, the **inv** function can be used to verify the result.

```
>> inv(a)
ans =
   -0.2000   -0.6000    0.6000
    0.4000    0.2000   -0.2000
         0    0.6667   -0.3333
```

14.6 SYMBOLIC MATHEMATICS

Symbolic mathematics means doing mathematics on symbols (not numbers!). For example, a + a is 2a. The symbolic math functions are in Symbolic Math Toolbox in MATLAB. Toolboxes contain related functions and are add-ons to MATLAB. (Therefore, this may or may not be part of your own system.) Symbolic Math Toolbox includes an alternative method for solving equations and is therefore covered in this chapter.

14.6.1 Symbolic Variables and Expressions

MATLAB has a type called **sym** for symbolic variables and expressions; these work with character vectors. For example, to create a symbolic variable *a* and perform the addition just described, a symbolic variable would first be created by passing the character vector 'a' to the **sym** function:

```
>> a = sym('a');
>> a+a
ans =
2*a
```

Symbolic variables can also store expressions. For example, the variables *b* and *c* store symbolic expressions:

```
>> b = a^2
b =
```

```
a^2
>> c = a^4;
>> class(b)
ans =
    'sym'
```

Note: it used to be possible to pass expressions to the **sym** function, e.g., sym('a^2'), but this is no longer supported.

All basic mathematical operations can be performed on symbolic variables and expressions (e.g., add, subtract, multiply, divide, raise to a power, etc.). The following are examples:

```
>> c/b
ans =
a^2
>> b^3
ans =
a^6
>> c*b
ans =
a^6
>> b + 4*a^2
ans =
5*a^2
```

It can be seen from the last example that MATLAB will collect like terms in these expressions, adding the a^2 and $4a^2$ to result in $5a^2$.

If using multiple variables as symbolic variable names is desired, the **syms** function is a shortcut instead of using **sym** repeatedly. For example,

```
>> syms x y z
```

is equivalent to

```
>> x = sym('x');
>> y = sym('y');
>> z = sym('z');
```

The built-in functions **sym2poly** and **poly2sym** convert from symbolic expressions to polynomial vectors and vice versa. For example:

```
>> myp = [1 2 -4 3];
>> poly2sym(myp)
ans =
x^3+2*x^2-4*x+3
>> mypoly = [2 0 -1 0 5];
>> poly2sym(mypoly)
```

```
ans =
2*x^4-x^2+5

>> sym2poly(ans)
ans =
    2    0    -1    0    5
```

14.6.2 Simplification Functions

There are several functions that work with symbolic expressions and simplify the terms. Not all expressions can be simplified, but the **simplify** function does whatever it can to simplify expressions, including gathering like terms. For example:

```
>> x = sym('x');
>> myexpr = cos(x)^2 + sin(x)^2
myexpr =
cos(x)^2+sin(x)^2

>> simplify(myexpr)
ans =
1
```

The functions **collect**, **expand**, and **factor** work with polynomial expressions. The **collect** function collects coefficients, such as the following:

```
>> x = sym('x');
>> collect(x^2 + 4*x^3 + 3*x^2)
ans =
4*x^3 + 4*x^2
```

The **expand** function will multiply out terms, and **factor** will do the reverse:

```
>> ea = expand((x+2)*(x-1))
ea =
x^2 + x - 2
>> f = factor(ea)
f =
[ x + 2, x - 1]
>> expand(f(1)*f(2))
ans =
x^2 + x – 2
```

If the argument is not factorable, the original input argument will be returned unmodified.

The **subs** function will substitute a value for a symbolic variable in an expression. For example,

```
>> myexp = x^3 + 3*x^2 - 2
myexp =
x^3+3*x^2-2
>> subs(myexp,3)
ans =
52
```

If there are multiple variables in the expression, one will be chosen by default for the substitution (in this case, *x*), or the variable for which the substitution is to be made can be specified:

```
>> syms a b x
>> varexp = a*x^2 + b*x;
>> subs(varexp,3)
ans =
9*a + 3*b
>> subs(varexp,'a',3)
ans =
3*x^2 + b*x
```

With symbolic math, MATLAB works by default with rational numbers, meaning that results are kept in fractional forms. For example, performing the addition 1/3 + 1/2 would normally result in a **double** value:

```
>> 1/3 + 1/2
ans =
    0.8333
```

However, by making the expression symbolic, the result is symbolic also. Any numeric function (e.g., **double)** could change that:

```
>> sym(1/3 + 1/2)
ans =
5/6
>> double(ans)
ans =
    0.8333
```

The **numden** function will return separately the numerator and denominator of a symbolic expression:

```
>> frac = sym(1/3 + 1/2)
frac =
5/6
>> [n, d] = numden(frac)
n =
5
d =
6
```

14.6.3 Displaying Expressions

The **pretty** function will display symbolic expressions using exponents. For example:

```
>> syms x b
>> b = x^2
b =
x^2
>> pretty(b)
  2
x
```

The function **fplot** will draw a two-dimensional plot in the default x-range from −5 to 5. The following code produces the figure that is shown in Figure 14.5. The DisplayName property of the plot is used for the title of the plot.

```
>> expr = x^3 + 3*x^2 - 2;
>> fpe = fplot(expr);
>> title(fpe.DisplayName)
```

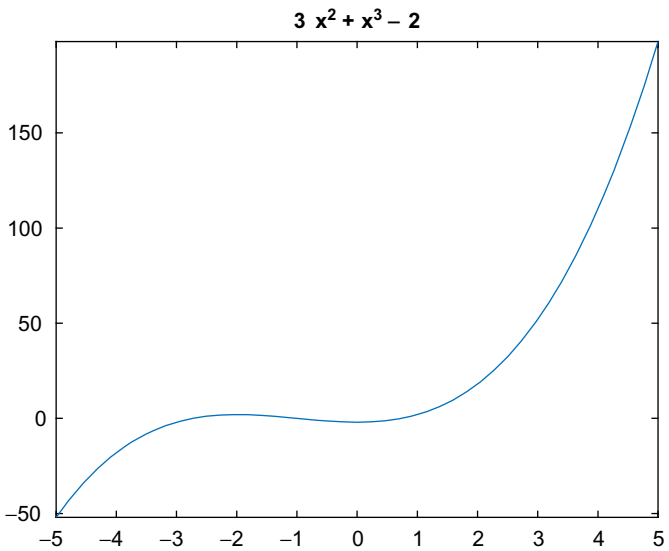


FIGURE 14.5
Plot produced using **fplot**.

14.6.4 Solving Equations

We've already seen several methods for solving simultaneous linear equations using a matrix representation. MATLAB can also solve sets of equations using symbolic math.

The function **solve** solves an equation and returns the solution(s) as symbolic expressions. The solution can be converted to numbers using any numeric function, such as **double**:

```
>> syms a b x
>> solve(2*x^2 + x == 6)
ans =
  -2
3/2
>> double(ans)
ans =
  -2.0000
   1.5000
```

If an expression is passed to the **solve** function rather than an equation, the **solve** function will set the expression equal to 0 and solve the resulting equation. For example, the following will solve $3x^2 + x = 0$:

```
>> solve(3*x^2 + x)
ans =
-1/3
   0
```

If there is more than one variable, MATLAB chooses which to solve for. In the following example, the equation $ax^2 + bx = 0$ is solved. There are three variables. As can be seen from the result, which is given in terms of *a* and *b*, the equation was solved for *x*. MATLAB has rules built-in that specify how to choose which variable to solve for. For example, *x* will always be the first choice if it is in the equation or expression.

```
>> solve(a*x^2 + b*x)
ans =
   0
-b/a
```

However, it is possible to specify which variable to solve for:

```
>> solve(a*x^2 + b*x, b)
ans =
-a*x
```

MATLAB can also solve sets of equations. In this example, the solutions for x, y, and z are returned as a structure consisting of fields for *x*, *y*, and *z*. The individual solutions are symbolic expressions stored in fields of the structure.

```
>> syms x y z
>> solve(4*x-2*y+z==7, x+y+5*z==10, -2*x+3*y-z==2)
ans =
  struct with fields:
    x: [1×1 sym]
    y: [1×1 sym]
    z: [1×1 sym]
```

To refer to the individual solutions, which are in the structure fields, the dot operator is used.

```
>> x = ans.x
x =
124/41
>> y = ans.y
y =
121/41
>> z = ans.z
z =
33/41
```

The **double** function can then be used to convert the symbolic expressions to numbers and store the results from the three unknowns in a vector.

```
>> double([x y z])
ans =
    3.0244      2.9512      0.8049
```

PRACTICE 14.8

For each of the following expressions, show what the MATLAB result would be. Assume that all expressions are typed SEQUENTIALLY.

```
x = sym('x');
a = x^3 - 2*x^2 + 1;
b = x^3 + x^2;
res = a+b
p = sym2poly(res)
polyval(p,2)
sym(1/2 + 1/4)
solve(x^2 - 16)
```

14.7 CALCULUS: INTEGRATION AND DIFFERENTIATION

MATLAB has functions that perform common calculus operations on a mathematical function f(x), such as ***integration*** and ***differentiation***.

14.7.1 Integration and the Trapezoidal Rule

The integral of a function f(x) between the limits given by x = a and x = b is written as

$$\int_a^b f(x)dx$$

and is defined as the area under the curve f(x) from a to b, as long as the function is above the x-axis. Numerical integration techniques involve approximating this.

One simple method of approximating the area under a curve is to draw a straight line from f(a) to f(b) and calculate the area of the resulting trapezoid as

$$(\mathrm{b}-\mathrm{a})\frac{f(a)+f(b)}{2}$$

In MATLAB, this could be implemented as a function.

An improvement on this is to divide the range from a to b into n intervals, apply the trapezoidal rule to each interval, and sum them. For example, for the preceding if there are two intervals, you would draw a straight line from f(a) to f((a+b)/2) and then from f((a+b)/2) to f(b).

THE PROGRAMMING CONCEPT

Here is a function to which the function handle and limits a and b are passed:

trapint.m

```
function int = trapint(fnh, a, b)
% trapint approximates area under a curve f(x)
%  from a to b using a trapezoid
% Format: trapint(handle of f, a, b)
int = (b-a) * (fnh(a) + fnh(b))/2;
end
```

To call it, for example, for the function $f(x) = 3x^2 - 1$, an anonymous function is defined and its handle is passed to the *trapint* function.

```
>> f = @ (x) 3 * x .^ 2 - 1;
approxint = trapint(f, 2, 4)
approxint =

    58
```

THE EFFICIENT METHOD

MATLAB has a built-in function **trapz** that will implement the trapezoidal rule. Vectors with the values of x and y = f(x) are passed to it. For example, using the anonymous function defined previously:

```
>> x = [2 4];
>> y = f(x);
>> trapz(x,y)
ans =
    58
```

THE PROGRAMMING CONCEPT

The following is a modification of the previous function to which the function handle, limits, and the number of intervals are passed:

trapintn.m

```
function intsum = trapintn(fnh, lowrange,highrange, n)
% trapintn approximates area under a curve f(x) from
% a to b using trapezoids with n intervals
% Format: trapintn(handle of f, a, b, n)
intsum = 0;
increm = (highrange - lowrange)/n;
for a = lowrange: increm : highrange - increm
    b = a + increm;
    intsum = intsum + (b-a) * (fnh(a) + fnh(b))/2;
end
end
```

For example, this approximates the integral of the previous function f with two intervals:

```
>> trapintn(f,2,4,2)
ans =
    55
```

In these examples, straight lines, which are first-order polynomials, were used. Other methods involve higher-order polynomials. The built-in function **quad** uses Simpson's method. Three arguments are normally passed to it: the handle of the function, and the limits a and b. For example, for the previous function:

```
>> quad(f,2,4)
ans =
    54
```

THE EFFICIENT METHOD

To use the built-in function **trapz** to accomplish the same thing, the *x* vector is created with the values 2, 3, and 4:

```
>> x = 2:4;
>> y = f(x)
>> trapz(x,y)
ans =
     55
```

MATLAB has a function **polyint**, which will find the integral of a polynomial. For example, for the polynomial $3x^2 + 4x - 4$, which would be represented by the vector `[3 4 -4]`, the integral is found by:

```
>> origp = [3 4 -4];
>> intp = polyint(origp)
intp =
     1     2    -4     0
```

which shows that the integral is the polynomial $x^3 + 2x^2 - 4x$.

14.7.2 Differentiation

The derivative of a function y = f(x) is written as $\frac{dy}{dx}f(x)$ or `f'(x)` and is defined as the rate of change of the dependent variable y with respect to x. The derivative is the slope of the line tangent to the function at a given point.

MATLAB has a function **polyder**, which will find the derivative of a polynomial. For example, for the polynomial $x^3 + 2x^2 - 4x + 3$, which would be represented by the vector `[1 2 −4 3]`, the derivative is found by:

```
>> origp = [1 2 -4 3];
>> diffp = polyder(origp)
diffp =
     3     4    -4
```

which shows that the derivative is the polynomial $3x^2 + 4x - 4$. The function **polyval** can then be used to find the derivative for certain values of x, such as for x = 1, 2, and 3:

```
>> polyval(diffp, 1:3)
ans =
     3    16    35
```

The derivative can be written as the limit

$$f'(x) = \lim_{h \to 0} \frac{f(x+h) - f(x)}{h}$$

and can be approximated by a difference equation.

Recall that MATLAB has a built-in function, **diff**, which returns the differences between consecutive elements in a vector. For a function `y = f(x)` where x is a vector, the values of `f'(x)` can be approximated as **diff(y)** divided by **diff(x)**. For example, the equation $x^3 + 2x^2 - 4x + 3$ can be written as an anonymous function. It can be seen that the approximate derivative is close to the values found using **polyder** and **polyval**.

```
>> f = @(x) x.^3 + 2*x.^2 - 4*x + 3;
>> x = 0.5 : 3.5
x =
    0.5000    1.5000    2.5000    3.5000
>> y = f(x)
y =
    1.6250    4.8750   21.1250   56.3750
>> diff(y)
ans =
    3.2500   16.2500   35.2500
>> diff(x)
ans =
     1     1     1
>> diff(y) ./ diff(x)
ans =
    3.2500   16.2500   35.2500
```

14.7.3 Calculus in Symbolic Math Toolbox

There are several functions in Symbolic Math Toolbox™ to perform calculus operations symbolically (e.g., **diff** to differentiate and **int** to integrate). To learn about the **int** function, for example, from the Command Window:

```
>> help sym/int
```

For instance, to find the indefinite integral of the function $f(x) = 3x^2 - 1$:

```
>> syms x
>> int(3*x^2 - 1)
ans =
x^3 - x
```

To instead find the definite integral of this function from $x = 2$ to $x = 4$:

```
>> int(3*x^2 - 1, 2, 4)
ans =
54
```

Limits can be found using the **limit** function. For example, for the difference equation described previously:

```
>> syms x h
>> f = @ (x) x .^ 3 + 2 * x .^ 2 - 4 * x + 3;
>> limit((f(x+h)-f(x))/h,h,0)
ans =
3*x^2 + 4*x - 4
```

To differentiate, instead of the anonymous function we write it symbolically:

```
>> syms x f
>> f = x^3 + 2*x^2 - 4*x + 3
f =
x^3 + 2*x^2 - 4*x + 3

>> diff(f)
ans =
3*x^2 + 4*x - 4
```

PRACTICE 14.9

For the function $3x^2 - 4x + 2$:

- Find the indefinite integral of the function.
- Find the definite integral of the function from x = 2 to x = 5.
- Approximate the area under the curve from x = 2 to x = 5.
- Find its derivative.
- Approximate the derivative for x = 2.

Explore Other Interesting Features

Investigate the **corrcoef** function, which returns correlation coefficients.

Investigate filtering data, for example using the **filter** function.

Investigate the moving statistical functions introduced in R2016a **movmean**, **movsum**, etc.

Investigate the flag introduced in R2015a to omit NaN in functions such as **max**, **min median**, etc.

Investigate the index vectors returned by the set functions.

Investigate the use of 'R2012a' to see future changes to the set functions versus the use of 'legacy' to preserve the previous values.

Investigate passing matrices to the set functions, using the 'rows' specifier.

Investigate the **interp1** function, which does a table look-up to interpolate or extrapolate.

Investigate the **fminsearch** function, which finds local minima for a function.

Investigate the **fzero** function, which attempts to find a zero of a function near a specified x value.

Investigate linear algebra functions, such as **rank** for the rank of a matrix, or **null**, which returns the null space of a matrix.

Investigate the **blkdiag** function, which will create a block diagonal matrix.

Investigate the functions that return eigenvalues and eigenvectors, such as **eig** and **eigs**.

Investigate the **norm** function to find a vector or matrix norm.

Investigate the Ordinary Differential Equation (ODE) solve functions, such as **ode23** and **ode45**, which use the Runge-Kutta integration methods.

In the Command Window, type "odeexamples" to see some ODE example codes.

Investigate some of the other numerical integration functions, such as **integral**, **integral2** for double integrals, and **integral3** for triple integrals.

Investigate the **poly** function, which finds the characteristic equation for a matrix, and the **polyeig** function, which solves a polynomial eigenvalue problem of a specified degree. ■

SUMMARY

COMMON PITFALLS

- Forgetting that **max** and **min** return the index of only the first occurrence of the maximum or minimum value.
- Not realizing that a data set has outliers that can drastically alter the results obtained from the statistical functions.
- Extrapolating too far away from the data set
- Forgetting that the **fprintf** function by default only prints the real part of a complex number
- Forgetting that, to augment one matrix with another, the number of rows must be the same in each

PROGRAMMING STYLE GUIDELINES

- Remove the largest and smallest numbers from a large data set before performing statistical analyses, in order to handle the problem of outliers.

MATLAB Functions and Commands			
mink	polyval	issymmetric	subs
maxk	polyfit	istril	numden
mean	complex	istriu	pretty
var	real	isbanded	ezplot
std	imag	inv	solve
mode	isreal	det	trapz
median	conj	rref	quad
union	angle	sym	polyint
intersect	polarplot	syms	polyder
unique	diag	sym2poly	int
setdiff	trace	poly2sym	limit
setxor	eye	simplify	
ismember	triu	collect	
issorted	tril	expand	
roots	isdiag	factor	

- The better the curve fit, the more accurate interpolated and extrapolated values will be.
- When working with symbolic expressions, it is generally easier to make all variables symbolic variables to begin with.

Exercises

1. In a marble manufacturing plant, a quality control engineer randomly selects eight marbles from each of the two production lines and measures the diameter of each marble in millimeters. For the each data set here, determine the mean, median, mode, and standard deviation using built-in functions.

```
Prod. line A:15.94  15.98  15.94  16.16  15.86  15.86  15.90  15.88
Prod. line B:15.96  15.94  16.02  16.10  15.92  16.00  15.96  16.02
```

   Suppose the desired diameter of the marbles is 16 mm. Based on the results you have, which production line is better in terms of meeting the specification? (Hint: think in terms of the mean and the standard deviation.)

2. Write a function *mymin* that will receive any number of arguments and will return the minimum. Note: the function is not receiving a vector; rather, all of the values are separate arguments.
3. Write a script that will do the following. Create two vectors with 20 random integers in each; in one, the integers should range from 1 to 5, and in the other, from 1 to 500 (inclusive). For each vector, would you expect the mean and median to be approximately the same? Would you expect the standard deviation of the two vectors to be approximately the same? Answer these questions, and then

use the built-in functions to find the minimum, maximum, mean, median, standard deviation, and mode of each. Do a histogram for each in a subplot. Run the script a few times to see the variations.

4. Write a function that will return the mean of the values in a vector, not including the minimum and maximum values. Assume that the values in the vector are unique. It is okay to use the built-in **mean** function. To test this, create a vector of 10 random integers, each in the range from 0 to 50, and pass this vector to the function.
5. Create a vector that has 10 elements. Use the **mink** function to determine the two smallest, and use a set function to create a new vector that does not include those two values.
6. Investigate the use of the **bounds** function to find the smallest and largest values in a vector.
7. A moving average of a data set $x = \{x_1, x_2, x_3, x_4, \ldots, x_n\}$ is defined as a set of averages of subsets of the original data set. For example, a moving average of every two terms would be $1/2 * \{x_1 + x_2, x_2 + x_3, x_3 + x_4, \ldots, x_{n-1} + x_n\}$. Write a function that will receive a vector as an input argument and will calculate and return the moving average of every two elements.
8. Redo problem 7, using the function **movmean** which was introduced in R2016a.
9. A median filter on a vector has a size, for example, a size of 3 means calculating the median of every three values in the vector. The first and last elements are left alone. Starting from the second element to the next-to-last element, every element of a vector vec(i) is replaced by the median of [vec(i − 1) vec(i) vec(i + 1)]. For example, if the signal vector is

   ```
   signal = [5 11 4 2 6 8 5 9]
   ```

   the median filter with a size of 3 is

   ```
   medianFilter3 = [5 5 4 4 6 6 8 9]
   ```

   Write a function to receive the original signal vector and return the median filtered vector.

10. What is the difference between the mean and the median of a data set if there are only two values in it?
11. A student missed one of four exams in a course and the professor decided to use the "average" of the other three grades for the missed exam grade. Which would be better for the student: the mean or the median if the three recorded grades were 99, 88, and 95? What if the grades were 99, 70, and 77?
12. The set functions can be used with cell arrays of character vectors. Create two cell arrays to store course numbers taken by two students. For example,

    ```
    s1 = {'EC 101', 'CH 100', 'MA 115'};
    s2 = {'CH 100', 'MA 112', 'BI 101'};
    ```

    Use a set function to determine which courses the students have in common.

13. Redo problem 12 using a string array.

14. A vector *v* is supposed to store unique random numbers. Use set functions to determine whether or not this is true.
15. A program has a vector of structures that stores information on experimental data that has been collected. For each experiment, up to 10 data values were obtained. Each structure stores the number of data values for that experiment, and then the data values. The program is to calculate and print the average value for each experiment. Write a script to create some data in this format and print the averages.
16. Express the following polynomials as row vectors of coefficients:

    $3x^3 - 4x^2 + 2x + 3$

    $2x^4 + x^2 + 4x - 1$

17. Find the roots of the equation f(x) = 0 for the following function. Also, create *x* and *y* vectors and plot this function in the range from −3 to 3 to visualize the solution.

    $f(x) = 3x^2 - 2x - 4$

18. Evaluate the polynomial expression $3x^3 + 4x^2 + 2x - 2$ at x = 3, x = 5, and x = 7.
19. What is a danger of extrapolation?
20. Create a vector y that stores 20 random integers, each in the range from −2 to +2, and an x vector which is 1:20. Fit a straight line through these points. Plot the data points and the straight line on the same graph with a legend.
21. Write a function *fitwcolors* that will receive 4 input arguments: x and y data vectors, a number *n*, and a colormap. The function should fit curves with polynomials of orders 1 through n, and plot each (all on the same graph). The curve with degree 1 should be plotted with the first color from the colormap (using the 'Color' property), the curve with degree 2 should be plotted with the second color from the colormap, etc. You may assume that *n* is an integer less than the length of the x and y vectors and that the colormap has at least *n* colors in it.
22. Write a function that will receive data points in the form of x and y vectors. If the lengths of the vectors are not the same, then they can't represent data points, so an error message should be printed. Otherwise, the function will fit a polynomial of a random degree through the points and will plot the points and the resulting curve with a title specifying the degree of the polynomial. The degree of the polynomial must be less than the number of data points, n, so the function must generate a random integer in the range from 1 to n − 1 for the polynomial degree.
23. Write a function *fit1or2* that will receive two input arguments, data vectors x and y. It may be assumed that the vectors are of the same length and each has at least 6 numbers. The function will fit a polynomial curve of a random order in the range from 2 to 4 to the data points and return the resulting polynomial vector. If the function is expected to return a second argument, it will fit a polynomial curve with order one less than the original and return that, also.

24. The voltage in a circuit is determined at various times, as follows:

Time	1	2	3	4	5	6	7
Voltage	1.1	1.9	3.3	3.4	3.1	3.3	7.1

Fit a straight line through the data points, and then plot this line along with the sample voltages. According to your straight line, determine at what time the voltage would be 5.

25. Write a function that will receive x and y vectors representing data points. The function will create, in one Figure Window, a plot showing these data points as circles and also in the top part a second-order polynomial that best fits these points and on the bottom a third-order polynomial. The top plot will have a line width of 3 and will be a gray color. The bottom plot will be blue and have a line width of 2.
26. Store the following complex numbers in variables and print them in the form a + bi.

$3 - 2i$

$\sqrt{-3}$

27. Create the following complex variables

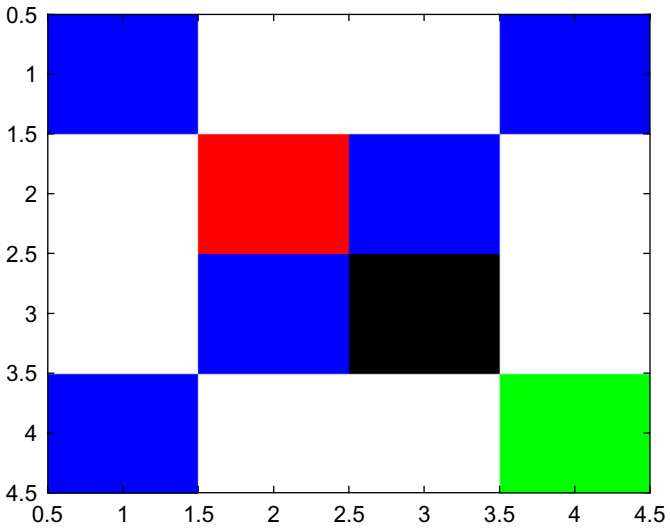
```
c1 = 1 - 3i;
c2 = 5 + 2i;
```

Perform the following operations on them:

- add them
- multiply them
- get the complex conjugate and magnitude of each
- put them in polar form

28. Represent the expression $z^3 - 2z^2 + 3 - 5i$ as a row vector of coefficients and store this in a variable *compoly*. Use the **roots** function to solve $z^3 - 2z^2 + 3 - 5i = 0$. Also, find the value of *compoly* when $z = 2$ using **polyval**.
29. What is the value of the trace of an $n \times n$ identity matrix?
30. For an $m \times n$ matrix, what are the dimensions of its transpose?
31. What is the transpose of a diagonal matrix A?
32. When is a square matrix both an upper triangular and lower triangular matrix?
33. Is the transpose of an upper triangular matrix also upper triangular? If not, how would you characterize it?

34. Write a script that will generate the following image, using the colormap method.



35. The function **magic** returns an n × n matrix for which the sum of all rows, columns, and the diagonal are equal to each other. Investigate **magic(5)** and verify the sums.
36. Write a function *myeye* to return an n × n identity matrix (without using **eye**).
37. Write a function *myupp* that will receive an integer argument *n* and will return an *n* × *n* upper triangular matrix of random integers.
38. Analyzing electric circuits can be accomplished by solving sets of equations. For a particular circuit, the voltages V_1, V_2, and V_3 are found through the system:

$$V_1 = 5$$
$$-6V_1 + 10V_2 - 3V_3 = 0$$
$$-V_2 + 51V_3 = 0$$

Put these equations in matrix form and solve in MATLAB.

39. Rewrite the following system of equations in matrix form:

$$4x_1 - x_2 + 3x_4 = 10$$
$$-2x_1 + 3x_2 + x_3 - 5x_4 = -3$$
$$x_1 + x_2 - x_3 + 2x_4 = 2$$
$$3x_1 + 2x_2 - 4x_3 = 4$$

Set it up in MATLAB and use any method to solve.

40. Solve the simultaneous equations $x - y = 2$ and $x^2 + y = 0$ using **solve**. Plot the corresponding functions, $y = x - 2$ and $y = -x^2$, on the same graph with an x-range from −5 to 5.

41. For the following set of equations,

$$2x_1 + 2x_2 + x_3 = 2$$
$$x_2 + 2x_3 = 1$$
$$x_1 + x_2 + 3x_3 = 3$$

write it in symbolic form and solve using the **solve** function. From the symbolic solution, create a vector of the numerical (**double**) equivalents.

42. The reproduction of cells in a bacterial colony is important for many environmental engineering applications such as wastewater treatments. The formula

$$\log(N) = \log(N_0) + t/T \log(2)$$

can be used to simulate this, where N_0 is the original population, N is the population at time t, and T is the time it takes for the population to double. Use the **solve** function to determine the following: if $N_0 = 10^2$, $N = 10^8$, and $t = 8$ hours, what will be the doubling time T? Use **double** to get your result in hours.

43. Using the symbolic function **int**, find the indefinite integral of the function $4x^2 + 3$, and the definite integral of this function from $x = -1$ to $x = 3$. Also, approximate this using the **trapz** function.

44. Use the **quad** function to approximate the area under the curve $4x^2 + 3$ from -1 to 3. First, create an anonymous function and pass its handle to the **quad** function.

45. Use the **polyder** function to find the derivative of $2x^3 - x^2 + 4x - 5$.

46. Examine the motion, or *trajectory*, of a *projectile* moving through the air. Assume that it has an initial height of 0, and neglect the air resistance for simplicity. The projectile has an initial velocity v_0, an angle of departure θ_0, and is subject to the gravity constant $g = 9.81$ m/s^2. The position of the projectile is given by x and y coordinates, where the origin is the initial position of the projectile at time $t = 0$. The total horizontal distance that the projectile travels is called its *range* (the point at which it hits the ground) and the highest peak (or vertical distance) is called its *apex*. Equations for the trajectory can be given in terms of the time t or in terms of x and y. The position of the projectile at any time t is given by:

$$x = v_0 \cos(\theta_0)\, t$$

$$y = v_0 \sin(\theta_0)\, t - \tfrac{1}{2} g\, t^2$$

For a given initial velocity v_0, and angle of departure θ_0, describe the motion of the projectile by writing a script to answer the following:

- What is the range?
- Plot the position of the projectile at suitable x values
- Plot the height versus time.
- How long does it take to reach its apex?

47. Write a GUI function that creates four random points. Radio buttons are used to choose the order of a polynomial to fit through the points. The points are plotted along with the chosen curve.