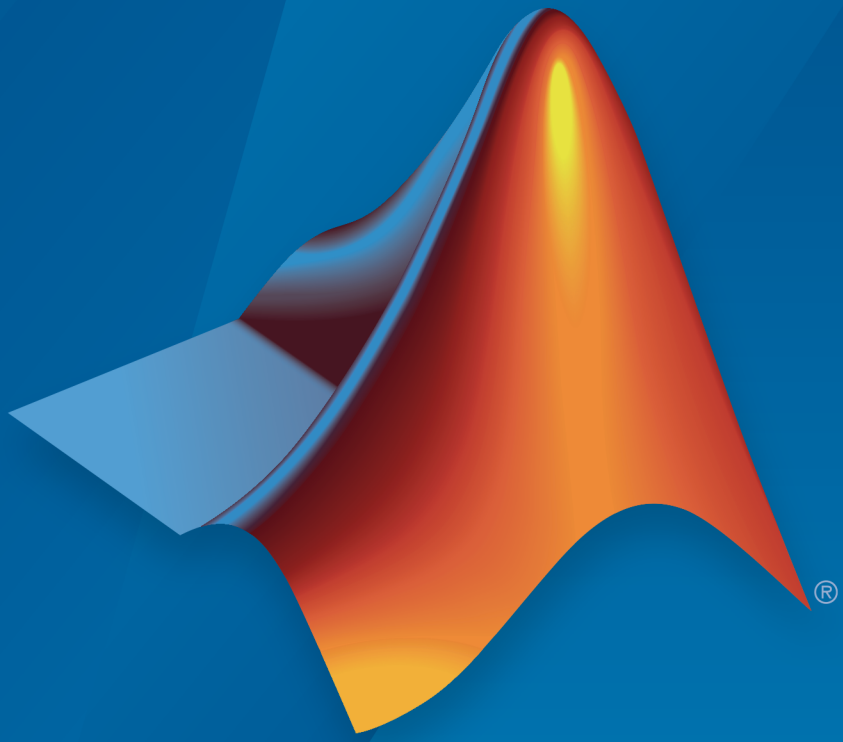
MATLAB®

Primer



MATLAB®

R2016b

MathWorks®

How to Contact MathWorks

Latest news: www.mathworks.com

Sales and services: www.mathworks.com/sales_and_services

User community: www.mathworks.com/matlabcentral

Technical support: www.mathworks.com/support/contact_us

Phone: 508-647-7000

The MathWorks, Inc.
3 Apple Hill Drive
Natick, MA 01760-2098

MATLAB® Primer

Revision History

December 1996	First printing	For MATLAB 5
May 1997	Second printing	For MATLAB 5.1
September 1998	Third printing	For MATLAB 5.3
September 2000	Fourth printing	Revised for MATLAB 6 (Release 12)
June 2001	Online only	Revised for MATLAB 6.1 (Release 12.1)
July 2002	Online only	Revised for MATLAB 6.5 (Release 13)
August 2002	Fifth printing	Revised for MATLAB 6.5
June 2004	Sixth printing	Revised for MATLAB 7.0 (Release 14)
October 2004	Online only	Revised for MATLAB 7.0.1 (Release 14SP1)
March 2005	Online only	Revised for MATLAB 7.0.4 (Release 14SP2)
June 2005	Seventh printing	Minor revision for MATLAB 7.0.4 (Release 14SP2)
September 2005	Online only	Minor revision for MATLAB 7.1 (Release 14SP3)
March 2006	Online only	Minor revision for MATLAB 7.2 (Release 2006a)
September 2006	Eighth printing	Minor revision for MATLAB 7.3 (Release 2006b)
March 2007	Ninth printing	Minor revision for MATLAB 7.4 (Release 2007a)
September 2007	Tenth printing	Minor revision for MATLAB 7.5 (Release 2007b)
March 2008	Eleventh printing	Minor revision for MATLAB 7.6 (Release 2008a)
October 2008	Twelfth printing	Minor revision for MATLAB 7.7 (Release 2008b)
March 2009	Thirteenth printing	Minor revision for MATLAB 7.8 (Release 2009a)
September 2009	Fourteenth printing	Minor revision for MATLAB 7.9 (Release 2009b)
March 2010	Fifteenth printing	Minor revision for MATLAB 7.10 (Release 2010a)
September 2010	Sixteenth printing	Revised for MATLAB 7.11 (R2010b)
April 2011	Online only	Revised for MATLAB 7.12 (R2011a)
September 2011	Seventeenth printing	Revised for MATLAB 7.13 (R2011b)
March 2012	Eighteenth printing	Revised for Version 7.14 (R2012a) (Renamed from *MATLAB Getting Started Guide*)
September 2012	Nineteenth printing	Revised for Version 8.0 (R2012b)
March 2013	Twentieth printing	Revised for Version 8.1 (R2013a)
September 2013	Twenty-first printing	Revised for Version 8.2 (R2013b)
March 2014	Twenty-second printing	Revised for Version 8.3 (R2014a)
October 2014	Twenty-third printing	Revised for Version 8.4 (R2014b)
March 2015	Twenty-fourth printing	Revised for Version 8.5 (R2015a)
September 2015	Twenty-fifth printing	Revised for Version 8.6 (R2015b)
March 2016	Twenty-sixth printing	Revised for Version 9.0 (R2016a)
September 2016	Twenty-seventh printing	Revised for Version 9.1 (R2016b)

Contents

1 Quick Start

2 Language Fundamentals

3 Mathematics

4 Graphics

5 Programming

Quick Start

MATLAB Product Description

The Language of Technical Computing

Millions of engineers and scientists worldwide use MATLAB® to analyze and design the systems and products transforming our world. MATLAB is in automobile active safety systems, interplanetary spacecraft, health monitoring devices, smart power grids, and LTE cellular networks. It is used for machine learning, signal processing, image processing, computer vision, communications, computational finance, control design, robotics, and much more.

Math. Graphics. Programming.

The MATLAB platform is optimized for solving engineering and scientific problems. The matrix-based MATLAB language is the world's most natural way to express computational mathematics. Built-in graphics make it easy to visualize and gain insights from data. A vast library of pre-built toolboxes lets you get started right away with algorithms essential to your domain. The desktop environment invites experimentation, exploration, and discovery. These MATLAB tools and capabilities are all rigorously tested and designed to work together.

Scale. Integrate. Deploy.

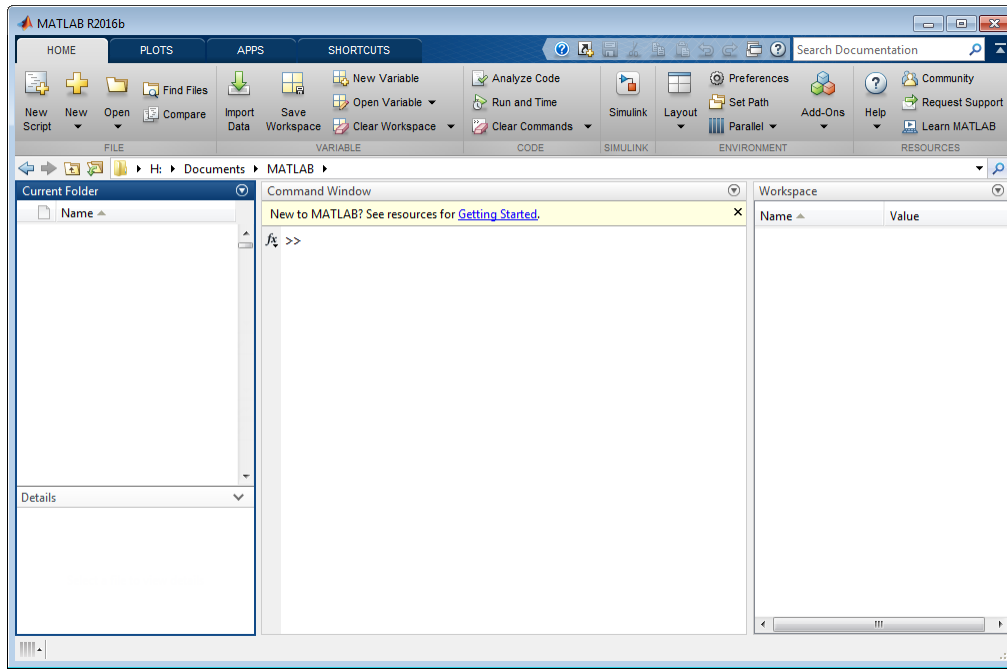
MATLAB helps you take your ideas beyond the desktop. You can run your analyses on larger data sets, and scale up to clusters and clouds. MATLAB code can be integrated with other languages, enabling you to deploy algorithms and applications within web, enterprise, and production systems.

Key Features

- High-level language for scientific and engineering computing
- Desktop environment tuned for iterative exploration, design, and problem-solving
- Graphics for visualizing data and tools for creating custom plots
- Apps for curve fitting, data classification, signal analysis, control system tuning, and many other tasks
- Add-on toolboxes for a wide range of engineering and scientific applications
- Tools for building applications with custom user interfaces
- Interfaces to C/C++, Java®, .NET, Python, SQL, Hadoop, and Microsoft® Excel®
- Royalty-free deployment options for sharing MATLAB programs with end users

Desktop Basics

When you start MATLAB, the desktop appears in its default layout.



The desktop includes these panels:

- **Current Folder** — Access your files.
- **Command Window** — Enter commands at the command line, indicated by the prompt (>>).
- **Workspace** — Explore data that you create or import from files.

As you work in MATLAB, you issue commands that create variables and call functions. For example, create a variable named `a` by typing this statement at the command line:

```
a = 1
```

MATLAB adds variable `a` to the workspace and displays the result in the Command Window.

```
a =

     1
```

Create a few more variables.

```
b = 2

b =

     2

c = a + b

c =

     3

d = cos(a)

d =

    0.5403
```

When you do not specify an output variable, MATLAB uses the variable `ans`, short for *answer*, to store the results of your calculation.

```
sin(a)

ans =

    0.8415
```

If you end a statement with a semicolon, MATLAB performs the computation, but suppresses the display of output in the Command Window.

```
e = a*b;
```

You can recall previous commands by pressing the up- and down-arrow keys, ↑ and ↓. Press the arrow keys either at an empty command line or after you type the first few characters of a command. For example, to recall the command `b = 2`, type `b`, and then press the up-arrow key.

See Also

"Matrices and Arrays" on page 1-5

Matrices and Arrays

MATLAB is an abbreviation for "matrix laboratory." While other programming languages mostly work with numbers one at a time, MATLAB® is designed to operate primarily on whole matrices and arrays.

All MATLAB variables are multidimensional *arrays*, no matter what type of data. A *matrix* is a two-dimensional array often used for linear algebra.

Array Creation

To create an array with four elements in a single row, separate the elements with either a comma (`,`) or a space.

```
a = [1 2 3 4]
```

```
a =

     1     2     3     4
```

This type of array is a *row vector*.

To create a matrix that has multiple rows, separate the rows with semicolons.

```
a = [1 2 3; 4 5 6; 7 8 10]
```

```
a =

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

Another way to create a matrix is to use a function, such as `ones`, `zeros`, or `rand`. For example, create a 5-by-1 column vector of zeros.

```
z = zeros(5,1)
```

```
z =

     0
```

```
     0
     0
     0
     0
```

Matrix and Array Operations

MATLAB allows you to process all of the values in a matrix using a single arithmetic operator or function.

```
a + 10
```

```
ans =

    11    12    13
    14    15    16
    17    18    20
```

```
sin(a)
```

```
ans =

    0.8415    0.9093    0.1411
   -0.7568   -0.9589   -0.2794
    0.6570    0.9894   -0.5440
```

To transpose a matrix, use a single quote (`'`):

```
a'
```

```
ans =

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

You can perform standard matrix multiplication, which computes the inner products between rows and columns, using the `*` operator. For example, confirm that a matrix times its inverse returns the identity matrix:

```
p = a*inv(a)


p =

    1.0000         0   -0.0000
         0    1.0000         0
         0         0    1.0000
```

Notice that `p` is not a matrix of integer values. MATLAB stores numbers as floating-point values, and arithmetic operations are sensitive to small differences between the actual value and its floating-point representation. You can display more decimal digits using the `format` command:

```
format long
p = a*inv(a)


p =

   1.000000000000000                   0  -0.000000000000000
                   0   1.000000000000000                   0
                   0                   0   0.999999999999998
```

Reset the display to the shorter format using

```
format short
```

`format` affects only the display of numbers, not the way MATLAB computes or saves them.

To perform element-wise multiplication rather than matrix multiplication, use the `.*` operator:

```
p = a.*a


p =

     1     4     9
    16    25    36
    49    64   100
```

The matrix operators for multiplication, division, and power each have a corresponding array operator that operates element-wise. For example, raise each element of `a` to the third power:

```
a.^3
```

```
ans =

           1           8          27
          64         125         216
         343         512        1000
```

Concatenation

Concatenation is the process of joining arrays to make larger ones. In fact, you made your first array by concatenating its individual elements. The pair of square brackets `[]` is the concatenation operator.

```
A = [a,a]
```

```
A =

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

Concatenating arrays next to one another using commas is called *horizontal* concatenation. Each array must have the same number of rows. Similarly, when the arrays have the same number of columns, you can concatenate *vertically* using semicolons.

```
A = [a; a]
```

```
A =

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

```
     7     8    10
```

Complex Numbers

Complex numbers have both real and imaginary parts, where the imaginary unit is the square root of `-1`.

```
sqrt(-1)
```

```
ans =

   0.0000 + 1.0000i
```

To represent the imaginary part of complex numbers, use either `i` or `j` .

```
c = [3+4i, 4+3j; -i, 10j]
```

```
c =

   3.0000 + 4.0000i   4.0000 + 3.0000i
   0.0000 - 1.0000i   0.0000 +10.0000i
```

See Also

"Array Indexing" on page 1-10

Array Indexing

Every variable in MATLAB® is an array that can hold many numbers. When you want to access selected elements of an array, use indexing.

For example, consider the 4-by-4 magic square A:

```
A = magic(4)
```

```
A =

    16     2     3    13
     5    11    10     8
     9     7     6    12
     4    14    15     1
```

There are two ways to refer to a particular element in an array. The most common way is to specify row and column subscripts, such as

```
A(4,2)
```

```
ans =

    14
```

Less common, but sometimes useful, is to use a single subscript that traverses down each column in order:

```
A(8)
```

```
ans =

    14
```

Using a single subscript to refer to a particular element in an array is called *linear indexing*.

If you try to refer to elements outside an array on the right side of an assignment statement, MATLAB throws an error.

```
test = A(4,5)
```

```
Index exceeds matrix dimensions.
```

However, on the left side of an assignment statement, you can specify elements outside the current dimensions. The size of the array increases to accommodate the newcomers.

```
A(4,5) = 17
```

```
A =

    16     2     3    13     0
     5    11    10     8     0
     9     7     6    12     0
     4    14    15     1    17
```

To refer to multiple elements of an array, use the colon operator, which allows you to specify a range of the form `start:end`. For example, list the elements in the first three rows and the second column of `A`:

```
A(1:3,2)
```

```
ans =

     2
    11
     7
```

The colon alone, without start or end values, specifies all of the elements in that dimension. For example, select all the columns in the third row of `A`:

```
A(3,:)
```

```
ans =

     9     7     6    12     0
```

The colon operator also allows you to create an equally spaced vector of values using the more general form `start:step:end`.

```
B = 0:10:100

B =

     0    10    20    30    40    50    60    70    80    90   100
```

If you omit the middle step, as in `start:end`, MATLAB uses the default step value of `1`.

See Also

"Workspace Variables" on page 1-13

Workspace Variables

The *workspace* contains variables that you create within or import into MATLAB from data files or other programs. For example, these statements create variables `A` and `B` in the workspace.

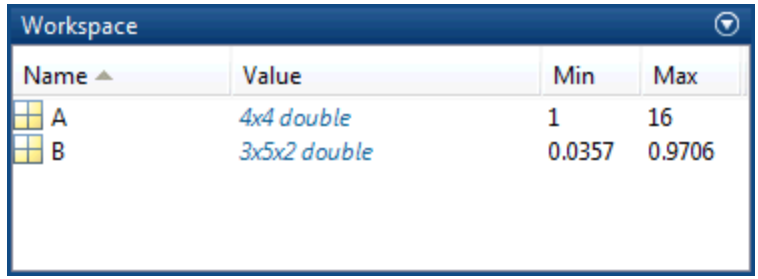
```
A = magic(4);
B = rand(3,5,2);
```

You can view the contents of the workspace using `whos`.

```
whos
```

```
  Name      Size              Bytes  Class     Attributes

  A         4x4                 128  double
  B         3x5x2               240  double
```

The variables also appear in the Workspace pane on the desktop.



Workspace variables do not persist after you exit MATLAB. Save your data for later use with the `save` command,

```
save myfile.mat
```

Saving preserves the workspace in your current working folder in a compressed file with a `.mat` extension, called a MAT-file.

To clear all the variables from the workspace, use the `clear` command.

Restore data from a MAT-file into the workspace using `load`.

```
load myfile.mat
```

See Also

"Text and Characters" on page 1-15

Text and Characters

When you are working with text, enclose sequences of characters in single quotes. You can assign text to a variable.

```
myText = 'Hello, world';
```

If the text includes a single quote, use two single quotes within the definition.

```
otherText = 'You''re right'
```

```
otherText =

You're right
```

`myText` and `otherText` are arrays, like all MATLAB® variables. Their *class* or data type is `char`, which is short for *character*.

```
whos myText
```

```
  Name        Size            Bytes  Class    Attributes

  myText      1x12               24  char
```

You can concatenate character arrays with square brackets, just as you concatenate numeric arrays.

```
longText = [myText,' - ',otherText]
```

```
longText =

Hello, world - You're right
```

To convert numeric values to characters, use functions, such as `num2str` or `int2str`.

```
f = 71;
c = (f-32)/1.8;
tempText = ['Temperature is ',num2str(c),'C']
```

```
tempText =

Temperature is 21.6667C
```

See Also

"Calling Functions" on page 1-17

Calling Functions

MATLAB® provides a large number of functions that perform computational tasks. Functions are equivalent to *subroutines* or *methods* in other programming languages.

To call a function, such as `max`, enclose its input arguments in parentheses:

```
A = [1 3 5];
max(A)
```

```
ans =

     5
```

If there are multiple input arguments, separate them with commas:

```
B = [10 6 4];
max(A,B)
```

```
ans =

    10     6     5
```

Return output from a function by assigning it to a variable:

```
maxA = max(A)
```

```
maxA =

     5
```

When there are multiple output arguments, enclose them in square brackets:

```
[maxA,location] = max(A)
```

```
maxA =

     5
```

```
location =

     3
```

Enclose any character inputs in single quotes:

```
disp('hello world')
```

```
hello world
```

To call a function that does not require any inputs and does not return any outputs, type only the function name:

```
clc
```

The `clc` function clears the Command Window.

See Also

"2-D and 3-D Plots" on page 1-19

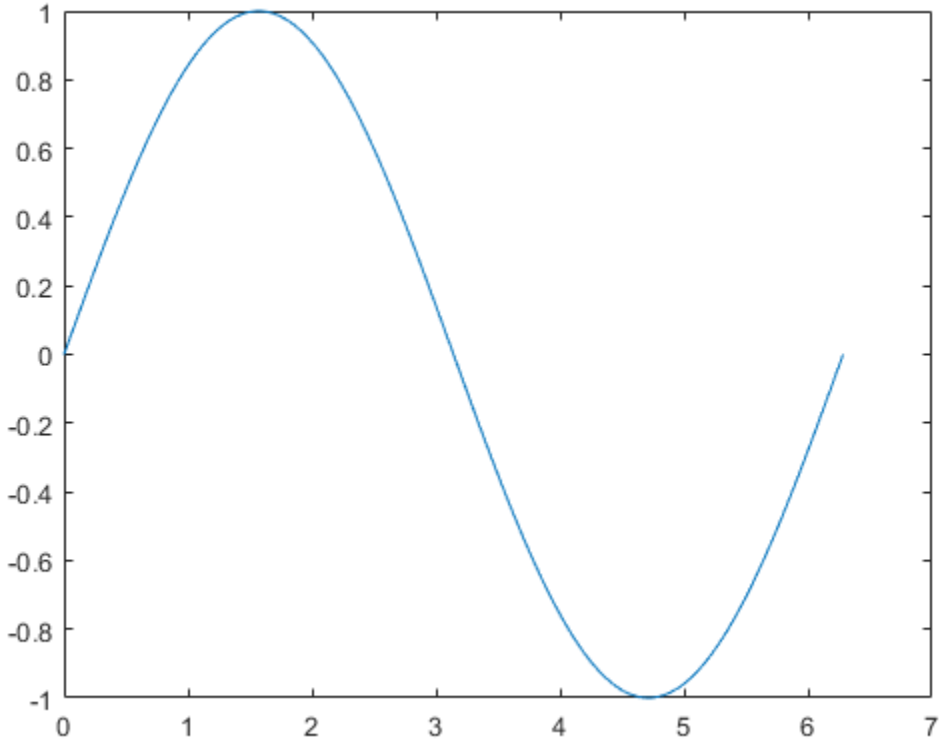
2-D and 3-D Plots

In this section...

Line Plots

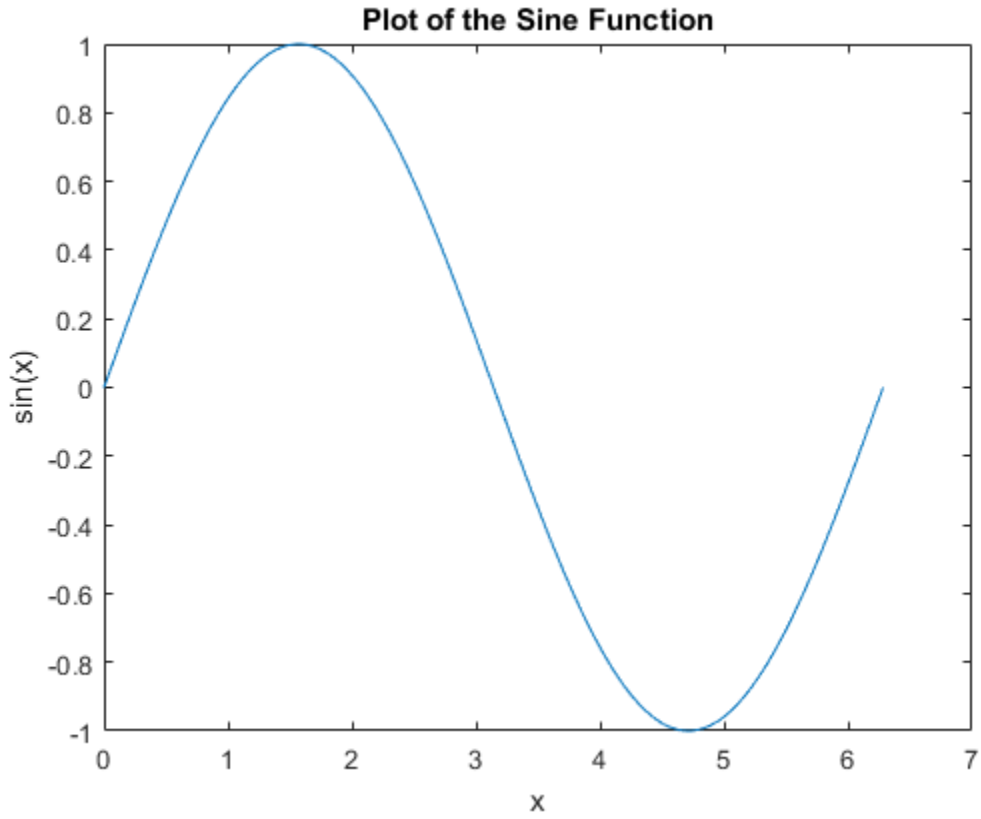
To create two-dimensional line plots, use the `plot` function. For example, plot the value of the sine function from 0 to 2π:

```
x = 0:pi/100:2*pi;
y = sin(x);
plot(x,y)
```



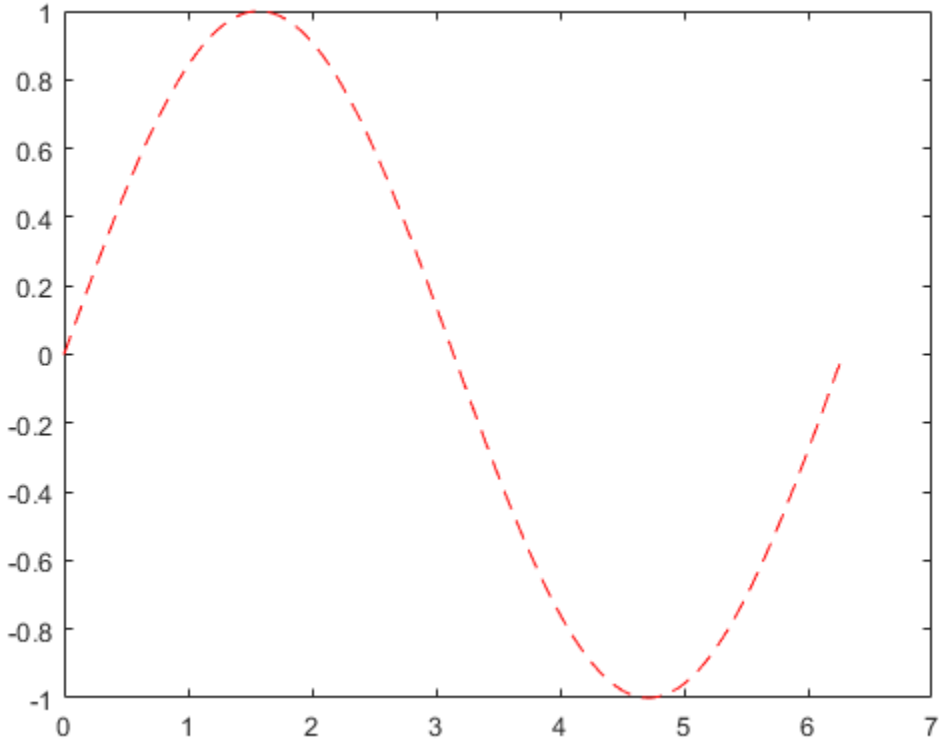
You can label the axes and add a title.

```
xlabel('x')
ylabel('sin(x)')
title('Plot of the Sine Function')
```



By adding a third input argument to the `plot` function, you can plot the same variables using a red dashed line.

```
plot(x,y,'r--')
```



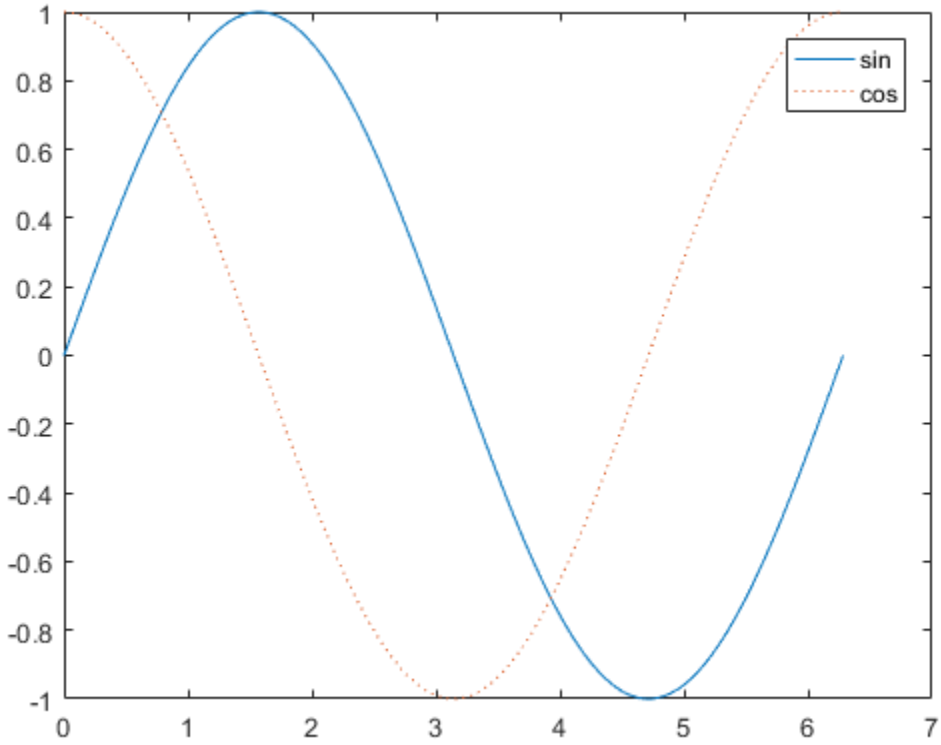
The `'r--'` string is a *line specification*. Each specification can include characters for the line color, style, and marker. A marker is a symbol that appears at each plotted data point, such as a `+`, `o`, or `*`. For example, `'g:*'` requests a dotted green line with `*` markers.

Notice that the titles and labels that you defined for the first plot are no longer in the current *figure* window. By default, MATLAB® clears the figure each time you call a plotting function, resetting the axes and other elements to prepare the new plot.

To add plots to an existing figure, use `hold`.

```
x = 0:pi/100:2*pi;
y = sin(x);
plot(x,y)
```

```
hold on

y2 = cos(x);
plot(x,y2,':')
legend('sin','cos')
```



Until you use `hold off` or close the window, all plots appear in the current figure window.

3-D Plots

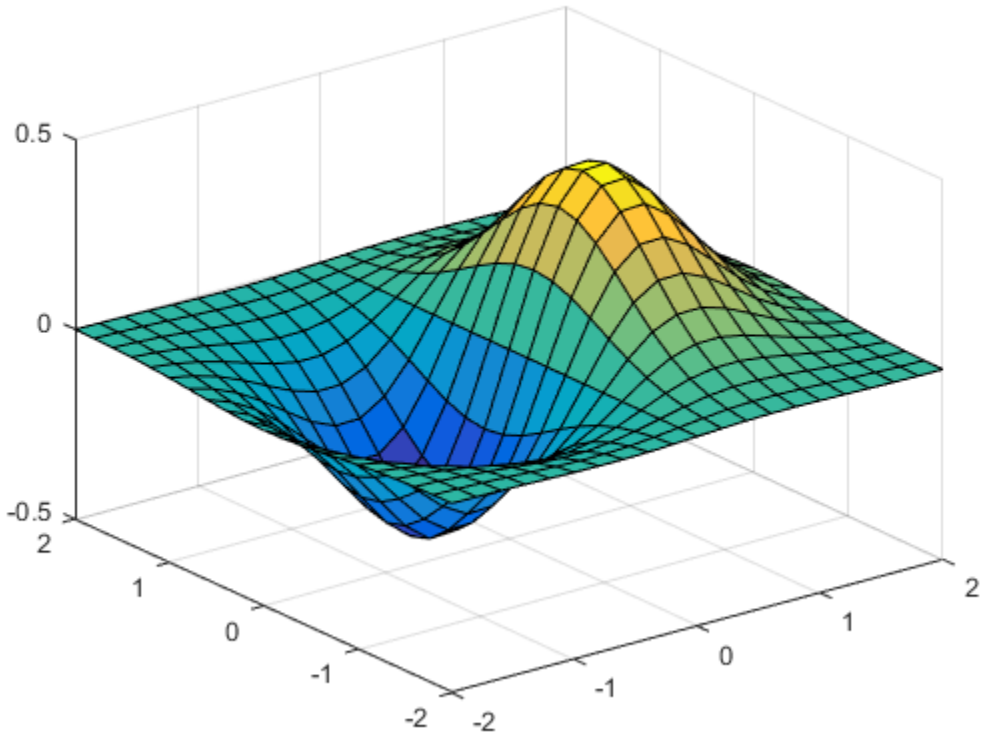
Three-dimensional plots typically display a surface defined by a function in two variables, $z = f(x,y)$.

To evaluate z, first create a set of (x,y) points over the domain of the function using `meshgrid`.

```
[X,Y] = meshgrid(-2:.2:2);
Z = X .* exp(-X.^2 - Y.^2);
```

Then, create a surface plot.

```
surf(X,Y,Z)
```



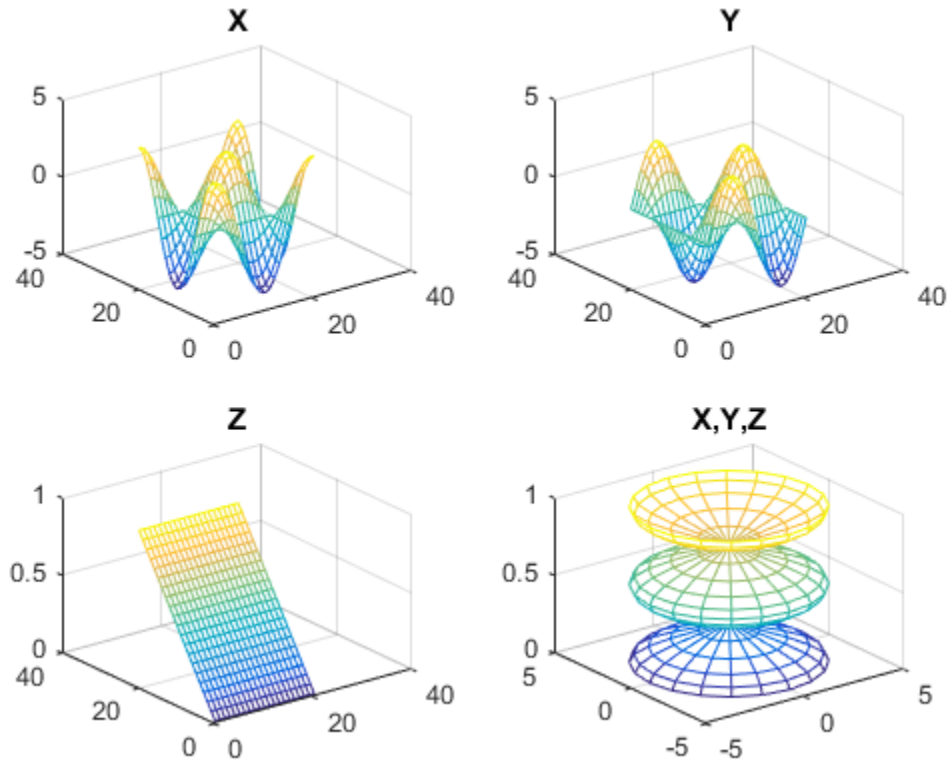
Both the `surf` function and its companion `mesh` display surfaces in three dimensions. `surf` displays both the connecting lines and the faces of the surface in color. `mesh` produces wireframe surfaces that color only the lines connecting the defining points.

Subplots

You can display multiple plots in different subregions of the same window using the `subplot` function.

The first two inputs to `subplot` indicate the number of plots in each row and column. The third input specifies which plot is active. For example, create four plots in a 2-by-2 grid within a figure window.

```
t = 0:pi/10:2*pi;
[X,Y,Z] = cylinder(4*cos(t));
subplot(2,2,1); mesh(X); title('X');
subplot(2,2,2); mesh(Y); title('Y');
subplot(2,2,3); mesh(Z); title('Z');
subplot(2,2,4); mesh(X,Y,Z); title('X,Y,Z');
```



See Also

"Programming and Scripts" on page 1-27

Programming and Scripts

In this section...

The simplest type of MATLAB program is called a *script*. A script is a file with a `.m` extension that contains multiple sequential lines of MATLAB commands and function calls. You can run a script by typing its name at the command line.

Sample Script

To create a script, use the `edit` command,

```
edit plotrand
```

This opens a blank file named `plotrand.m`. Enter some code that plots a vector of random data:

```
n = 50;
r = rand(n,1);
plot(r)
```

Next, add code that draws a horizontal line on the plot at the mean:

```
m = mean(r);
hold on
plot([0,n],[m,m])
hold off
title('Mean of Random Uniform Data')
```

Whenever you write code, it is a good practice to add comments that describe the code. Comments allow others to understand your code, and can refresh your memory when you return to it later. Add comments using the percent (%) symbol.

```
% Generate random data from a uniform distribution
% and calculate the mean. Plot the data and the mean.

n = 50;             % 50 data points
r = rand(n,1);
```

```
plot(r)

% Draw a line from (0,m) to (n,m)
m = mean(r);
hold on
plot([0,n],[m,m])
hold off
title('Mean of Random Uniform Data')
```

Save the file in the current folder. To run the script, type its name at the command line:

```
plotrand
```

You can also run scripts from the Editor by pressing the **Run** button,  .

Loops and Conditional Statements

Within a script, you can loop over sections of code and conditionally execute sections using the keywords `for`, `while`, `if`, and `switch`.

For example, create a script named `calcmean.m` that uses a `for` loop to calculate the mean of five random samples and the overall mean.

```
nsamples = 5;
npoints = 50;

for k = 1:nsamples
    currentData = rand(npoints,1);
    sampleMean(k) = mean(currentData);
end
overallMean = mean(sampleMean)
```

Now, modify the `for` loop so that you can view the results at each iteration. Display text in the Command Window that includes the current iteration number, and remove the semicolon from the assignment to `sampleMean`.

```
for k = 1:nsamples
   iterationString = ['Iteration #',int2str(k)];
   disp(iterationString)
   currentData = rand(npoints,1);
   sampleMean(k) = mean(currentData)
end
overallMean = mean(sampleMean)
```

When you run the script, it displays the intermediate results, and then calculates the overall mean.

```
calcmean
```

```
Iteration #1

sampleMean =

    0.3988

Iteration #2

sampleMean =

    0.3988    0.4950

Iteration #3

sampleMean =

    0.3988    0.4950    0.5365

Iteration #4

sampleMean =

    0.3988    0.4950    0.5365    0.4870

Iteration #5

sampleMean =

    0.3988    0.4950    0.5365    0.4870    0.5501


overallMean =

    0.4935
```

In the Editor, add conditional statements to the end of `calcmean.m` that display a different message depending on the value of `overallMean`.

```
if overallMean < .49
   disp('Mean is less than expected')
```

```
elseif overallMean > .51
   disp('Mean is greater than expected')
else
   disp('Mean is within the expected range')
end
```

Run `calcmean` and verify that the correct message displays for the calculated `overallMean`. For example:

```
overallMean =

    0.5178

Mean is greater than expected
```

Script Locations

MATLAB looks for scripts and other files in certain places. To run a script, the file must be in the current folder or in a folder on the *search path*.

By default, the `MATLAB` folder that the MATLAB Installer creates is on the search path. If you want to store and run programs in another folder, add it to the search path. Select the folder in the Current Folder browser, right-click, and then select **Add to Path**.

See Also

“Help and Documentation” on page 1-31

Help and Documentation

All MATLAB functions have supporting documentation that includes examples and describes the function inputs, outputs, and calling syntax. There are several ways to access this information from the command line:

- Open the function documentation in a separate window using the `doc` command.

  ```
  doc mean
  ```

- Display function hints (the syntax portion of the function documentation) in the Command Window by pausing after you type the open parentheses for the function input arguments.

  ```
  mean(
  ```

- View an abbreviated text version of the function documentation in the Command Window using the `help` command.

  ```
  help mean
  ```

Access the complete product documentation by clicking the help icon  .

2

Language Fundamentals

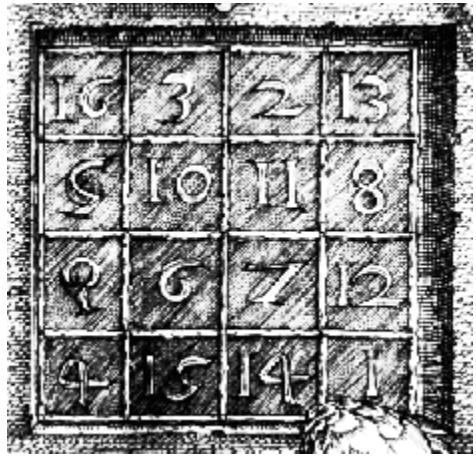
Matrices and Magic Squares

In this section...

About Matrices

In the MATLAB environment, a matrix is a rectangular array of numbers. Special meaning is sometimes attached to 1-by-1 matrices, which are scalars, and to matrices with only one row or column, which are vectors. MATLAB has other ways of storing both numeric and nonnumeric data, but in the beginning, it is usually best to think of everything as a matrix. The operations in MATLAB are designed to be as natural as possible. Where other programming languages work with numbers one at a time, MATLAB allows you to work with entire matrices quickly and easily. A good example matrix, used throughout this book, appears in the Renaissance engraving Melencolia I by the German artist and amateur mathematician Albrecht Dürer.

This image is filled with mathematical symbolism, and if you look carefully, you will see a matrix in the upper-right corner. This matrix is known as a magic square and was believed by many in Dürer's time to have genuinely magical properties. It does turn out to have some fascinating characteristics worth exploring.



Entering Matrices

The best way for you to get started with MATLAB is to learn how to handle matrices. Start MATLAB and follow along with each example.

You can enter matrices into MATLAB in several different ways:

- Enter an explicit list of elements.
- Load matrices from external data files.
- Generate matrices using built-in functions.
- Create matrices with your own functions and save them in files.

Start by entering Dürer's matrix as a list of its elements. You only have to follow a few basic conventions:

- Separate the elements of a row with blanks or commas.
- Use a semicolon, `;` , to indicate the end of each row.
- Surround the entire list of elements with square brackets, `[ ]`.

To enter Dürer's matrix, simply type in the Command Window

```
A = [16 3 2 13; 5 10 11 8; 9 6 7 12; 4 15 14 1]
```

MATLAB displays the matrix you just entered:

```
A =
    16     3     2    13
     5    10    11     8
     9     6     7    12
     4    15    14     1
```

This matrix matches the numbers in the engraving. Once you have entered the matrix, it is automatically remembered in the MATLAB workspace. You can refer to it simply as `A`. Now that you have `A` in the workspace, take a look at what makes it so interesting. Why is it magic?

sum, transpose, and diag

You are probably already aware that the special properties of a magic square have to do with the various ways of summing its elements. If you take the sum along any row or column, or along either of the two main diagonals, you will always get the same number. Let us verify that using MATLAB. The first statement to try is

```
sum(A)
```

MATLAB replies with

```
ans =
    34    34    34    34
```

When you do not specify an output variable, MATLAB uses the variable `ans`, short for *answer*, to store the results of a calculation. You have computed a row vector containing the sums of the columns of `A`. Each of the columns has the same sum, the *magic* sum, 34.

How about the row sums? MATLAB has a preference for working with the columns of a matrix, so one way to get the row sums is to transpose the matrix, compute the column sums of the transpose, and then transpose the result.

MATLAB has two transpose operators. The apostrophe operator (for example, `A'`) performs a complex conjugate transposition. It flips a matrix about its main diagonal, and also changes the sign of the imaginary component of any complex elements of the matrix. The dot-apostrophe operator (`A.'`), transposes without affecting the sign of complex elements. For matrices containing all real elements, the two operators return the same result.

So

```
A'
```

produces

```
ans =
    16     5     9     4
     3    10     6    15
     2    11     7    14
    13     8    12     1
```

and

```
sum(A')'
```

produces a column vector containing the row sums

```
ans =
    34
    34
    34
    34
```

For an additional way to sum the rows that avoids the double transpose use the dimension argument for the `sum` function:

```
sum(A,2)
```

produces

```
ans =
    34
    34
    34
    34
```

The sum of the elements on the main diagonal is obtained with the `sum` and the `diag` functions:

```
diag(A)
```

produces

```
ans =
    16
    10
     7
     1
```

and

```
sum(diag(A))
```

produces

```
ans =
    34
```

The other diagonal, the so-called *antidiagonal,* is not so important mathematically, so MATLAB does not have a ready-made function for it. But a function originally intended for use in graphics, `fliplr`, flips a matrix from left to right:

```
sum(diag(fliplr(A)))
ans =
    34
```

You have verified that the matrix in Dürer's engraving is indeed a magic square and, in the process, have sampled a few MATLAB matrix operations. The following sections continue to use this matrix to illustrate additional MATLAB capabilities.

The magic Function

MATLAB actually has a built-in function that creates magic squares of almost any size. Not surprisingly, this function is named `magic`:

```
B = magic(4)
B =
    16     2     3    13
     5    11    10     8
     9     7     6    12
     4    14    15     1
```

This matrix is almost the same as the one in the Dürer engraving and has all the same "magic" properties; the only difference is that the two middle columns are exchanged.

You can swap the two middle columns of `B` to look like Dürer's `A`. For each row of `B`, rearrange the columns in the order specified by 1, 3, 2, 4:

```
A = B(:,[1 3 2 4])

A =
    16     3     2    13
     5    10    11     8
```

```
     9     6     7    12
     4    15    14     1
```

Generating Matrices

MATLAB software provides four functions that generate basic matrices.

`zeros`	All zeros
`ones`	All ones
`rand`	Uniformly distributed random elements
`randn`	Normally distributed random elements

Here are some examples:

```
Z = zeros(2,4)
Z =
     0     0     0     0
     0     0     0     0

F = 5*ones(3,3)
F =
     5     5     5
     5     5     5
     5     5     5

N = fix(10*rand(1,10))
N =
     9     2     6     4     8     7     4     0     8     4

R = randn(4,4)
R =
    0.6353    0.0860   -0.3210   -1.2316
   -0.6014   -2.0046    1.2366    1.0556
    0.5512   -0.4931   -0.6313   -0.1132
   -1.0998    0.4620   -2.3252    0.3792
```

Expressions

In this section...

Variables

Like most other programming languages, the MATLAB language provides mathematical *expressions*, but unlike most programming languages, these expressions involve entire matrices.

MATLAB does not require any type declarations or dimension statements. When MATLAB encounters a new variable name, it automatically creates the variable and allocates the appropriate amount of storage. If the variable already exists, MATLAB changes its contents and, if necessary, allocates new storage. For example,

```
num_students = 25
```

creates a 1-by-1 matrix named `num_students` and stores the value 25 in its single element. To view the matrix assigned to any variable, simply enter the variable name.

Variable names consist of a letter, followed by any number of letters, digits, or underscores. MATLAB is case sensitive; it distinguishes between uppercase and lowercase letters. `A` and `a` are *not* the same variable.

Although variable names can be of any length, MATLAB uses only the first `N` characters of the name, (where `N` is the number returned by the function `namelengthmax`), and ignores the rest. Hence, it is important to make each variable name unique in the first `N` characters to enable MATLAB to distinguish variables.

```
N = namelengthmax
N =
    63
```

Numbers

MATLAB uses conventional decimal notation, with an optional decimal point and leading plus or minus sign, for numbers. Scientific notation uses the letter `e` to specify a power-of-ten scale factor. Imaginary numbers use either `i` or `j` as a suffix. Some examples of legal numbers are

```
3              -99            0.0001
9.6397238      1.60210e-20    6.02252e23
1i             -3.14159j      3e5i
```

MATLAB stores all numbers internally using the *long* format specified by the IEEE® floating-point standard. Floating-point numbers have a finite *precision* of roughly 16 significant decimal digits and a finite *range* of roughly 10^{-308} to 10^{+308}.

Numbers represented in the double format have a maximum precision of 52 bits. Any double requiring more bits than 52 loses some precision. For example, the following code shows two unequal values to be equal because they are both truncated:

```
x = 36028797018963968;
y = 36028797018963972;
x == y
ans =
      1
```

Integers have available precisions of 8-bit, 16-bit, 32-bit, and 64-bit. Storing the same numbers as 64-bit integers preserves precision:

```
x = uint64(36028797018963968);
y = uint64(36028797018963972);
x == y
ans =
      0
```

MATLAB software stores the real and imaginary parts of a complex number. It handles the magnitude of the parts in different ways depending on the context. For instance, the `sort` function sorts based on magnitude and resolves ties by phase angle.

```
sort([3+4i, 4+3i])
ans =
   4.0000 + 3.0000i   3.0000 + 4.0000i
```

This is because of the phase angle:

```
angle(3+4i)
```

```
ans =
    0.9273
angle(4+3i)
ans =
    0.6435
```

The “equal to” relational operator == requires both the real and imaginary parts to be equal. The other binary relational operators > <, >=, and <= ignore the imaginary part of the number and consider the real part only.

Matrix Operators

Expressions use familiar arithmetic operators and precedence rules.

+	Addition
-	Subtraction
*	Multiplication
/	Division
\	Left division
^	Power
'	Complex conjugate transpose
( )	Specify evaluation order

Array Operators

When they are taken away from the world of linear algebra, matrices become two-dimensional numeric arrays. Arithmetic operations on arrays are done element by element. This means that addition and subtraction are the same for arrays and matrices, but that multiplicative operations are different. MATLAB uses a dot, or decimal point, as part of the notation for multiplicative array operations.

The list of operators includes

+	Addition
-	Subtraction
.*	Element-by-element multiplication

`./`	Element-by-element division
`.\`	Element-by-element left division
`.^`	Element-by-element power
`.'`	Unconjugated array transpose

If the Dürer magic square is multiplied by itself with array multiplication

```
A.*A
```

the result is an array containing the squares of the integers from 1 to 16, in an unusual order:

```
ans =
   256     9     4   169
    25   100   121    64
    81    36    49   144
    16   225   196     1
```

Building Tables

Array operations are useful for building tables. Suppose `n` is the column vector

```
n = (0:9)';
```

Then

```
pows = [n  n.^2  2.^n]
```

builds a table of squares and powers of 2:

```
pows =
     0     0     1
     1     1     2
     2     4     4
     3     9     8
     4    16    16
     5    25    32
     6    36    64
     7    49   128
     8    64   256
     9    81   512
```

The elementary math functions operate on arrays element by element. So

```
format short g
x = (1:0.1:2)';
logs = [x log10(x)]
```

builds a table of logarithms.

```
 logs =
      1.0            0
      1.1      0.04139
      1.2      0.07918
      1.3      0.11394
      1.4      0.14613
      1.5      0.17609
      1.6      0.20412
      1.7      0.23045
      1.8      0.25527
      1.9      0.27875
      2.0      0.30103
```

Functions

MATLAB provides a large number of standard elementary mathematical functions, including `abs`, `sqrt`, `exp`, and `sin`. Taking the square root or logarithm of a negative number is not an error; the appropriate complex result is produced automatically. MATLAB also provides many more advanced mathematical functions, including Bessel and gamma functions. Most of these functions accept complex arguments. For a list of the elementary mathematical functions, type

```
help elfun
```

For a list of more advanced mathematical and matrix functions, type

```
help specfun
help elmat
```

Some of the functions, like `sqrt` and `sin`, are *built in*. Built-in functions are part of the MATLAB core so they are very efficient, but the computational details are not readily accessible. Other functions are implemented in the MATLAB programing language, so their computational details are accessible.

There are some differences between built-in functions and other functions. For example, for built-in functions, you cannot see the code. For other functions, you can see the code and even modify it if you want.

Several special functions provide values of useful constants.

`pi`	3.14159265...
`i`	Imaginary unit, $\sqrt{-1}$
`j`	Same as `i`
`eps`	Floating-point relative precision, $\varepsilon = 2^{-52}$
`realmin`	Smallest floating-point number, 2^{-1022}
`realmax`	Largest floating-point number, $(2-\varepsilon)2^{1023}$
`Inf`	Infinity
`NaN`	Not-a-number

Infinity is generated by dividing a nonzero value by zero, or by evaluating well defined mathematical expressions that *overflow*, that is, exceed `realmax`. Not-a-number is generated by trying to evaluate expressions like `0/0` or `Inf-Inf` that do not have well defined mathematical values.

The function names are not reserved. It is possible to overwrite any of them with a new variable, such as

```
eps = 1.e-6
```

and then use that value in subsequent calculations. The original function can be restored with

```
clear eps
```

Examples of Expressions

You have already seen several examples of MATLAB expressions. Here are a few more examples, and the resulting values:

```
rho = (1+sqrt(5))/2
rho =
    1.6180

a = abs(3+4i)
a =
```

```
     5

z = sqrt(besselk(4/3,rho-i))
z =
   0.3730+ 0.3214i

huge = exp(log(realmax))
huge =
  1.7977e+308

toobig = pi*huge
toobig =
   Inf
```

Entering Commands

In this section...

The format Function

The `format` function controls the numeric format of the values displayed. The function affects only how numbers are displayed, not how MATLAB software computes or saves them. Here are the different formats, together with the resulting output produced from a vector `x` with components of different magnitudes.

Note To ensure proper spacing, use a fixed-width font, such as Courier.

```
x = [4/3 1.2345e-6]

format short

   1.3333    0.0000

format short e

   1.3333e+000  1.2345e-006

format short g

   1.3333  1.2345e-006

format long

   1.33333333333333   0.00000123450000

format long e

   1.333333333333333e+000    1.234500000000000e-006
```

```
format long g

   1.33333333333333                1.2345e-006

format bank

   1.33          0.00

format rat

   4/3          1/810045

format hex

   3ff5555555555555   3eb4b6231abfd271
```

If the largest element of a matrix is larger than 10^3 or smaller than 10^{-3}, MATLAB applies a common scale factor for the short and long formats.

In addition to the `format` functions shown above

```
format compact
```

suppresses many of the blank lines that appear in the output. This lets you view more information on a screen or window. If you want more control over the output format, use the `sprintf` and `fprintf` functions.

Suppressing Output

If you simply type a statement and press **Return** or **Enter**, MATLAB automatically displays the results on screen. However, if you end the line with a semicolon, MATLAB performs the computation, but does not display any output. This is particularly useful when you generate large matrices. For example,

```
A = magic(100);
```

Entering Long Statements

If a statement does not fit on one line, use an ellipsis (three periods), `...`, followed by **Return** or **Enter** to indicate that the statement continues on the next line. For example,

```
s = 1 -1/2 + 1/3 -1/4 + 1/5 - 1/6 + 1/7 ...
```

```
    - 1/8 + 1/9 - 1/10 + 1/11 - 1/12;
```

Blank spaces around the =, +, and - signs are optional, but they improve readability.

Command Line Editing

Various arrow and control keys on your keyboard allow you to recall, edit, and reuse statements you have typed earlier. For example, suppose you mistakenly enter

```
rho = (1 + sqt(5))/2
```

You have misspelled `sqrt`. MATLAB responds with

```
Undefined function 'sqt' for input arguments of type 'double'.
```

Instead of retyping the entire line, simply press the ↑ key. The statement you typed is redisplayed. Use the ← key to move the cursor over and insert the missing `r`. Repeated use of the ↑ key recalls earlier lines. Typing a few characters, and then pressing the ↑ key finds a previous line that begins with those characters. You can also copy previously executed statements from the Command History.

Indexing

In this section...

Subscripts

The element in row `i` and column `j` of `A` is denoted by `A(i,j)`. For example, `A(4,2)` is the number in the fourth row and second column. For the magic square, `A(4,2)` is `15`. So to compute the sum of the elements in the fourth column of `A`, type

```
A(1,4) + A(2,4) + A(3,4) + A(4,4)
```

This subscript produces

```
ans =
     34
```

but is not the most elegant way of summing a single column.

It is also possible to refer to the elements of a matrix with a single subscript, `A(k)`. A single subscript is the usual way of referencing row and column vectors. However, it can also apply to a fully two-dimensional matrix, in which case the array is regarded as one long column vector formed from the columns of the original matrix. So, for the magic square, `A(8)` is another way of referring to the value `15` stored in `A(4,2)`.

If you try to use the value of an element outside of the matrix, it is an error:

```
t = A(4,5)
Index exceeds matrix dimensions.
```

Conversely, if you store a value in an element outside of the matrix, the size increases to accommodate the newcomer:

```
X = A;
X(4,5) = 17

X =
    16     3     2    13     0
     5    10    11     8     0
     9     6     7    12     0
     4    15    14     1    17
```

The Colon Operator

The colon, `:`, is one of the most important MATLAB operators. It occurs in several different forms. The expression

```
1:10
```

is a row vector containing the integers from 1 to 10:

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

To obtain nonunit spacing, specify an increment. For example,

```
100:-7:50
```

is

```
100    93    86    79    72    65    58    51
```

and

```
0:pi/4:pi
```

is

```
0    0.7854    1.5708    2.3562    3.1416
```

Subscript expressions involving colons refer to portions of a matrix:

```
A(1:k,j)
```

is the first `k` elements of the `j`th column of `A`. Thus,

```
sum(A(1:4,4))
```

computes the sum of the fourth column. However, there is a better way to perform this computation. The colon by itself refers to *all* the elements in a row or column of a matrix and the keyword `end` refers to the *last* row or column. Thus,

```
sum(A(:,end))
```

computes the sum of the elements in the last column of A:

```
ans =
     34
```

Why is the magic sum for a 4-by-4 square equal to 34? If the integers from 1 to 16 are sorted into four groups with equal sums, that sum must be

```
sum(1:16)/4
```

which, of course, is

```
ans =
     34
```

Concatenation

Concatenation is the process of joining small matrices to make bigger ones. In fact, you made your first matrix by concatenating its individual elements. The pair of square brackets, `[]`, is the concatenation operator. For an example, start with the 4-by-4 magic square, A, and form

```
B = [A  A+32; A+48  A+16]
```

The result is an 8-by-8 matrix, obtained by joining the four submatrices:

```
B =

    16     3     2    13    48    35    34    45
     5    10    11     8    37    42    43    40
     9     6     7    12    41    38    39    44
     4    15    14     1    36    47    46    33
    64    51    50    61    32    19    18    29
    53    58    59    56    21    26    27    24
    57    54    55    60    25    22    23    28
    52    63    62    49    20    31    30    17
```

This matrix is halfway to being another magic square. Its elements are a rearrangement of the integers `1:64`. Its column sums are the correct value for an 8-by-8 magic square:

```
sum(B)

ans =
```

```
   260   260   260   260   260   260   260   260
```

But its row sums, `sum(B')'`, are not all the same. Further manipulation is necessary to make this a valid 8-by-8 magic square.

Deleting Rows and Columns

You can delete rows and columns from a matrix using just a pair of square brackets. Start with

```
X = A;
```

Then, to delete the second column of `X`, use

```
X(:,2) = []
```

This changes `X` to

```
X =
    16     2    13
     5    11     8
     9     7    12
     4    14     1
```

If you delete a single element from a matrix, the result is not a matrix anymore. So, expressions like

```
X(1,2) = []
```

result in an error. However, using a single subscript deletes a single element, or sequence of elements, and reshapes the remaining elements into a row vector. So

```
X(2:2:10) = []
```

results in

```
X =
    16     9     2     7    13    12     1
```

Scalar Expansion

Matrices and scalars can be combined in several different ways. For example, a scalar is subtracted from a matrix by subtracting it from each element. The average value of the elements in our magic square is 8.5, so

```
B = A - 8.5
```

forms a matrix whose column sums are zero:

```
B =
      7.5     -5.5     -6.5      4.5
     -3.5      1.5      2.5     -0.5
      0.5     -2.5     -1.5      3.5
     -4.5      6.5      5.5     -7.5

sum(B)

ans =
     0     0     0     0
```

With scalar expansion, MATLAB assigns a specified scalar to all indices in a range. For example,

```
B(1:2,2:3) = 0
```

zeros out a portion of `B`:

```
B =
      7.5        0        0      4.5
     -3.5        0        0     -0.5
      0.5     -2.5     -1.5      3.5
     -4.5      6.5      5.5     -7.5
```

Logical Subscripting

The logical vectors created from logical and relational operations can be used to reference subarrays. Suppose `X` is an ordinary matrix and `L` is a matrix of the same size that is the result of some logical operation. Then `X(L)` specifies the elements of `X` where the elements of `L` are nonzero.

This kind of subscripting can be done in one step by specifying the logical operation as the subscripting expression. Suppose you have the following set of data:

```
x = [2.1 1.7 1.6 1.5 NaN 1.9 1.8 1.5 5.1 1.8 1.4 2.2 1.6 1.8];
```

The `NaN` is a marker for a missing observation, such as a failure to respond to an item on a questionnaire. To remove the missing data with logical indexing, use `isfinite(x)`, which is true for all finite numerical values and false for `NaN` and `Inf`:

```
x = x(isfinite(x))
x =
  2.1 1.7 1.6 1.5 1.9 1.8 1.5 5.1 1.8 1.4 2.2 1.6 1.8
```

Now there is one observation, `5.1`, which seems to be very different from the others. It is an *outlier*. The following statement removes outliers, in this case those elements more than three standard deviations from the mean:

```
x = x(abs(x-mean(x)) <= 3*std(x))
x =
  2.1 1.7 1.6 1.5 1.9 1.8 1.5 1.8 1.4 2.2 1.6 1.8
```

For another example, highlight the location of the prime numbers in Dürer's magic square by using logical indexing and scalar expansion to set the nonprimes to 0. (See "The magic Function" on page 2-7.)

```
A(~isprime(A)) = 0

A =
     0     3     2    13
     5     0    11     0
     0     0     7     0
     0     0     0     0
```

The find Function

The `find` function determines the indices of array elements that meet a given logical condition. In its simplest form, `find` returns a column vector of indices. Transpose that vector to obtain a row vector of indices. For example, start again with Dürer's magic square. (See "The magic Function" on page 2-7.)

```
k = find(isprime(A))'
```

picks out the locations, using one-dimensional indexing, of the primes in the magic square:

```
k =
     2     5     9    10    11    13
```

Display those primes, as a row vector in the order determined by `k`, with

```
A(k)
```

```
ans =
     5     3     2    11     7    13
```

When you use `k` as a left-side index in an assignment statement, the matrix structure is preserved:

```
A(k) = NaN

A =
    16   NaN   NaN   NaN
   NaN    10   NaN     8
     9     6   NaN    12
     4    15    14     1
```

Types of Arrays

In this section...

Multidimensional Arrays

Multidimensional arrays in the MATLAB environment are arrays with more than two subscripts. One way of creating a multidimensional array is by calling `zeros`, `ones`, `rand`, or `randn` with more than two arguments. For example,

```
R = randn(3,4,5);
```

creates a 3-by-4-by-5 array with a total of `3*4*5 = 60` normally distributed random elements.

A three-dimensional array might represent three-dimensional physical data, say the temperature in a room, sampled on a rectangular grid. Or it might represent a sequence of matrices, $A^{(k)}$, or samples of a time-dependent matrix, $A(t)$. In these latter cases, the (i, j)th element of the kth matrix, or the t_kth matrix, is denoted by `A(i,j,k)`.

MATLAB and Dürer's versions of the magic square of order 4 differ by an interchange of two columns. Many different magic squares can be generated by interchanging columns. The statement

```
p = perms(1:4);
```

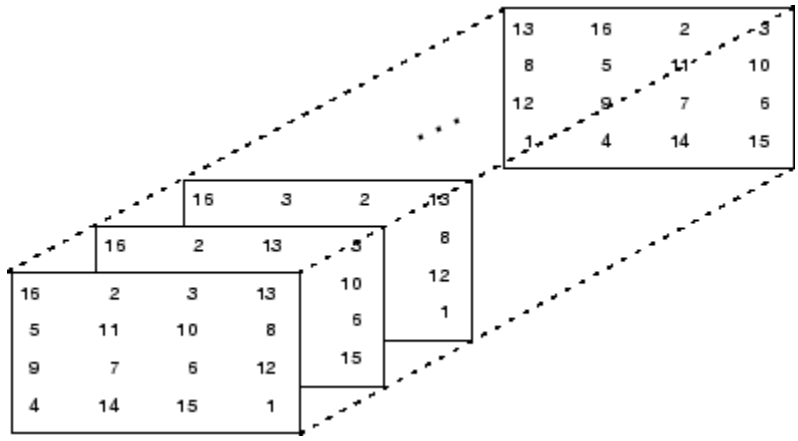
generates the 4! = 24 permutations of `1:4`. The `k`th permutation is the row vector `p(k,:)`. Then

```
A = magic(4);
M = zeros(4,4,24);

for k = 1:24
```

```
   M(:,:,k) = A(:,p(k,:));
end
```

stores the sequence of 24 magic squares in a three-dimensional array, `M`. The size of `M` is

```
size(M)

ans =
     4     4    24
```



Note The order of the matrices shown in this illustration might differ from your results. The `perms` function always returns all permutations of the input vector, but the order of the permutations might be different for different MATLAB versions.

The statement

```
sum(M,d)
```

computes sums by varying the `d`th subscript. So

```
sum(M,1)
```

is a 1-by-4-by-24 array containing 24 copies of the row vector

```
34     34     34     34
```

and

```
sum(M,2)
```

is a 4-by-1-by-24 array containing 24 copies of the column vector

```
34
34
34
34
```

Finally,

```
S = sum(M,3)
```

adds the 24 matrices in the sequence. The result has size 4-by-4-by-1, so it looks like a 4-by-4 array:

```
S =
   204   204   204   204
   204   204   204   204
   204   204   204   204
   204   204   204   204
```

Cell Arrays

Cell arrays in MATLAB are multidimensional arrays whose elements are copies of other arrays. A cell array of empty matrices can be created with the `cell` function. But, more often, cell arrays are created by enclosing a miscellaneous collection of things in curly braces, `{}`. The curly braces are also used with subscripts to access the contents of various cells. For example,

```
C = {A sum(A) prod(prod(A))}
```

produces a 1-by-3 cell array. The three cells contain the magic square, the row vector of column sums, and the product of all its elements. When `C` is displayed, you see

```
C =
    [4x4 double]    [1x4 double]    [20922789888000]
```

This is because the first two cells are too large to print in this limited space, but the third cell contains only a single number, 16!, so there is room to print it.

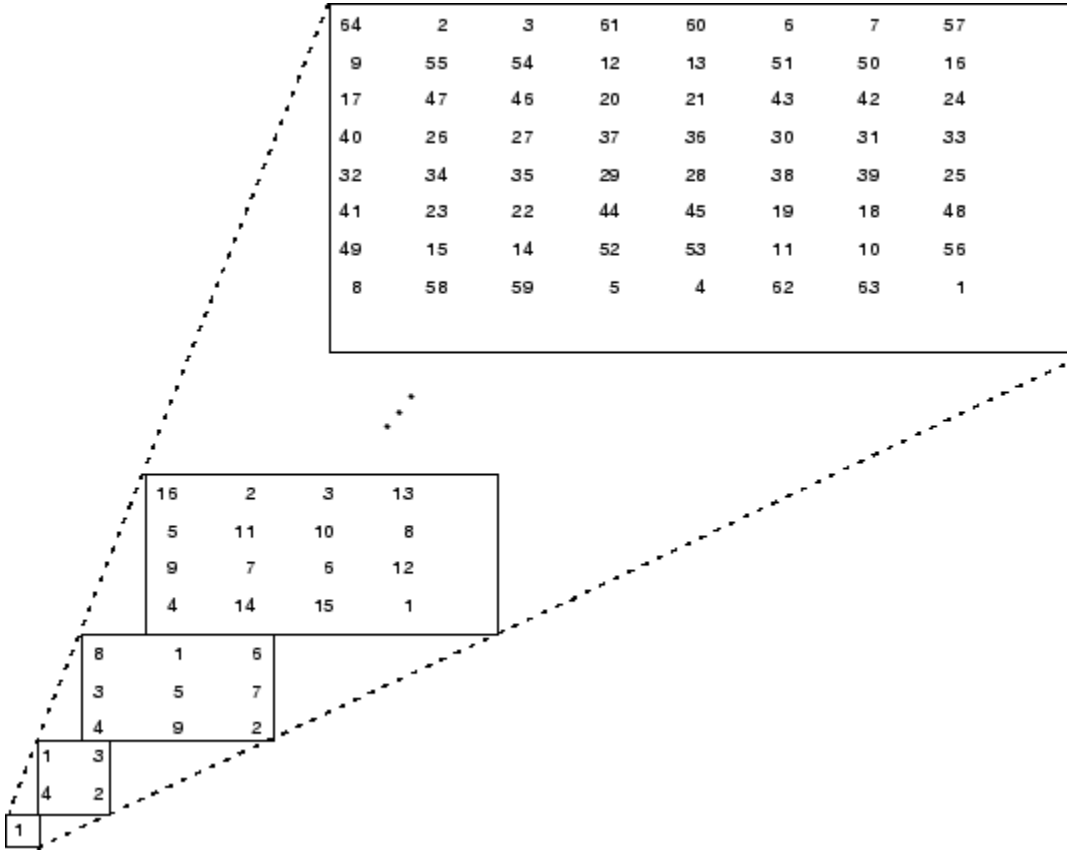
Here are two important points to remember. First, to retrieve the contents of one of the cells, use subscripts in curly braces. For example, `C{1}` retrieves the magic square and `C{3}` is 16!. Second, cell arrays contain *copies* of other arrays, not *pointers* to those arrays. If you subsequently change `A`, nothing happens to `C`.

You can use three-dimensional arrays to store a sequence of matrices of the *same* size. Cell arrays can be used to store a sequence of matrices of *different* sizes. For example,

```
M = cell(8,1);
for n = 1:8
   M{n} = magic(n);
end
M
```

produces a sequence of magic squares of different order:

```
M =
    [           1]
    [ 2x2  double]
    [ 3x3  double]
    [ 4x4  double]
    [ 5x5  double]
    [ 6x6  double]
    [ 7x7  double]
    [ 8x8  double]
```



You can retrieve the 4-by-4 magic square matrix with

```
M{4}
```

Characters and Text

Enter text into MATLAB using single quotes. For example,

```
s = 'Hello'
```

The result is not the same kind of numeric matrix or array you have been dealing with up to now. It is a 1-by-5 character array.

Internally, the characters are stored as numbers, but not in floating-point format. The statement

```
a = double(s)
```

converts the character array to a numeric matrix containing floating-point representations of the ASCII codes for each character. The result is

```
a =
    72    101    108    108    111
```

The statement

```
s = char(a)
```

reverses the conversion.

Converting numbers to characters makes it possible to investigate the various fonts available on your computer. The printable characters in the basic ASCII character set are represented by the integers `32:127`. (The integers less than 32 represent nonprintable control characters.) These integers are arranged in an appropriate 6-by-16 array with

```
F = reshape(32:127,16,6)';
```

The printable characters in the extended ASCII character set are represented by `F+128`. When these integers are interpreted as characters, the result depends on the font currently being used. Type the statements

```
char(F)
char(F+128)
```

and then vary the font being used for the Command Window. To change the font, on the **Home** tab, in the **Environment** section, click **Preferences > Fonts**. If you include tabs in lines of code, use a fixed-width font, such as `Monospaced`, to align the tab positions on different lines.

Concatenation with square brackets joins text variables together. The statement

```
h = [s, ' world']
```

joins the characters horizontally and produces

```
h =
   Hello world
```

The statement

```
v = [s; 'world']
```

joins the characters vertically and produces

```
v =
   Hello
   world
```

Notice that a blank has to be inserted before the `'w'` in `h` and that both words in `v` have to have the same length. The resulting arrays are both character arrays; `h` is 1-by-11 and `v` is 2-by-5.

To manipulate a body of text containing lines of different lengths, you have two choices—a padded character array or a cell array of character vectors. When creating a character array, you must make each row of the array the same length. (Pad the ends of the shorter rows with spaces.) The `char` function does this padding for you. For example,

```
S = char('A','rolling','stone','gathers','momentum.')
```

produces a 5-by-9 character array:

```
S =
A
rolling
stone
gathers
momentum.
```

Alternatively, you can store the text in a cell array. For example,

```
C = {'A';'rolling';'stone';'gathers';'momentum.'}
```

creates a 5-by-1 cell array that requires no padding because each row of the array can have a different length:

```
C =
    'A'
    'rolling'
    'stone'
    'gathers'
    'momentum.'
```

You can convert a padded character array to a cell array of character vectors with

```
C = cellstr(S)
```

and reverse the process with

```
S = char(C)
```

Structures

Structures are multidimensional MATLAB arrays with elements accessed by textual *field designators*. For example,

```
S.name = 'Ed Plum';
S.score = 83;
S.grade = 'B+'
```

creates a scalar structure with three fields:

```
S =
     name: 'Ed Plum'
    score: 83
    grade: 'B+'
```

Like everything else in the MATLAB environment, structures are arrays, so you can insert additional elements. In this case, each element of the array is a structure with several fields. The fields can be added one at a time,

```
S(2).name = 'Toni Miller';
S(2).score = 91;
S(2).grade = 'A-';
```

or an entire element can be added with a single statement:

```
S(3) = struct('name','Jerry Garcia',...
              'score',70,'grade','C')
```

Now the structure is large enough that only a summary is printed:

```
S =
1x3 struct array with fields:
    name
    score
    grade
```

There are several ways to reassemble the various fields into other MATLAB arrays. They are mostly based on the notation of a *comma-separated list*. If you type

```
S.score
```

it is the same as typing

```
S(1).score, S(2).score, S(3).score
```

which is a comma-separated list.

If you enclose the expression that generates such a list within square brackets, MATLAB stores each item from the list in an array. In this example, MATLAB creates a numeric row vector containing the `score` field of each element of structure array `S`:

```
scores = [S.score]
scores =
    83    91    70

avg_score = sum(scores)/length(scores)
avg_score =
   81.3333
```

To create a character array from one of the text fields (`name`, for example), call the `char` function on the comma-separated list produced by `S.name`:

```
names = char(S.name)
names =
   Ed Plum
   Toni Miller
   Jerry Garcia
```

Similarly, you can create a cell array from the `name` fields by enclosing the list-generating expression within curly braces:

```
names = {S.name}
names =
    'Ed Plum'    'Toni Miller'    'Jerry Garcia'
```

To assign the fields of each element of a structure array to separate variables outside of the structure, specify each output to the left of the equals sign, enclosing them all within square brackets:

```
[N1 N2 N3] = S.name
N1 =
   Ed Plum
N2 =
   Toni Miller
```

```
N3 =
   Jerry Garcia
```

Dynamic Field Names

The most common way to access the data in a structure is by specifying the name of the field that you want to reference. Another means of accessing structure data is to use dynamic field names. These names express the field as a variable expression that MATLAB evaluates at run time. The dot-parentheses syntax shown here makes `expression` a dynamic field name:

```
structName.(expression)
```

Index into this field using the standard MATLAB indexing syntax. For example, to evaluate `expression` into a field name and obtain the values of that field at columns `1` through `25` of row `7`, use

```
structName.(expression)(7,1:25)
```

Dynamic Field Names Example

The `avgscore` function shown below computes an average test score, retrieving information from the `testscores` structure using dynamic field names:

```
function avg = avgscore(testscores, student, first, last)
for k = first:last
   scores(k) = testscores.(student).week(k);
end
avg = sum(scores)/(last - first + 1);
```

You can run this function using different values for the dynamic field `student`. First, initialize the structure that contains scores for a 25-week period:

```
testscores.Ann_Lane.week(1:25) = ...
  [95 89 76 82 79 92 94 92 89 81 75 93 ...
   85 84 83 86 85 90 82 82 84 79 96 88 98];

testscores.William_King.week(1:25) = ...
  [87 80 91 84 99 87 93 87 97 87 82 89 ...
   86 82 90 98 75 79 92 84 90 93 84 78 81];
```

Now run `avgscore`, supplying the students name fields for the `testscores` structure at run time using dynamic field names:

```
avgscore(testscores, 'Ann_Lane', 7, 22)
```

```
ans =
   85.2500

avgscore(testscores, 'William_King', 7, 22)
ans =
   87.7500
```

3

Mathematics

Linear Algebra

In this section...

Matrices in the MATLAB Environment

Creating Matrices

The MATLAB environment uses the term *matrix* to indicate a variable containing real or complex numbers arranged in a two-dimensional grid. An *array* is, more generally, a vector, matrix, or higher dimensional grid of numbers. All arrays in MATLAB are rectangular, in the sense that the component vectors along any dimension are all the same length.

Symbolic Math Toolbox software extends the capabilities of MATLAB software to matrices of mathematical expressions.

MATLAB has dozens of functions that create different kinds of matrices. There are two functions you can use to create a pair of 3-by-3 example matrices for use throughout this chapter. The first example is symmetric:

```
A = pascal(3)

A =
       1     1     1
       1     2     3
       1     3     6
```

The second example is not symmetric:

```
B = magic(3)

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

Another example is a 3-by-2 rectangular matrix of random integers:

```
  C = fix(10*rand(3,2))

  C =
       9     4
       2     8
       6     7
```

A column vector is an m-by-1 matrix, a row vector is a 1-by-n matrix, and a scalar is a 1-by-1 matrix. The statements

```
u = [3; 1; 4]

v = [2 0 -1]

s = 7
```

produce a column vector, a row vector, and a scalar:

```
u =
       3
       1
       4

v =
       2     0    -1

s =
       7
```

Adding and Subtracting Matrices

Addition and subtraction of matrices is defined just as it is for arrays, element by element. Adding A to B, and then subtracting A from the result recovers B:

```
A = pascal(3);
B = magic(3);
X = A + B

X =
       9     2     7
       4     7    10
       5    12     8

Y = X - A

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

Addition and subtraction require both matrices to have the same dimension, or one of them be a scalar. If the dimensions are incompatible, an error results:

```
C = fix(10*rand(3,2))
X = A + C
Error using  +
Matrix dimensions must agree.
```

Vector Products and Transpose

A row vector and a column vector of the same length can be multiplied in either order. The result is either a scalar, the *inner* product, or a matrix, the *outer product* :

```
u = [3; 1; 4];
v = [2 0 -1];
x = v*u

x =
       2

X = u*v

X =
       6     0     -3
```

```
     2     0    -1
     8     0    -4
```

For real matrices, the *transpose* operation interchanges a_{ij} and a_{ji}. MATLAB uses the apostrophe operator (`'`) to perform a complex conjugate transpose, and uses the dot-apostrophe operator (`.'`) to transpose without conjugation. For matrices containing all real elements, the two operators return the same result.

The example matrix `A` is *symmetric*, so `A'` is equal to `A`. But, `B` is not symmetric:

```
B = magic(3);
X = B'

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

Transposition turns a row vector into a column vector:

```
x = v'

x =
     2
     0
    -1
```

If `x` and `y` are both real column vectors, the product `x*y` is not defined, but the two products

```
x'*y
```

and

```
y'*x
```

are the same scalar. This quantity is used so frequently, it has three different names: *inner* product, *scalar* product, or *dot* product.

For a complex vector or matrix, `z`, the quantity `z'` not only transposes the vector or matrix, but also converts each complex element to its complex conjugate. That is, the sign of the imaginary part of each complex element changes. So if

```
z = [1+2i 7-3i 3+4i; 6-2i 9i 4+7i]
z =
```

```
   1.0000 + 2.0000i   7.0000 - 3.0000i   3.0000 + 4.0000i
   6.0000 - 2.0000i        0 + 9.0000i   4.0000 + 7.0000i
```

then

```
z'
ans =
   1.0000 - 2.0000i   6.0000 + 2.0000i
   7.0000 + 3.0000i        0 - 9.0000i
   3.0000 - 4.0000i   4.0000 - 7.0000i
```

The unconjugated complex transpose, where the complex part of each element retains its sign, is denoted by `z.'`:

```
z.'
ans =
   1.0000 + 2.0000i   6.0000 - 2.0000i
   7.0000 - 3.0000i        0 + 9.0000i
   3.0000 + 4.0000i   4.0000 + 7.0000i
```

For complex vectors, the two scalar products `x'*y` and `y'*x` are complex conjugates of each other, and the scalar product `x'*x` of a complex vector with itself is real.

Multiplying Matrices

Multiplication of matrices is defined in a way that reflects composition of the underlying linear transformations and allows compact representation of systems of simultaneous linear equations. The matrix product $C = AB$ is defined when the column dimension of A is equal to the row dimension of B, or when one of them is a scalar. If A is m-by-p and B is p-by-n, their product C is m-by-n. The product can actually be defined using MATLAB `for` loops, `colon` notation, and vector dot products:

```
A = pascal(3);
B = magic(3);
m = 3; n = 3;
for i = 1:m
     for j = 1:n
        C(i,j) = A(i,:)*B(:,j);
     end
end
```

MATLAB uses a single asterisk to denote matrix multiplication. The next two examples illustrate the fact that matrix multiplication is not commutative; AB is usually not equal to BA:

```
X = A*B

X =
      15    15    15
      26    38    26
      41    70    39

Y = B*A

Y =
      15    28    47
      15    34    60
      15    28    43
```

A matrix can be multiplied on the right by a column vector and on the left by a row vector:

```
u = [3; 1; 4];
x = A*u

x =
       8
      17
      30

v = [2 0 -1];
y = v*B

y =
      12    -7    10
```

Rectangular matrix multiplications must satisfy the dimension compatibility conditions:

```
C = fix(10*rand(3,2));
X = A*C

X =
      17    19
      31    41
      51    70

Y = C*A

Error using  *
Inner matrix dimensions must agree.
```

Anything can be multiplied by a scalar:

```
s = 7;
w = s*v

w =
      14     0    -7
```

Identity Matrix

Generally accepted mathematical notation uses the capital letter *I* to denote identity matrices, matrices of various sizes with ones on the main diagonal and zeros elsewhere. These matrices have the property that $AI = A$ and $IA = A$ whenever the dimensions are compatible. The original version of MATLAB could not use *I* for this purpose because it did not distinguish between uppercase and lowercase letters and *i* already served as a subscript and as the complex unit. So an English language pun was introduced. The function

```
eye(m,n)
```

returns an *m*-by-*n* rectangular identity matrix and `eye(n)` returns an *n*-by-*n* square identity matrix.

Kronecker Tensor Product

The Kronecker product, `kron(X,Y)`, of two matrices is the larger matrix formed from all possible products of the elements of `X` with those of `Y`. If `X` is *m*-by-*n* and `Y` is *p*-by-*q*, then `kron(X,Y)` is *mp*-by-*nq*. The elements are arranged in the following order:

```
[X(1,1)*Y  X(1,2)*Y  . . .  X(1,n)*Y
                     . . .
 X(m,1)*Y  X(m,2)*Y  . . .  X(m,n)*Y]
```

The Kronecker product is often used with matrices of zeros and ones to build up repeated copies of small matrices. For example, if `X` is the 2-by-2 matrix

```
X =
       1     2
       3     4
```

and `I = eye(2,2)` is the 2-by-2 identity matrix, then the two matrices

```
kron(X,I)
```

and

```
kron(I,X)
```

are

```
     1     0     2     0
     0     1     0     2
     3     0     4     0
     0     3     0     4
```

and

```
     1     2     0     0
     3     4     0     0
     0     0     1     2
     0     0     3     4
```

Vector and Matrix Norms

The p-norm of a vector x,

$$\|x\|_p = \left(\sum |x_i|^p\right)^{1/p},$$

is computed by `norm(x,p)`. This is defined by any value of $p > 1$, but the most common values of p are 1, 2, and ∞. The default value is $p = 2$, which corresponds to *Euclidean length*:

```
v = [2 0 -1];
[norm(v,1) norm(v) norm(v,inf)]

ans =
    3.0000    2.2361    2.0000
```

The p-norm of a matrix A,

$$\|A\|_p = \max_x \frac{\|Ax\|_p}{\|x\|_p},$$

can be computed for $p = 1$, 2, and ∞ by `norm(A,p)`. Again, the default value is $p = 2$:

```
C = fix(10*rand(3,2));
[norm(C,1) norm(C) norm(C,inf)]

ans =
     19.0000   14.8015   13.0000
```

Using Multithreaded Computation with Linear Algebra Functions

MATLAB software supports multithreaded computation for a number of linear algebra and element-wise numerical functions. These functions automatically execute on multiple threads. For a function or expression to execute faster on multiple CPUs, a number of conditions must be true:

1. The function performs operations that easily partition into sections that execute concurrently. These sections must be able to execute with little communication between processes. They should require few sequential operations.
2. The data size is large enough so that any advantages of concurrent execution outweigh the time required to partition the data and manage separate execution threads. For example, most functions speed up only when the array contains than several thousand elements or more.
3. The operation is not memory-bound; processing time is not dominated by memory access time. As a general rule, complex functions speed up more than simple functions.

The matrix multiply (`X*Y`) and matrix power (`X^p`) operators show significant increase in speed on large double-precision arrays (on order of 10,000 elements). The matrix analysis functions `det`, `rcond`, `hess`, and `expm` also show significant increase in speed on large double-precision arrays.

Systems of Linear Equations

• "Iterative Methods for Solving Systems of Linear Equations" on page 3-20

Computational Considerations

One of the most important problems in technical computing is the solution of systems of simultaneous linear equations.

In matrix notation, the general problem takes the following form: Given two matrices A and b, does there exist a unique matrix x, so that $Ax = b$ or $xA = b$?

It is instructive to consider a 1-by-1 example. For example, does the equation
$7x = 21$

have a unique solution?

The answer, of course, is yes. The equation has the unique solution $x = 3$. The solution is easily obtained by division:
$x = 21/7 = 3$.

The solution is *not* ordinarily obtained by computing the inverse of 7, that is $7^{-1} = 0.142857...$, and then multiplying 7^{-1} by 21. This would be more work and, if 7^{-1} is represented to a finite number of digits, less accurate. Similar considerations apply to sets of linear equations with more than one unknown; the MATLAB software solves such equations without computing the inverse of the matrix.

Although it is not standard mathematical notation, MATLAB uses the division terminology familiar in the scalar case to describe the solution of a general system of simultaneous equations. The two division symbols, *slash*, /, and *backslash*, \, correspond to the two MATLAB functions `mrdivide` and `mldivide`. `mrdivide` and `mldivide` are used for the two situations where the unknown matrix appears on the left or right of the coefficient matrix:

`x = b/A`	Denotes the solution to the matrix equation $xA = b$.
`x = A\b`	Denotes the solution to the matrix equation $Ax = b$.

Think of "dividing" both sides of the equation $Ax = b$ or $xA = b$ by A. The coefficient matrix `A` is always in the "denominator."

The dimension compatibility conditions for `x = A\b` require the two matrices `A` and `b` to have the same number of rows. The solution `x` then has the same number of columns as

`b` and its row dimension is equal to the column dimension of `A`. For `x = b/A`, the roles of rows and columns are interchanged.

In practice, linear equations of the form $Ax = b$ occur more frequently than those of the form $xA = b$. Consequently, the backslash is used far more frequently than the slash. The remainder of this section concentrates on the backslash operator; the corresponding properties of the slash operator can be inferred from the identity:

```
(b/A)' = (A'\b').
```

The coefficient matrix `A` need not be square. If `A` is m-by-n, there are three cases:

m = n	Square system. Seek an exact solution.
m > n	Overdetermined system. Find a least-squares solution.
m < n	Underdetermined system. Find a basic solution with at most m nonzero components.

The mldivide Algorithm

The `mldivide` operator employs different solvers to handle different kinds of coefficient matrices. The various cases are diagnosed automatically by examining the coefficient matrix. For more information, see the "Algorithms" section of the `mldivide` reference page.

General Solution

The general solution to a system of linear equations $Ax = b$ describes all possible solutions. You can find the general solution by:

1. Solving the corresponding homogeneous system $Ax = 0$. Do this using the `null` command, by typing `null(A)`. This returns a basis for the solution space to $Ax = 0$. Any solution is a linear combination of basis vectors.
2. Finding a particular solution to the nonhomogeneous system $Ax = b$

You can then write any solution to $Ax = b$ as the sum of the particular solution to $Ax = b$, from step 2, plus a linear combination of the basis vectors from step 1.

The rest of this section describes how to use MATLAB to find a particular solution to $Ax = b$, as in step 2.

Square Systems

The most common situation involves a square coefficient matrix `A` and a single right-side column vector `b`.

Nonsingular Coefficient Matrix

If the matrix `A` is nonsingular, the solution, `x = A\b`, is then the same size as `b`. For example:

```
A = pascal(3);
u = [3; 1; 4];
x = A\u

x =
      10
     -12
       5
```

It can be confirmed that `A*x` is exactly equal to `u`.

If `A` and `b` are square and the same size, `x= A\b` is also that size:

```
b = magic(3);
X = A\b

X =
      19    -3    -1
     -17     4    13
       6     0    -6
```

It can be confirmed that `A*x` is exactly equal to `b`.

Both of these examples have exact, integer solutions. This is because the coefficient matrix was chosen to be `pascal(3)`, which is a full rank matrix (nonsingular).

Singular Coefficient Matrix

A square matrix A is singular if it does not have linearly independent columns. If A is singular, the solution to $Ax = b$ either does not exist, or is not unique. The backslash operator, `A\b`, issues a warning if `A` is nearly singular and raises an error condition if it detects exact singularity.

If A is singular and $Ax = b$ has a solution, you can find a particular solution that is not unique, by typing

```
P = pinv(A)*b
```

`P` is a pseudoinverse of A. If $Ax = b$ does not have an exact solution, `pinv(A)` returns a least-squares solution.

For example:

```
A = [ 1     3     7
     -1     4     4
      1    10    18 ]
```

is singular, as you can verify by typing

```
rank(A)

ans =

     2
```

Since A is not full rank, it has some singular values equal to zero.

Note: For information about using `pinv` to solve systems with rectangular coefficient matrices, see "Pseudoinverses" on page 3-23.

Exact Solutions

For `b =[5;2;12]`, the equation $Ax = b$ has an exact solution, given by

```
pinv(A)*b

ans =
    0.3850
   -0.1103
    0.7066
```

Verify that `pinv(A)*b` is an exact solution by typing

```
A*pinv(A)*b

ans =
    5.0000
    2.0000
```

```
   12.0000
```

Least-Squares Solutions

However, if `b = [3;6;0]`, $Ax = b$ does not have an exact solution. In this case, `pinv(A)*b` returns a least-squares solution. If you type

```
A*pinv(A)*b

ans =
   -1.0000
    4.0000
    2.0000
```

you do not get back the original vector `b`.

You can determine whether $Ax = b$ has an exact solution by finding the row reduced echelon form of the augmented matrix `[A b]`. To do so for this example, enter

```
rref([A b])
ans =
    1.0000         0    2.2857         0
         0    1.0000    1.5714         0
         0         0         0    1.0000
```

Since the bottom row contains all zeros except for the last entry, the equation does not have a solution. In this case, `pinv(A)` returns a least-squares solution.

Overdetermined Systems

This example shows how overdetermined systems are often encountered in various kinds of curve fitting to experimental data.

A quantity, `y`, is measured at several different values of time, `t`, to produce the following observations. You can enter the data and view it in a table with the following statements.

```
t = [0 .3 .8 1.1 1.6 2.3]';
y = [.82 .72 .63 .60 .55 .50]';
B = table(t,y)


B =
```

```
    t       y
   ___    ____

     0    0.82
   0.3    0.72
   0.8    0.63
   1.1     0.6
   1.6    0.55
   2.3     0.5
```

Try modeling the data with a decaying exponential function

$y(t) = c_1 + c_2 e^{-t}$.

The preceding equation says that the vector `y` should be approximated by a linear combination of two other vectors. One is a constant vector containing all ones and the other is the vector with components `exp(-t)`. The unknown coefficients, c_1 and c_2, can be computed by doing a least-squares fit, which minimizes the sum of the squares of the deviations of the data from the model. There are six equations in two unknowns, represented by a 6-by-2 matrix.

```
E = [ones(size(t)) exp(-t)]
```

```
E =

    1.0000    1.0000
    1.0000    0.7408
    1.0000    0.4493
    1.0000    0.3329
    1.0000    0.2019
    1.0000    0.1003
```

Use the backslash operator to get the least-squares solution.

```
c = E\y
```

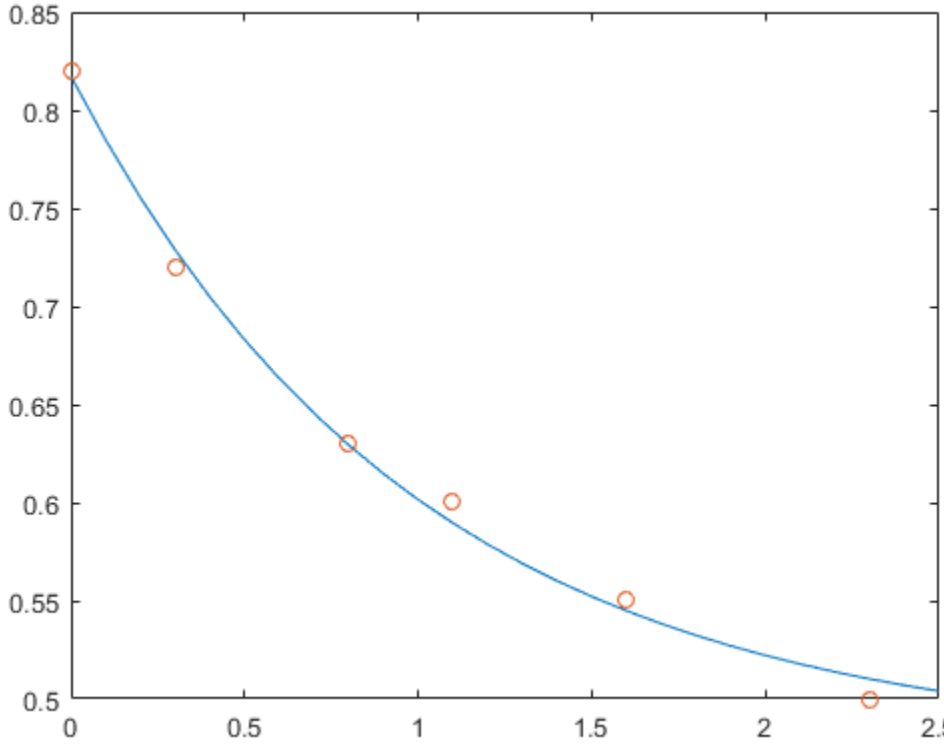
```
c =

    0.4760
    0.3413
```

In other words, the least-squares fit to the data is

$$y(t) = 0.4760 + 0.3413e^{-t}.$$

The following statements evaluate the model at regularly spaced increments in `t`, and then plot the result together with the original data:

```
T = (0:0.1:2.5)';
Y = [ones(size(T)) exp(-T)]*c;
plot(T,Y,'-',t,y,'o')
```



`E*c` is not exactly equal to `y`, but the difference might well be less than measurement errors in the original data.

A rectangular matrix `A` is rank deficient if it does not have linearly independent columns. If `A` is rank deficient, the least-squares solution to `AX = B` is not unique. The backslash operator, `A\B`, issues a warning if `A` is rank deficient and produces a least-squares solution if the system has no solution and a basic solution if the system has infinitely many solutions.

Underdetermined Systems

This example shows how the solution to underdetermined systems is not unique. Underdetermined linear systems involve more unknowns than equations. The matrix left division operation in MATLAB finds a basic solution, which has at most `m` nonzero components for an `m`-by-`n` coefficient matrix.

Here is a small, random example:

```
R = [6 8 7 3; 3 5 4 1]
rng(0);
b = randi(8,2,1)
```

```
R =

     6     8     7     3
     3     5     4     1


b =

     7
     8
```

The linear system `Rp = b` involves two equations in four unknowns. Since the coefficient matrix contains small integers, it is appropriate to use the `format` command to display the solution in rational format. The particular solution is obtained with

```
format rat
p = R\b
```

```
p =

     0
    17/7
```

```
     0
-29/7
```

One of the nonzero components is `p(2)` because `R(:,2)` is the column of `R` with largest norm. The other nonzero component is `p(4)` because `R(:,4)` dominates after `R(:,2)` is eliminated.

The complete general solution to the underdetermined system can be characterized by adding `p` to an arbitrary linear combination of the null space vectors, which can be found using the `null` function with an option requesting a rational basis.

```
Z = null(R,'r')
```

```
Z =

    -1/2    -7/6
    -1/2     1/2
     1       0
     0       1
```

It can be confirmed that `R*Z` is zero and that the residual `R*x - b` is small for any vector `x`, where

```
x = p + Z*q.
```

Since the columns of `Z` are the null space vectors, the product `Z*q` is a linear combination of those vectors:

$$Z * q = \begin{pmatrix} \vec{x}_1 & \vec{x}_2 \end{pmatrix} \begin{pmatrix} u \\ w \end{pmatrix} = u\vec{x}_1 + w\vec{x}_2 \ .$$

To illustrate, choose an arbitrary `q` and construct `x`.

```
q = [-2; 1];
x = p + Z*q;
```

Calculate the norm of the residual.

```
format short
norm(R*x - b)
```

```
ans =
```

```
2.6645e-15
```

Using Multithreaded Computation with Systems of Linear Equations

MATLAB software supports multithreaded computation for a number of linear algebra and element-wise numerical functions. These functions automatically execute on multiple threads. For a function or expression to execute faster on multiple CPUs, a number of conditions must be true:

1. The function performs operations that easily partition into sections that execute concurrently. These sections must be able to execute with little communication between processes. They should require few sequential operations.
2. The data size is large enough so that any advantages of concurrent execution outweigh the time required to partition the data and manage separate execution threads. For example, most functions speed up only when the array contains several thousand elements or more.
3. The operation is not memory-bound; processing time is not dominated by memory access time. As a general rule, complicated functions speed up more than simple functions.

`inv`, `lscov`, `linsolve`, and `mldivide` show significant increase in speed on large double-precision arrays (on order of 10,000 elements or more) when multithreading is enabled.

Iterative Methods for Solving Systems of Linear Equations

If the coefficient matrix A is large and sparse, factorization methods are generally not efficient. *Iterative methods* generate a series of approximate solutions. MATLAB provides several iterative methods to handle large, sparse input matrices.

`pcg`

Preconditioned conjugate gradients method. This method is appropriate for Hermitian positive definite coefficient matrix A.

`bicg`

BiConjugate Gradients Method

`bicgstab`

BiConjugate Gradients Stabilized Method

`bicgstabl`

BiCGStab(l) Method

`cgs`

Conjugate Gradients Squared Method

`gmres`

Generalized Minimum Residual Method

`lsqr`

LSQR Method

`minres`

Minimum Residual Method. This method is appropriate for Hermitian coefficient matrix A.

`qmr`

Quasi-Minimal Residual Method

`symmlq`

Symmetric LQ Method

`tfqmr`

Transpose-Free QMR Method

Inverses and Determinants

Introduction

If A is square and nonsingular, the equations $AX = I$ and $XA = I$ have the same solution, X. This solution is called the inverse of A, is denoted by A^{-1}, and is computed by the function `inv`.

The *determinant* of a matrix is useful in theoretical considerations and some types of symbolic computation, but its scaling and round-off error properties make it far less satisfactory for numeric computation. Nevertheless, the function `det` computes the determinant of a square matrix:

```
A = pascal(3)
```

```
A =
       1     1     1
       1     2     3
       1     3     6
d = det(A)
X = inv(A)

d =
       1

X =
       3    -3     1
      -3     5    -2
       1    -2     1
```

Again, because A is symmetric, has integer elements, and has determinant equal to one, so does its inverse. However,

```
B = magic(3)

B =
       8     1     6
       3     5     7
       4     9     2
d = det(B)
X = inv(B)

d =
     -360

X =
      0.1472   -0.1444    0.0639
     -0.0611    0.0222    0.1056
     -0.0194    0.1889   -0.1028
```

Closer examination of the elements of X, or use of `format rat`, would reveal that they are integers divided by 360.

If A is square and nonsingular, then, without round-off error, `X = inv(A)*B` is theoretically the same as `X = A\B` and `Y = B*inv(A)` is theoretically the same as `Y = B/A`. But the computations involving the backslash and slash operators are preferable because they require less computer time, less memory, and have better error-detection properties.

Pseudoinverses

Rectangular matrices do not have inverses or determinants. At least one of the equations $AX = I$ and $XA = I$ does not have a solution. A partial replacement for the inverse is provided by the *Moore-Penrose pseudoinverse*, which is computed by the `pinv` function:

```
format short
C = fix(10*gallery('uniformdata',[3 2],0));
X = pinv(C)

X =
    0.1159   -0.0729    0.0171
   -0.0534    0.1152    0.0418
```

The matrix

```
Q = X*C

Q =
    1.0000    0.0000
    0.0000    1.0000
```

is the 2-by-2 identity, but the matrix

```
P = C*X

P =
    0.8293   -0.1958    0.3213
   -0.1958    0.7754    0.3685
    0.3213    0.3685    0.3952
```

is not the 3-by-3 identity. However, `P` acts like an identity on a portion of the space in the sense that `P` is symmetric, `P*C` is equal to `C`, and `X*P` is equal to `X`.

Solving a Rank-Deficient System

If `A` is m-by-n with $m > n$ and full rank n, each of the three statements

```
x = A\b
x = pinv(A)*b
x = inv(A'*A)*A'*b
```

theoretically computes the same least-squares solution `x`, although the backslash operator does it faster.

However, if A does not have full rank, the solution to the least-squares problem is not unique. There are many vectors x that minimize

```
norm(A*x -b)
```

The solution computed by `x = A\b` is a basic solution; it has at most *r* nonzero components, where *r* is the rank of A. The solution computed by `x = pinv(A)*b` is the minimal norm solution because it minimizes `norm(x)`. An attempt to compute a solution with `x = inv(A'*A)*A'*b` fails because `A'*A` is singular.

Here is an example that illustrates the various solutions:

```
A = [ 1  2  3
      4  5  6
      7  8  9
     10 11 12 ];
```

does not have full rank. Its second column is the average of the first and third columns. If

```
b = A(:,2)
```

is the second column, then an obvious solution to `A*x = b` is `x = [0 1 0]'`. But none of the approaches computes that x. The backslash operator gives

```
x = A\b

Warning: Rank deficient, rank = 2, tol = 1.4594e-014.
x =
      0.5000
      0
      0.5000
```

This solution has two nonzero components. The pseudoinverse approach gives

```
y = pinv(A)*b

y =
      0.3333
      0.3333
      0.3333
```

There is no warning about rank deficiency. But `norm(y) = 0.5774` is less than `norm(x) = 0.7071`. Finally,

```
z = inv(A'*A)*A'*b
```

fails completely:

```
Warning: Matrix is close to singular or badly scaled.
         Results may be inaccurate. RCOND = 9.868649e-018.
z =
   -0.8594
    1.3438
   -0.6875
```

Factorizations

Introduction

All three of the matrix factorizations discussed in this section make use of *triangular* matrices, where all the elements either above or below the diagonal are zero. Systems of linear equations involving triangular matrices are easily and quickly solved using either *forward* or *back substitution*.

Cholesky Factorization

The Cholesky factorization expresses a symmetric matrix as the product of a triangular matrix and its transpose
$A = R'R$,

where R is an upper triangular matrix.

Not all symmetric matrices can be factored in this way; the matrices that have such a factorization are said to be positive definite. This implies that all the diagonal elements of A are positive and that the offdiagonal elements are "not too big." The Pascal matrices provide an interesting example. Throughout this chapter, the example matrix A has been the 3-by-3 Pascal matrix. Temporarily switch to the 6-by-6:

```
A = pascal(6)
```

```
A =
     1     1     1     1     1     1
     1     2     3     4     5     6
     1     3     6    10    15    21
     1     4    10    20    35    56
     1     5    15    35    70   126
     1     6    21    56   126   252
```

The elements of A are binomial coefficients. Each element is the sum of its north and west neighbors. The Cholesky factorization is

```
R = chol(A)

R =
     1     1     1     1     1     1
     0     1     2     3     4     5
     0     0     1     3     6    10
     0     0     0     1     4    10
     0     0     0     0     1     5
     0     0     0     0     0     1
```

The elements are again binomial coefficients. The fact that `R'*R` is equal to A demonstrates an identity involving sums of products of binomial coefficients.

Note: The Cholesky factorization also applies to complex matrices. Any complex matrix that has a Cholesky factorization satisfies $A' = A$ and is said to be *Hermitian positive definite*.

The Cholesky factorization allows the linear system
$Ax = b$

to be replaced by
$R'Rx = b$.

Because the backslash operator recognizes triangular systems, this can be solved in the MATLAB environment quickly with

```
x = R\(R'\b)
```

If A is n-by-n, the computational complexity of `chol(A)` is $O(n^3)$, but the complexity of the subsequent backslash solutions is only $O(n^2)$.

LU Factorization

LU factorization, or Gaussian elimination, expresses any square matrix A as the product of a permutation of a lower triangular matrix and an upper triangular matrix
$A = LU$,

where L is a permutation of a lower triangular matrix with ones on its diagonal and U is an upper triangular matrix.

The permutations are necessary for both theoretical and computational reasons. The matrix

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

cannot be expressed as the product of triangular matrices without interchanging its two rows. Although the matrix

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

can be expressed as the product of triangular matrices, when ε is small, the elements in the factors are large and magnify errors, so even though the permutations are not strictly necessary, they are desirable. Partial pivoting ensures that the elements of L are bounded by one in magnitude and that the elements of U are not much larger than those of A.

For example:

```
[L,U] = lu(B)

L =
    1.0000         0         0
    0.3750    0.5441    1.0000
    0.5000    1.0000         0

U =
    8.0000    1.0000    6.0000
         0    8.5000   -1.0000
```

```
         0         0    5.2941
```

The LU factorization of A allows the linear system

```
A*x = b
```

to be solved quickly with

```
x = U\(L\b)
```

Determinants and inverses are computed from the LU factorization using

```
det(A) = det(L)*det(U)
```

and

```
inv(A) = inv(U)*inv(L)
```

You can also compute the determinants using `det(A) = prod(diag(U))`, though the signs of the determinants might be reversed.

QR Factorization

An *orthogonal* matrix, or a matrix with orthonormal columns, is a real matrix whose columns all have unit length and are perpendicular to each other. If Q is orthogonal, then $Q'Q = 1$.

The simplest orthogonal matrices are two-dimensional coordinate rotations:

$$\begin{bmatrix} \cos(\theta) & \sin(\theta) \\ -\sin(\theta) & \cos(\theta) \end{bmatrix}.$$

For complex matrices, the corresponding term is *unitary*. Orthogonal and unitary matrices are desirable for numerical computation because they preserve length, preserve angles, and do not magnify errors.

The orthogonal, or QR, factorization expresses any rectangular matrix as the product of an orthogonal or unitary matrix and an upper triangular matrix. A column permutation might also be involved:
$A = QR$

or
$AP = QR$,

where Q is orthogonal or unitary, R is upper triangular, and P is a permutation.

There are four variants of the QR factorization—full or economy size, and with or without column permutation.

Overdetermined linear systems involve a rectangular matrix with more rows than columns, that is m-by-n with $m > n$. The full-size QR factorization produces a square, m-by-m orthogonal Q and a rectangular m-by-n upper triangular R:

```
C=gallery('uniformdata',[5 4], 0);
[Q,R] = qr(C)

Q =

    0.6191    0.1406   -0.1899   -0.5058    0.5522
    0.1506    0.4084    0.5034    0.5974    0.4475
    0.3954   -0.5564    0.6869   -0.1478   -0.2008
    0.3167    0.6676    0.1351   -0.1729   -0.6370
    0.5808   -0.2410   -0.4695    0.5792   -0.2207


R =

    1.5346    1.0663    1.2010    1.4036
         0    0.7245    0.3474   -0.0126
         0         0    0.9320    0.6596
         0         0         0    0.6648
         0         0         0         0
```

In many cases, the last $m - n$ columns of Q are not needed because they are multiplied by the zeros in the bottom portion of R. So the economy-size QR factorization produces a rectangular, m-by-nQ with orthonormal columns and a square n-by-n upper triangular R. For the 5-by-4 example, this is not much of a saving, but for larger, highly rectangular matrices, the savings in both time and memory can be quite important:

```
[Q,R] = qr(C,0)
Q =

    0.6191    0.1406   -0.1899   -0.5058
    0.1506    0.4084    0.5034    0.5974
```

```
    0.3954   -0.5564    0.6869   -0.1478
    0.3167    0.6676    0.1351   -0.1729
    0.5808   -0.2410   -0.4695    0.5792


R =

    1.5346    1.0663    1.2010    1.4036
         0    0.7245    0.3474   -0.0126
         0         0    0.9320    0.6596
         0         0         0    0.6648
```

In contrast to the LU factorization, the QR factorization does not require any pivoting or permutations. But an optional column permutation, triggered by the presence of a third output argument, is useful for detecting singularity or rank deficiency. At each step of the factorization, the column of the remaining unfactored matrix with largest norm is used as the basis for that step. This ensures that the diagonal elements of R occur in decreasing order and that any linear dependence among the columns is almost certainly be revealed by examining these elements. For the small example given here, the second column of `C` has a larger norm than the first, so the two columns are exchanged:

```
[Q,R,P] = qr(C)

Q =
   -0.3522    0.8398   -0.4131
   -0.7044   -0.5285   -0.4739
   -0.6163    0.1241    0.7777

R =
  -11.3578   -8.2762
         0    7.2460
         0         0

P =
     0     1
     1     0
```

When the economy-size and column permutations are combined, the third output argument is a permutation vector, rather than a permutation matrix:

```
[Q,R,p] = qr(C,0)

Q =
   -0.3522    0.8398
```

```
    -0.7044   -0.5285
    -0.6163    0.1241

R =
   -11.3578   -8.2762
         0    7.2460


p =
     2     1
```

The QR factorization transforms an overdetermined linear system into an equivalent triangular system. The expression

```
norm(A*x - b)
```

equals

```
norm(Q*R*x - b)
```

Multiplication by orthogonal matrices preserves the Euclidean norm, so this expression is also equal to

```
norm(R*x - y)
```

where `y = Q'*b`. Since the last m-n rows of R are zero, this expression breaks into two pieces:

```
norm(R(1:n,1:n)*x - y(1:n))
```

and

```
norm(y(n+1:m))
```

When `A` has full rank, it is possible to solve for `x` so that the first of these expressions is zero. Then the second expression gives the norm of the residual. When `A` does not have full rank, the triangular structure of `R` makes it possible to find a basic solution to the least-squares problem.

Using Multithreaded Computation for Factorization

MATLAB software supports multithreaded computation for a number of linear algebra and element-wise numerical functions. These functions automatically execute on

multiple threads. For a function or expression to execute faster on multiple CPUs, a number of conditions must be true:

1. The function performs operations that easily partition into sections that execute concurrently. These sections must be able to execute with little communication between processes. They should require few sequential operations.
2. The data size is large enough so that any advantages of concurrent execution outweigh the time required to partition the data and manage separate execution threads. For example, most functions speed up only when the array contains several thousand elements or more.
3. The operation is not memory-bound; processing time is not dominated by memory access time. As a general rule, complex functions speed up more than simple functions.

`lu` and `qr` show significant increase in speed on large double-precision arrays (on order of 10,000 elements).

Powers and Exponentials

Positive Integer Powers

If `A` is a square matrix and `p` is a positive integer, `A^p` effectively multiplies `A` by itself `p-1` times. For example:

```
A = [1 1 1;1 2 3;1 3 6]

A =

     1     1     1
     1     2     3
     1     3     6

X = A^2

X =
```

```
     3      6     10
     6     14     25
    10     25     46
```

Inverse and Fractional Powers

If `A` is square and nonsingular, `A^(-p)` effectively multiplies `inv(A)` by itself `p-1` times:

```
Y = A^(-3)

Y =

  145.0000 -207.0000   81.0000
 -207.0000  298.0000 -117.0000
   81.0000 -117.0000   46.0000
```

Fractional powers, like `A^(2/3)`, are also permitted; the results depend upon the distribution of the eigenvalues of the matrix.

Element-by-Element Powers

The `.^` operator produces element-by-element powers. For example:

```
X = A.^2

A =
     1     1     1
     1     4     9
     1     9    36
```

Exponentials

The function

```
sqrtm(A)
```

computes `A^(1/2)` by a more accurate algorithm. The `m` in `sqrtm` distinguishes this function from `sqrt(A)`, which, like `A.^(1/2)`, does its job element-by-element.

A system of linear, constant coefficient, ordinary differential equations can be written

$$dx/dt = Ax,$$

where $x = x(t)$ is a vector of functions of `t` and `A` is a matrix independent of `t`. The solution can be expressed in terms of the matrix exponential

$$x(t) = e^{tA}x(0).$$

The function

```
expm(A)
```

computes the matrix exponential. An example is provided by the 3-by-3 coefficient matrix,

```
A = [0 -6 -1; 6 2 -16; -5 20 -10]
```

```
A =

     0    -6    -1
     6     2   -16
    -5    20   -10
```

and the initial condition, *x(0)*.

```
x0 = [1 1 1]'
```

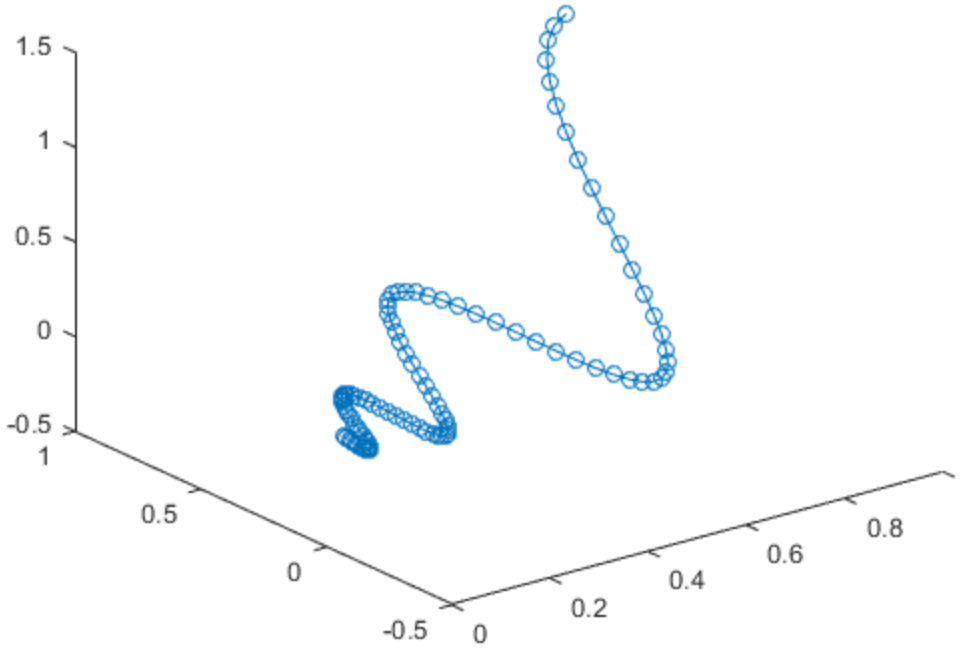
```
x0 =

     1
     1
     1
```

The matrix exponential is used to compute the solution, *x(t)*, to the differential equation at 101 points on the interval $0 \le t \le 1$.

```
X = [];
for t = 0:.01:1
   X = [X expm(t*A)*x0];
end
```

A three-dimensional phase plane plot shows the solution spiraling in towards the origin. This behavior is related to the eigenvalues of the coefficient matrix.

```
plot3(X(1,:),X(2,:),X(3,:),'-o')
```



Eigenvalues

Eigenvalue Decomposition

An *eigenvalue* and *eigenvector* of a square matrix A are, respectively, a scalar λ and a nonzero vector v that satisfy

$Av = \lambda v$.

With the eigenvalues on the diagonal of a diagonal matrix Λ and the corresponding eigenvectors forming the columns of a matrix V, you have
$AV = V\Lambda$.

If V is nonsingular, this becomes the eigenvalue decomposition
$A = V\Lambda V^{-1}$.

A good example is provided by the coefficient matrix of the ordinary differential equation of the previous section:

```
A =
     0    -6    -1
     6     2   -16
    -5    20   -10
```

The statement

```
lambda = eig(A)
```

produces a column vector containing the eigenvalues. For this matrix, the eigenvalues are complex:

```
lambda =
     -3.0710
     -2.4645+17.6008i
     -2.4645-17.6008i
```

The real part of each of the eigenvalues is negative, so $e^{\lambda t}$ approaches zero as t increases. The nonzero imaginary part of two of the eigenvalues, $\pm\omega$, contributes the oscillatory component, $\sin(\omega t)$, to the solution of the differential equation.

With two output arguments, `eig` computes the eigenvectors and stores the eigenvalues in a diagonal matrix:

```
[V,D] = eig(A)

V =
  -0.8326          0.2003 - 0.1394i   0.2003 + 0.1394i
  -0.3553         -0.2110 - 0.6447i  -0.2110 + 0.6447i
  -0.4248         -0.6930            -0.6930

D =
  -3.0710                  0                  0
```

```
         0            -2.4645+17.6008i         0
         0                  0            -2.4645-17.6008i
```

The first eigenvector is real and the other two vectors are complex conjugates of each other. All three vectors are normalized to have Euclidean length, `norm(v,2)`, equal to one.

The matrix `V*D*inv(V)`, which can be written more succinctly as `V*D/V`, is within round-off error of `A`. And, `inv(V)*A*V`, or `V\A*V`, is within round-off error of `D`.

Multiple Eigenvalues

Some matrices do not have an eigenvector decomposition. These matrices are not diagonalizable. For example:

```
A = [ 1    -2    1
      0     1    4
      0     0    3 ]
```

For this matrix

```
[V,D] = eig(A)
```

produces

```
V =

    1.0000    1.0000   -0.5571
         0    0.0000    0.7428
         0         0    0.3714


D =

     1     0     0
     0     1     0
     0     0     3
```

There is a double eigenvalue at $\lambda = 1$. The first and second columns of `V` are the same. For this matrix, a full set of linearly independent eigenvectors does not exist.

Schur Decomposition

The MATLAB advanced matrix computations do not require eigenvalue decompositions. They are based, instead, on the Schur decomposition

$A = USU^T$.

where U is an orthogonal matrix and S is a block upper triangular matrix with 1-by-1 and 2-by-2 blocks on the diagonal. The eigenvalues are revealed by the diagonal elements and blocks of S, while the columns of U provide a basis with much better numerical properties than a set of eigenvectors. The Schur decomposition of this defective example is

```
[U,S] = schur(A)

U =
   -0.4741    0.6648    0.5774
    0.8127    0.0782    0.5774
   -0.3386   -0.7430    0.5774

S =
   -1.0000   20.7846   -44.6948
         0    1.0000    -0.6096
         0         0     1.0000
```

The double eigenvalue is contained in the lower 2-by-2 block of `S`.

Note: If `A` is complex, `schur` returns the complex Schur form, which is upper triangular with the eigenvalues of `A` on the diagonal.

Singular Values

A *singular value* and corresponding *singular vectors* of a rectangular matrix A are, respectively, a scalar σ and a pair of vectors u and v that satisfy

$$Av = \sigma u$$
$$A^H u = \sigma v,$$

where A^H is the Hermitian transpose of A. The singular vectors u and v are typically scaled to have a norm of 1. Also, if u and v are singular vectors of A, then $-u$ and $-v$ are singular vectors of `A` as well.

The singular values σ are always real and nonnegative, even if A is complex. With the singular values on the diagonal of a diagonal matrix Σ and the corresponding

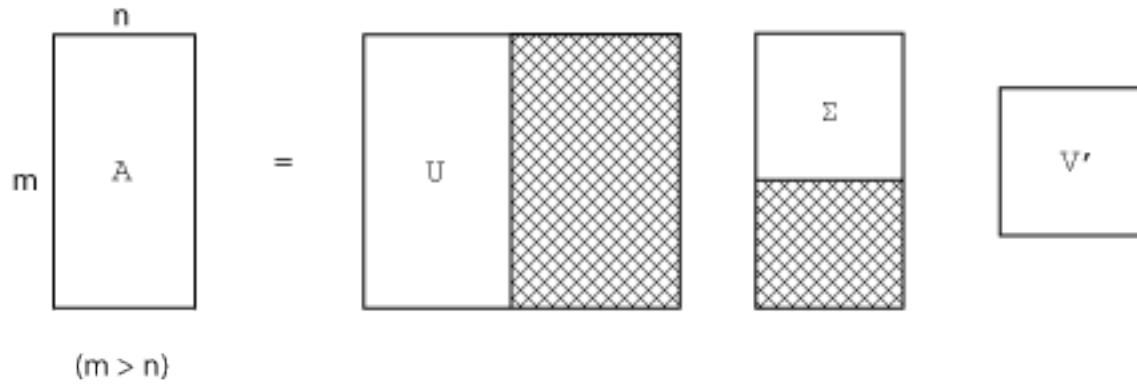
singular vectors forming the columns of two orthogonal matrices U and V, you obtain the equations

$$AV = U\Sigma$$
$$A^H U = V\Sigma.$$

Since U and V are unitary matrices, multiplying the first equation by V^H on the right yields the singular value decomposition equation

$$A = U\Sigma V^H.$$

The full singular value decomposition of an m-by-n matrix involves an m-by-m U, an m-by-n Σ, and an n-by-n V. In other words, U and V are both square, and Σ is the same size as A. If A has many more rows than columns, the resulting U can be quite large, but most of its columns are multiplied by zeros in Σ. In this situation, the *economy-sized* decomposition saves both time and storage by producing an m-by-n U, an n-by-n Σ and the same V:



The eigenvalue decomposition is the appropriate tool for analyzing a matrix when it represents a mapping from a vector space into itself, as it does for an ordinary differential equation. However, the singular value decomposition is the appropriate tool for analyzing a mapping from one vector space into another vector space, possibly with a different dimension. Most systems of simultaneous linear equations fall into this second category.

If A is square, symmetric, and positive definite, then its eigenvalue and singular value decompositions are the same. But, as A departs from symmetry and positive definiteness, the difference between the two decompositions increases. In particular, the singular value decomposition of a real matrix is always real, but the eigenvalue decomposition of a real, nonsymmetric matrix might be complex.

For the example matrix

```
A =
     9     4
     6     8
     2     7
```

the full singular value decomposition is

```
[U,S,V] = svd(A)

U =

    0.6105   -0.7174    0.3355
    0.6646    0.2336   -0.7098
    0.4308    0.6563    0.6194


S =

   14.9359         0
         0    5.1883
         0         0


V =

    0.6925   -0.7214
    0.7214    0.6925
```

You can verify that `U*S*V'` is equal to `A` to within round-off error. For this small problem, the economy size decomposition is only slightly smaller.

```
[U,S,V] = svd(A,0)

U =

    0.6105   -0.7174
```

```
    0.6646    0.2336
    0.4308    0.6563


S =

   14.9359         0
         0    5.1883


V =

    0.6925   -0.7214
    0.7214    0.6925
```

Again, `U*S*V'` is equal to `A` to within round-off error.

If the matrix `A` is large and sparse, then using `svd` to calculate *all* of the singular values and vectors is not always practical. For example, if you need to know just a few of the largest singular values, then calculating all of the singular values of a 5000-by-5000 sparse matrix is a lot of extra work. In cases where only a subset of the singular values and vectors are required, the `svds` function is preferred over `svd`.

For a 1000-by-1000 random sparse matrix with a density of about 30%,

```
n = 1000;
A = sprand(n,n,0.3);
```

the six largest singular values are

```
S = svds(A)

S =

  130.2184
   16.4358
   16.4119
   16.3688
   16.3242
   16.2838
```

Also, the six smallest singular values are

```
S = svds(A,6,'smallest')
```

```
S =

    0.0740
    0.0574
    0.0388
    0.0282
    0.0131
    0.0066
```

For smaller matrices that can fit in memory as a full matrix, `full(A)`, using `svd(full(A))` might still be quicker than `svds`. However, for truly large and sparse matrices, using `svds` becomes necessary.

Operations on Nonlinear Functions

In this section...

Function Handles

You can create a handle to any MATLAB function, and then use that handle as a means of referencing the function. A function handle is typically passed in an argument list to other functions, which can then execute, or *evaluate*, the function using the handle.

Construct a function handle in MATLAB using the *at* sign, `@`, before the function name. The following example creates a function handle for the `sin` function and assigns it to the variable `fhandle`:

```
fhandle = @sin;
```

You can call a function by means of its handle in the same way that you would call the function using its name. The syntax is

```
fhandle(arg1, arg2, ...);
```

The function `plot_fhandle`, shown below, receives a function handle and data, generates *y*-axis data using the function handle, and plots it:

```
function plot_fhandle(fhandle, data)
plot(data, fhandle(data))
```

When you call `plot_fhandle` with a handle to the `sin` function and the argument shown below, the resulting evaluation produces a sine wave plot:

```
plot_fhandle(@sin, -pi:0.01:pi)
```

Function Functions

A class of functions called “function functions” works with nonlinear functions of a scalar variable. That is, one function works on another function. The function functions include

- Zero finding

- Optimization
- Quadrature
- Ordinary differential equations

MATLAB represents the nonlinear function by the file that defines it. For example, here is a simplified version of the function `humps` from the `matlab/demos` folder:

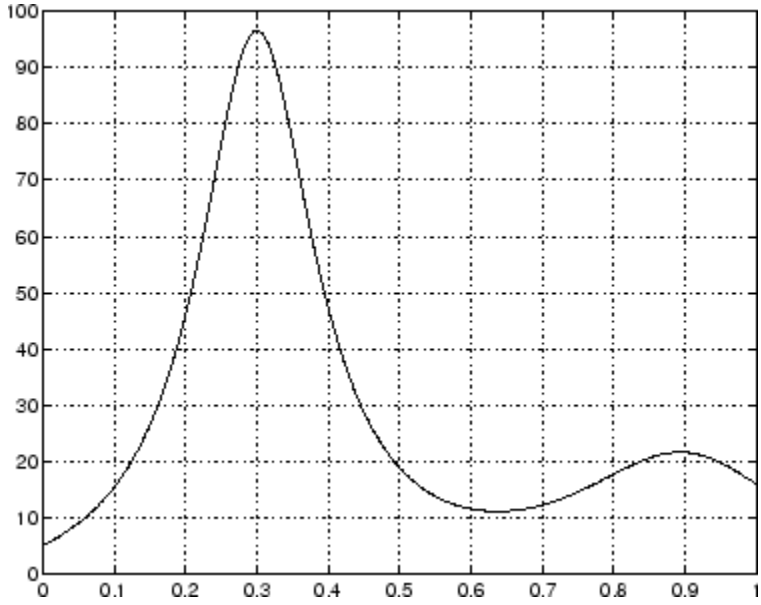
```
function y = humps(x)
y = 1./((x-.3).^2 + .01) + 1./((x-.9).^2 + .04) - 6;
```

Evaluate this function at a set of points in the interval $0 \le x \le 1$ with

```
x = 0:.002:1;
y = humps(x);
```

Then plot the function with

```
plot(x,y)
```



The graph shows that the function has a local minimum near $x = 0.6$. The function `fminsearch` finds the *minimizer*, the value of x where the function takes on this

minimum. The first argument to `fminsearch` is a function handle to the function being minimized and the second argument is a rough guess at the location of the minimum:

```
p = fminsearch(@humps,.5)
p =
    0.6370
```

To evaluate the function at the minimizer,

```
humps(p)

ans =
   11.2528
```

Numerical analysts use the terms *quadrature* and *integration* to distinguish between numerical approximation of definite integrals and numerical integration of ordinary differential equations. MATLAB quadrature routines are `quad` and `quadl`. The statement

```
Q = quadl(@humps,0,1)
```

computes the area under the curve in the graph and produces

```
Q =
   29.8583
```

Finally, the graph shows that the function is never zero on this interval. So, if you search for a zero with

```
z = fzero(@humps,.5)
```

you will find one outside the interval

```
z =
   -0.1316
```

Multivariate Data

MATLAB uses column-oriented analysis for multivariate statistical data. Each column in a data set represents a variable and each row an observation. The `(i,j)`th element is the `i`th observation of the `j`th variable.

As an example, consider a data set with three variables:

- Heart rate
- Weight
- Hours of exercise per week

For five observations, the resulting array might look like

```
D = [ 72          134          3.2
      81          201          3.5
      69          156          7.1
      82          148          2.4
      75          170          1.2 ]
```

The first row contains the heart rate, weight, and exercise hours for patient 1, the second row contains the data for patient 2, and so on. Now you can apply many MATLAB data analysis functions to this data set. For example, to obtain the mean and standard deviation of each column, use

```
mu = mean(D), sigma = std(D)

mu =
      75.8       161.8         3.48

sigma =
    5.6303      25.499       2.2107
```

For a list of the data analysis functions available in MATLAB, type

```
help datafun
```

If you have access to the Statistics and Machine Learning Toolbox™ software, type

```
help stats
```

Data Analysis

Introduction

Every data analysis has some standard components:

- Preprocessing — Consider outliers and missing values, and smooth data to identify possible models.
- Summarizing — Compute basic statistics to describe the overall location, scale, and shape of the data.
- Visualizing — Plot data to identify patterns and trends.
- Modeling — Give data trends fuller descriptions, suitable for predicting new values.

Data analysis moves among these components with two basic goals in mind:

1. Describe the patterns in the data with simple models that lead to accurate predictions.
2. Understand the relationships among variables that lead to the model.

This section explains how to carry out a basic data analysis in the MATLAB environment.

Preprocessing Data

This example shows how to preprocess data for analysis.

Overview

Begin a data analysis by loading data into suitable MATLAB® container variables and sorting out the "good" data from the "bad." This is a preliminary step that assures meaningful conclusions in subsequent parts of the analysis.

Loading the Data

Begin by loading the data in `count.dat`:

```
load count.dat
```

The 24-by-3 array `count` contains hourly traffic counts (the rows) at three intersections (the columns) for a single day.

Missing Data

The MATLAB NaN (Not a Number) value is normally used to represent missing data. NaN values allow variables with missing data to maintain their structure - in this case, 24-by-1 vectors with consistent indexing across all three intersections.

Check the data at the third intersection for NaN values using the isnan function:

```
c3 = count(:,3); % Data at intersection 3
c3NaNCount = sum(isnan(c3))
```

```
c3NaNCount =

     0
```

isnan returns a logical vector the same size as c3, with entries indicating the presence (1) or absence (0) of NaN values for each of the 24 elements in the data. In this case, the logical values sum to 0, so there are no NaN values in the data.

NaN values are introduced into the data in the section on Outliers.

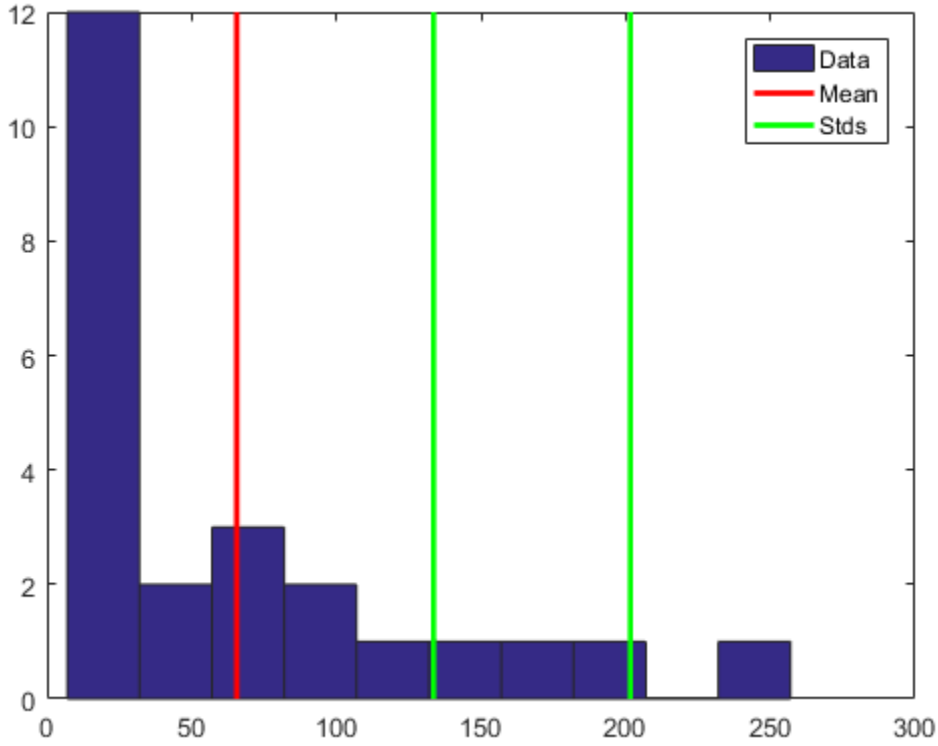
Outliers

Outliers are data values that are dramatically different from patterns in the rest of the data. They might be due to measurement error, or they might represent significant features in the data. Identifying outliers, and deciding what to do with them, depends on an understanding of the data and its source.

One common method for identifying outliers is to look for values more than a certain number of standard deviations σ from the mean μ. The following code plots a histogram of the data at the third intersection together with lines at μ and $\mu + \eta$, for $\eta = 1, 2$:

```
bin_counts = hist(c3); % Histogram bin counts
N = max(bin_counts); % Maximum bin count
mu3 = mean(c3); % Data mean
sigma3 = std(c3); % Data standard deviation

hist(c3) % Plot histogram
hold on
plot([mu3 mu3],[0 N],'r','LineWidth',2) % Mean
X = repmat(mu3+(1:2)*sigma3,2,1);
Y = repmat([0;N],1,2);
plot(X,Y,'g','LineWidth',2) % Standard deviations
```

```
legend('Data','Mean','Stds')
hold off
```



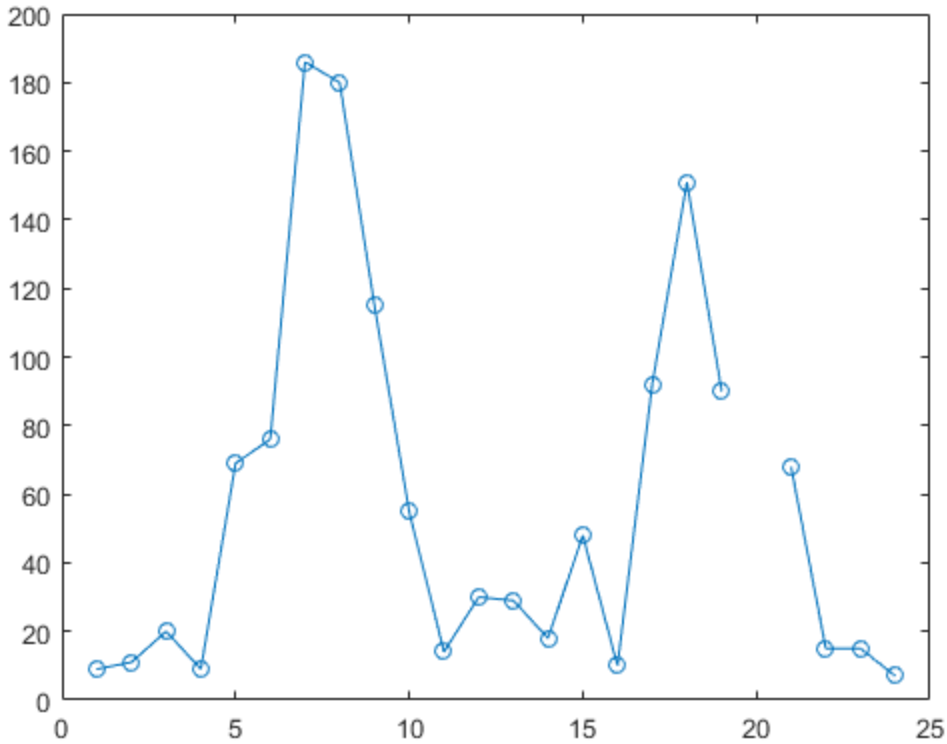
The plot shows that some of the data are more than two standard deviations above the mean. If you identify these data as errors (not features), replace them with `NaN` values as follows:

```
outliers = (c3 - mu3) > 2*sigma3;
c3m = c3; % Copy c3 to c3m
c3m(outliers) = NaN; % Add NaN values
```

Smoothing and Filtering

A time-series plot of the data at the third intersection (with the outlier removed in Outliers) results in the following plot:

```
plot(c3m,'o-')
hold on
```



The `NaN` value at hour 20 appears as a gap in the plot. This handling of `NaN` values is typical of MATLAB plotting functions.

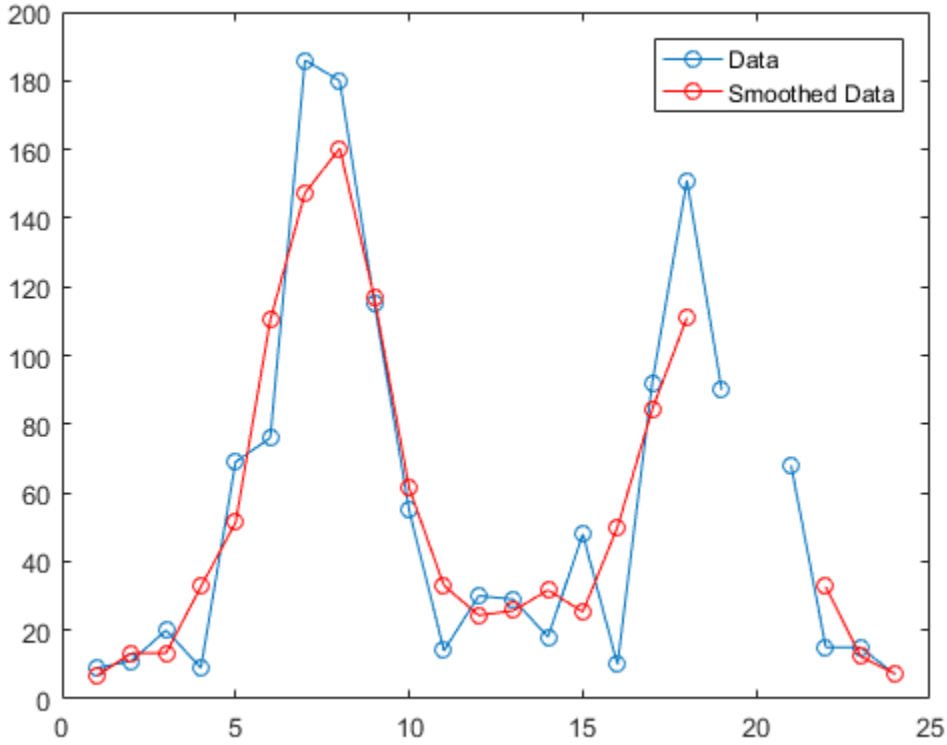
Noisy data shows random variations about expected values. You might want to smooth the data to reveal its main features before building a model. Two basic assumptions underlie smoothing:

- The relationship between the predictor (time) and the response (traffic volume) is smooth.

- The smoothing algorithm results in values that are better estimates of expected values because the noise has been reduced.

Apply a simple moving average smoother to the data using the MATLAB `convn` function:

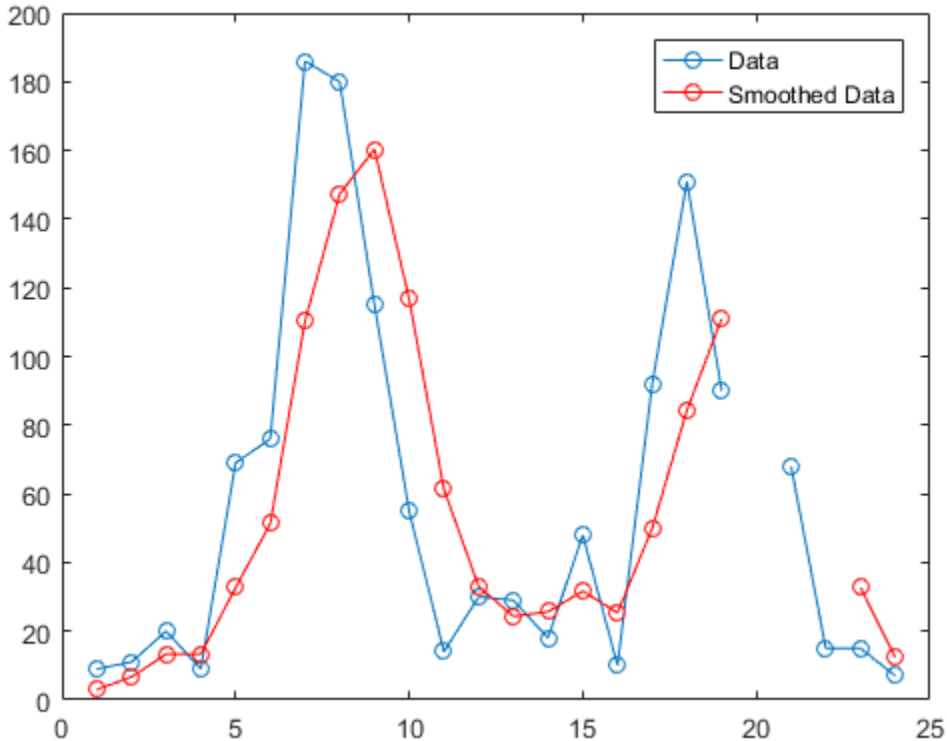
```
span = 3; % Size of the averaging window
window = ones(span,1)/span;
smoothed_c3m = convn(c3m,window,'same');

h = plot(smoothed_c3m,'ro-');
legend('Data','Smoothed Data')
```



The extent of the smoothing is controlled with the variable `span`. The averaging calculation returns `NaN` values whenever the smoothing window includes the `NaN` value in the data, thus increasing the size of the gap in the smoothed data.

The `filter` function is also used for smoothing data:

```
smoothed2_c3m = filter(window,1,c3m);

delete(h)
plot(smoothed2_c3m,'ro-');
```



The smoothed data are shifted from the previous plot. `convn` with the `'same'` parameter returns the central part of the convolution, the same length as the data. `filter` returns

the initial part of the convolution, the same length as the data. Otherwise, the algorithms are identical.

Smoothing estimates the center of the distribution of response values at each value of the predictor. It invalidates a basic assumption of many fitting algorithms, namely, that *the errors at each value of the predictor are independent*. Accordingly, you can use smoothed data to *identify* a model, but avoid using smoothed data to *fit* a model.

Summarizing Data

This example shows how to summarize data.

Overview

Many MATLAB® functions enable you to summarize the overall location, scale, and shape of a data sample.

One of the advantages of working in MATLAB® is that functions operate on entire arrays of data, not just on single scalar values. The functions are said to be *vectorized*. Vectorization allows for both efficient problem formulation, using array-based data, and efficient computation, using vectorized statistical functions.

Measures of Location

Summarize the location of a data sample by finding a "typical" value. Common measures of location or "central tendency" are computed by the functions `mean`, `median`, and `mode`:

```
load count.dat
x1 = mean(count)
x2 = median(count)
x3 = mode(count)
```

```
x1 =

   32.0000   46.5417   65.5833


x2 =

   23.5000   36.0000   39.0000


x3 =
```

```
    11     9     9
```

Like all of its statistical functions, the MATLAB® functions above summarize data across observations (rows) while preserving variables (columns). The functions compute the location of the data at each of the three intersections in a single call.

Measures of Scale

There are many ways to measure the scale or "dispersion" of a data sample. The MATLAB® functions `max`, `min`, `std`, and `var` compute some common measures:

```
dx1 = max(count)-min(count)
dx2 = std(count)
dx3 = var(count)
```

```
dx1 =

   107   136   250


dx2 =

   25.3703   41.4057   68.0281


dx3 =

   1.0e+03 *

    0.6437    1.7144    4.6278
```

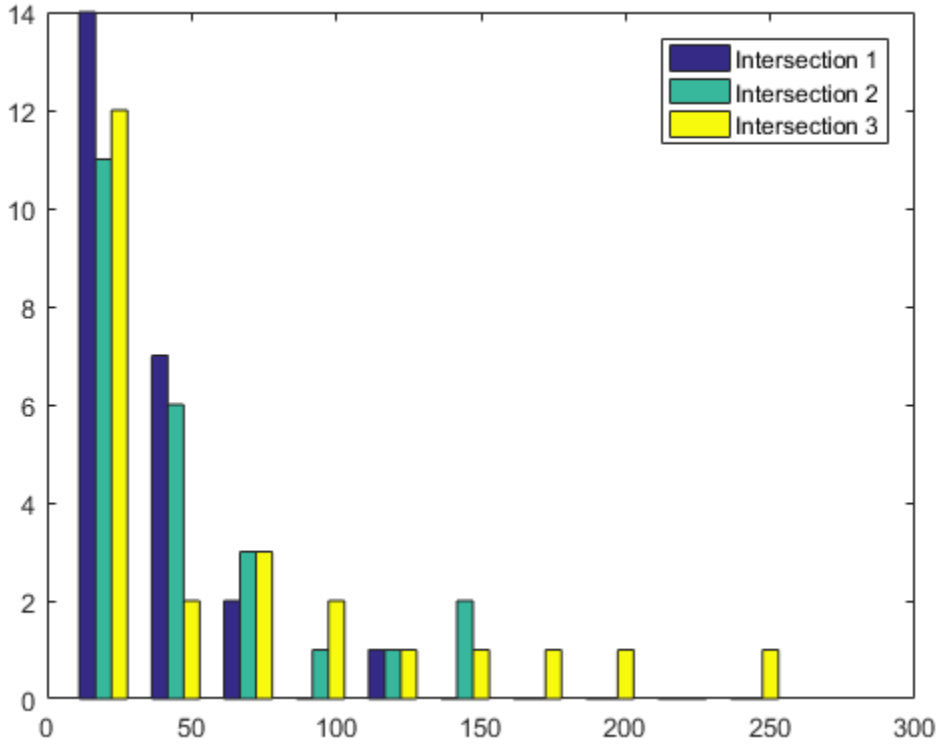
Like all of its statistical functions, the MATLAB® functions above summarize data across observations (rows) while preserving variables (columns). The functions compute the scale of the data at each of the three intersections in a single call.

Shape of a Distribution

The shape of a distribution is harder to summarize than its location or scale. The MATLAB® `hist` function plots a histogram that provides a visual summary:

```
figure
```

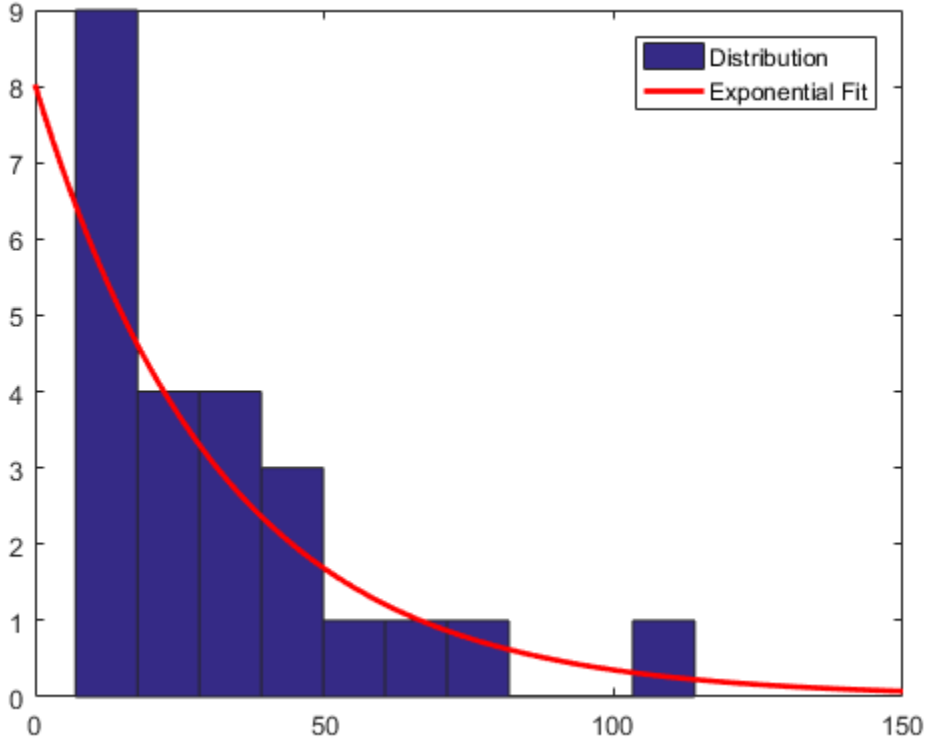
```
hist(count)
legend('Intersection 1',...
       'Intersection 2',...
       'Intersection 3')
```



Parametric models give analytic summaries of distribution shapes. Exponential distributions, with parameter `mu` given by the data mean, are a good choice for the traffic data:

```
c1 = count(:,1); % Data at intersection 1
[bin_counts,bin_locations] = hist(c1);
bin_width = bin_locations(2) - bin_locations(1);
hist_area = (bin_width)*(sum(bin_counts));
```

```
figure
hist(c1)
hold on

mu1 = mean(c1);
exp_pdf = @(t)(1/mu1)*exp(-t/mu1); % Integrates
                                   % to 1
t = 0:150;
y = exp_pdf(t);
plot(t,(hist_area)*y,'r','LineWidth',2)
legend('Distribution','Exponential Fit')
```



Methods for fitting general parametric models to data distributions are beyond the scope of this section. Statistics and Machine Learning Toolbox™ software provides functions for computing maximum likelihood estimates of distribution parameters.

Visualizing Data

Overview

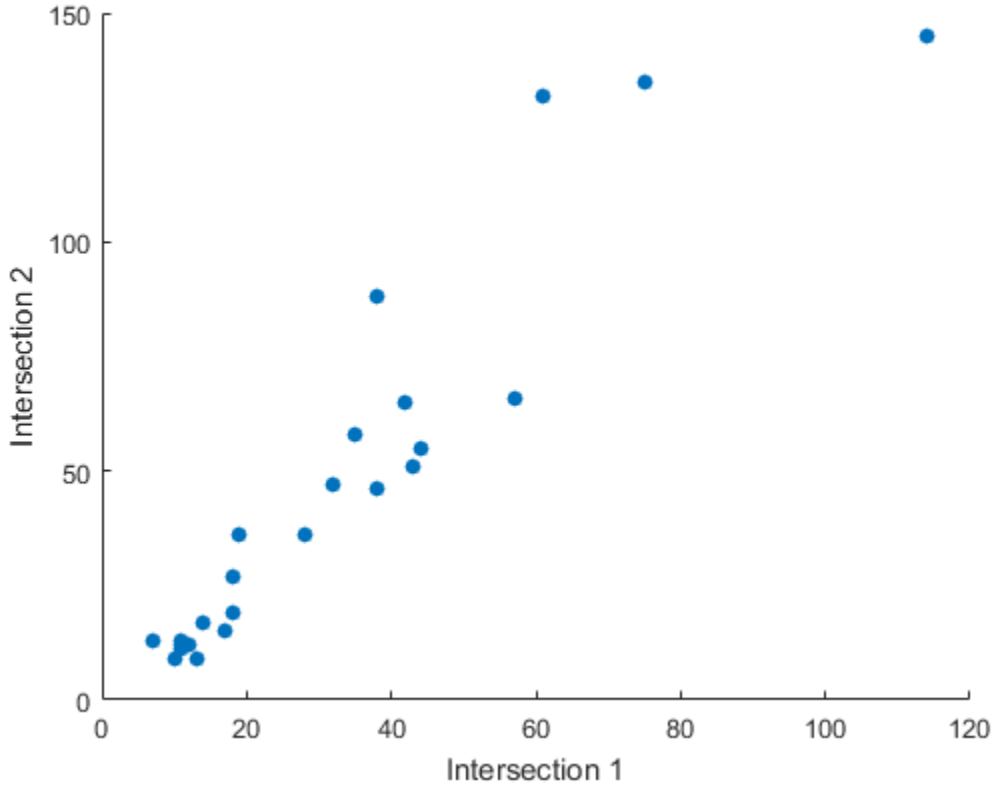
You can use many MATLAB graph types for visualizing data patterns and trends. Scatter plots, described in this section, help to visualize relationships among the traffic data at different intersections. Data exploration tools let you query and interact with individual data points on graphs.

Note: This section continues the data analysis from "Summarizing Data" on page 3-53.

2-D Scatter Plots

A two-dimensional scatter plot, created with the `scatter` function, shows the relationship between the traffic volume at the first two intersections:

```
load count.dat
c1 = count(:,1); % Data at intersection 1
c2 = count(:,2); % Data at intersection 2

figure
scatter(c1,c2,'filled')
xlabel('Intersection 1')
ylabel('Intersection 2')
```



The *covariance*, computed by the `cov` function measures the strength of the linear relationship between the two variables (how tightly the data lies along a least-squares line through the scatter):

```
C12 = cov([c1 c2])
```

```
C12 =

   1.0e+03 *

    0.6437    0.9802
    0.9802    1.7144
```

The results are displayed in a symmetric square matrix, with the covariance of the *i*th and *j*th variables in the (*i*, *j*)th position. The *i*th diagonal element is the variance of the *i*th variable.

Covariances have the disadvantage of depending on the units used to measure the individual variables. You can divide a covariance by the standard deviations of the variables to normalize values between +1 and –1. The `corrcoef` function computes *correlation coefficients*:

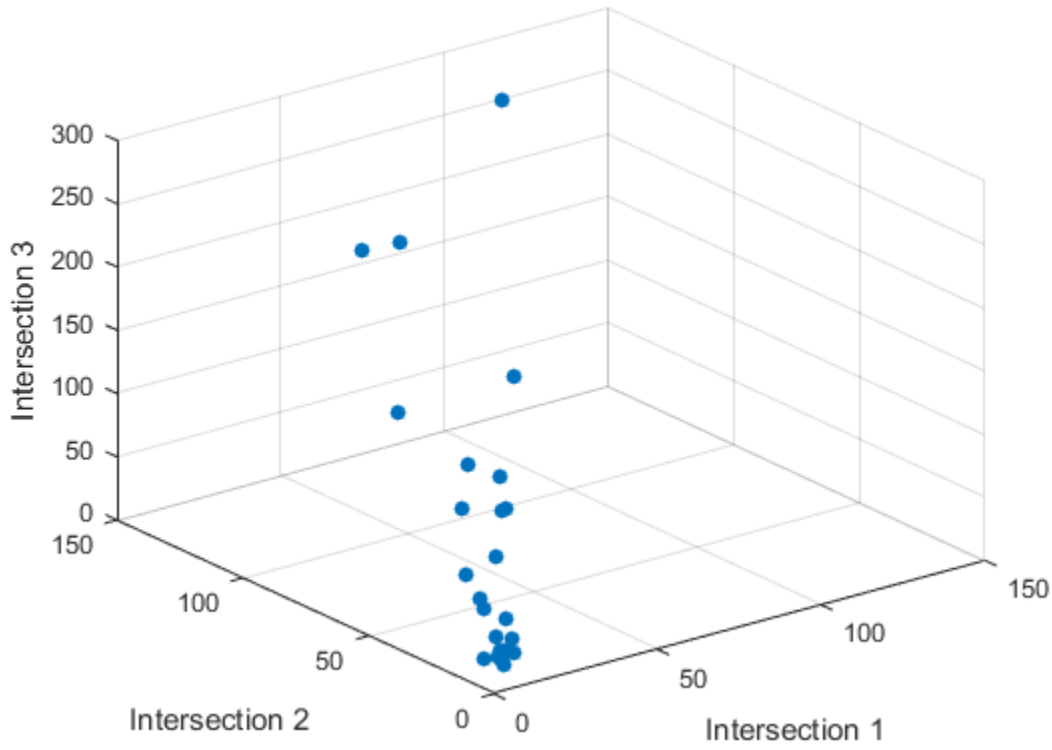
```
R12 = corrcoef([c1 c2])


R12 =

    1.0000    0.9331
    0.9331    1.0000


r12 = R12(1,2) % Correlation coefficient


r12 =

    0.9331


r12sq = r12^2 % Coefficient of determination


r12sq =

    0.8707
```

Because it is normalized, the value of the correlation coefficient is readily comparable to values for other pairs of intersections. Its square, the *coefficient of determination*, is the variance about the least-squares line divided by the variance about the mean. Thus, it is the proportion of variation in the response (in this case, the traffic volume at intersection 2) that is eliminated or statistically explained by a least-squares line through the scatter.

3-D Scatter Plots

A three-dimensional scatter plot, created with the `scatter3` function, shows the relationship between the traffic volume at all three intersections. Use the variables `c1`, `c2`, and `c3` that you created in the previous step:

```
figure
c3 = count(:,3); % Data at intersection 3
scatter3(c1,c2,c3,'filled')
xlabel('Intersection 1')
ylabel('Intersection 2')
zlabel('Intersection 3')
```



Measure the strength of the linear relationship among the variables in the three-dimensional scatter by computing eigenvalues of the covariance matrix with the `eig` function:

```
vars = eig(cov([c1 c2 c3]))
```

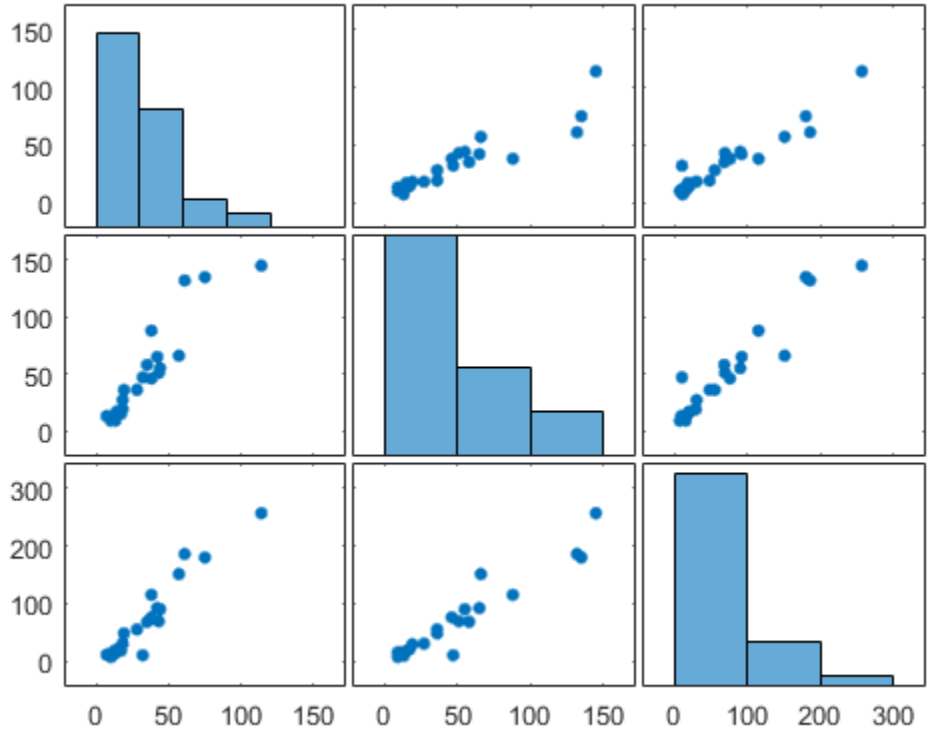
```
vars =

   1.0e+03 *

    0.0442
    0.1118
    6.8300


explained = max(vars)/sum(vars)


explained =

    0.9777
```

The eigenvalues are the variances along the *principal components* of the data. The variable `explained` measures the proportion of variation explained by the first principal component, along the axis of the data. Unlike the coefficient of determination for two-dimensional scatters, this measure distinguishes predictor and response variables.

Scatter Plot Arrays

Use the `plotmatrix` function to make comparisons of the relationships between multiple pairs of intersections:

```
figure
plotmatrix(count)
```



The plot in the (*i*, *j*)th position of the array is a scatter with the *i*th variable on the vertical axis and the *j*th variable on the horizontal axis. The plot in the *i*th diagonal position is a histogram of the *i*th variable.

Exploring Data in Graphs

Using your mouse, you can pick observations on almost any MATLAB graph with two tools from the figure toolbar:

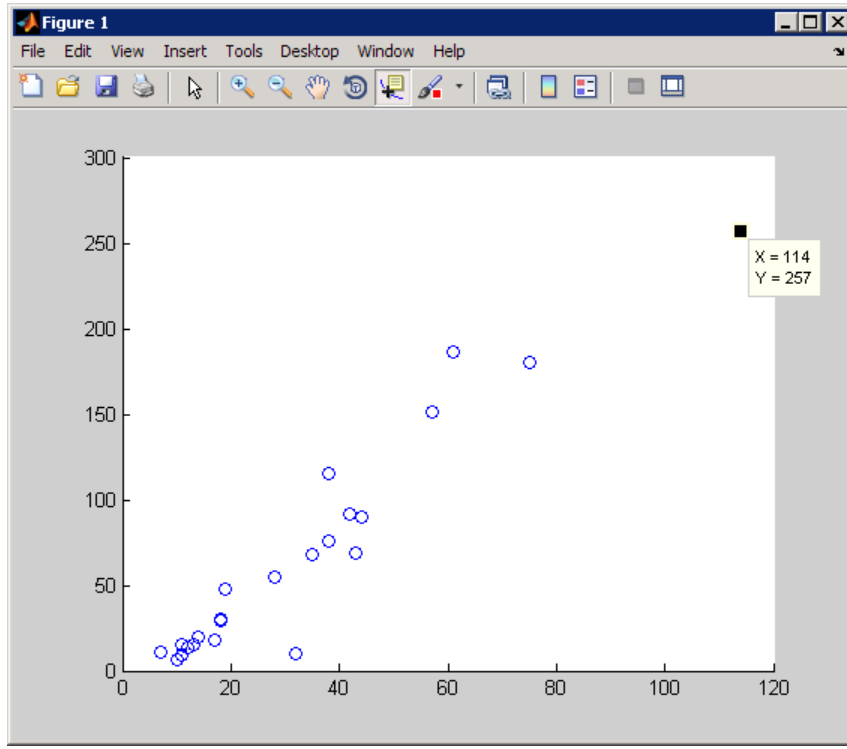
- Data Cursor 
- Data Brushing 

These tools each place you in exploratory modes in which you can select data points on graphs to identify their values and create workspace variables to contain specific observations. When you use data brushing, you can also copy, remove or replace the selected observations.

For example, make a scatter plot of the first and third columns of `count`:

```
load count.dat
scatter(count(:,1),count(:,3))
```

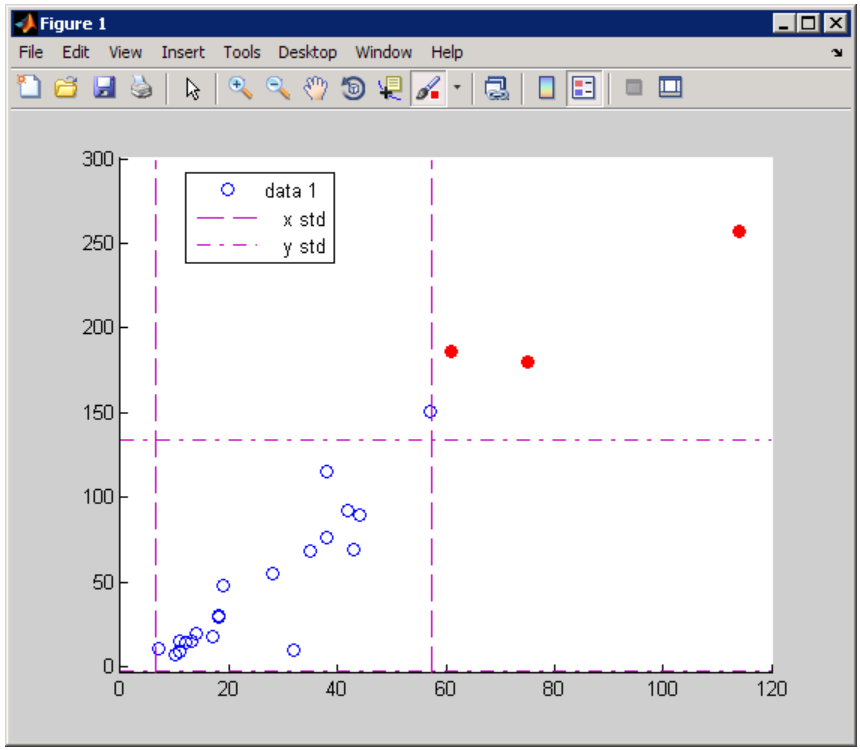
Select the Data Cursor Tool  and click the rightmost data point. A datatip displaying the point's *x* and *y* value is placed there.



Datatips display *x*-, *y*-, and *z*- (for three-dimensional plots) coordinates by default. You can drag a datatip from one data point to another to see new values or add additional datatips by right-clicking a datatip and using the context menu. You can also customize the text that datatips display using MATLAB code.

Data brushing is a related feature that lets you highlight one or more observations on a graph by clicking or dragging. To enter data brushing mode, click the left side of the Data Brushing tool on the figure toolbar. Clicking the arrow on the right side of the tool icon drops down a color palette for selecting the color with which to brush observations. This figure shows the same scatter plot as the previous figure, but with all observations beyond one standard deviation of the mean (as identified using the **Tools** > **Data Statistics** GUI) brushed in red.

```
scatter(count(:,1),count(:,3))
```

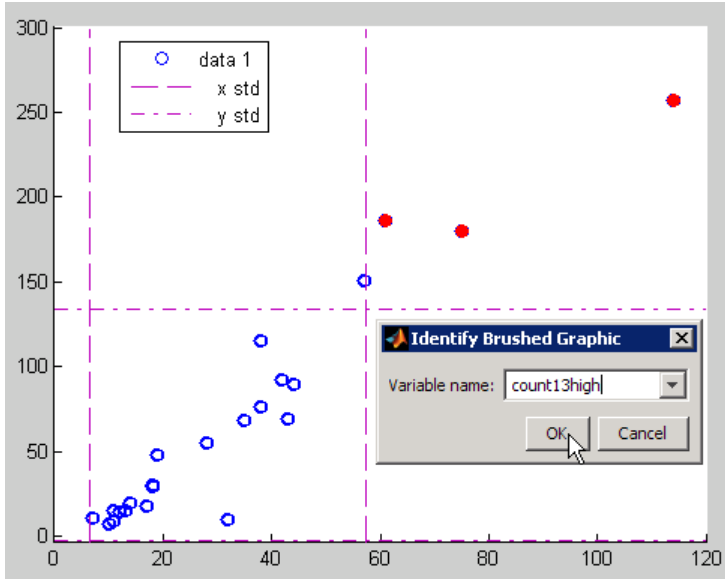


After you brush data observations, you can perform the following operations on them:

- Delete them.
- Replace them with constant values.
- Replace them with NaN values.

- Drag or copy, and paste them to the Command Window.
- Save them as workspace variables.

For example, use the Data Brush context menu or the **Tools > Brushing > Create new variable** option to create a new variable called `count13high`.



A new variable in the workspace results:

```
count13high

count13high =
    61   186
    75   180
   114   257
```

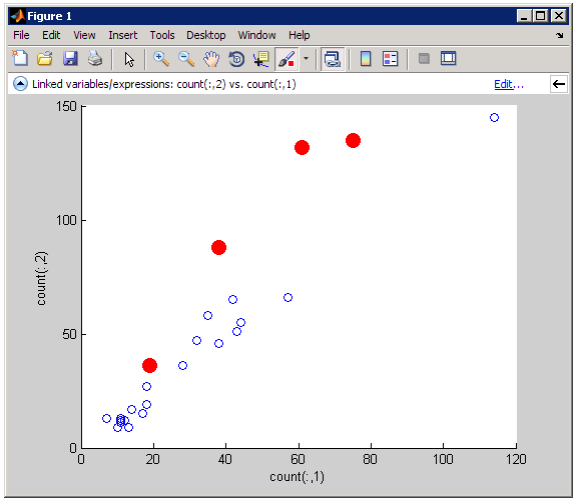
Linked plots, or *data linking*, is a feature closely related to data brushing. A plot is said to be linked when it has a live connection to the workspace data it depicts. The copies of variables stored in an object's `XData`, `YData`, (and, where appropriate, `ZData`), are automatically updated whenever the workspace variables to which they are linked change or are deleted. This causes the graphs on which they appear to update automatically.

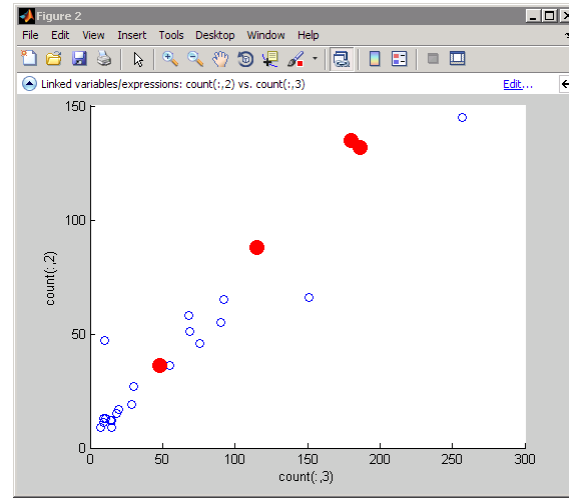
Linking plots to variables lets you track specific observations through different presentations of them. When you brush data points in linked plots, brushing one graph highlights the same observations in every graph that is linked to the same variables.

Data linking establishes immediate, two-way communication between figures and workspace variables, in the same way that the Variable Editor communicates with workspace variables. You create links by activating the Data Linking tool  on a figure's toolbar. Activating this tool causes the Linked Plot information bar, displayed in the next figure, to appear at the top of the plot (possibly obscuring its title). You can dismiss the bar (shown in the following figure) without unlinking the plot; it does not print and is not saved with the figure.

The following two graphs depict scatter plot displays of linked data after brushing some observations on the left graph. The common variable, `count` carries the brush marks to the right figure. Even though the right graph is not in data brushing mode, it displays brush marks because it is linked to its variables.

```
figure
scatter(count(:,1),count(:,2))
xlabel ('count(:,1)')
ylabel ('count(:,2)')
figure
scatter(count(:,3),count(:,2))
xlabel ('count(:,3)')
ylabel ('count(:,2)')
```

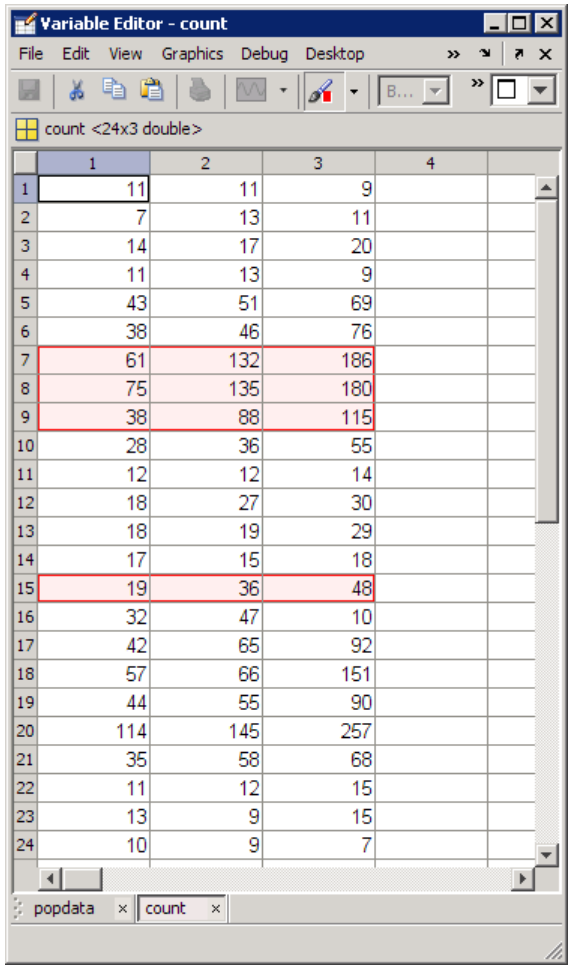




The right plot shows that the brushed observations are more linearly related than in the left plot.

Brushed data observations appear highlighted in the brushing color when you display those variables in the Variable Editor, as you can see here:

```
openvar count
```



	1	2	3	4
1	11	11	9	
2	7	13	11	
3	14	17	20	
4	11	13	9	
5	43	51	69	
6	38	46	76	
7	61	132	186	
8	75	135	180	
9	38	88	115	
10	28	36	55	
11	12	12	14	
12	18	27	30	
13	18	19	29	
14	17	15	18	
15	19	36	48	
16	32	47	10	
17	42	65	92	
18	57	66	151	
19	44	55	90	
20	114	145	257	
21	35	58	68	
22	11	12	15	
23	13	9	15	
24	10	9	7	

In the Variable Editor, you can alter any values of linked plot data, and the graphs will reflect your edits. To brush data observation from the Variable Editor, click its Brushing

Tool  button. If the variable you brush is currently depicted in a linked plot, the observations you brush highlight in the plot, as well as in the Variable Editor. When you brush a variable that is a column in a matrix, the other columns in that row are also brushed. That is, you can brush individual observations in a row or column vector, but all columns in a matrix highlight in any row you brush, not just the observations you click.

Modeling Data

Overview

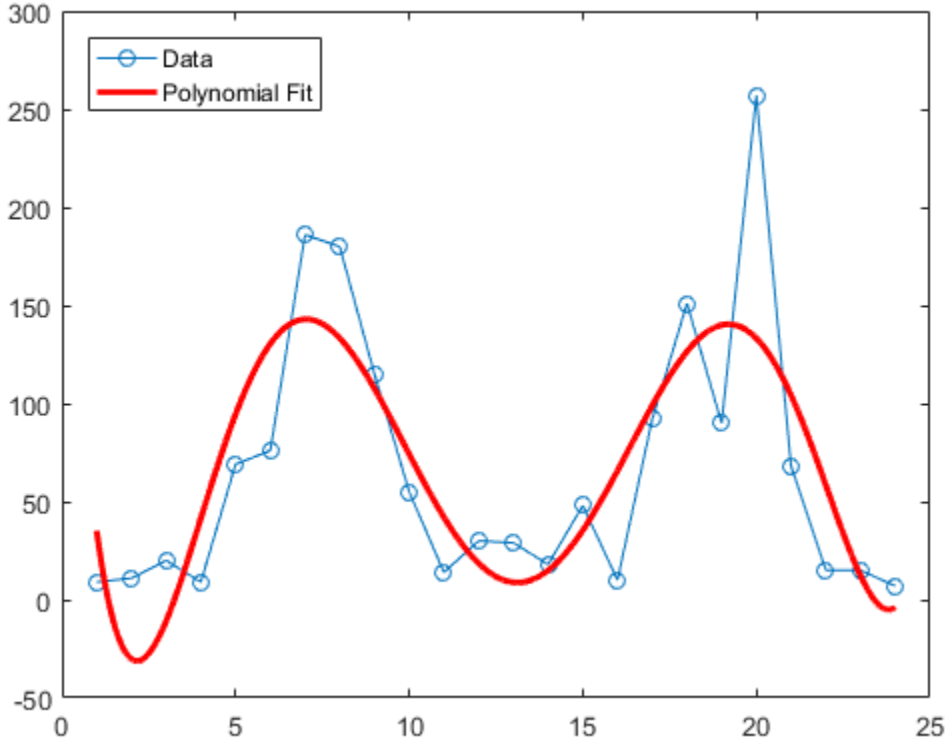
Parametric models translate an understanding of data relationships into analytic tools with predictive power. Polynomial and sinusoidal models are simple choices for the up and down trends in the traffic data.

Polynomial Regression

Use the `polyfit` function to estimate coefficients of polynomial models, then use the `polyval` function to evaluate the model at arbitrary values of the predictor.

The following code fits the traffic data at the third intersection with a polynomial model of degree six:

```
load count.dat
c3 = count(:,3); % Data at intersection 3
tdata = (1:24)';
p_coeffs = polyfit(tdata,c3,6);

figure
plot(c3,'o-')
hold on
tfit = (1:0.01:24)';
yfit = polyval(p_coeffs,tfit);
plot(tfit,yfit,'r-','LineWidth',2)
legend('Data','Polynomial Fit','Location','NW')
```



The model has the advantage of being simple while following the up-and-down trend. The accuracy of its predictive power, however, is questionable, especially at the ends of the data.

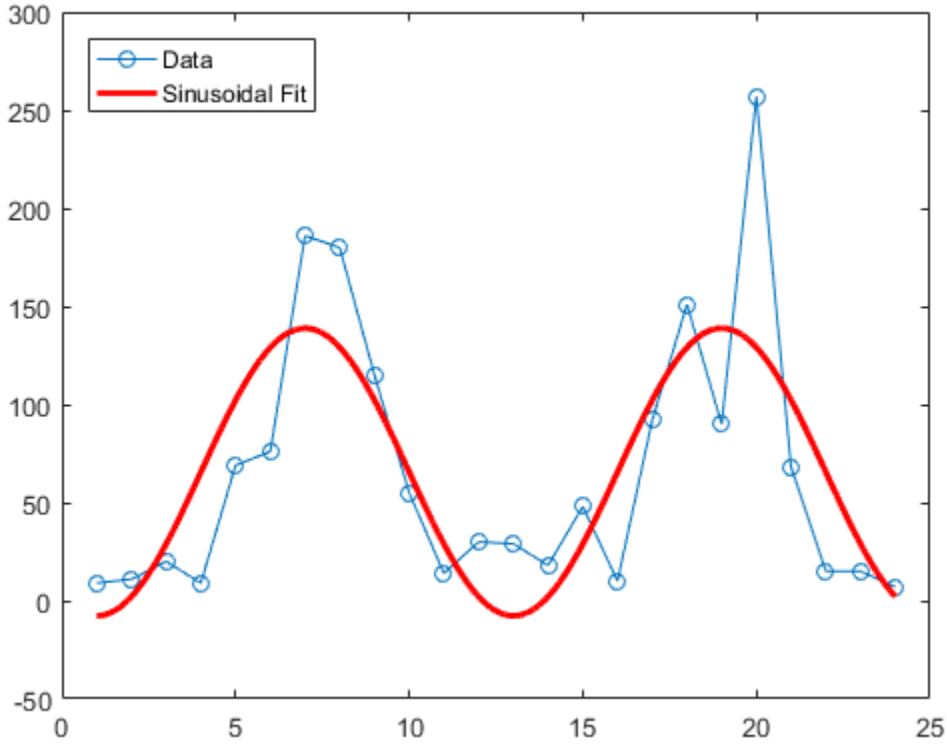
General Linear Regression

Assuming that the data are periodic with a 12-hour period and a peak around hour 7, it is reasonable to fit a sinusoidal model of the form:

$$y = a + b\cos((2\pi / 12)(t - 7))$$

The coefficients a and b appear linearly. Use the MATLAB® `mldivide` (backslash) operator to fit general linear models:

```
load count.dat
c3 = count(:,3); % Data at intersection 3
tdata = (1:24)';
X = [ones(size(tdata)) cos((2*pi/12)*(tdata-7))];
s_coeffs = X\c3;

figure
plot(c3,'o-')
hold on
tfit = (1:0.01:24)';
yfit = [ones(size(tfit)) cos((2*pi/12)*(tfit-7))]*s_coeffs;
plot(tfit,yfit,'r-','LineWidth',2)
legend('Data','Sinusoidal Fit','Location','NW')
```



Use the `lscov` function to compute statistics on the fit, such as estimated standard errors of the coefficients and the mean squared error:

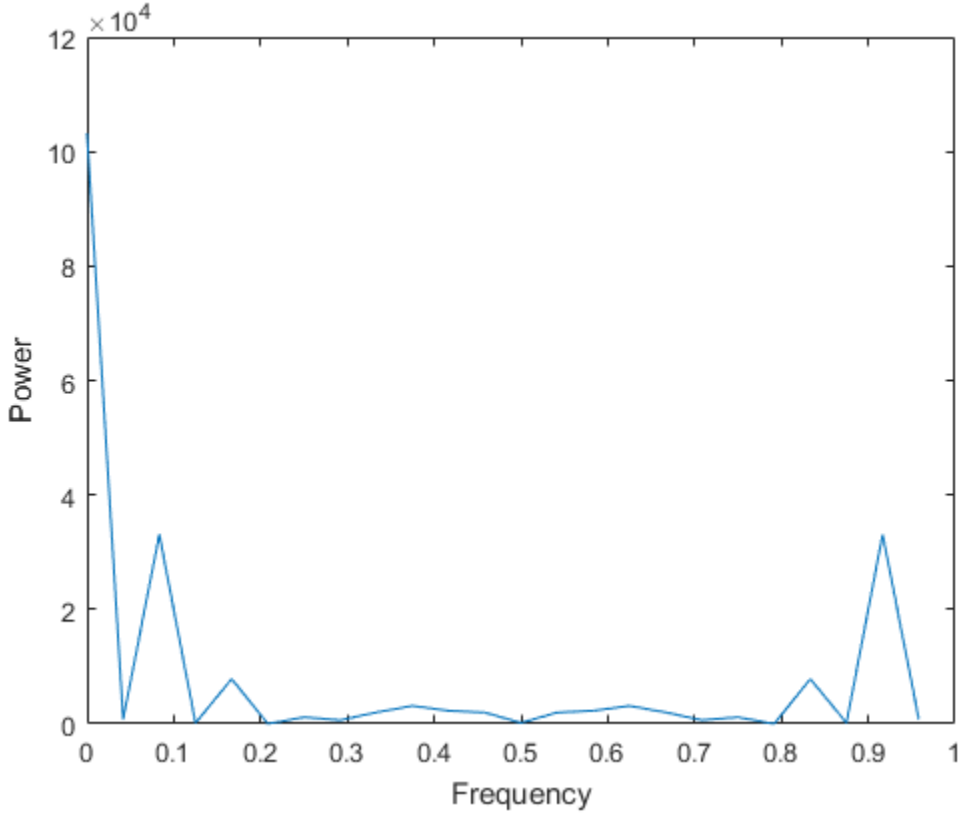
```
[s_coeffs,stdx,mse] = lscov(X,c3)
```

```
s_coeffs =

   65.5833
   73.2819


stdx =

    8.9185
   12.6127


mse =

   1.9090e+03
```

Check the assumption of a 12-hour period in the data with a *periodogram*, computed using the `fft` function:

```
Fs = 1; % Sample frequency (per hour)
n = length(c3); % Window length
Y = fft(c3); % DFT of data
f = (0:n-1)*(Fs/n); % Frequency range
P = Y.*conj(Y)/n; % Power of the DFT

figure
plot(f,P)
xlabel('Frequency')
ylabel('Power')

predicted_f = 1/12
```

```
predicted_f =

    0.0833
```



The peak near `0.0833` supports the assumption, although it occurs at a slightly higher frequency. The model can be adjusted accordingly.

See Also

`convn` | `corrcoef` | `cov` | `eig` | `fft` | `filter` | `histogram` | `isnan` | `lscov` | `max` | `mean` | `median` | `min` | `mldivide` | `mode` | `plotmatrix` | `polyfit` | `polyval` | `scatter` | `scatter3` | `std` | `var`

4

Graphics

Basic Plotting Functions

In this section...

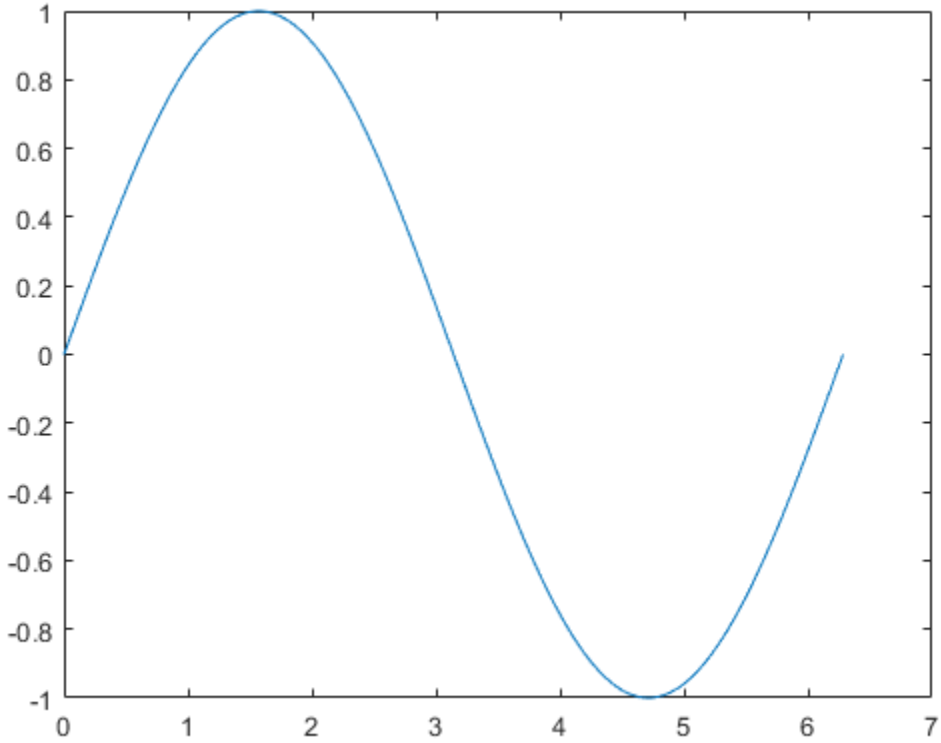
Creating a Plot

The `plot` function has different forms, depending on the input arguments.

- If `y` is a vector, `plot(y)` produces a piecewise linear graph of the elements of `y` versus the index of the elements of `y`.
- If you specify two vectors as arguments, `plot(x,y)` produces a graph of `y` versus `x`.

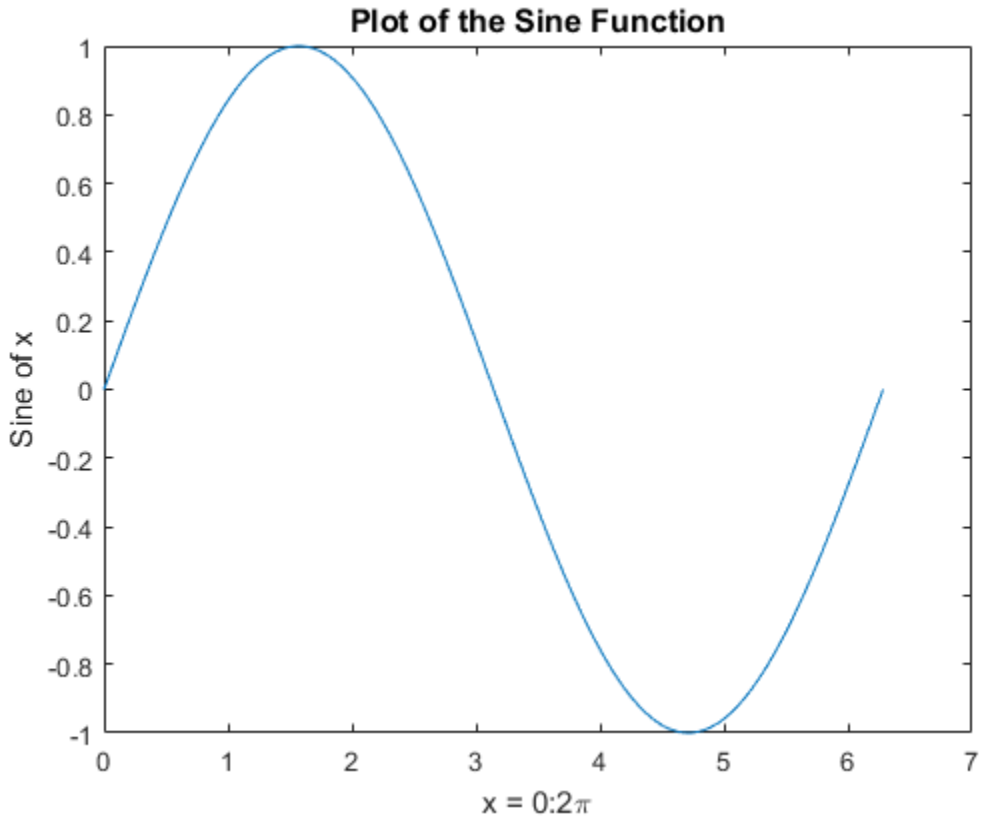
Use the colon operator to create a vector of `x` values ranging from `0` to 2π, compute the sine of these values, and plot the result.

```
x = 0:pi/100:2*pi;
y = sin(x);
plot(x,y)
```



Add axis labels and a title. The characters `\pi` in the `xlabel` function create the symbol π. The `FontSize` property in the `title` function increases the size the text used for the title.

```
xlabel('x = 0:2\pi')
ylabel('Sine of x')
title('Plot of the Sine Function','FontSize',12)
```

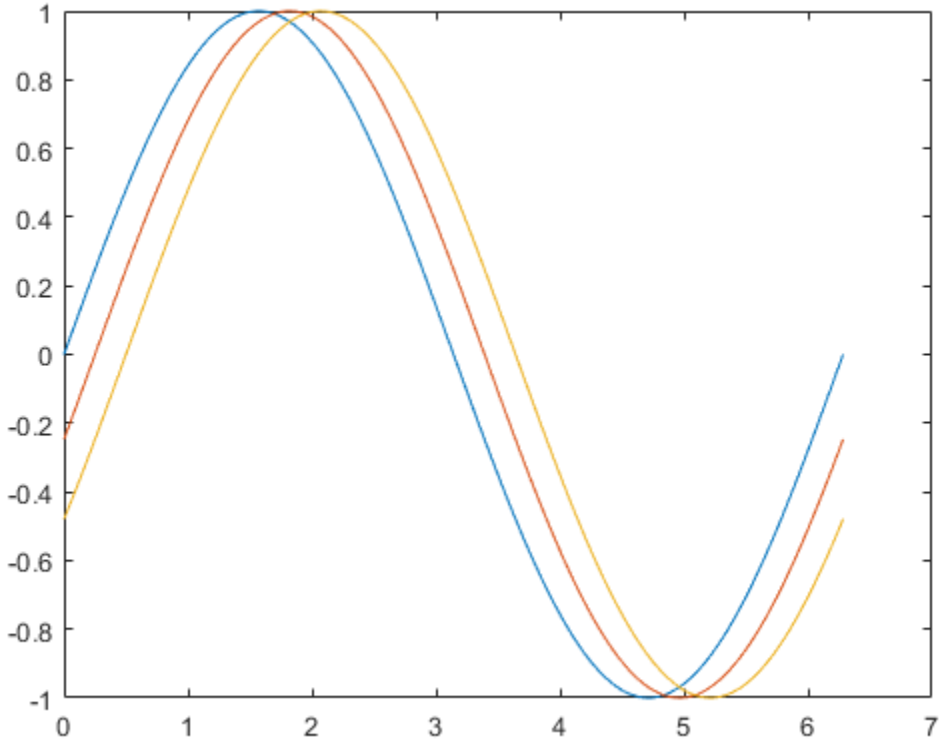


Plotting Multiple Data Sets in One Graph

Multiple `x-y` pair arguments create multiple graphs with a single call to `plot`. MATLAB® uses a different color for each line.

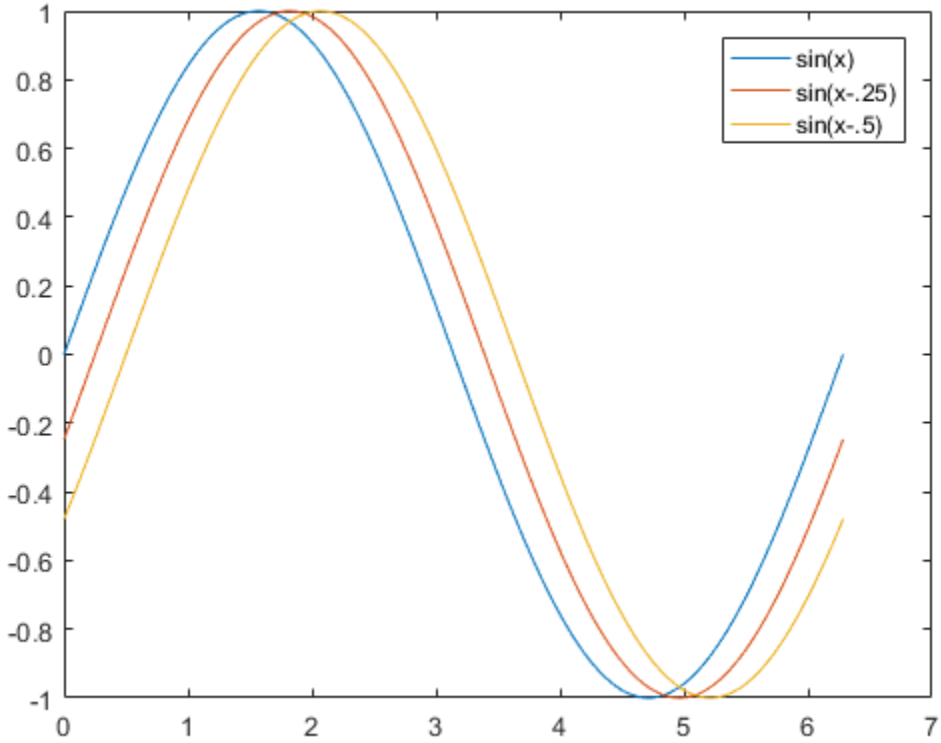
For example, these statements plot three related functions of `x` :

```
x = 0:pi/100:2*pi;
y = sin(x);
y2 = sin(x-.25);
y3 = sin(x-.5);
plot(x,y,x,y2,x,y3)
```



The `legend` function provides an easy way to identify the individual lines:

```
legend('sin(x)','sin(x-.25)','sin(x-.5)')
```



Specifying Line Styles and Colors

It is possible to specify color, line styles, and markers (such as plus signs or circles) when you plot your data using the `plot` command:

```
plot(x,y,'color_style_marker')
```

`color_style_marker` contains one to four characters (enclosed in single quotes) constructed from a color, a line style, and a marker type. For example,

```
plot(x,y,'r:+')
```

plots the data using a red-dotted line and places a + marker at each data point.

color_style_marker is composed of combinations of the following elements.

Type	Values	Meanings
Color	'c' 'm' 'y' 'r' 'g' 'b' 'w' 'k'	cyan magenta yellow red green blue white black
Line style	'-' '--' ':' '-.' no character	solid dashed dotted dash-dot no line
Marker type	'+' 'o' '*' 'x' 's' 'd' '^' 'v' '>' '<' 'p' 'h' no character	plus mark unfilled circle asterisk letter x filled square filled diamond filled upward triangle filled downward triangle filled right-pointing triangle filled left-pointing triangle filled pentagram filled hexagram no marker

Plotting Lines and Markers

If you specify a marker type, but not a line style, MATLAB creates the graph using only markers, but no line. For example,

```
plot(x,y,'ks')
```

plots black squares at each data point, but does not connect the markers with a line.

The statement

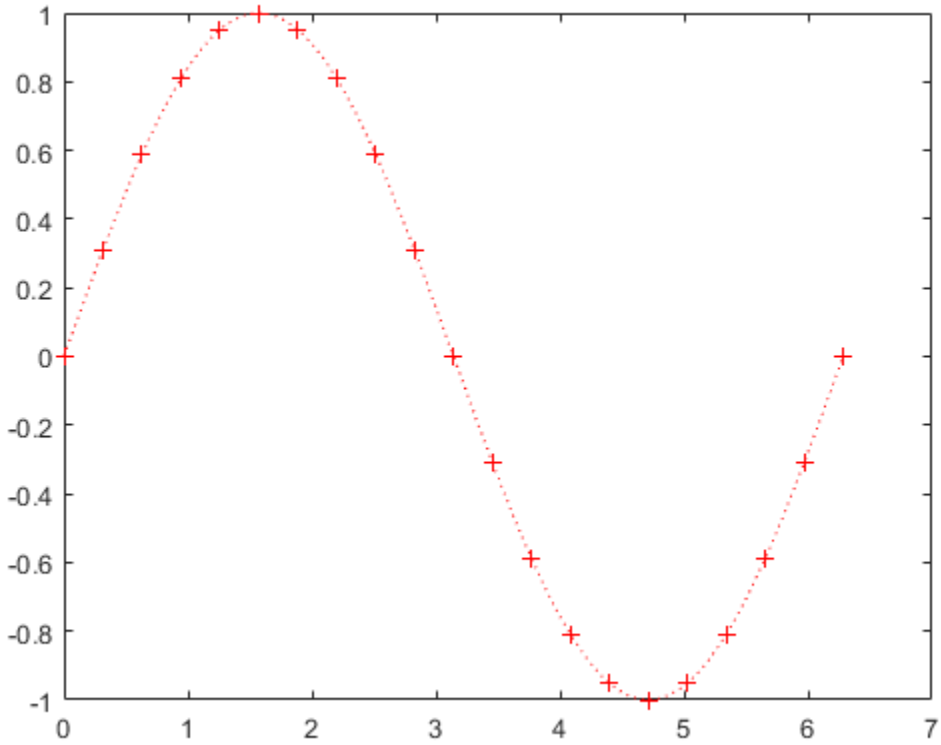
```
plot(x,y,'r:+')
```

plots a red-dotted line and places plus sign markers at each data point.

Placing Markers at Every Tenth Data Point

This example shows how to use fewer data points to plot the markers than you use to plot the lines. This example plots the data twice using a different number of points for the dotted line and marker plots.

```
x1 = 0:pi/100:2*pi;
x2 = 0:pi/10:2*pi;
plot(x1,sin(x1),'r:',x2,sin(x2),'r+')
```



Graphing Imaginary and Complex Data

When you pass complex values as arguments to `plot`, MATLAB ignores the imaginary part, *except* when you pass a single complex argument. For this special case, the command is a shortcut for a graph of the real part versus the imaginary part. Therefore,

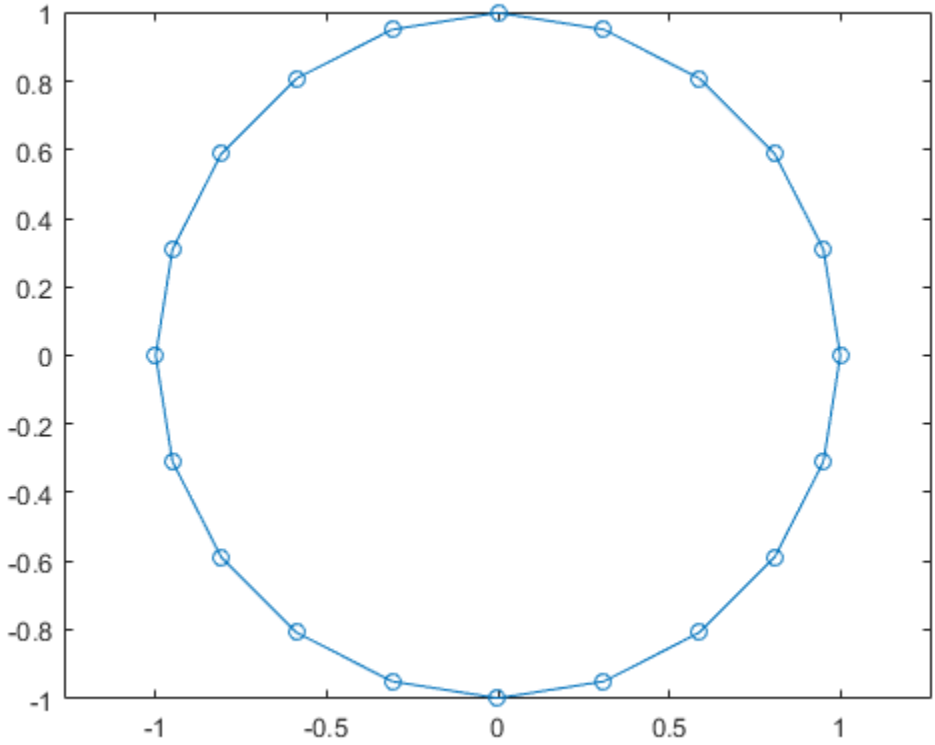
```
plot(Z)
```

where `Z` is a complex vector or matrix, is equivalent to

```
plot(real(Z),imag(Z))
```

The following statements draw a 20-sided polygon with little circles at the vertices.

```
t = 0:pi/10:2*pi;
plot(exp(i*t),'-o')
axis equal
```



The `axis equal` command makes the individual tick-mark increments on the *x*- and *y*-axes the same length, which makes this plot more circular in appearance.

Adding Plots to an Existing Graph

The `hold` command enables you to add plots to an existing graph. When you type,

```
hold on
```

MATLAB does not replace the existing graph when you issue another plotting command. Instead, MATLAB combines the new graph with the current graph.

For example, these statements first create a surface plot of the `peaks` function, then superimpose a contour plot of the same function.

```
[x,y,z] = peaks;
% Create surface plot
surf(x,y,z)
% Remove edge lines a smooth colors
shading interp
% Hold the current graph
hold on
% Add the contour graph to the pcolor graph
contour3(x,y,z,20,'k')
% Return to default
hold off
```

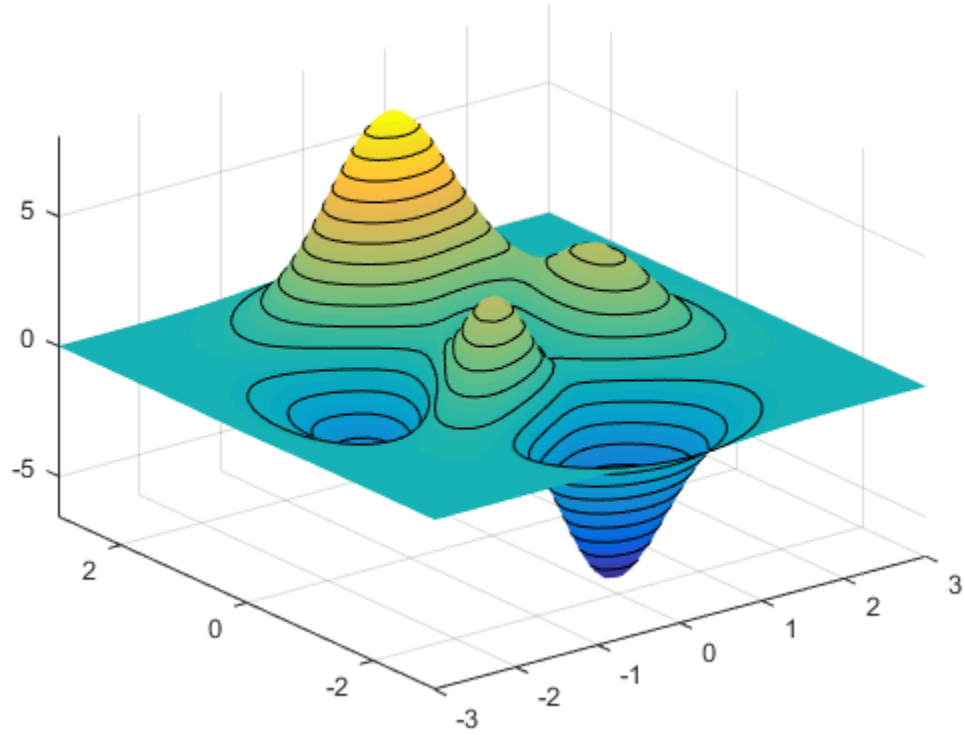


Figure Windows

Plotting functions automatically open a new figure window if there are no figure windows already created. If there are multiple figure windows open, MATLAB uses the one that is designated as the “current figure” (usually, the last figure used).

To make an existing figure window the current figure, you can click the mouse while the pointer is in that window or you can type,

```
figure(n)
```

where `n` is the number in the figure title bar.

To open a new figure window and make it the current figure, type

```
figure
```

Clearing the Figure for a New Plot

When a figure already exists, most plotting commands clear the axes and use this figure to create the new plot. However, these commands do not reset figure properties, such as the background color or the colormap. If you have set any figure properties in the previous plot, you can use the `clf` command with the `reset` option,

```
clf reset
```

before creating your new plot to restore the figure's properties to their defaults.

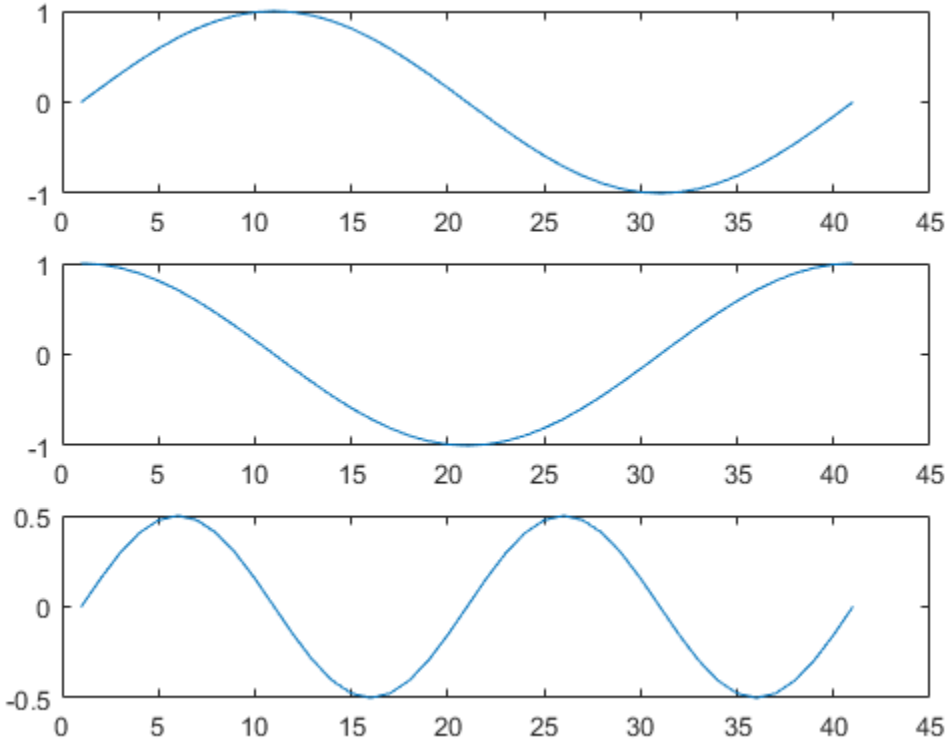
Displaying Multiple Plots in One Figure

The `subplot` command enables you to display multiple plots in the same window or print them on the same piece of paper. Typing

```
subplot(m,n,p)
```

partitions the figure window into an `m`-by-`n` matrix of small subplots and selects the p^{th} subplot for the current plot. The plots are numbered along the first row of the figure window, then the second row, and so on. For example, these statements plot data in three subregions of the figure window.

```
x = 0:pi/20:2*pi;
subplot(3,1,1); plot(sin(x))
subplot(3,1,2); plot(cos(x))
subplot(3,1,3); plot(sin(x).*cos(x))
```



Controlling the Axes

The `axis` command provides a number of options for setting the scaling, orientation, and aspect ratio of graphs.

Automatic Axis Limits and Tick Marks

By default, MATLAB finds the maxima and minima of the data and chooses the axis limits to span this range. MATLAB selects the limits and axis tick mark values to produce a graph that clearly displays the data. However, you can set your own limits using the `axis` or `xlim`, `ylim`, and `zlim` functions.

Note: Changing the limits of one axis can cause other limits to change to better represent the data. To disable automatic limit setting, enter the `axis manual` command.

Setting Axis Limits

The `axis` command enables you to specify your own limits:

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

or for three-dimensional graphs,

```
axis([xmin xmax ymin ymax zmin zmax])
```

Use the command

```
axis auto
```

to enable automatic limit selection again.

Setting the Axis Aspect Ratio

The `axis` command also enables you to specify a number of predefined modes. For example,

```
axis square
```

makes the *x*-axis and *y*-axis the same length.

```
axis equal
```

makes the individual tick mark increments on the *x*-axes and *y*-axes the same length. This means

```
plot(exp(i*[0:pi/10:2*pi]))
```

followed by either `axis square` or `axis equal` turns the oval into a proper circle:

```
axis auto normal
```

returns the axis scaling to its default automatic mode.

Setting Axis Visibility

You can use the `axis` command to make the axis visible or invisible.

```
axis on
```

makes the axes visible. This is the default.

```
axis off
```

makes the axes invisible.

Setting Grid Lines

The `grid` command toggles grid lines on and off. The statement

```
grid on
```

turns the grid lines on, and

```
grid off
```

turns them back off again.

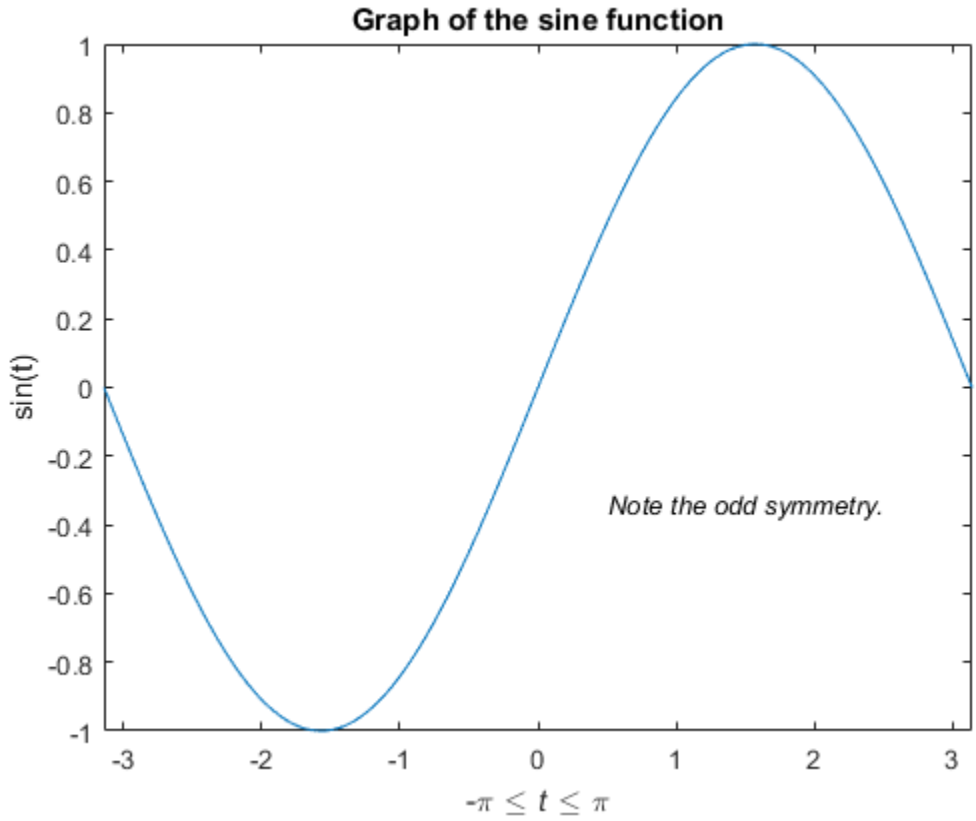
Adding Axis Labels and Titles

This example show how to create a graph and enhance its presentation:

- Define the x- and y-axis limits (`axis`)
- Label the x- and y-axes (`xlabel`, `ylabel`)
- Add a title (`title`)
- Add a text note on the graph (`text`)

Produce mathematical symbols using LaTeX notation.

```
t = -pi:pi/100:pi;
y = sin(t);
plot(t,y)

axis([-pi pi -1 1])
xlabel('-\pi \leq {\itt} \leq \pi')
ylabel('sin(t)')
title('Graph of the sine function')
text(0.5,-1/3,'{\itNote the odd symmetry.}')
```



For information on placing arrows, boxes, and circles in your graph, see the `annotation` function.

Saving Figures

Save a figure by selecting **Save** from the **File** menu. This writes the figure to a file, including property data, its menus, uicontrols, and all annotations (i.e., the entire window). If you have not saved the figure before, the **Save As** dialog displays. This dialog box provides options to save the figure as a `.fig` file or export it to a graphics format.

If you have previously saved the figure, using **Save** again saves the figure "silently," without displaying the **Save As** dialog.

To save a figure using a standard graphics format for use with other applications, such as TIFF or JPG, select **Save As** (or **Export Setup**, if you want additional control) from the **File** menu.

Note: Whenever you specify a format for saving a figure, that file format is used again the next time you save that figure or a new one. If you do not want to save in the previously used format, use **Save As** and be sure to set the **Save as type** drop-down menu to the kind of file you want to write.

You can also save from the command line:

- Use the `savefig` function to save a figure and the graphics objects it contains as a `.fig` file.
- Use the `saveas` command, including any options to save the figure in a variety of formats.

Loading a Figure

You can load figures into MATLAB using these functions:

- Use the `openfig` function to load figures saved as `.fig` files.
- Use the `imread` function to read standard graphics files (including save figures) into MATLAB.

Generating MATLAB Code to Recreate a Figure

You can generate MATLAB code that recreates a figure and the graph it contains by selecting **Generate code** from the figure **File** menu. This option is particularly useful if you have developed a graph using plotting tools and want to create a similar graph using the same or different data.

Saving Workspace Data

You can save the variables in your workspace by selecting **Save Workspace As** from the figure **File** menu. You can reload saved data using the **Import Data** item in the figure **File** menu. MATLAB supports a variety of data file formats, including MATLAB data files, which have a `.mat` extension.

Creating Mesh and Surface Plots

In this section...

About Mesh and Surface Plots

MATLAB defines a surface by the *z*-coordinates of points above a grid in the *x*-*y* plane, using straight lines to connect adjacent points. The `mesh` and `surf` functions display surfaces in three dimensions.

- `mesh` produces wireframe surfaces that color only the lines connecting the defining points.
- `surf` displays both the connecting lines and the faces of the surface in color.

MATLAB colors surfaces by mapping z-data values to indexes into the figure colormap.

Visualizing Functions of Two Variables

To display a function of two variables, $z = f(x,y)$,

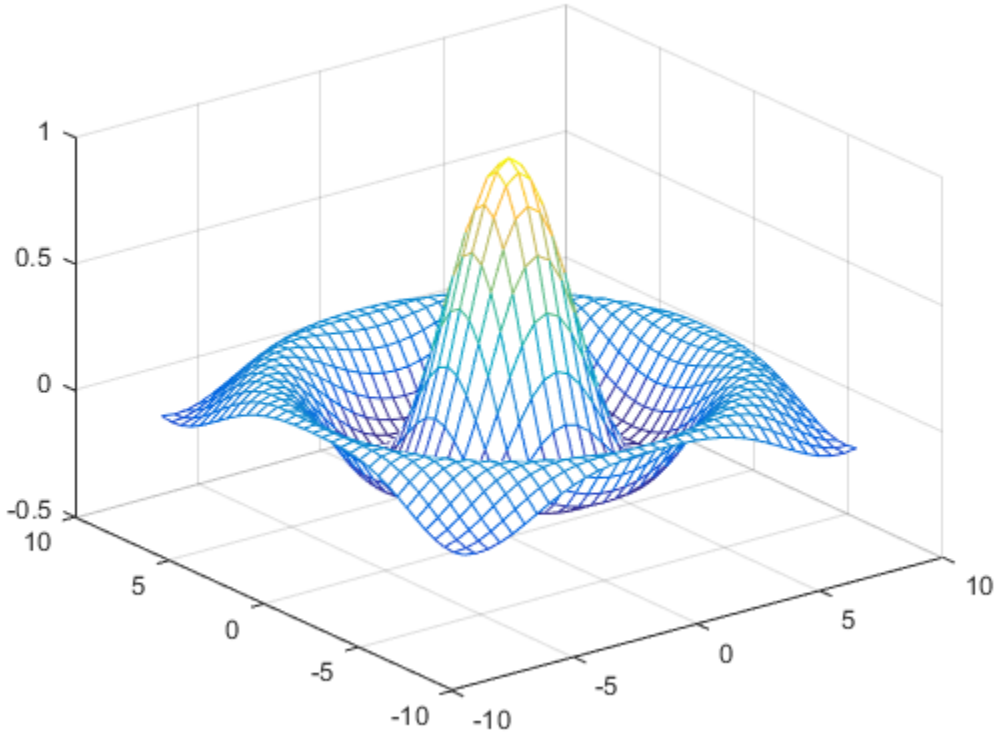
1. Generate `X` and `Y` matrices consisting of repeated rows and columns, respectively, over the domain of the function.
2. Use `X` and `Y` to evaluate and graph the function.

The `meshgrid` function transforms the domain specified by a single vector or two vectors `x` and `y` into matrices `X` and `Y` for use in evaluating functions of two variables. The rows of `X` are copies of the vector `x` and the columns of `Y` are copies of the vector `y`.

Graphing the sinc Function

This example shows how to evaluate and graph the two-dimensional `sinc` function, $\sin(r)/r$, between the *x* and *y* directions. `R` is the distance from the origin, which is at the center of the matrix. Adding `eps` (a very small value) prevents a hole in the mesh at the point where `R = 0`.

```
[X,Y] = meshgrid(-8:.5:8);
R = sqrt(X.^2 + Y.^2) + eps;
Z = sin(R)./R;
mesh(X,Y,Z)
```



By default, MATLAB uses the current colormap to color the mesh.

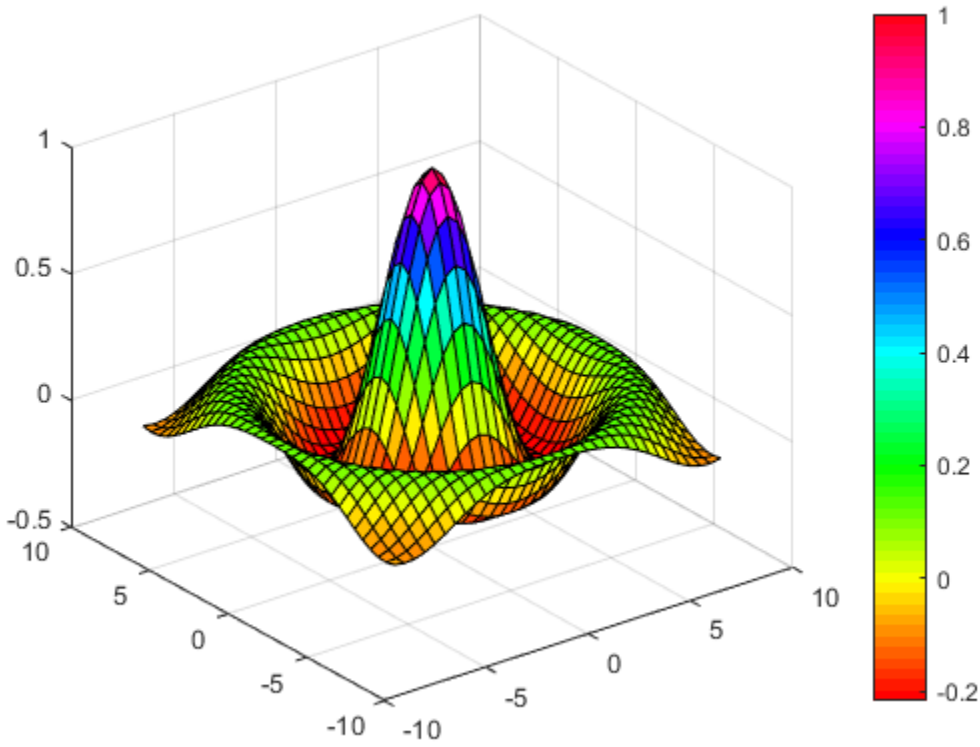
Colored Surface Plots

This example shows how to graph the `sinc` function as a surface plot, specify a colormap, and add a color bar to show the mapping of data to color.

A surface plot is similar to a mesh plot except that the rectangular faces of the surface are colored. The color of each face is determined by the values of `Z` and the colormap (a colormap is an ordered list of colors).

```
[X,Y] = meshgrid(-8:.5:8);
R = sqrt(X.^2 + Y.^2) + eps;
Z = sin(R)./R;
```

```
surf(X,Y,Z)
colormap hsv
colorbar
```

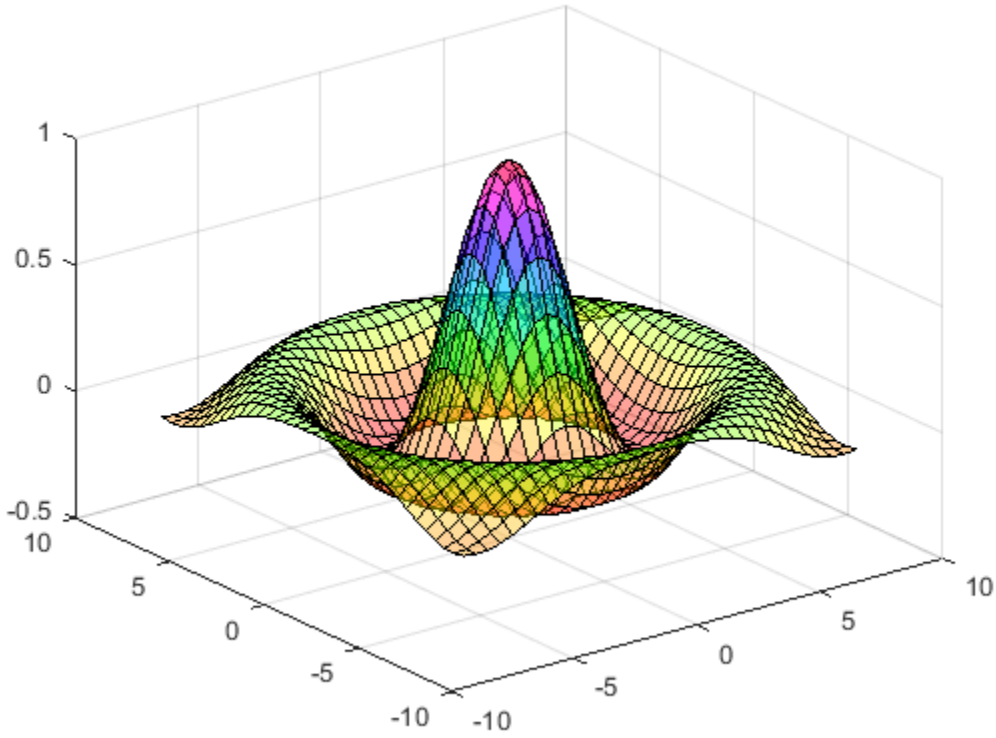


Making Surfaces Transparent

This example shows how you can make the faces of a surface transparent to a varying degree. Transparency (referred to as the alpha value) can be specified for the whole object or can be based on an `alphamap` , which behaves similarly to colormaps.

```
[X,Y] = meshgrid(-8:.5:8);
R = sqrt(X.^2 + Y.^2) + eps;
Z = sin(R)./R;
surf(X,Y,Z)
```

```
colormap hsv
alpha(.4)
```



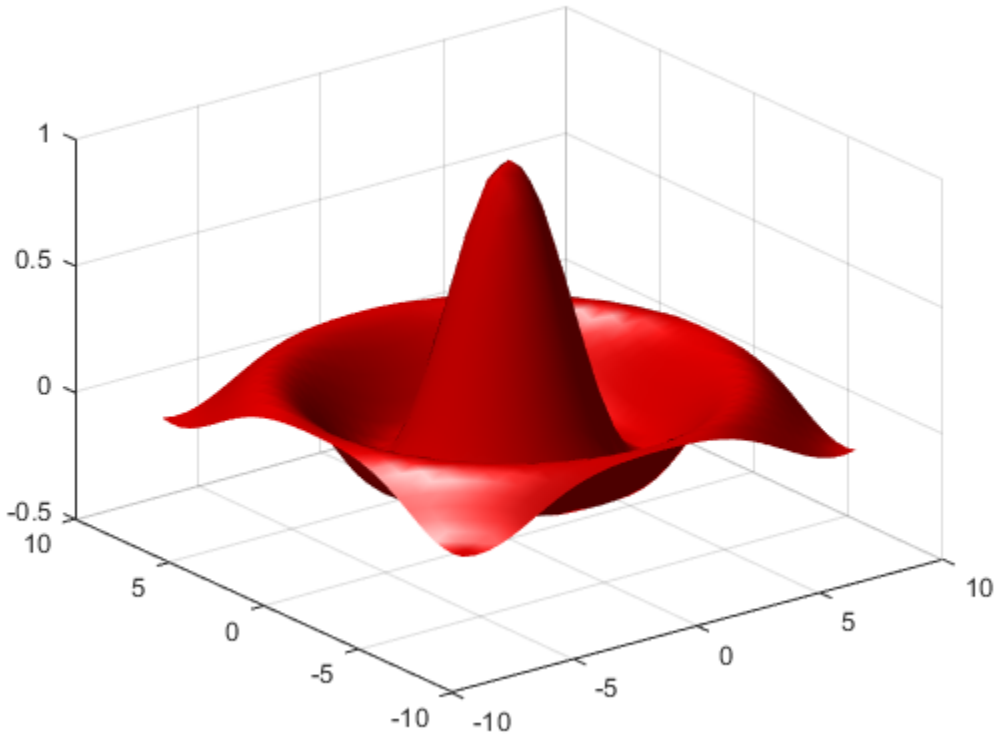
MATLAB displays a surface with a face alpha value of 0.4. Alpha values range from 0 (completely transparent) to 1 (not transparent).

Illuminating Surface Plots with Lights

This example shows the same surface as the previous examples, but colors it red and removes the mesh lines. A light object is then added to the left of the "camera" (the camera is the location in space from where you are viewing the surface).

```
[X,Y] = meshgrid(-8:.5:8);
R = sqrt(X.^2 + Y.^2) + eps;
```

```
Z = sin(R)./R;
surf(X,Y,Z,'FaceColor','red','EdgeColor','none')
camlight left;
lighting phong
```

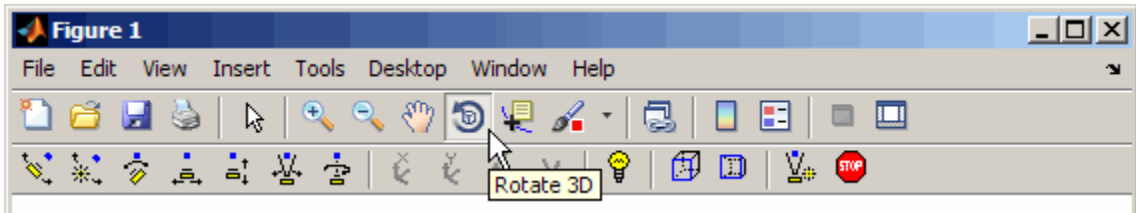


Lighting is the technique of illuminating an object with a directional light source. In certain cases, this technique can make subtle differences in surface shape easier to see. Lighting can also be used to add realism to three-dimensional graphs.

Manipulating the Surface

The figure toolbar and the camera toolbar provide ways to explore three-dimensional graphics interactively. Display the camera toolbar by selecting **Camera Toolbar** from the figure **View** menu.

The following picture shows both toolbars with the **Rotate 3D** tool selected.



These tools enable you to move the camera around the surface object, zoom, add lighting, and perform other viewing operations without issuing commands.

Display Images

In this section...

Image Data

You can display a 2-D numeric arrays as an *image*. In images, the array elements determine brightness or color of the images. For example, load an image array and its colormap:

```
load durer
whos
Name          Size         Bytes  Class

  X           648x509      2638656  double array
  caption     2x28             112  char array
  map         128x3           3072  double array
```

load the file `durer.mat`, adding three variables to the workspace. The array `X` is a 648-by-509 matrix and `map` is a 128-by-3 array that is the colormap for this image.

MAT-files, such as `durer.mat`, are binary files that provide a way to save MATLAB variables.

The elements of `X` are integers between 1 and 128, which serve as indices into the colormap, `map`. To display the image, use the imshow function:

```
imshow(X)
colormap(map)
```

reproduces Albrecht Dürer's etching.


MELENCOLIA I

Reading and Writing Images

You can read standard image files (TIFF, JPEG, PNG, and so on, using the `imread` function. The type of data returned by `imread` depends on the type of image you are reading.

You can write MATLAB data to a variety of standard image formats using the `imwrite` function.

Printing Graphics

In this section...

Overview of Printing

You can print a MATLAB figure directly on a printer connected to your computer or you can export the figure to one of the standard graphics file formats that MATLAB supports. There are two ways to print and export figures:

- Use the **Print**, **Print Preview**, or **Export Setup** GUI options under the **File** menu.
- Use the `print` command to print or export the figure from the command line.

The `print` command provides greater control over drivers and file formats. The Print Preview dialog box gives you greater control over figure size, proportions, placement, and page headers.

Printing from the File Menu

There are two menu options under the **File** menu that pertain to printing:

- The **Print Preview** option displays a dialog box that lets you lay out and style figures for printing while previewing the output page, and from which you can print the figure. It includes options that formerly were part of the Page Setup dialog box.
- The **Print** option displays a dialog box that lets you choose a printer, select standard printing options, and print the figure.

Use **Print Preview** to determine whether the printed output is what you want. Click the Print Preview dialog box **Help** button to display information on how to set up the page.

Exporting the Figure to a Graphics File

The **Export Setup** option in the **File** menu opens a GUI that enables you to set graphic characteristics, such as text size, font, and style, for figures you save as graphics

files. The Export Setup dialog lets you define and apply templates to customize and standardize output. After setup, you can export the figure to a number of standard graphics file formats, such as EPS, PNG, and TIFF.

Using the Print Command

The `print` command provides more flexibility in the type of output sent to the printer and allows you to control printing from function and script files. The result can be sent directly to your default printer or stored in a specified output file. A wide variety of output formats is available, including TIFF, JPEG, and PNG.

For example, this statement saves the contents of the current figure window as a PNG graphic in the file called `magicsquare.png`.

```
print -dpng magicsquare.png
```

To save the figure at the same size as the figure on the screen, use these statements:

```
set(gcf,'PaperPositionMode','auto')
print -dpng -r0 magicsquare.png
```

To save the same figure as a TIFF file with a resolution of 200 dpi, use the following command:

```
print -dtiff -r200 magicsquare.tiff
```

If you type `print` on the command line

```
print
```

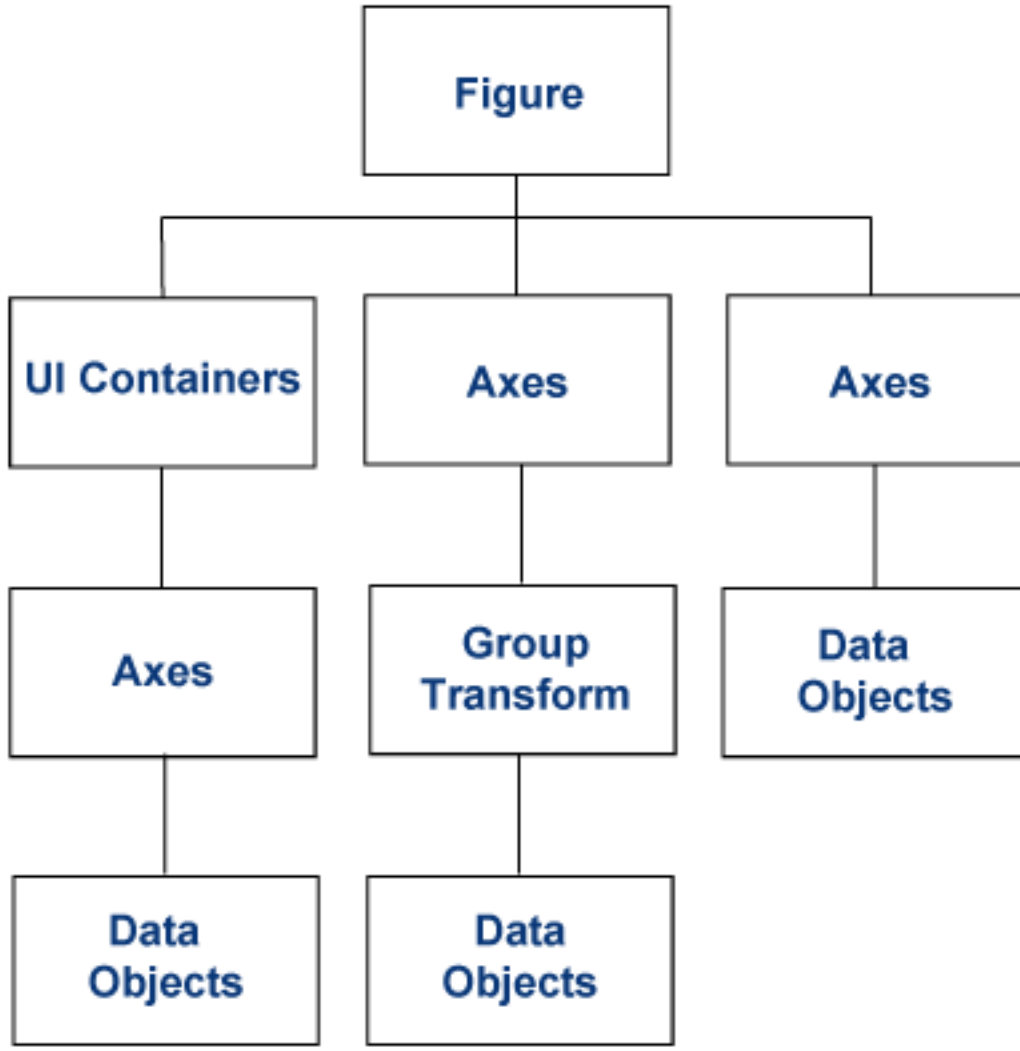
the current figure prints on your default printer.

Working with Graphics Objects

In this section...

Graphics Objects

Graphics objects are the basic elements used to display graphs. These objects are organized into a hierarchy, as shown by the following diagram.



When you call a plotting function, MATLAB creates the graph using various graphics objects, such as a figure window, axes, lines, text, and so on. Each object has a fixed set of properties, which you can use to control the behavior and appearance of the graph.

For example, the following statement creates a bar graph from the data in the variable y and sets properties that how the bars look:

```
y = [75 91 105 123.5 131 150 179 203 226 249 281.5];
bar(y,'FaceColor','green','EdgeColor','black','LineWidth',1.5)
```

Common Graphics Objects

When you call a function to create a graph, MATLAB creates a hierarchy of graphics objects. For example, calling the `plot` function creates the following graphics objects:

- Figure — Window that contains axes, toolbars, menus, and so on.
- Axes — Coordinate system that contains the object representing the data
- Line — Lines that represent the value of data passed to the `plot` function.
- Text — Labels for axes tick marks and optional titles and annotations.

Different types of graphs use different objects to represent data. Because there are many kinds of graphs, there are many types of data objects. Some are general purpose, such as lines and rectangles and some are highly specialized, such as errorbars, colorbars, and legends.

Access Object Properties

Plotting functions can return the objects used to create the graph. For example, the following statements create a graph and return the line object created by the `plot` function:

```
x = 1:10;
y = x.^3;
h = plot(x,y);
```

Use `h` to set the properties of the line object. For example, set its `Color` property.

```
h.Color = 'red';
```

You can also specify the line properties when you call the plotting function.

```
h = plot(x,y,'Color','red');
```

You can query the line properties to see the current value:

```
h.LineWidth

ans =
```

```
0.5000
```

Finding the Properties of an Object

To view the properties of an object, type:

```
get(h)
```

MATLAB returns a list of the object's properties with the current values.

To view an object's properties with information about possible values, type:

```
set(h)
```

Setting Object Properties

You can set multiple properties at once using the `set` function.

Setting Properties of Existing Objects

To set the same property to the same value on multiple objects, use the `set` function.

For example, the following statements plot a 5-by-5 matrix (creating five line objects, one per column), and then set the `Marker` property to square and the `MarkerFaceColor` property to green.

```
y = magic(5);
h = plot(y);
set(h,'Marker','s','MarkerFaceColor','g')
```

In this case, `h` is a vector containing five handles, one for each of the five lines in the graph. The `set` statement sets the `Marker` and `MarkerFaceColor` properties of all lines to the same value.

To set a property value on one object, index into the handle array:

```
h(1).LineWidth = 2;
```

Setting Multiple Property Values

If you want to set the properties of each line to a different value, you can use cell arrays to store all the data and pass it to the `set` command. For example, create a plot and save the line handles:

```
figure
y = magic(5);
h = plot(y);
```

Suppose you want to add different markers to each lines and color the marker's face color the same color as the line. You need to define two cell arrays—one containing the property names and the other containing the desired values of the properties.

The `prop_name` cell array contains two elements:

```
prop_name(1) = {'Marker'};
prop_name(2) = {'MarkerFaceColor'};
```

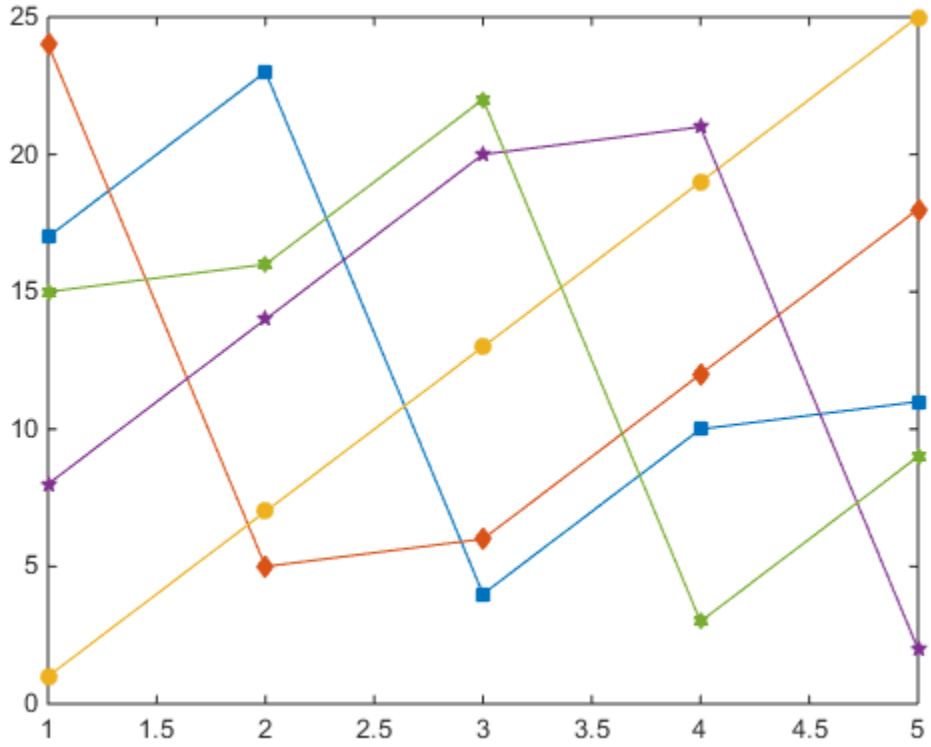
The `prop_values` cell array contains 10 values: five values for the `Marker` property and five values for the `MarkerFaceColor` property. Notice that `prop_values` is a two-dimensional cell array. The first dimension indicates which handle in `h` the values apply to and the second dimension indicates which property the value is assigned to:

```
prop_values(1,1) = {'s'};
prop_values(1,2) = {h(1).Color};
prop_values(2,1) = {'d'};
prop_values(2,2) = {h(2).Color};
prop_values(3,1) = {'o'};
prop_values(3,2) = {h(3).Color};
prop_values(4,1) = {'p'};
prop_values(4,2) = {h(4).Color};
prop_values(5,1) = {'h'};
prop_values(5,2) = {h(5).Color};
```

The `MarkerFaceColor` is always assigned the value of the corresponding line's color (obtained by getting the line `Color` property).

After defining the cell arrays, call `set` to specify the new property values:

```
set(h,prop_name,prop_values)
```



Functions for Working with Objects

This table lists functions commonly used when working with objects.

Function	Purpose
`allchild`	Find all children of specified objects.
`ancestor`	Find ancestor of graphics object.
`copyobj`	Copy graphics object.
`delete`	Delete an object.
`findall`	Find all graphics objects (including hidden handles).

Function	Purpose
`findobj`	Find the handles of objects having specified property values.
`gca`	Return the handle of the current axes.
`gcf`	Return the handle of the current figure.
`gco`	Return the handle of the current object.
`get`	Query the values of an object's properties.
`ishandle`	True if the value is a valid object handle.
`set`	Set the values of an object's properties.

Passing Arguments

You can define specialized plotting functions to simplify the creation of customized graphs. By defining a function, you can pass arguments like MATLAB plotting functions.

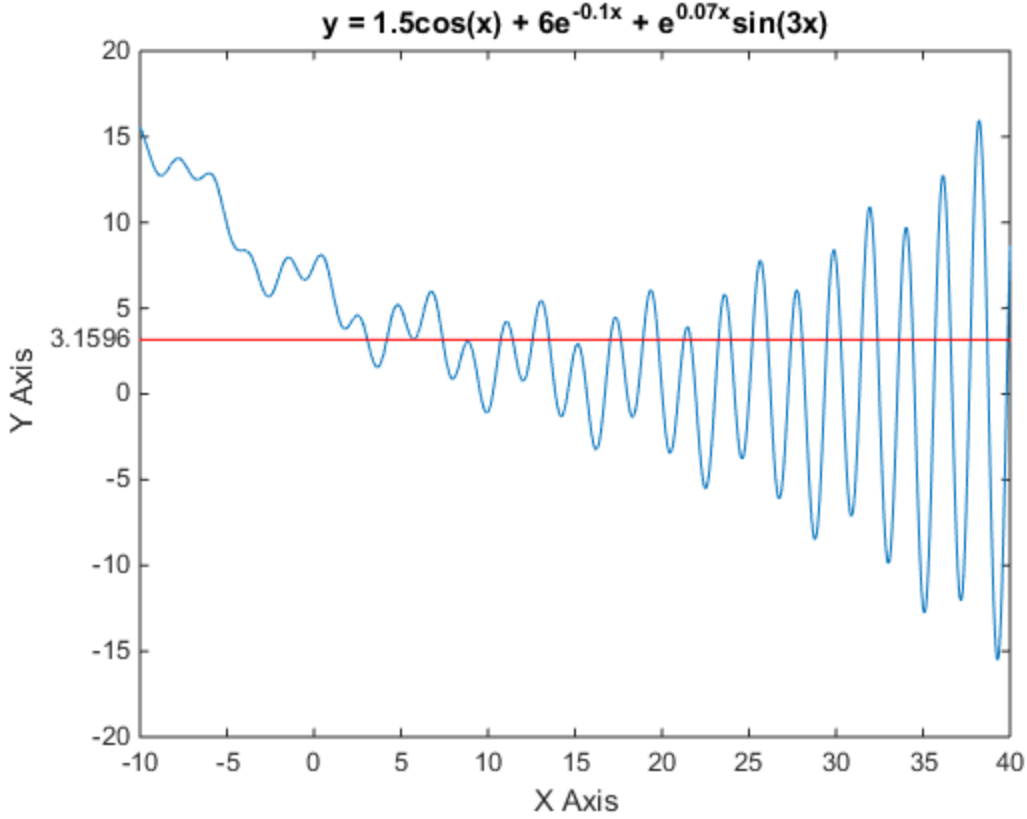
The following example shows a MATLAB function that evaluates a mathematical expression over the range specified in the input argument `x`, and then plots the results. A second call to the `plot` function plots the `mean` value of the results as a red line.

The function modifies the y-axis ticks based on the values calculated. Axis labels and a title finish the custom graph.

```
function plotFunc(x)
   y = 1.5*cos(x) + 6*exp(-.1*x) + exp(.07*x).*sin(3*x);
   ym = mean(y);
   hfig = figure('Name','Function and Mean');
   hax = axes('Parent',hfig);
   plot(hax,x,y)
   hold on
   plot(hax,[min(x) max(x)],[ym ym],'Color','red')
   hold off
   ylab = hax.YTick;
   new_ylab = sort([ylab, ym]);
   hax.YTick = new_ylab;
   title ('y = 1.5cos(x) + 6e^{-0.1x} + e^{0.07x}sin(3x)')
   xlabel('X Axis')
   ylabel('Y Axis')
end
```

Define a value for the input argument and call the function.

```
x = -10:.005:40;
plotFunc(x)
```



Finding the Handles of Existing Objects

The `findobj` function enables you to obtain the handles of graphics objects by searching for objects with particular property values. With `findobj` you can specify the values of any combination of properties, which makes it easy to pick one object out of many. `findobj` also recognizes regular expressions.

Finding All Objects of a Certain Type

Because all objects have a `Type` property that identifies the type of object, you can find the handles of all occurrences of a particular type of object. For example,

```
h = findobj('Type','patch');
```

finds the handles of all patch objects.

Finding Objects with a Particular Property

You can specify multiple properties to narrow the search. For example,

```
plot(rand(5),'r:')
h = findobj('Type','line','Color','r','LineStyle',':');
```

finds the handles of all red dotted lines.

```
h =

  5x1 Line array:

  Line
  Line
  Line
  Line
  Line
```

Limiting the Scope of the Search

You can specify the starting point in the object hierarchy by passing the handle of the starting figure or axes as the first argument. For example,

```
h = findobj(gca,'Type','text','String','\pi/2');
```

finds π/2 only within the current axes.

5

Programming

Control Flow

In this section...

Conditional Control — if, else, switch

Conditional statements enable you to select at run time which block of code to execute. The simplest conditional statement is an `if` statement. For example:

```
% Generate a random number
a = randi(100, 1);

% If it is even, divide by 2
if rem(a, 2) == 0
    disp('a is even')
    b = a/2;
end
```

`if` statements can include alternate choices, using the optional keywords `elseif` or `else`. For example:

```
a = randi(100, 1);

if a < 30
    disp('small')
elseif a < 80
    disp('medium')
else
    disp('large')
end
```

Alternatively, when you want to test for equality against a set of known values, use a `switch` statement. For example:

```
[dayNum, dayString] = weekday(date, 'long', 'en_US');
```

```
switch dayString
   case 'Monday'
      disp('Start of the work week')
   case 'Tuesday'
      disp('Day 2')
   case 'Wednesday'
      disp('Day 3')
   case 'Thursday'
      disp('Day 4')
   case 'Friday'
      disp('Last day of the work week')
   otherwise
      disp('Weekend!')
end
```

For both `if` and `switch`, MATLAB executes the code corresponding to the first true condition, and then exits the code block. Each conditional statement requires the `end` keyword.

In general, when you have many possible discrete, known values, `switch` statements are easier to read than `if` statements. However, you cannot test for inequality between `switch` and `case` values. For example, you cannot implement this type of condition with a `switch`:

```
yourNumber = input('Enter a number: ');

if yourNumber < 0
    disp('Negative')
elseif yourNumber > 0
    disp('Positive')
else
    disp('Zero')
end
```

Array Comparisons in Conditional Statements

It is important to understand how relational operators and `if` statements work with matrices. When you want to check for equality between two variables, you might use

```
if A == B, ...
```

This is valid MATLAB code, and does what you expect when `A` and `B` are scalars. But when `A` and `B` are matrices, `A == B` does not test *if* they are equal, it tests *where* they are

equal; the result is another matrix of 0s and 1s showing element-by-element equality. (In fact, if A and B are not the same size, then A == B is an error.)

```
A = magic(4);     B = A;     B(1,1) = 0;

A == B
ans =
     0     1     1     1
     1     1     1     1
     1     1     1     1
     1     1     1     1
```

The proper way to check for equality between two variables is to use the `isequal` function:

```
if isequal(A, B), ...
```

`isequal` returns a *scalar* logical value of `1` (representing `true`) or `0` (`false`), instead of a matrix, as the expression to be evaluated by the `if` function. Using the A and B matrices from above, you get

```
isequal(A, B)
ans =
    0
```

Here is another example to emphasize this point. If A and B are scalars, the following program will never reach the "unexpected situation". But for most pairs of matrices, including our magic squares with interchanged columns, none of the matrix conditions A > B, A < B, or A == B is true for *all* elements and so the `else` clause is executed:

```
if A > B
   'greater'
elseif A < B
   'less'
elseif A == B
   'equal'
else
   error('Unexpected situation')
end
```

Several functions are helpful for reducing the results of matrix comparisons to scalar conditions for use with `if`, including

```
isequal
```

```
isempty
all
any
```

Loop Control — for, while, continue, break

This section covers those MATLAB functions that provide control over program loops.

for

The `for` loop repeats a group of statements a fixed, predetermined number of times. A matching `end` delineates the statements:

```
for n = 3:32
   r(n) = rank(magic(n));
end
r
```

The semicolon terminating the inner statement suppresses repeated printing, and the `r` after the loop displays the final result.

It is a good idea to indent the loops for readability, especially when they are nested:

```
for i = 1:m
   for j = 1:n
      H(i,j) = 1/(i+j);
   end
end
```

while

The `while` loop repeats a group of statements an indefinite number of times under control of a logical condition. A matching `end` delineates the statements.

Here is a complete program, illustrating `while`, `if`, `else`, and `end`, that uses interval bisection to find a zero of a polynomial:

```
a = 0; fa = -Inf;
b = 3; fb = Inf;
while b-a > eps*b
   x = (a+b)/2;
   fx = x^3-2*x-5;
   if sign(fx) == sign(fa)
```

```
      a = x; fa = fx;
   else
      b = x; fb = fx;
   end
end
x
```

The result is a root of the polynomial $x^3 - 2x - 5$, namely

```
x =
   2.09455148154233
```

The cautions involving matrix comparisons that are discussed in the section on the `if` statement also apply to the `while` statement.

continue

The `continue` statement passes control to the next iteration of the `for` loop or `while` loop in which it appears, skipping any remaining statements in the body of the loop. The same holds true for `continue` statements in nested loops. That is, execution continues at the beginning of the loop in which the `continue` statement was encountered.

The example below shows a `continue` loop that counts the lines of code in the file `magic.m`, skipping all blank lines and comments. A `continue` statement is used to advance to the next line in `magic.m` without incrementing the count whenever a blank line or comment line is encountered:

```
fid = fopen('magic.m','r');
count = 0;
while ~feof(fid)
    line = fgetl(fid);
    if isempty(line) || strncmp(line,'%',1) || ~ischar(line)
        continue
    end
    count = count + 1;
end
fprintf('%d lines\n',count);
fclose(fid);
```

break

The `break` statement lets you exit early from a `for` loop or `while` loop. In nested loops, `break` exits from the innermost loop only.

Here is an improvement on the example from the previous section. Why is this use of `break` a good idea?

```
a = 0; fa = -Inf;
b = 3; fb = Inf;
while b-a > eps*b
   x = (a+b)/2;
   fx = x^3-2*x-5;
   if fx == 0
      break
   elseif sign(fx) == sign(fa)
      a = x; fa = fx;
   else
      b = x; fb = fx;
   end
end
x
```

Program Termination — return

This section covers the MATLAB `return` function that enables you to terminate your program before it runs to completion.

return

`return` terminates the current sequence of commands and returns control to the invoking function or to the keyboard. `return` is also used to terminate `keyboard` mode. A called function normally transfers control to the function that invoked it when it reaches the end of the function. You can insert a `return` statement within the called function to force an early termination and to transfer control to the invoking function.

Vectorization

One way to make your MATLAB programs run faster is to vectorize the algorithms you use in constructing the programs. Where other programming languages might use `for` loops or `DO` loops, MATLAB can use vector or matrix operations. A simple example involves creating a table of logarithms:

```
x = .01;
for k = 1:1001
   y(k) = log10(x);
   x = x + .01;
```

```
end
```

A vectorized version of the same code is

```
x = .01:.01:10;
y = log10(x);
```

For more complicated code, vectorization options are not always so obvious.

Preallocation

If you cannot vectorize a piece of code, you can make your `for` loops go faster by preallocating any vectors or arrays in which output results are stored. For example, this code uses the function `zeros` to preallocate the vector created in the `for` loop. This makes the `for` loop execute significantly faster:

```
r = zeros(32,1);
for n = 1:32
    r(n) = rank(magic(n));
end
```

Without the preallocation in the previous example, the MATLAB interpreter enlarges the `r` vector by one element each time through the loop. Vector preallocation eliminates this step and results in faster execution.

Scripts and Functions

In this section...

Overview

The MATLAB product provides a powerful programming language, as well as an interactive computational environment. You can enter commands from the language one at a time at the MATLAB command line, or you can write a series of commands to a file that you then execute as you would any MATLAB function. Use the MATLAB Editor or any other text editor to create your own function files. Call these functions as you would any other MATLAB function or command.

There are two kinds of program files:

- Scripts, which do not accept input arguments or return output arguments. They operate on data in the workspace.
- Functions, which can accept input arguments and return output arguments. Internal variables are local to the function.

If you are a new MATLAB programmer, just create the program files that you want to try out in the current folder. As you develop more of your own files, you will want to organize them into other folders and personal toolboxes that you can add to your MATLAB search path.

If you duplicate function names, MATLAB executes the one that occurs first in the search path.

To view the contents of a program file, for example, `myfunction.m`, use

```
type myfunction
```

Scripts

When you invoke a *script*, MATLAB simply executes the commands found in the file. Scripts can operate on existing data in the workspace, or they can create new data on which to operate. Although scripts do not return output arguments, any variables that they create remain in the workspace, to be used in subsequent computations. In addition, scripts can produce graphical output using functions like `plot`.

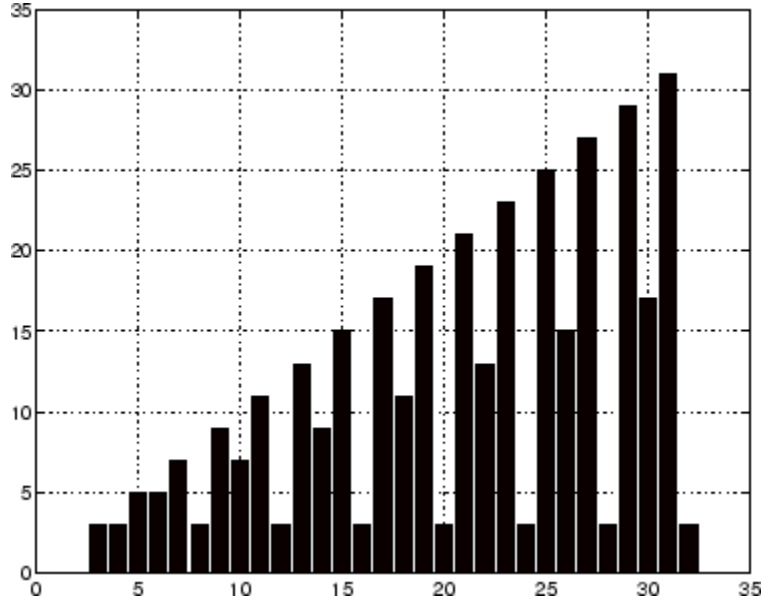
For example, create a file called `magicrank.m` that contains these MATLAB commands:

```
% Investigate the rank of magic squares
r = zeros(1,32);
for n = 3:32
   r(n) = rank(magic(n));
end
r
bar(r)
```

Typing the statement

```
magicrank
```

causes MATLAB to execute the commands, compute the rank of the first 30 magic squares, and plot a bar graph of the result. After execution of the file is complete, the variables `n` and `r` remain in the workspace.



Functions

Functions are files that can accept input arguments and return output arguments. The names of the file and of the function should be the same. Functions operate on variables within their own workspace, separate from the workspace you access at the MATLAB command prompt.

A good example is provided by `rank`. The file `rank.m` is available in the folder

```
toolbox/matlab/matfun
```

You can see the file with

```
type rank
```

Here is the file:

```
function r = rank(A,tol)
%   RANK Matrix rank.
%   RANK(A) provides an estimate of the number of linearly
%   independent rows or columns of a matrix A.
%   RANK(A,tol) is the number of singular values of A
%   that are larger than tol.
```

```
%   RANK(A) uses the default tol = max(size(A)) * norm(A) * eps.

s = svd(A);
if nargin==1
   tol = max(size(A)') * max(s) * eps;
end
r = sum(s > tol);
```

The first line of a function starts with the keyword `function`. It gives the function name and order of arguments. In this case, there are up to two input arguments and one output argument.

The next several lines, up to the first blank or executable line, are comment lines that provide the help text. These lines are printed when you type

```
help rank
```

The first line of the help text is the H1 line, which MATLAB displays when you use the `lookfor` command or request `help` on a folder.

The rest of the file is the executable MATLAB code defining the function. The variable `s` introduced in the body of the function, as well as the variables on the first line, `r`, `A` and `tol`, are all *local* to the function; they are separate from any variables in the MATLAB workspace.

This example illustrates one aspect of MATLAB functions that is not ordinarily found in other programming languages—a variable number of arguments. The `rank` function can be used in several different ways:

```
rank(A)
r = rank(A)
r = rank(A,1.e-6)
```

Many functions work this way. If no output argument is supplied, the result is stored in `ans`. If the second input argument is not supplied, the function computes a default value. Within the body of the function, two quantities named `nargin` and `nargout` are available that tell you the number of input and output arguments involved in each particular use of the function. The `rank` function uses `nargin`, but does not need to use `nargout`.

Types of Functions

MATLAB offers several different types of functions to use in your programming.

Anonymous Functions

An *anonymous function* is a simple form of the MATLAB function that is defined within a single MATLAB statement. It consists of a single MATLAB expression and any number of input and output arguments. You can define an anonymous function right at the MATLAB command line, or within a function or script. This gives you a quick means of creating simple functions without having to create a file for them each time.

The syntax for creating an anonymous function from an expression is

```
f = @(arglist)expression
```

The statement below creates an anonymous function that finds the square of a number. When you call this function, MATLAB assigns the value you pass in to variable `x`, and then uses `x` in the equation `x.^2`:

```
sqr = @(x) x.^2;
```

To execute the `sqr` function defined above, type

```
a = sqr(5)
a =
   25
```

Primary and Subfunctions

Any function that is not anonymous must be defined within a file. Each such function file contains a required *primary function* that appears first, and any number of *subfunctions* that can follow the primary. Primary functions have a wider scope than subfunctions. That is, primary functions can be called from outside of the file that defines them (for example, from the MATLAB command line or from functions in other files) while subfunctions cannot. Subfunctions are visible only to the primary function and other subfunctions within their own file.

The `rank` function shown in the section on “Functions” on page 5-11 is an example of a primary function.

Private Functions

A *private function* is a type of primary function. Its unique characteristic is that it is visible only to a limited group of other functions. This type of function can be useful if you

want to limit access to a function, or when you choose not to expose the implementation of a function.

Private functions reside in subfolders with the special name `private`. They are visible only to functions in the parent folder. For example, assume the folder `newmath` is on the MATLAB search path. A subfolder of `newmath` called `private` can contain functions that only the functions in `newmath` can call.

Because private functions are invisible outside the parent folder, they can use the same names as functions in other folders. This is useful if you want to create your own version of a particular function while retaining the original in another folder. Because MATLAB looks for private functions before standard functions, it will find a private function named `test.m` before a nonprivate file named `test.m`.

Nested Functions

You can define functions within the body of another function. These are said to be *nested* within the outer function. A nested function contains any or all of the components of any other function. In this example, function `B` is nested in function `A`:

```
function x = A(p1, p2)
...
B(p2)
   function y = B(p3)
   ...
   end
...
end
```

Like other functions, a nested function has its own workspace where variables used by the function are stored. But it also has access to the workspaces of all functions in which it is nested. So, for example, a variable that has a value assigned to it by the primary function can be read or overwritten by a function nested at any level within the primary. Similarly, a variable that is assigned in a nested function can be read or overwritten by any of the functions containing that function.

Global Variables

If you want more than one function to share a single copy of a variable, simply declare the variable as `global` in all the functions. Do the same thing at the command line if you want the base workspace to access the variable. The global declaration must occur before the variable is actually used in a function. Although it is not required, using capital

letters for the names of global variables helps distinguish them from other variables. For example, create a new function in a file called `falling.m`:

```
function h = falling(t)
global GRAVITY
h = 1/2*GRAVITY*t.^2;
```

Then interactively enter the statements

```
global GRAVITY
GRAVITY = 32;
y = falling((0:.1:5)');
```

The two global statements make the value assigned to `GRAVITY` at the command prompt available inside the function. You can then modify `GRAVITY` interactively and obtain new solutions without editing any files.

Command vs. Function Syntax

You can write MATLAB functions that accept character arguments without the parentheses and quotes. That is, MATLAB interprets

```
foo a b c
```

as

```
foo('a','b','c')
```

However, when you use the unquoted command form, MATLAB cannot return output arguments. For example,

```
legend apples oranges
```

creates a legend on a plot using `apples` and `oranges` as labels. If you want the `legend` command to return its output arguments, then you must use the quoted form:

```
[legh,objh] = legend('apples','oranges');
```

In addition, you must use the quoted form if any of the arguments is not a character vector.

Caution While the unquoted command syntax is convenient, in some cases it can be used incorrectly without causing MATLAB to generate an error.

Constructing Character Arguments in Code

The quoted function form enables you to construct character arguments within the code. The following example processes multiple data files, `August1.dat`, `August2.dat`, and so on. It uses the function `int2str`, which converts an integer to a character, to build the file name:

```
for d = 1:31
   s = ['August' int2str(d) '.dat'];
   load(s)
   % Code to process the contents of the d-th file
end
```