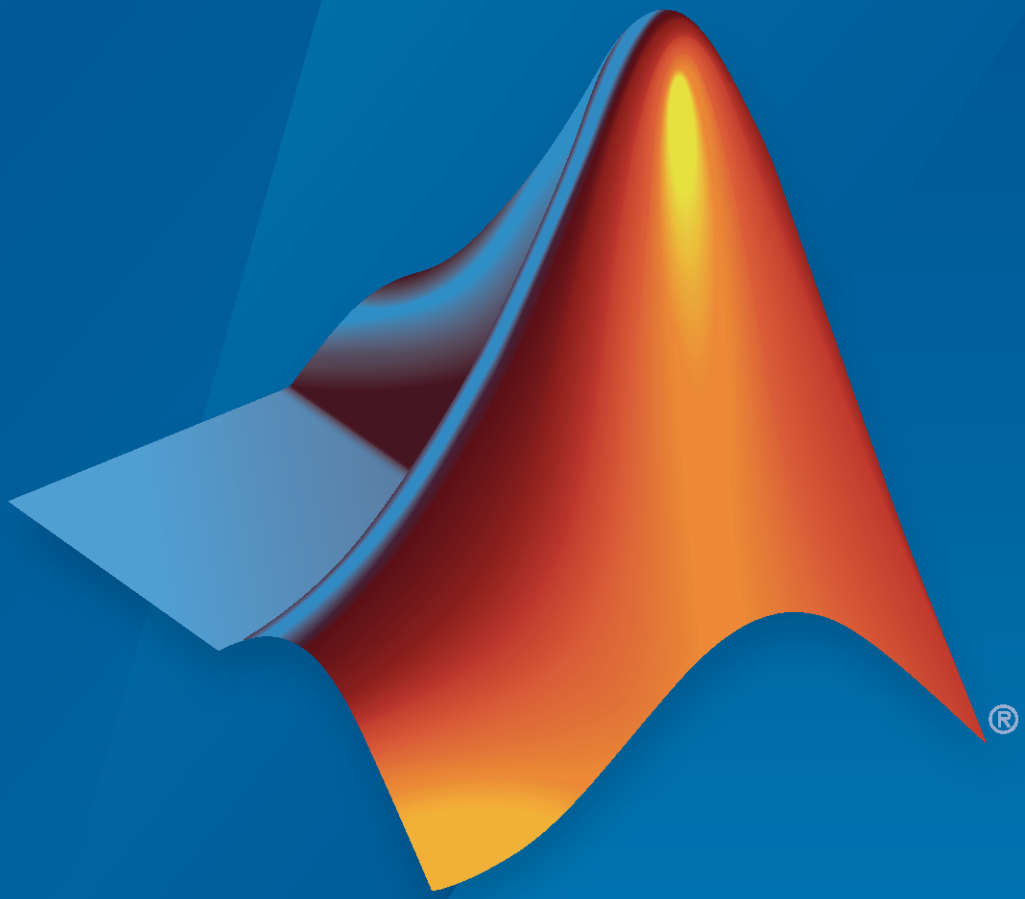
MATLAB®

Primer



MATLAB®

R2025a

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.
1 Apple Hill Drive
Natick, MA 01760-2098

MATLAB® Primer

Revision History

December 1996	First printing	For MATLAB 5
May 1997	Second printing	Revised for MATLAB 5.1
September 1998	Third printing	Revised 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 (Release 2010b)
April 2011	Online only	Revised for MATLAB 7.12 (Release 2011a)
September 2011	Seventeenth printing	Revised for MATLAB 7.13 (Release 2011b)
March 2012	Eighteenth printing	Revised for MATLAB 7.14 (Release 2012a) (Renamed from *MATLAB Getting Started Guide*)
September 2012	Nineteenth printing	Revised for MATLAB 8.0 (Release 2012b)
March 2013	Twentieth printing	Revised for MATLAB 8.1 (Release 2013a)
September 2013	Twenty-first printing	Revised for MATLAB 8.2 (Release 2013b)
March 2014	Twenty-second printing	Revised for MATLAB 8.3 (Release 2014a)
October 2014	Twenty-third printing	Revised for MATLAB 8.4 (Release 2014b)
March 2015	Twenty-fourth printing	Revised for MATLAB 8.5 (Release 2015a)
September 2015	Twenty-fifth printing	Revised for MATLAB 8.6 (Release 2015b)
March 2016	Twenty-sixth printing	Revised for MATLAB 9.0 (Release 2016a)
September 2016	Twenty-seventh printing	Revised for MATLAB 9.1 (Release 2016b)
March 2017	Twenty-eighth printing	Revised for MATLAB 9.2 (Release 2017a)
September 2017	Twenty-ninth printing	Revised for MATLAB 9.3 (Release 2017b)
March 2018	Thirtieth printing	Revised for MATLAB 9.4 (Release 2018a)
September 2018	Thirty-first printing	Revised for MATLAB 9.5 (Release 2018b)
March 2019	Online only	Revised for MATLAB 9.6 (Release 2019a)
September 2019	Online only	Revised for MATLAB 9.7 (Release 2019b)
March 2020	Online only	Revised for MATLAB 9.8 (Release 2020a)
September 2020	Online only	Revised for MATLAB 9.9 (Release 2020b)
March 2021	Online only	Revised for MATLAB 9.10 (Release 2021a)
September 2021	Online only	Revised for MATLAB 9.11 (Release 2021b)
March 2022	Online only	Revised for MATLAB 9.12 (Release 2022a)
September 2022	Online only	Revised for MATLAB 9.13 (Release 2022b)
March 2023	Online only	Revised for MATLAB 9.14 (Release 2023a)
September 2023	Online only	Revised for Version 23.2 (R2023b)
March 2024	Online only	Revised for Version 24.1 (R2024a)
September 2024	Online only	Revised for Version 24.2 (R2024b)
March 2025	Online only	Revised for Version 25.1 (R2025a)

Contents

1 Quick Start

MATLAB Product Description **1-2**
Key Features **1-2**

Desktop Basics **1-3**

Matrices and Arrays **1-5**

Array Indexing **1-9**

Workspace Variables **1-11**

Text and Characters **1-12**
Text in String Arrays **1-12**
Data in Character Arrays **1-12**

Calling Functions **1-14**

2-D and 3-D Plots **1-15**

Programming and Scripts **1-20**
Scripts **1-20**
Live Scripts **1-21**
Loops and Conditional Statements **1-21**
Script Locations **1-22**

Help and Documentation **1-23**

2 Language Fundamentals

Matrices and Magic Squares **2-2**
About Matrices **2-2**
Entering Matrices **2-3**
sum, transpose, and diag **2-4**
The magic Function **2-5**
Generating Matrices **2-6**

Matrix Operations **2-7**
Removing Rows or Columns from a Matrix **2-7**
Reshaping and Rearranging Arrays **2-7**

Array vs. Matrix Operations 2-12
Find Array Elements That Meet Conditions 2-16
Multidimensional Arrays 2-19

Data Types 2-27
Text in String and Character Arrays 2-27
Tables of Mixed Data 2-29
Access Data in Cell Array 2-34
Structure Arrays 2-39
Floating-Point Numbers 2-43
Integers 2-50

3 Mathematics

Linear Algebra 3-2
Matrices in the MATLAB Environment 3-2
Powers and Exponentials 3-10
Systems of Linear Equations 3-13
Eigenvalues 3-22
Singular Values 3-25

Create Arrays of Random Numbers 3-29
Random Number Functions 3-29
Random Number Generators 3-30
Random Number Data Types 3-30

Operations on Nonlinear Functions 3-32
Create Function Handle 3-32
Pass Function to Another Function 3-34

4 Graphics

Create 2-D Line Plot 4-2

Format and Annotate Charts 4-7
Add Title and Axis Labels to Chart 4-7
Specify Axis Limits 4-11
Specify Axis Tick Values and Labels 4-16
Add Legend to Graph 4-22

Combine Multiple Plots 4-29

Create Chart with Two y-Axes 4-36

Surface and Mesh Plots 4-43

5 Programming

Control Flow **5-2**
Conditional Control — if, else, switch **5-2**
Loop Control — for, while, continue, break **5-4**
Program Termination — return **5-6**
Vectorization **5-6**
Preallocation **5-6**

Scripts and Functions **5-8**
Overview **5-8**
Scripts **5-8**
Functions **5-9**
Types of Functions **5-10**
Global Variables **5-12**
Command vs. Function Syntax **5-12**

Quick Start

- “MATLAB Product Description” on page 1-2
- “Desktop Basics” on page 1-3
- “Matrices and Arrays” on page 1-5
- “Array Indexing” on page 1-9
- “Workspace Variables” on page 1-11
- “Text and Characters” on page 1-12
- “Calling Functions” on page 1-14
- “2-D and 3-D Plots” on page 1-15
- “Programming and Scripts” on page 1-20
- “Help and Documentation” on page 1-23

MATLAB Product Description

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.

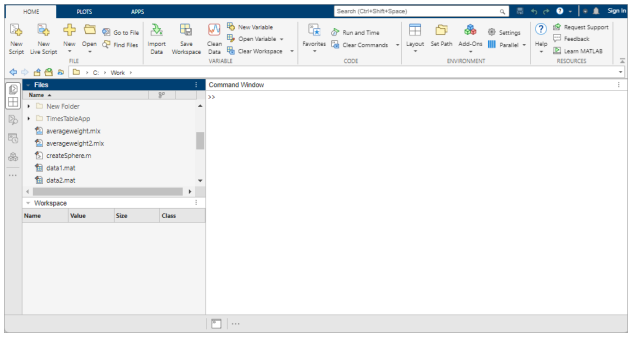
MATLAB helps you take your ideas beyond the desktop. You can run your analyses on larger data sets, and scale up to clusters and clouds. MATLAB code can be integrated with other languages, enabling you to deploy algorithms and applications within web, enterprise, and production systems.

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 areas:

- Files panel — Access your files.
- Workspace panel — Explore data that you create or import from files.
- Command Window — Enter commands at the command line, indicated by the prompt (>>).
- Sidebars — Access tools docked in the desktop and additional panels.

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.

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 = 1×4

     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 3 5; 2 4 6; 7 8 10]
```

```
a = 3×3

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

You can also define each row on its own line of code and separate the rows with a newline.

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

```
a = 3×3

     1     3     5
     2     4     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 = 5×1

     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 = 3×3

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

```
sin(a)
```

```
ans = 3×3

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

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

```
a'
```

```
ans = 3×3

     1     2     7
     3     4     8
     5     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 = 3×3

    1.0000    0.0000   -0.0000
         0    1.0000   -0.0000
         0    0.0000    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 = 3×3
```

```
   0.999999999999996   0.000000000000007  -0.000000000000002
                   0   1.000000000000000  -0.000000000000003
                   0   0.000000000000014   0.999999999999995
```

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 = 3×3

     1     9    25
     4    16    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 = 3×3

           1          27         125
           8          64         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 = 3×6

     1     3     5     1     3     5
     2     4     6     2     4     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 = 6×3

     1     3     5
     2     4     6
```

```
     7     8    10
     1     3     5
     2     4     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 = 2×2 complex

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

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 matrix `A`:

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

```
A = 4×4

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

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 in position 2 exceeds array bounds (must not exceed 4).
```

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 = 4×5

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

     2
     6
    10
```

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 = 1×5

     9    10    11    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 = 1×11

     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`.

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 panel 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
```

Text and Characters

Text in String Arrays

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

```
t = "Hello, world";
```

If the text includes double quotes, use two double quotes within the definition.

```
q = "Something ""quoted"" and something else."
```

```
q =

    "Something "quoted" and something else."
```

`t` and `q` are arrays, like all MATLAB variables. Their *class* or data type is `string`.

```
whos t
```

```
  Name        Size            Bytes  Class     Attributes
  t           1x1               174  string
```

To add text to the end of a string, use the plus operator, `+`.

```
f = 71;
c = (f-32)/1.8;
tempText = "Temperature is " + c + "C"
```

```
tempText =
"Temperature is 21.6667C"
```

Similar to numeric arrays, string arrays can have multiple elements. Use the `strlength` function to find the length of each string within an array.

```
A = ["a","bb","ccc"; "dddd","eeeeee","fffffff"]
```

```
A =
  2×3 string array
    "a"       "bb"        "ccc"
    "dddd"    "eeeeee"    "fffffff"
```

```
strlength(A)
```

```
ans =

     1     2     3
     4     6     7
```

Data in Character Arrays

Sometimes characters represent data that does not correspond to text, such as a DNA sequence. You can store this type of data in a character array, which has data type `char`. Character arrays use single quotes.

```
seq = 'GCTAGAATCC';
whos seq
```

```
  Name      Size            Bytes  Class    Attributes
  seq       1x10               20  char
```

Each element of the array contains a single character.

```
seq(4)
```

```
ans =
    'A'
```

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

```
seq2 = [seq 'ATTAGAAACC']
```

```
seq2 =
    'GCTAGAATCCATTAGAAACC'
```

Character arrays are common in programs that were written before the introduction of double quotes for string creation in R2017a. All MATLAB functions that accept `string` data also accept `char` data, and vice versa.

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 = [3 6 9];
union(A,B)
```

```
ans = 1×5

     1     3     5     6     9
```

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:

```
[minA,maxA] = bounds(A)
```

```
minA =
1

maxA =
5
```

Enclose any text inputs in 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
```

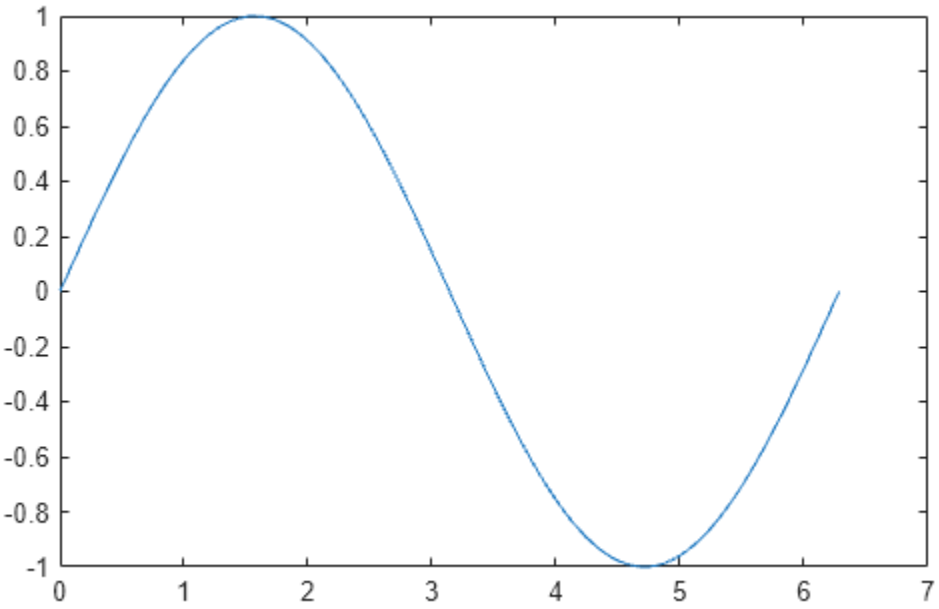
The `clc` function clears the Command Window.

2-D and 3-D Plots

Line Plots

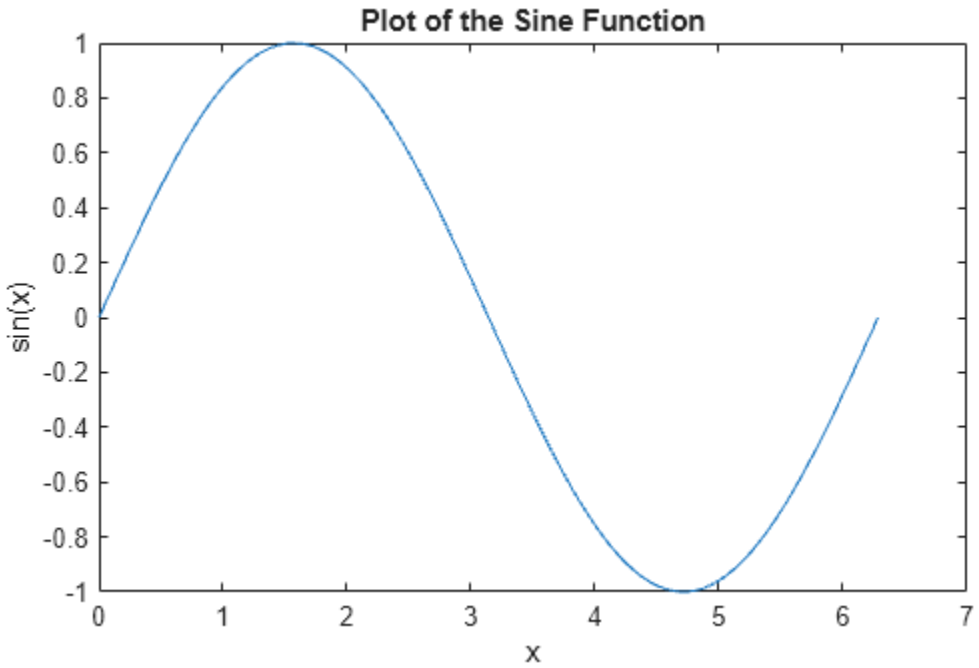
To create two-dimensional line plots, use the `plot` function. For example, plot the sine function over a linearly spaced vector of values from 0 to 2π:

```
x = linspace(0,2*pi);
y = sin(x);
plot(x,y)
```



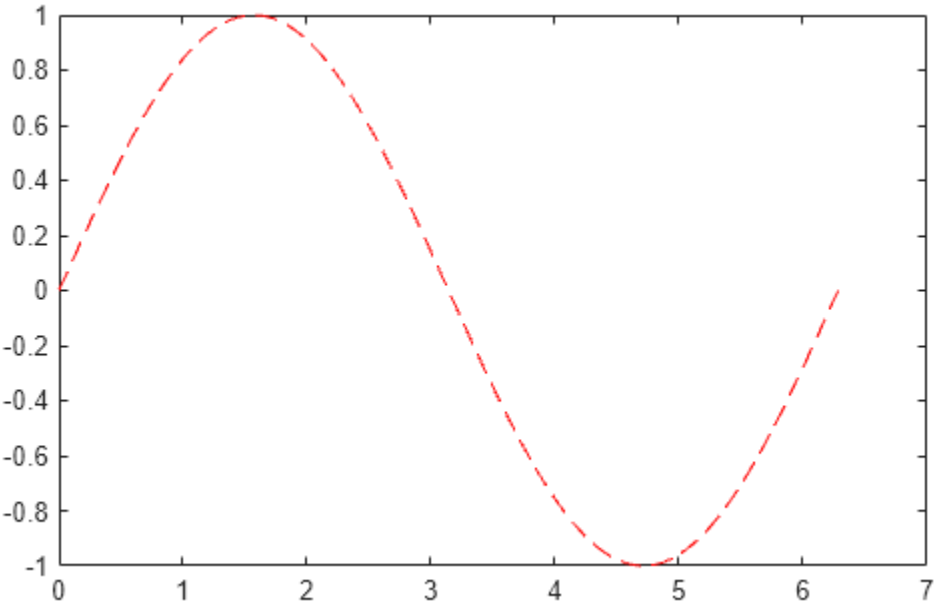
You can label the axes and add a title.

```
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--")
```

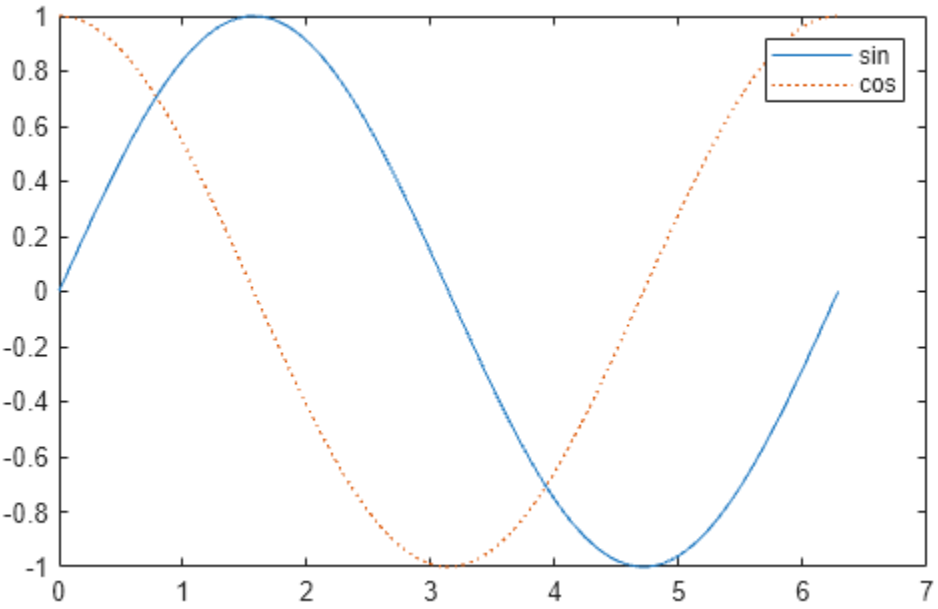


`"r--"` is a *line specification*. Each specification can include characters for the line color, style, and marker. A marker is a symbol that appears at each plotted data point, such as a `+`, `o`, or `*`. For example, `"g:*"` requests a dotted green line with * markers.

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

```
x = linspace(0,2*pi);
y = sin(x);
plot(x,y)

hold on

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

hold off
```



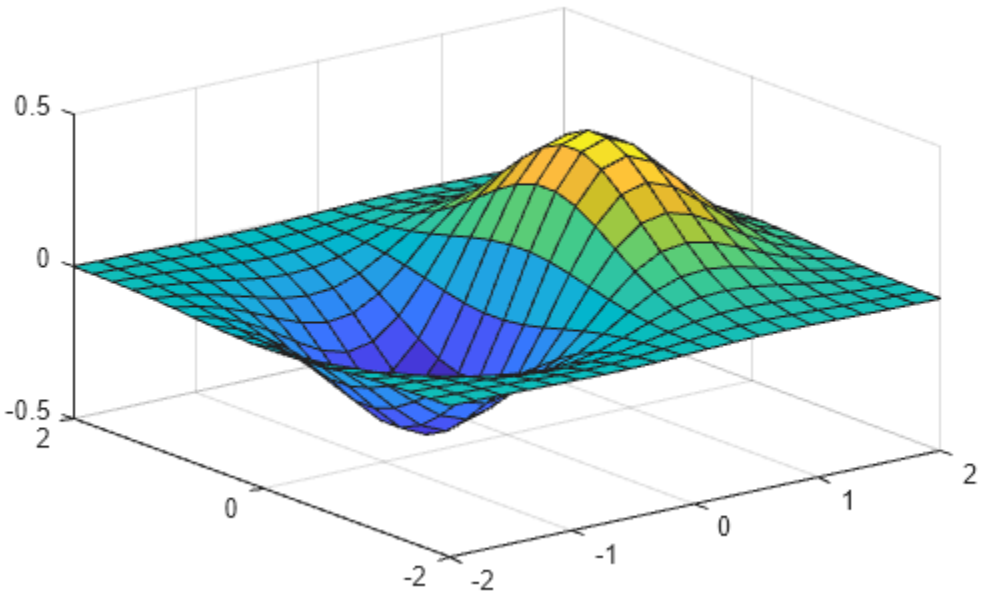
3-D Plots

Three-dimensional plots typically display a surface defined by a function in two variables, $z = f(x, y)$. For instance, calculate $z = xe^{-x^2 - y^2}$ given row and column vectors `x` and `y` with 20 points each in the range [-2,2].

```
x = linspace(-2,2,20);
y = x';
z = x .* exp(-x.^2 - y.^2);
```

Then, create a surface plot.

```
surf(x,y,z)
```



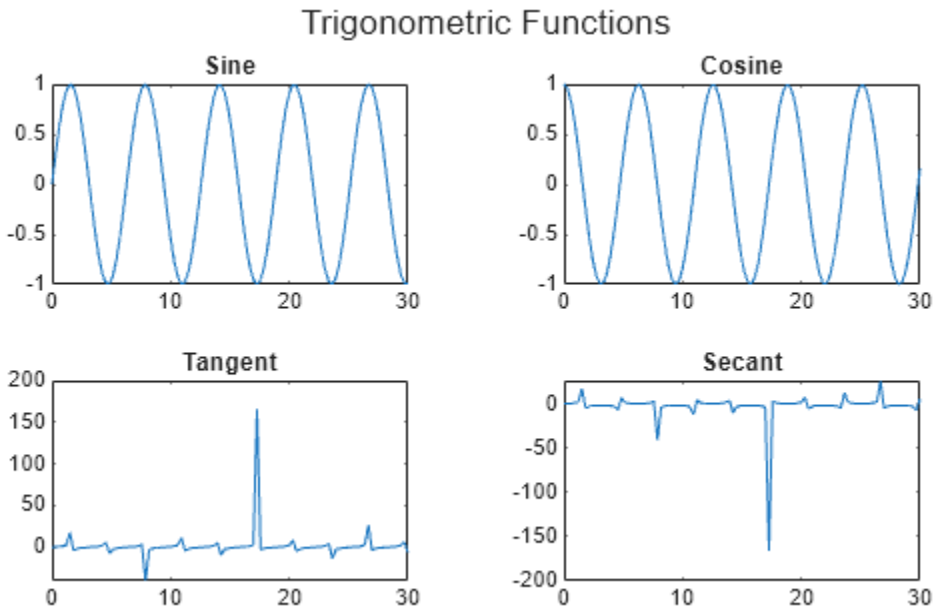
Both the `surf` function and its companion `mesh` display surfaces in three dimensions. `surf` displays both the connecting lines and the faces of the surface in color. `mesh` produces wireframe surfaces that color only the connecting lines.

Multiple Plots

You can display multiple plots in different parts of the same window using either `tiledlayout` or `subplot`.

The `tiledlayout` function was introduced in R2019b and provides more control over labels and spacing than `subplot`. For example, create a 2-by-2 layout within a figure window. Then, call `nexttile` each time you want a plot to appear in the next region.

```
t = tiledlayout(2,2);
title(t,"Trigonometric Functions")
x = linspace(0,30);

nexttile
plot(x,sin(x))
title("Sine")

nexttile
plot(x,cos(x))
title("Cosine")

nexttile
plot(x,tan(x))
title("Tangent")

nexttile
plot(x,sec(x))
title("Secant")
```



If you are using a release earlier than R2019b, see `subplot`.

Programming and Scripts

In this section...
“Scripts” on page 1-20
“Live Scripts” on page 1-21
“Loops and Conditional Statements” on page 1-21
“Script Locations” on page 1-22

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

Scripts

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

```
edit mysphere
```

This command opens a blank file named `mysphere.m`. Enter some code that creates a unit sphere, doubles the radius, and plots the results:

```
[x,y,z] = sphere;
r = 2;
surf(x*r,y*r,z*r)
axis equal
```

Next, add code that calculates the surface area and volume of a sphere:

```
A = 4*pi*r^2;
V = (4/3)*pi*r^3;
```

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

```
% Create and plot a sphere with radius r.
[x,y,z] = sphere;       % Create a unit sphere.
r = 2;
surf(x*r,y*r,z*r)       % Adjust each dimension and plot.
axis equal              % Use the same scale for each axis.

% Find the surface area and volume.
A = 4*pi*r^2;
V = (4/3)*pi*r^3;
```

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

```
mysphere
```

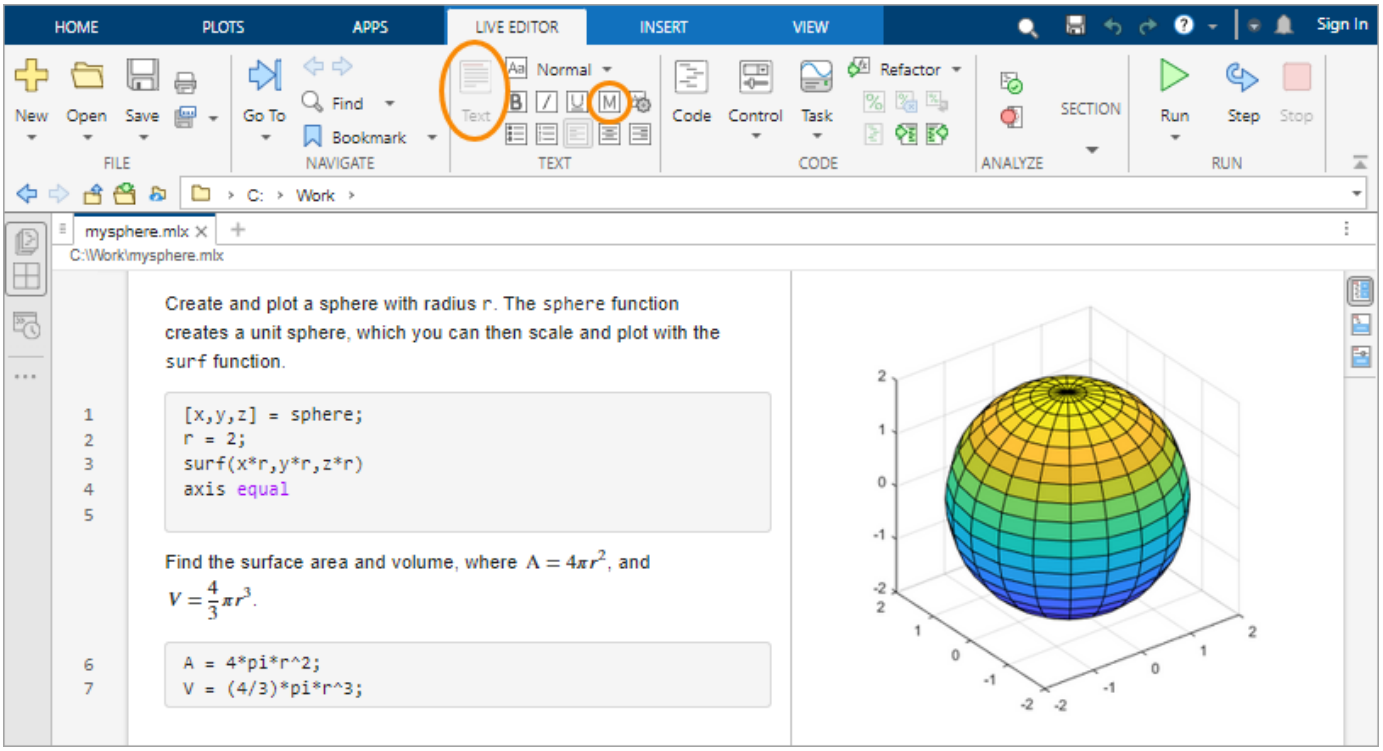
You can also run scripts from the Editor using the **Run** button,  .

Live Scripts

Instead of writing code and comments in plain text, you can use formatting options in *live scripts* to enhance your code. Live scripts allow you to view and interact with both code and output and can include formatted text, equations, and images.

For example, convert `mysphere` to a live script by selecting **Save As** and changing the file type to a MATLAB Live Code File (`*.mlx`). Then, replace the code comments with formatted text. For instance:

- Convert the comment lines to text. Select each line that begins with a percent symbol, and then select **Text**. Remove the percent symbols.
- Rewrite the text to replace the comments at the end of code lines. To apply a monospace font to function names in the text, select M. To add an equation, select **Equation** on the **Insert** tab.



To create a new live script using the `edit` command, include the `.mlx` extension with the file name:

```
edit newfile.mlx
```

Loops and Conditional Statements

Within any script, you can define sections of code that either repeat in a loop or conditionally execute. Loops use a `for` or `while` keyword, and conditional statements use `if` or `switch`.

Loops are useful for creating sequences. For example, create a script named `fibseq` that uses a `for` loop to calculate the first 100 numbers of the Fibonacci sequence. In this sequence, the first two numbers are 1, and each subsequent number is the sum of the previous two, $F_n = F_{n-1} + F_{n-2}$.

```
N = 100;
f(1) = 1;
```

```
f(2) = 1;

for n = 3:N
    f(n) = f(n-1) + f(n-2);
end
f(1:10)
```

When you run the script, the `for` statement defines a counter named `n` that starts at 3. Then, the loop repeatedly assigns to `f(n)`, incrementing `n` on each execution until it reaches 100. The last command in the script, `f(1:10)`, displays the first 10 elements of `f`.

```
ans =
     1     1     2     3     5     8    13    21    34    55
```

Conditional statements execute only when given expressions are true. For example, assign a value to a variable depending on the size of a random number: `'low'`, `'medium'`, or `'high'`. In this case, the random number is an integer between 1 and 100.

```
num = randi(100)
if num < 34
   sz = 'low'
elseif num < 67
   sz = 'medium'
else
   sz = 'high'
end
```

The statement `sz = 'high'` only executes when `num` is greater than or equal to 67.

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 Files panel, right-click, and then select **Add to Path**.

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.

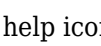
  ```
  doc mean
  ```

- 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  .

Language Fundamentals

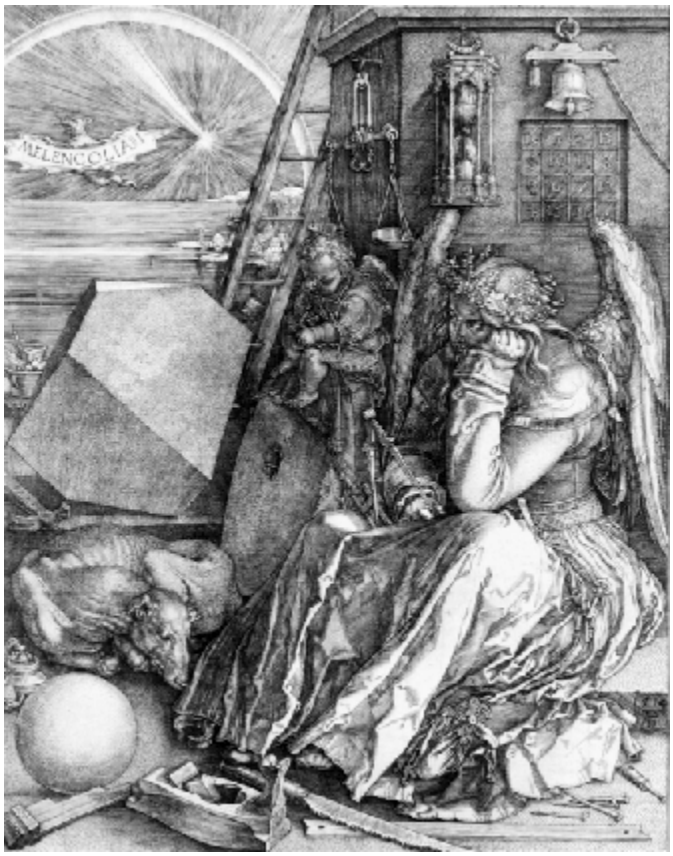
- “Matrices and Magic Squares” on page 2-2
- “Matrix Operations” on page 2-7
- “Data Types” on page 2-27

Matrices and Magic Squares

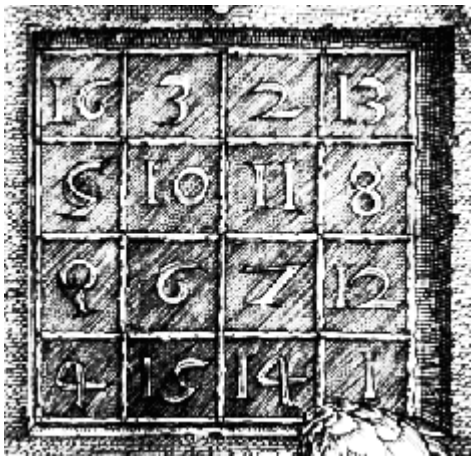
In this section...
“About Matrices” on page 2-2
“Entering Matrices” on page 2-3
“sum, transpose, and diag” on page 2-4
“The magic Function” on page 2-5
“Generating Matrices” on page 2-6

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 functions that generate basic matrices.

`zeros`	All zeros
`ones`	All ones
`rand`	Uniformly distributed random numbers
`randn`	Normally distributed random numbers
`randi`	Uniformly distributed random integers

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

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

N = randi([1,10],2,5)
N =
     5     8     7     9     7
    10    10     1    10     8
```

Matrix Operations

In this section...
“Removing Rows or Columns from a Matrix” on page 2-7
“Reshaping and Rearranging Arrays” on page 2-7
“Array vs. Matrix Operations” on page 2-12
“Find Array Elements That Meet Conditions” on page 2-16
“Multidimensional Arrays” on page 2-19

Removing Rows or Columns from a Matrix

The easiest way to remove a row or column from a matrix is to set that row or column equal to a pair of empty square brackets `[]`. For example, create a 4-by-4 matrix and remove the second row.

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

```
A = 4×4

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

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

```
A = 3×4

    16     2     3    13
     9     7     6    12
     4    14    15     1
```

Now remove the third column.

```
A(:,3) = []
```

```
A = 3×3

    16     2    13
     9     7    12
     4    14     1
```

You can extend this approach to any array. For example, create a random 3-by-3-by-3 array and remove all of the elements in the first matrix of the third dimension.

```
B = rand(3,3,3);
B(:,:,1) = [];
```

Reshaping and Rearranging Arrays

Many functions in MATLAB® can take the elements of an existing array and put them in a different shape or sequence. This can be helpful for preprocessing your data for subsequent computations or analyzing the data.

Reshaping

The `reshape` function changes the size and shape of an array. For example, reshape a 3-by-4 matrix to a 2-by-6 matrix.

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

```
A = 3×4

     1     4     7    10
     2     5     8    11
     3     6     9    12
```

```
B = reshape(A,2,6)
```

```
B = 2×6

     1     3     5     7     9    11
     2     4     6     8    10    12
```

As long as the number of elements in each shape are the same, you can reshape them into an array with any number of dimensions. Using the elements from `A`, create a 2-by-2-by-3 multidimensional array.

```
C = reshape(A,2,2,3)
```

```
C =
C(:,:,1) =

     1     3
     2     4


C(:,:,2) =

     5     7
     6     8


C(:,:,3) =

     9    11
    10    12
```

Transposing and Flipping

A common task in linear algebra is to work with the transpose of a matrix, which turns the rows into columns and the columns into rows. To do this, use the `transpose` function or the `.'` operator.

Create a 3-by-3 matrix and compute its transpose.

```
A = magic(3)
```

```
A = 3×3

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

```
B = A.'
```

```
B = 3×3

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

A similar operator ' computes the conjugate transpose for complex matrices. This operation computes the complex conjugate of each element and transposes it. Create a 2-by-2 complex matrix and compute its conjugate transpose.

```
A = [1+i 1-i; -i i]
```

```
A = 2×2 complex

   1.0000 + 1.0000i   1.0000 - 1.0000i
   0.0000 - 1.0000i   0.0000 + 1.0000i
```

```
B = A'
```

```
B = 2×2 complex

   1.0000 - 1.0000i   0.0000 + 1.0000i
   1.0000 + 1.0000i   0.0000 - 1.0000i
```

`flipud` flips the rows of a matrix in an up-to-down direction, and `fliplr` flips the columns in a left-to-right direction.

```
A = [1 2; 3 4]
```

```
A = 2×2

     1     2
     3     4
```

```
B = flipud(A)
```

```
B = 2×2

     3     4
     1     2
```

```
C = fliplr(A)
```

```
C = 2×2

     2     1
```

```
     4     3
```

Shifting and Rotating

You can shift elements of an array by a certain number of positions using the `circshift` function. For example, create a 3-by-4 matrix and shift its columns to the right by 2. The second argument `[0 2]` tells `circshift` to shift the rows 0 places and shift the columns 2 places to the right.

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

```
A = 3×4

     1     2     3     4
     5     6     7     8
     9    10    11    12
```

```
B = circshift(A,[0 2])
```

```
B = 3×4

     3     4     1     2
     7     8     5     6
    11    12     9    10
```

To shift the rows of `A` up by 1 and keep the columns in place, specify the second argument as `[-1 0]`.

```
C = circshift(A,[-1 0])
```

```
C = 3×4

     5     6     7     8
     9    10    11    12
     1     2     3     4
```

The `rot90` function can rotate a matrix counterclockwise by 90 degrees.

```
A = [1 2; 3 4]
```

```
A = 2×2

     1     2
     3     4
```

```
B = rot90(A)
```

```
B = 2×2

     2     4
     1     3
```

If you rotate 3 more times by using the second argument to specify the number of rotations, you end up with the original matrix `A`.

```
C = rot90(B,3)
```

```
C = 2×2

     1     2
     3     4
```

Sorting

Sorting the data in an array is also a valuable tool, and MATLAB offers a number of approaches. For example, the `sort` function sorts the elements of each row or column of a matrix separately in ascending or descending order. Create a matrix `A` and sort each column of `A` in ascending order.

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

```
A = 4×4

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

```
B = sort(A)
```

```
B = 4×4

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

Sort each row in descending order. The second argument value `2` specifies that you want to sort row-wise.

```
C = sort(A,2,'descend')
```

```
C = 4×4

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

To sort entire rows or columns relative to each other, use the `sortrows` function. For example, sort the rows of `A` in ascending order according to the elements in the first column. The positions of the rows change, but the order of the elements in each row are preserved.

```
D = sortrows(A)
```

```
D = 4×4

     4    14    15     1
     5    11    10     8
     9     7     6    12
```

```
    16     2     3    13
```

Array vs. Matrix Operations

Introduction

MATLAB has two different types of arithmetic operations: array operations and matrix operations. You can use these arithmetic operations to perform numeric computations, for example, adding two numbers, raising the elements of an array to a given power, or multiplying two matrices.

Matrix operations follow the rules of linear algebra. By contrast, array operations execute element by element operations and support multidimensional arrays. The period character (`.`) distinguishes the array operations from the matrix operations. However, since the matrix and array operations are the same for addition and subtraction, the character pairs `.+` and `.-` are unnecessary.

Array Operations

Array operations execute element by element operations on corresponding elements of vectors, matrices, and multidimensional arrays. If the operands have the same size, then each element in the first operand gets matched up with the element in the same location in the second operand. If the operands have compatible sizes, then each input is implicitly expanded as needed to match the size of the other.

As a simple example, you can add two vectors with the same size.

```
A = [1 1 1]

A =

     1     1     1

B = [1 2 3]

B =

     1     2     3

A+B

ans =

     2     3     4
```

If one operand is a scalar and the other is not, then MATLAB implicitly expands the scalar to be the same size as the other operand. For example, you can compute the element-wise product of a scalar and a matrix.

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

A =

     1     2     3
     1     2     3

3.*A
```

```
ans =

     3     6     9
     3     6     9
```

Implicit expansion also works if you subtract a 1-by-3 vector from a 3-by-3 matrix because the two sizes are compatible. When you perform the subtraction, the vector is implicitly expanded to become a 3-by-3 matrix.

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

A =

     1     1     1
     2     2     2
     3     3     3

m = [2 4 6]

m =

     2     4     6

A - m

ans =

    -1    -3    -5
     0    -2    -4
     1    -1    -3
```

A row vector and a column vector have compatible sizes. If you add a 1-by-3 vector to a 2-by-1 vector, then each vector implicitly expands into a 2-by-3 matrix before MATLAB executes the element-wise addition.

```
x = [1 2 3]

x =

     1     2     3

y = [10; 15]

y =

    10
    15

x + y

ans =

    11    12    13
    16    17    18
```

If the sizes of the two operands are incompatible, then you get an error.

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

A =
```

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

m = [2 4]

m =

     2     4

A - m

Arrays have incompatible sizes for this operation.
```

For more information, see “Compatible Array Sizes for Basic Operations”.

The following table provides a summary of arithmetic array operators in MATLAB. For function-specific information, click the link to the function reference page in the last column.

Operator	Purpose	Description	Reference Page
`+`	Addition	`A+B` adds `A` and `B`.	`plus`
`+`	Unary plus	`+A` returns `A`.	`uplus`
`-`	Subtraction	`A-B` subtracts `B` from `A`	`minus`
`-`	Unary minus	`-A` negates the elements of `A`.	`uminus`
`.*`	Element-wise multiplication	`A.*B` is the element-by-element product of `A` and `B`.	`times`
`.^`	Element-wise power	`A.^B` is the matrix with elements `A(i,j)` to the `B(i,j)` power.	`power`
`./`	Right array division	`A./B` is the matrix with elements `A(i,j)/B(i,j)`.	`rdivide`
`.\`	Left array division	`A.\B` is the matrix with elements `B(i,j)/A(i,j)`.	`ldivide`
`.'`	Array transpose	`A.'` is the array transpose of `A`. For complex matrices, this does not involve conjugation.	`transpose`

Matrix Operations

Matrix operations follow the rules of linear algebra and are not compatible with multidimensional arrays. The required size and shape of the inputs in relation to one another depends on the operation. For nonscalar inputs, the matrix operators generally calculate different answers than their array operator counterparts.

For example, if you use the matrix right division operator, `/`, to divide two matrices, the matrices must have the same number of columns. But if you use the matrix multiplication operator, `*`, to multiply two matrices, then the matrices must have a common *inner dimension*. That is, the number of columns in the first input must be equal to the number of rows in the second input. The matrix multiplication operator calculates the product of two matrices with the formula,

$$C(i,j) = \sum_{k=1}^{n} A(i,k)B(k,j).$$

To see this, you can calculate the product of two matrices.

```
A = [1 3;2 4]

A =

     1     3
     2     4

B = [3 0;1 5]

B =

     3     0
     1     5

A*B

ans =

     6    15
    10    20
```

The previous matrix product is not equal to the following element-wise product.

```
A.*B

ans =

     3     0
     2    20
```

The following table provides a summary of matrix arithmetic operators in MATLAB. For function-specific information, click the link to the function reference page in the last column.

Operator	Purpose	Description	Reference Page
*	Matrix multiplication	`C = A*B` is the linear algebraic product of the matrices `A` and `B`. The number of columns of `A` must equal the number of rows of `B`.	`mtimes`
\	Matrix left division	`x = A\B` is the solution to the equation $Ax = B$. Matrices `A` and `B` must have the same number of rows.	`mldivide`
/	Matrix right division	`x = B/A` is the solution to the equation $xA = B$. Matrices `A` and `B` must have the same number of columns. In terms of the left division operator, `B/A = (A'\B')'`.	`mrdivide`
^	Matrix power	`A^B` is `A` to the power `B`, if `B` is a scalar. For other values of `B`, the calculation involves eigenvalues and eigenvectors.	`mpower`
'	Complex conjugate transpose	`A'` is the linear algebraic transpose of `A`. For complex matrices, this is the complex conjugate transpose.	`ctranspose`

Find Array Elements That Meet Conditions

This example shows how to filter the elements of an array by applying conditions to the array. For instance, you can examine the even elements in a matrix, find the location of all 0s in a multidimensional array, or replace `NaN` values in data. You can perform these tasks using a combination of the relational and logical operators. The relational operators (>, <, >=, <=, ==, ~=) impose conditions on the array, and you can apply multiple conditions by connecting them with the logical operators `and`, `or`, and `not`, respectively denoted by the symbols &, |, and ~.

Apply Single Condition

To apply a single condition, start by creating a 5-by-5 matrix that contains random integers between 1 and 15. Reset the random number generator to the default state for reproducibility.

```
rng("default")
A = randi(15,5)
```

```
A = 5×5

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

Use the relational *less than* operator, <, to determine which elements of `A` are less than 9. Store the result in `B`.

```
B = A < 9
```

```
B = 5×5 logical array

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

The result is a logical matrix. Each value in `B` is a logical `1` (`true`) or logical `0` (`false`) that indicates whether the corresponding element of `A` fulfills the condition `A < 9`. For example, `A(1,1)` is `13`, so `B(1,1)` is logical `0` (`false`). However, `A(1,2)` is `2`, so `B(1,2)` is logical `1` (`true`).

Although `B` contains information about *which* elements in `A` are less than 9, `B` does not tell you what their *values* are. Rather than comparing the two matrices element by element, you can use `B` to index into `A`.

```
A(B)
```

```
ans = 8×1

     2
     2
     5
     3
```

```
     8
     3
     7
     1
```

The result is a column vector of the elements in `A` that are less than 9. Since `B` is a logical matrix, this operation is called *logical indexing*. In this case, the logical array being used as an index is the same size as the array it is indexing, but this is not a requirement. For more information, see “Array Indexing”.

Some problems require information about the *locations* of the array elements that meet a condition rather than their actual values. In this example, you can use the `find` function to locate all of the elements in `A` less than 9.

```
I = find(A < 9)
```

```
I = 8×1

     3
     6
     7
    11
    14
    16
    17
    22
```

The result is a column vector of linear indices. Each index describes the location of an element in `A` that is less than 9, so in practice `A(I)` returns the same result as `A(B)`. The difference is that `A(B)` uses logical indexing, whereas `A(I)` uses linear indexing.

Apply Multiple Conditions

You can use the logical `and`, `or`, and `not` operators to apply any number of conditions to an array; the number of conditions is not limited to one or two.

First, use the logical `and` operator, denoted &, to specify two conditions: the elements must be **less than 9** and **greater than 2**. Specify the conditions as a logical index to view the elements that satisfy both conditions.

```
A(A<9 & A>2)
```

```
ans = 5×1

     5
     3
     8
     3
     7
```

The result is a list of the elements in `A` that satisfy both conditions. Be sure to specify each condition with a separate statement connected by a logical operator. For example, you cannot specify the conditions above using `A(2<A<9)` because it evaluates to `A(2<A | A<9)`.

Next, find the elements in `A` that are **less than 9** and **even**.

```
A(A<9 & ~mod(A,2))

ans = 3×1

     2
     2
     8
```

The result is a list of all even elements in `A` that are less than 9. The use of the logical `not` operator, `~`, converts the matrix `mod(A,2)` into a logical matrix, with a value of logical `1` (`true`) located where an element is divisible by 2 or even.

Finally, find the elements in `A` that are **less than 9** and **even** and **not equal to 2**.

```
A(A<9 & ~mod(A,2) & A~=2)

ans =
8
```

The result, 8, is less than 9, even, and not equal to 2. It is the only element in `A` that satisfies all three conditions.

Use the `find` function to get the index of the element equal to 8 that satisfies the conditions.

```
find(A<9 & ~mod(A,2) & A~=2)

ans =
14
```

The result indicates that `A(14) = 8`.

Replace Values That Meet Condition

Sometimes it is useful to simultaneously change the values of several existing array elements. Use logical indexing with a simple assignment statement to replace the values in an array that meet a condition.

For example, replace all values in `A` that are greater than 10 with the number 10.

```
A(A>10) = 10

A = 5×5

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

Next, replace all values in `A` that are not equal to 10 with a `NaN` value.

```
A(A~=10) = NaN

A = 5×5

    10   NaN   NaN   NaN    10
    10   NaN    10   NaN   NaN
```

```
   NaN   NaN    10    10    10
    10    10   NaN    10    10
    10    10    10    10    10
```

Lastly, replace all of the `NaN` values in `A` with zeros and apply the logical `not` operator on `A`, `~A`.

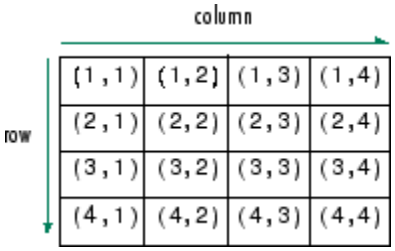
```
A(isnan(A)) = 0;
C = ~A
```

```
C = 5×5 logical array

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

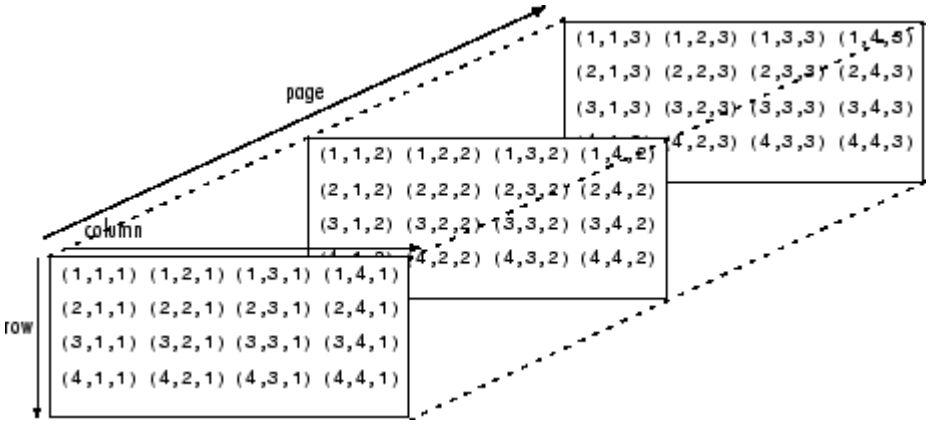
The resulting matrix has values of logical `1` (`true`) in place of the `NaN` values, and logical `0` (`false`) in place of the 10s. The logical `not` operation, `~A`, converts the numeric array `A` into a logical array `C` such that `A&C` returns a matrix of logical `0` (`false`) values and `A|C` returns a matrix of logical `1` (`true`) values.

Multidimensional Arrays

A multidimensional array in MATLAB® is an array with more than two dimensions. In a matrix, the two dimensions are represented by rows and columns.



Each element is defined by two subscripts, the row index and the column index. Multidimensional arrays are an extension of 2-D matrices and use additional subscripts for indexing. A 3-D array, for example, uses three subscripts. The first two are just like a matrix, but the third dimension represents *pages* or *sheets* of elements.



Creating Multidimensional Arrays

You can create a multidimensional array by creating a 2-D matrix first, and then extending it. For example, first define a 3-by-3 matrix as the first page in a 3-D array.

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

```
A = 3×3

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

Now add a second page. To do this, assign another 3-by-3 matrix to the index value 2 in the third dimension. The syntax `A(:,:,2)` uses a colon in the first and second dimensions to include all rows and all columns from the right-hand side of the assignment.

```
A(:,:,2) = [10 11 12; 13 14 15; 16 17 18]
```

```
A =
A(:,:,1) =

     1     2     3
     4     5     6
     7     8     9


A(:,:,2) =

    10    11    12
    13    14    15
    16    17    18
```

The `cat` function can be a useful tool for building multidimensional arrays. For example, create a new 3-D array `B` by concatenating `A` with a third page. The first argument indicates which dimension to concatenate along.

```
B = cat(3,A,[3 2 1; 0 9 8; 5 3 7])
```

```
B =
B(:,:,1) =
```

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


B(:,:,2) =

    10    11    12
    13    14    15
    16    17    18


B(:,:,3) =

     3     2     1
     0     9     8
     5     3     7
```

Another way to quickly expand a multidimensional array is by assigning a single element to an entire page. For example, add a fourth page to `B` that contains all zeros.

```
B(:,:,4) = 0

B =
B(:,:,1) =

     1     2     3
     4     5     6
     7     8     9


B(:,:,2) =

    10    11    12
    13    14    15
    16    17    18


B(:,:,3) =

     3     2     1
     0     9     8
     5     3     7


B(:,:,4) =

     0     0     0
     0     0     0
     0     0     0
```

Accessing Elements

To access elements in a multidimensional array, use integer subscripts just as you would for vectors and matrices. For example, find the 1,2,2 element of `A`, which is in the first row, second column, and second page of `A`.

```
A
```

```
A =
A(:,:,1) =

     1     2     3
     4     5     6
     7     8     9


A(:,:,2) =

    10    11    12
    13    14    15
    16    17    18
```

```
elA = A(1,2,2)
```

```
elA =
11
```

Use the index vector `[1 3]` in the second dimension to access only the first and last columns of each page of `A`.

```
C = A(:,[1 3],:)
```

```
C =
C(:,:,1) =

     1     3
     4     6
     7     9


C(:,:,2) =

    10    12
    13    15
    16    18
```

To find the second and third rows of each page, use the colon operator to create your index vector.

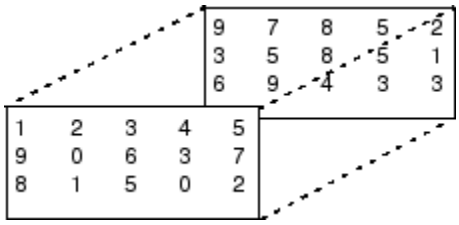
```
D = A(2:3,:,:)
```

```
D =
D(:,:,1) =

     4     5     6
     7     8     9
```

```
D(:,:,2) =

    13    14    15
    16    17    18
```

Manipulating Arrays

Elements of multidimensional arrays can be moved around in many ways, similar to vectors and matrices. `reshape`, `permute`, and `squeeze` are useful functions for rearranging elements. Consider a 3-D array with two pages.



Reshaping a multidimensional array can be useful for performing certain operations or visualizing the data. Use the `reshape` function to rearrange the elements of the 3-D array into a 6-by-5 matrix.

```
A = [1 2 3 4 5; 9 0 6 3 7; 8 1 5 0 2];
A(:,:,2) = [9 7 8 5 2; 3 5 8 5 1; 6 9 4 3 3];
B = reshape(A,[6 5])
```

```
B = 6×5

     1     3     5     7     5
     9     6     7     5     5
     8     5     2     9     3
     2     4     9     8     2
     0     3     3     8     1
     1     0     6     4     3
```

`reshape` operates columnwise, creating the new matrix by taking consecutive elements down each column of `A`, starting with the first page then moving to the second page.

Permutations are used to rearrange the order of the dimensions of an array. Consider a 3-D array `M`.

```
M(:,:,1) = [1 2 3; 4 5 6; 7 8 9];
M(:,:,2) = [0 5 4; 2 7 6; 9 3 1]
```

```
M =
M(:,:,1) =

     1     2     3
     4     5     6
     7     8     9


M(:,:,2) =

     0     5     4
     2     7     6
```

```
     9     3     1
```

Use the `permute` function to interchange row and column subscripts on each page by specifying the order of dimensions in the second argument. The original rows of `M` are now columns, and the columns are now rows.

```
P1 = permute(M,[2 1 3])
```

```
P1 =
P1(:,:,1) =

     1     4     7
     2     5     8
     3     6     9


P1(:,:,2) =

     0     2     9
     5     7     3
     4     6     1
```

Similarly, interchange row and page subscripts of `M`.

```
P2 = permute(M,[3 2 1])
```

```
P2 =
P2(:,:,1) =

     1     2     3
     0     5     4


P2(:,:,2) =

     4     5     6
     2     7     6


P2(:,:,3) =

     7     8     9
     9     3     1
```

When working with multidimensional arrays, you might encounter one that has an unnecessary dimension of length 1. The `squeeze` function performs another type of manipulation that eliminates dimensions of length 1. For example, use the `repmat` function to create a 2-by-3-by-1-by-4 array whose elements are each 5, and whose third dimension has length 1.

```
A = repmat(5,[2 3 1 4])
```

```
A =
A(:,:,1,1) =

     5     5     5
```

```
     5     5     5


A(:,:,1,2) =

     5     5     5
     5     5     5


A(:,:,1,3) =

     5     5     5
     5     5     5


A(:,:,1,4) =

     5     5     5
     5     5     5
```

```
szA = size(A)
```

```
szA = 1×4

     2     3     1     4
```

```
numdimsA = ndims(A)
```

```
numdimsA =
4
```

Use the `squeeze` function to remove the third dimension, resulting in a 3-D array.

```
B = squeeze(A)
```

```
B =
B(:,:,1) =

     5     5     5
     5     5     5


B(:,:,2) =

     5     5     5
     5     5     5


B(:,:,3) =

     5     5     5
     5     5     5


B(:,:,4) =

     5     5     5
```

```
     5     5     5
```

```
szB = size(B)
```

```
szB = 1×3

     2     3     4
```

```
numdimsB = ndims(B)
```

```
numdimsB =
3
```

Data Types

In this section...
“Text in String and Character Arrays” on page 2-27
“Tables of Mixed Data” on page 2-29
“Access Data in Cell Array” on page 2-34
“Structure Arrays” on page 2-39
“Floating-Point Numbers” on page 2-43
“Integers” on page 2-50

Text in String and Character Arrays

There are two ways to represent text in MATLAB®. You can store text in string arrays and in character vectors. MATLAB displays strings with double quotes and character vectors with single quotes.

Represent Text with String Arrays

You can store any 1-by-n sequence of characters as a string, using the `string` data type. Enclose text in double quotes to create a string.

```
str = "Hello, world"
```

```
str =
"Hello, world"
```

Though the text `"Hello, world"` is 12 characters long, `str` itself is a 1-by-1 string, or *string scalar*. You can use a string scalar to specify a file name, plot label, or any other piece of textual information.

To find the number of characters in a string, use the `strlength` function.

```
n = strlength(str)
```

```
n =
12
```

If the text includes double quotes, use two double quotes within the definition.

```
str = "They said, ""Welcome!"" and waved."
```

```
str =
"They said, "Welcome!" and waved."
```

To add text to the end of a string, use the plus operator, `+`. If a variable can be converted to a string, then `plus` converts it and appends it.

```
fahrenheit = 71;
celsius = (fahrenheit-32)/1.8;
tempText = "temperature is " + celsius + "C"
```

```
tempText =
"temperature is 21.6667C"
```

You can also concatenate text using the `append` function.

```
tempText2 = append("Today's ",tempText)
```

```
tempText2 =
"Today's temperature is 21.6667C"
```

The `string` function can convert different types of inputs, such as numeric, datetime, duration, and categorical values. For example, convert the output of `pi` to a string.

```
ps = string(pi)
```

```
ps =
"3.1416"
```

You can store multiple pieces of text in a string array. Each element of the array can contain a string having a different number of characters, without padding.

```
str = ["Mercury","Gemini","Apollo";...
       "Skylab","Skylab B","ISS"]
```

```
str = 2×3 string
    "Mercury"    "Gemini"      "Apollo"
    "Skylab"     "Skylab B"    "ISS"
```

`str` is a 2-by-3 string array. You can find the lengths of the strings with the `strlength` function.

```
N = strlength(str)
```

```
N = 2×3

     7     6     6
     6     8     3
```

String arrays are supported throughout MATLAB and MathWorks® products. Functions that accept character arrays (and cell arrays of character vectors) as inputs also accept string arrays.

Represent Text with Character Vectors

To store a 1-by-n sequence of characters as a character vector, using the `char` data type, enclose it in single quotes.

```
chr = 'Hello, world'
```

```
chr =
'Hello, world'
```

The text `'Hello, world'` is 12 characters long, and `chr` stores it as a 1-by-12 character vector.

```
whos chr
```

```
  Name      Size            Bytes  Class    Attributes

  chr       1x12               24  char
```

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

```
chr = 'They said, ''Welcome!'' and waved.'
```

```
chr =
'They said, 'Welcome!' and waved.'
```

Character vectors have two principal uses:

- To specify single pieces of text, such as file names and plot labels.
- To represent data that is encoded using characters. In such cases, you might need easy access to individual characters.

For example, you can store a DNA sequence as a character vector.

```
seq = 'GCTAGAATCC';
```

You can access individual characters or subsets of characters by indexing, just as you would index into a numeric array.

```
seq(4:6)
```

```
ans =
'AGA'
```

Concatenate character vector with square brackets, just as you concatenate other types of arrays.

```
seq2 = [seq 'ATTAGAAACC']
```

```
seq2 =
'GCTAGAATCCATTAGAAACC'
```

You can also concatenate text using `append`. The `append` function is recommended because it treats string arrays, character vectors, and cell arrays of character vectors consistently.

```
seq2 = append(seq,'ATTAGAAACC')
```

```
seq2 =
'GCTAGAATCCATTAGAAACC'
```

MATLAB functions that accept string arrays as inputs also accept character vectors and cell arrays of character vectors.

Tables of Mixed Data

Store Related Data in Single Container

You can use the `table` data type to collect mixed-type data and metadata properties, such as variable names, row names, descriptions, and variable units, in a single container. Tables are suitable for column-oriented or tabular data that is often stored as columns in a text file or in a spreadsheet. For example, you can use a table to store experimental data, with rows representing different observations and columns representing different measured variables.

Tables consist of rows and column-oriented variables. Variables in a table can have different data types and different sizes, but the variables must have the same number of rows. Also, the data within a variable is homogeneous, which enables you to treat a table variable like an array of data.

For example, load sample data about patients from the `patients.mat` MAT-file. Combine blood pressure data into a single variable. Convert a four-category variable called

`SelfAssessedHealthStatus`—which has values of `Poor`, `Fair`, `Good`, or `Excellent`—to a categorical array. View information about several of the variables.

```
load patients
BloodPressure = [Systolic Diastolic];
SelfAssessedHealthStatus = categorical(SelfAssessedHealthStatus);

whos("Age","Smoker","BloodPressure","SelfAssessedHealthStatus")
```

```
  Name                          Size            Bytes  Class          Attributes

  Age                         100x1               800  double
  BloodPressure               100x2              1600  double
  SelfAssessedHealthStatus    100x1               624  categorical
  Smoker                      100x1               100  logical
```

Now, create a table from these variables and display it. The variables can be stored together in a table because they all have the same number of rows, 100.

```
T = table(Age,Smoker,BloodPressure,SelfAssessedHealthStatus)
```

```
T=100×4 table
    Age    Smoker    BloodPressure    SelfAssessedHealthStatus
    ___    ______    _____________    ________________________

    38     true       124     93            Excellent
    43     false      109     77            Fair
    38     false      125     83            Good
    40     false      117     75            Fair
    49     false      122     80            Good
    46     false      121     70            Good
    33     true       130     88            Good
    40     false      115     82            Good
    28     false      115     78            Excellent
    31     false      118     86            Excellent
    45     false      114     77            Excellent
    42     false      115     68            Poor
    25     false      127     74            Poor
    39     true       130     95            Excellent
    36     false      114     79            Good
    48     true       130     92            Good
      ⋮
```

Each variable in a table has one data type. If you add a new row to the table, MATLAB® forces consistency of the data type between the new data and the corresponding table variables. For example, if you try to add information for a new patient where the first column contains the patient's health status instead of age, as in the expression `T(end+1,:) = {"Poor",true,[130 84],37}`, then you receive the error:

```
Right hand side of an assignment to a categorical array must be a categorical
or text representing a category name.
```

The error occurs because MATLAB® cannot assign numeric data, `37`, to the categorical array, `SelfAssessedHealthStatus`.

Access Data Using Numeric or Named Indexing

You can index into a table using parentheses, curly braces, or dot notation. Parentheses allow you to select a subset of the data in a table and preserve the table container. Curly braces and dot notation allow you to extract data from a table. Within each table indexing method, you can specify the rows or variables to access by name or by numeric index.

Consider the sample table from above. Each row in the table, `T`, represents a different patient. The workspace variable, `LastName`, contains unique identifiers for the 100 rows. Add row names to the table by setting the `RowNames` property to `LastName` and display the first five rows of the updated table.

```
T.Properties.RowNames = LastName;
T(1:5,:)
```

```
ans=5×4 table
                Age    Smoker    BloodPressure    SelfAssessedHealthStatus
                ___    ______    _____________    ________________________

    Smith       38     true       124     93            Excellent
    Johnson     43     false      109     77            Fair
    Williams    38     false      125     83            Good
    Jones       40     false      117     75            Fair
    Brown       49     false      122     80            Good
```

In addition to labeling the data, you can use row and variable names to access data in the table. For example, use named indexing to display the age and blood pressure of the patients `Williams` and `Brown`.

```
T(["Williams","Brown"],["Age","BloodPressure"])
```

```
ans=2×2 table
                Age    BloodPressure
                ___    _____________

    Williams    38      125     83
    Brown       49      122     80
```

Now, use numeric indexing to return an equivalent subtable. Return the third and fifth rows from the first and third variables.

```
T([3 5],[1 3])
```

```
ans=2×2 table
                Age    BloodPressure
                ___    _____________

    Williams    38      125     83
    Brown       49      122     80
```

For more information on table indexing, see “Access Data in Tables”.

Describe Data with Table Properties

In addition to storing data, tables have properties to store metadata, such as variable names, row names, descriptions, and variable units. You can access a property using *T*.Properties.*PropName*, where *T* is the name of the table and *PropName* is the name of a table property.

For example, add a table description, variable descriptions, and variable units for Age.

```
T.Properties.Description = "Simulated Patient Data";

T.Properties.VariableDescriptions = ...
    ["" ...
     "true or false" ...
     "Systolic/Diastolic" ...
     "Status Reported by Patient"];

T.Properties.VariableUnits("Age") = "Yrs";
```

Individual empty strings within VariableDescriptions indicate that the corresponding variable does not have a description. For more information, see the Properties section of table.

To print a table summary, use the summary function.

```
summary(T)
```

```
T: 100×4 table

Description: Simulated Patient Data

Variables:

    Age: double (Yrs)
    Smoker: logical (34 true, true or false)
    BloodPressure: 2-column double (Systolic/Diastolic)
    SelfAssessedHealthStatus: categorical (4 categories, Status Reported by Patient)

Statistics for applicable variables:

                              NumMissing      Min        Median         Max          Mean

    Age                            0            25            39         50         38.2800
    BloodPressure(:,1)             0           109           122        138        122.7800
    BloodPressure(:,2)             0            68       81.5000         99         82.9600
    SelfAssessedHealthStatus       0
```

Comparison to Cell Arrays

Like a table, a cell array can provide storage for mixed-type data in a single container. But unlike a table, a cell array does not provide metadata that describes its contents. It does not force data in its columns to remain homogenous. You cannot access the contents of a cell array using row names or column names.

For example, convert T to a cell array using the table2cell function. The output cell array contains the same data but has no information about that data. If it is important to keep such information attached to your data, then storing it in a table is a better choice than storing it in a cell array.

```
C = table2cell(T)
```

```
C=100×4 cell array
    {[38]}    {[1]}    {[124 93]}    {[Excellent]}
    {[43]}    {[0]}    {[109 77]}    {[Fair     ]}
    {[38]}    {[0]}    {[125 83]}    {[Good     ]}
    {[40]}    {[0]}    {[117 75]}    {[Fair     ]}
    {[49]}    {[0]}    {[122 80]}    {[Good     ]}
    {[46]}    {[0]}    {[121 70]}    {[Good     ]}
    {[33]}    {[1]}    {[130 88]}    {[Good     ]}
    {[40]}    {[0]}    {[115 82]}    {[Good     ]}
    {[28]}    {[0]}    {[115 78]}    {[Excellent]}
    {[31]}    {[0]}    {[118 86]}    {[Excellent]}
    {[45]}    {[0]}    {[114 77]}    {[Excellent]}
    {[42]}    {[0]}    {[115 68]}    {[Poor     ]}
    {[25]}    {[0]}    {[127 74]}    {[Poor     ]}
    {[39]}    {[1]}    {[130 95]}    {[Excellent]}
    {[36]}    {[0]}    {[114 79]}    {[Good     ]}
    {[48]}    {[1]}    {[130 92]}    {[Good     ]}
      ⋮
```

To access subsets of data in a cell array, you can only use indexing with parentheses or curly braces.

```
C(1:5,1:3)
```

```
ans=5×3 cell array
    {[38]}    {[1]}    {[124 93]}
    {[43]}    {[0]}    {[109 77]}
    {[38]}    {[0]}    {[125 83]}
    {[40]}    {[0]}    {[117 75]}
    {[49]}    {[0]}    {[122 80]}
```

Comparison to Structures

Structures also can provide storage for mixed-type data. A structure has fields that you can access by name, just as you can access table variables by name. However, it does not force data in its fields to remain homogenous. Structures do not provide any metadata to describe their contents.

For example, convert `T` to a scalar structure where every field is an array, in a way that resembles table variables. Use the `table2struct` function with the `ToScalar` name-value argument.

```
S = table2struct(T,ToScalar=true)
```

```
S = struct with fields:
                        Age: [100×1 double]
                     Smoker: [100×1 logical]
              BloodPressure: [100×2 double]
    SelfAssessedHealthStatus: [100×1 categorical]
```

In this structure, you can access arrays of data by using field names.

```
S.Age
```

```
ans = 100×1

    38
    43
    38
```

```
    40
    49
    46
    33
    40
    28
    31
    45
    42
    25
    39
    36
     ⋮
```

But to access subsets of data in the fields, you can only use numeric indices, and you can only access one field at a time. Table row and variable indexing provides more flexible access to data in a table.

```
S.Age(1:5)
```

```
ans = 5×1

    38
    43
    38
    40
    49
```

Access Data in Cell Array

Basic Indexing

A *cell array* is a data type with indexed data containers called cells. Each cell can contain any type of data. Cell arrays are often used to hold data from a file that has inconsistent formatting, such as columns that contain both numeric and text data.

For instance, consider a 2-by-3 cell array of mixed data.

```
C = {'one','two','three';
     100,200,rand(3,3)}
```

```
C=2×3 cell array
    {'one'}    {'two'}    {'three'   }
    {[100]}    {[200]}    {3×3 double}
```

Each element is within a cell. If you index into this array using standard parentheses, the result is a subset of the cell array that includes the cells.

```
C2 = C(1:2,1:2)
```

```
C2=2×2 cell array
    {'one'}    {'two'}
    {[100]}    {[200]}
```

To read or write the contents within a specific cell, enclose the indices in curly braces.

```
R = C{2,3}
```

```
R = 3×3

    0.8147    0.9134    0.2785
    0.9058    0.6324    0.5469
    0.1270    0.0975    0.9575
```

```
C{1,3} = 'longer text in a third location'
```

```
C=2×3 cell array
    {'one'}    {'two'}    {'longer text in a third location'}
    {[100]}    {[200]}    {3×3 double                       }
```

To replace the contents of multiple cells at the same time, use parentheses to refer to the cells and curly braces to define an equivalently sized cell array.

```
C(1,1:2) = {'first','second'}
```

```
C=2×3 cell array
    {'first'}    {'second'}    {'longer text in a third location'}
    {[  100]}    {[   200]}    {3×3 double                       }
```

Read Data from Multiple Cells

Most of the data processing functions in MATLAB® operate on a rectangular array with a uniform data type. Because cell arrays can contain a mix of types and sizes, you sometimes must extract and combine data from cells before processing that data. This section describes a few common scenarios.

Text in Specific Cells

When the entire cell array or a known subset of cells contains text, you can index and pass the cells directly to any of the text processing functions in MATLAB. For instance, find where the letter *t* appears in each element of the first row of `C`.

```
ts = strfind(C(1,:),'t')
```

```
ts=1×3 cell array
    {[5]}    {0×0 double}    {[8 11 18 28]}
```

Numbers in Specific Cells

The two main ways to process numeric data in a cell array are:

- Combine the contents of those cells into a single numeric array, and then process that array.
- Process the individual cells separately.

To combine numeric cells, use the `cell2mat` function. The arrays in each cell must have compatible sizes for concatenation. For instance, the first two elements of the second row of `C` are scalar values. Combine them into a 1-by-2 numeric vector.

```
v = cell2mat(C(2,1:2))
```

```
v = 1×2

   100   200
```

To process individual cells, you can use the `cellfun` function. When calling `cellfun`, specify the function to apply to each cell. Use the `@` symbol to indicate that it is a function and to create a function handle. For instance, find the length of each of the cells in the second row of `C`.

```
len = cellfun(@length,C(2,:))
```

```
len = 1×3

     1     1     3
```

Data in Cells with Unknown Indices

When some of the cells contain data that you want to process, but you do not know the exact indices, you can use one of these options:

- Find all the elements that meet a certain condition using logical indexing, and then process those elements.
- Check and process cells one at a time with a `for`- or `while`-loop.

For instance, suppose you want to process only the cells that contain character vectors. To take advantage of logical indexing, first use the `cellfun` function with `ischar` to find those cells.

```
idx = cellfun(@ischar,C)
```

```
idx = 2×3 logical array

   1   1   1
   0   0   0
```

Then, use the logical array to index into the cell array, `C(idx)`. The result of the indexing operation is a column vector, which you can pass to a text processing function, such as `strlength`.

```
len = strlength(C(idx))
```

```
len = 3×1

     5
     6
    31
```

The other approach is to use a loop to check and process the contents of each cell. For instance, find cells that contain the letter *t* and combine them into a string array by looping through the cells. Track how many elements the loop adds to the string array in variable `n`.

```
n = 0;
for k = 1:numel(C)
    if ischar(C{k}) && contains(C{k},"t")
        n = n + 1;
        txt(n) = string(C{k});
    end
```

```
end
txt
```

```
txt = 1×2 string
    "first"    "longer text in a third location"
```

Index into Multiple Cells

If you refer to multiple cells using curly brace indexing, MATLAB returns the contents of the cells as a *comma-separated list*. For example,

```
C{1:2,1:2}
```

is the same as

```
C{1,1}, C{2,1}, C{1,2}, C{2,2}.
```

Because each cell can contain a different type of data, you cannot assign this list to a single variable. However, you can assign the list to the same number of variables as cells.

```
[v1,v2,v3,v4] = C{1:2,1:2}
```

```
v1 =
'first'

v2 =
100

v3 =
'second'

v4 =
200
```

If each cell contains the same type of data with compatible sizes, you can create a single variable by applying the array concatenation operator `[]` to the comma-separated list.

```
v = [C{2,1:2}]
```

```
v = 1×2

   100   200
```

If the cell contents cannot be concatenated, store results in a new cell array, table, or other heterogeneous container. For instance, convert the numeric data in the second row of `C` to a table. Use the text data in the first row of `C` for variable names.

```
t = cell2table(C(2,:),VariableNames=C(1,:))
```

```
t=1×3 table
    first    second    longer text in a third location
    _____    ______    _______________________________

     100      200               {3×3 double}
```

Index into Arrays Within Cells

If a cell contains an array, you can access specific elements within that array using two levels of indices. First, use curly braces to access the contents of the cell. Then, use the standard indexing syntax for the type of array in that cell.

For example, `C{2,3}` returns a 3-by-3 matrix of random numbers. Index with parentheses to extract the second row of that matrix.

```
C{2,3}(2,:)
```

```
ans = 1×3

    0.9058    0.6324    0.5469
```

If the cell contains a cell array, use curly braces for indexing, and if it contains a structure array, use dot notation to refer to specific fields. For instance, consider a cell array that contains a 2-by-1 cell array and a scalar structure with fields `f1` and `f2`.

```
c = {'A'; ones(3,4)};
s = struct("f1",'B',"f2",ones(5,6));
C = {c,s}
```

```
C=1×2 cell array
    {2×1 cell}    {1×1 struct}
```

Extract the arrays of ones from the nested cell array and structure.

```
A1 = C{1}{2}
```

```
A1 = 3×4

     1     1     1     1
     1     1     1     1
     1     1     1     1
```

```
A2 = C{2}.f2
```

```
A2 = 5×6

     1     1     1     1     1     1
     1     1     1     1     1     1
     1     1     1     1     1     1
     1     1     1     1     1     1
     1     1     1     1     1     1
```

You can nest any number of cell and structure arrays. Apply the same indexing rules to lower levels in the hierarchy. For instance, these syntaxes are valid when the referenced cells contain the expected cell or structure array.

```
C{1}{2}{3}
```

```
C{4}.f1.f2(1)
```

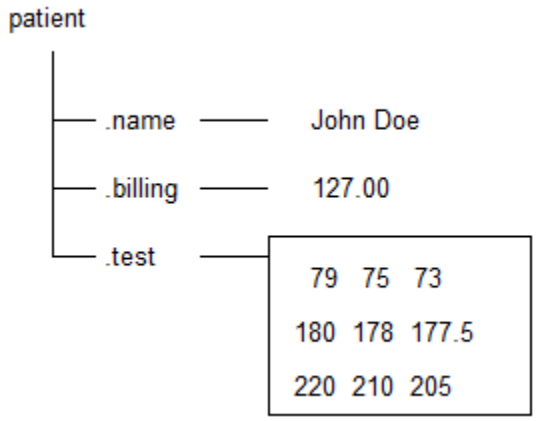
```
C{5}.f3.f4{1}
```

At any indexing level, if you refer to multiple cells, MATLAB returns a comma-separated list. For details, see Index into Multiple Cells on page 2-37.

Structure Arrays

Create Scalar Structure

First, create a structure named `patient` that has fields storing data about a patient. The diagram shows how the structure stores data. A structure like `patient` is also referred to as a *scalar structure* because the variable stores one structure.



Use dot notation to add the fields `name`, `billing`, and `test`, assigning data to each field. In this example, the syntax `patient.name` creates both the structure and its first field. The commands that follow add more fields.

```
patient.name = 'John Doe';
patient.billing = 127;
patient.test = [79 75 73; 180 178 177.5; 220 210 205]
```

```
patient = struct with fields:
       name: 'John Doe'
    billing: 127
       test: [3×3 double]
```

Access Values in Fields

After you create a field, you can keep using dot notation to access and change the value it stores.

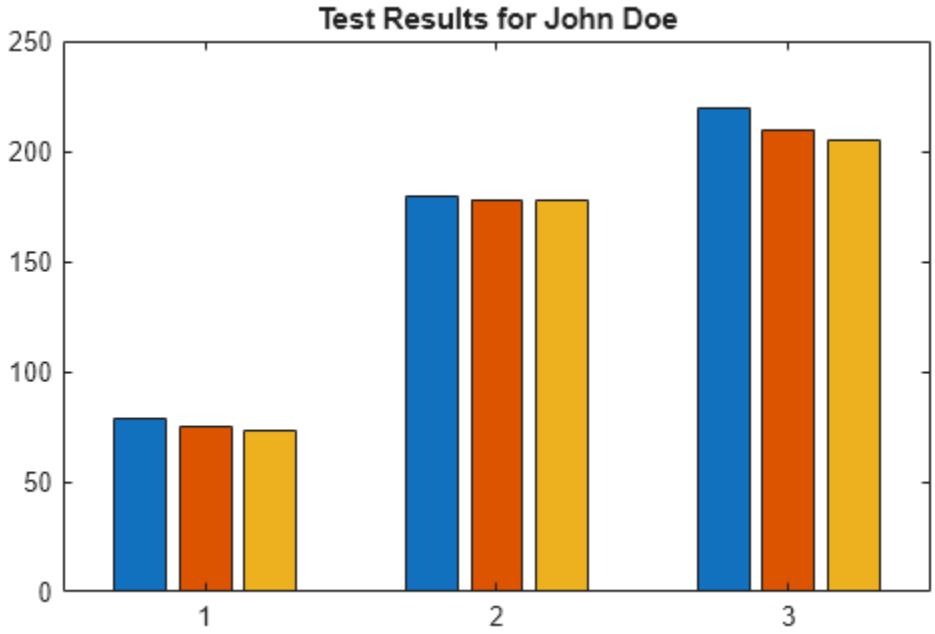
For example, change the value of the `billing` field.

```
patient.billing = 512.00
```

```
patient = struct with fields:
       name: 'John Doe'
    billing: 512
       test: [3×3 double]
```

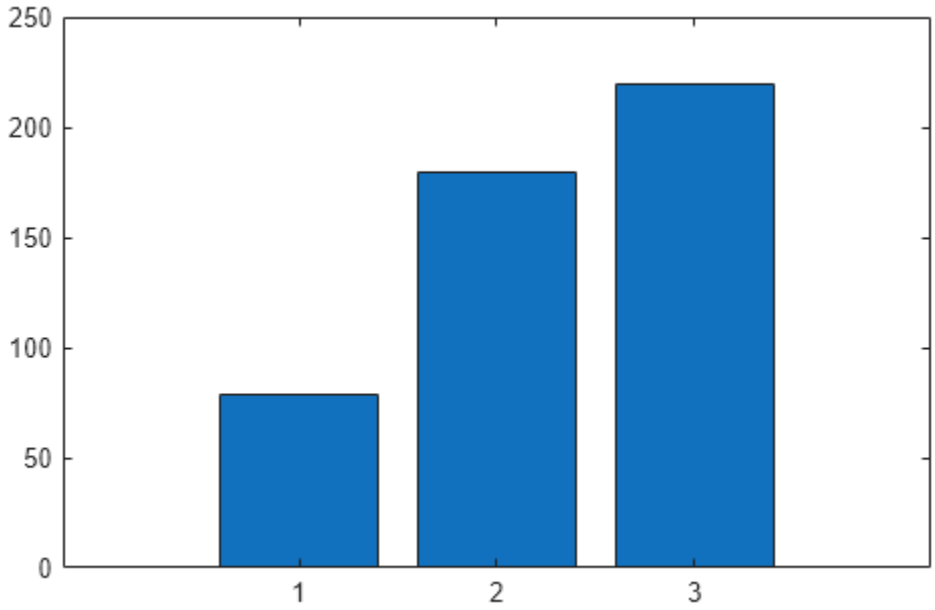
With dot notation, you also can access the value of any field. For example, make a bar chart of the values in `patient.test`. Add a title with the text in `patient.name`. If a field stores an array, then this syntax returns the whole array.

```
bar(patient.test)
title("Test Results for " + patient.name)
```



To access part of an array stored in a field, add indices that are appropriate for the size and type of the array. For example, create a bar chart of the data in one column of `patient.test`.

```
bar(patient.test(:,1))
```



Index into Nonscalar Structure Array

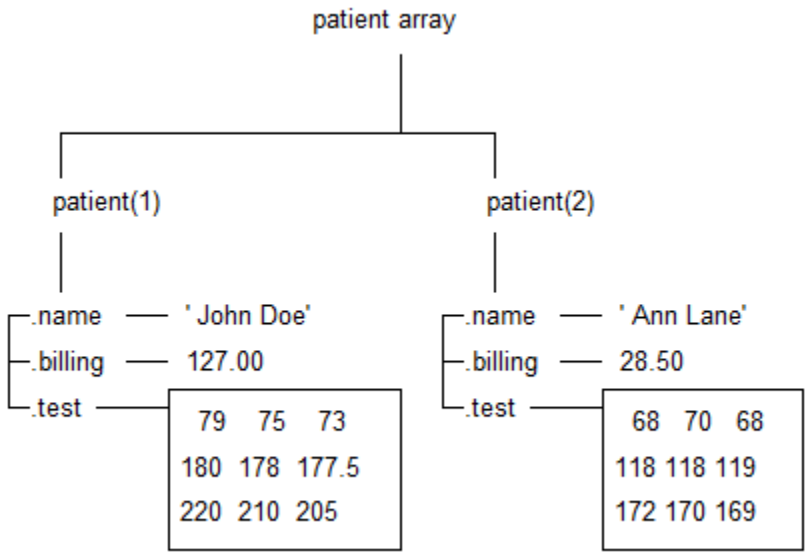
Structure arrays can be nonscalar. You can create a structure array having any size, as long as each structure in the array has the same fields.

For example, add a second structure to `patients` having data about a second patient. Also, assign the original value of `127` to the `billing` field of the first structure. Since the array now has two structures, you must access the first structure by indexing, as in `patient(1).billing = 127`.

```
patient(2).name = 'Ann Lane';
patient(2).billing = 28.50;
patient(2).test = [68 70 68; 118 118 119; 172 170 169];
patient(1).billing = 127
```

```
patient=1×2 struct array with fields:
    name
    billing
    test
```

As a result, `patient` is a 1-by-2 structure array with contents shown in the diagram.



Each patient record in the array is a structure of class `struct`. An array of structures is sometimes referred to as a *struct array*. However, the terms *struct array* and *structure array* mean the same thing. Like other MATLAB® arrays, a structure array can have any dimensions.

A structure array has the following properties:

- All structures in the array have the same number of fields.
- All structures have the same field names.
- Fields of the same name in different structures can contain different types or sizes of data.

If you add a new structure to the array without specifying all of its fields, then the unspecified fields contain empty arrays.

```
patient(3).name = 'New Name';
patient(3)
```

```
ans = struct with fields:
       name: 'New Name'
    billing: []
       test: []
```

To index into a structure array, use array indexing. For example, `patient(2)` returns the second structure.

```
patient(2)
```

```
ans = struct with fields:
       name: 'Ann Lane'
    billing: 28.5000
       test: [3×3 double]
```

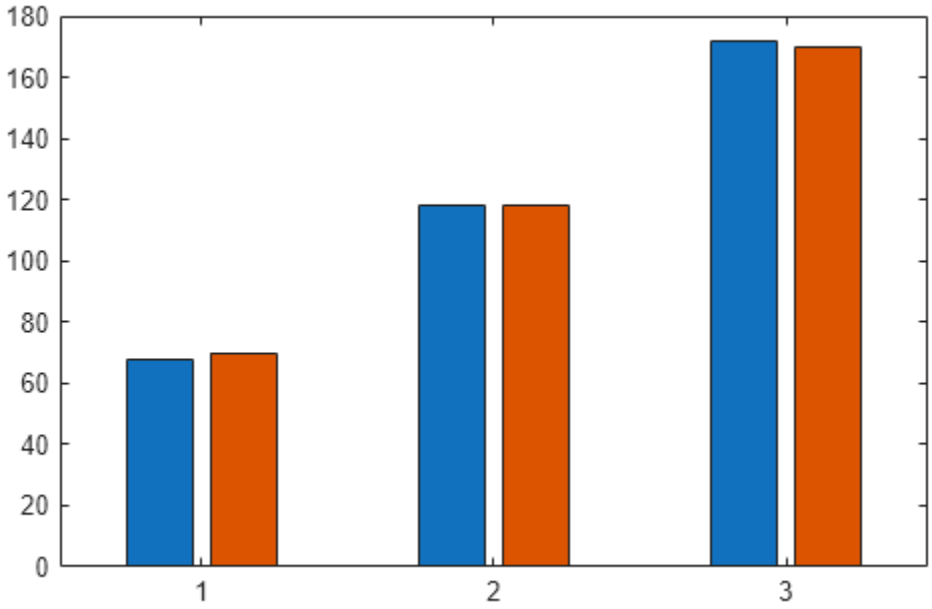
To access a field, use array indexing and dot notation. For example, return the value of the `billing` field for the second patient.

```
patient(2).billing
```

```
ans =
28.5000
```

You also can index into an array stored by a field. Create a bar chart displaying only the first two columns of `patient(2).test`.

```
bar(patient(2).test(:,[1 2]))
```



Floating-Point Numbers

“Floating point” refers to a set of data types that encode real numbers, including fractions and decimals. Floating-point data types allow for a varying number of digits after the decimal point, while fixed-point data types have a specific number of digits reserved before and after the decimal point. So, floating-point data types can represent a wider range of numbers than fixed-point data types.

Due to limited memory for number representation and storage, computers can represent a finite set of floating-point numbers that have finite precision. This finite precision can limit accuracy for floating-point computations that require exact values or high precision, as some numbers are not represented exactly. Despite their limitations, floating-point numbers are widely used due to their fast calculations and sufficient precision and range for solving real-world problems.

Floating-Point Numbers in MATLAB

MATLAB has data types for double-precision (`double`) and single-precision (`single`) floating-point numbers following IEEE® Standard 754. By default, MATLAB represents floating-point numbers in double precision. Double precision allows you to represent numbers to greater precision but requires more memory than single precision. To conserve memory, you can convert a number to single precision by using the `single` function.

You can store numbers between approximately –3.4 × 10^{38} and 3.4 × 10^{38} using either double or single precision. If you have numbers outside of that range, store them using double precision.

Create Double-Precision Data

Because the default numeric type for MATLAB is type `double`, you can create a double-precision floating-point number with a simple assignment statement.

```
x = 10;
c = class(x)

c =
   'double'
```

You can convert numeric data, characters or strings, and logical data to double precision by using the `double` function. For example, convert a signed integer to a double-precision floating-point number.

```
x = int8(-113);
y = double(x)

y =
   -113
```

Create Single-Precision Data

To create a single-precision number, use the `single` function.

```
x = single(25.783);
```

You can also convert numeric data, characters or strings, and logical data to single precision by using the `single` function. For example, convert a signed integer to a single-precision floating-point number.

```
x = int8(-113);
y = single(x)

y =
   single
   -113
```

How MATLAB Stores Floating-Point Numbers

MATLAB constructs its `double` and `single` floating-point data types according to IEEE format and follows the round to nearest, ties to even rounding mode by default.

A floating-point number x has the form:

$$x = -1^{s} \cdot (1 + f) \cdot 2^{e}$$

where:

- s determines the sign.
- f is the fraction, or mantissa, which satisfies $0 \le f < 1$.
- e is the exponent.

s, f, and e are each determined by a finite number of bits in memory, with f and e depending on the precision of the data type.

Storage of a `double` number requires 64 bits, as shown in this table.

Bits	Width	Usage
63	1	Stores the sign, where `0` is positive and `1` is negative
62 to 52	11	Stores the exponent, biased by `1023`
51 to 0	52	Stores the mantissa

Storage of a `single` number requires 32 bits, as shown in this table.

Bits	Width	Usage
31	1	Stores the sign, where `0` is positive and `1` is negative
30 to 23	8	Stores the exponent, biased by `127`
22 to 0	23	Stores the mantissa

Largest and Smallest Values for Floating-Point Data Types

The double- and single-precision data types have a largest and smallest value that you can represent. Numbers outside of the representable range are assigned positive or negative infinity. However, some numbers within the representable range cannot be stored exactly due to the gaps between consecutive floating-point numbers, and these numbers can have round-off errors.

Largest and Smallest Double-Precision Values

Find the largest and smallest positive values that can be represented with the `double` data type by using the `realmax` and `realmin` functions, respectively.

```
m = realmax
```

```
m =
   1.7977e+308
```

```
n = realmin
```

```
n =
   2.2251e-308
```

`realmax` and `realmin` return normalized IEEE values. You can find the largest and smallest negative values by multiplying `realmax` and `realmin` by `-1`. Numbers greater than `realmax` or less than `-realmax` are assigned the values of positive or negative infinity, respectively.

Largest and Smallest Single-Precision Values

Find the largest and smallest positive values that can be represented with the `single` data type by calling the `realmax` and `realmin` functions with the argument `"single"`.

```
m = realmax("single")
```

```
m =
   single
   3.4028e+38
```

```
n = realmin("single")
```

```
n =
   single
   1.1755e-38
```

You can find the largest and smallest negative values by multiplying `realmax("single")` and `realmin("single")` by `−1`. Numbers greater than `realmax("single")` or less than `−realmax("single")` are assigned the values of positive or negative infinity, respectively.

Largest Consecutive Floating-Point Integers

Not all integers are representable using floating-point data types. The largest consecutive integer, *x*, is the greatest integer for which all integers less than or equal to *x* can be exactly represented, but *x* + 1 cannot be represented in floating-point format. The `flintmax` function returns the largest consecutive integer. For example, find the largest consecutive integer in double-precision floating-point format, which is 2^{53}, by using the `flintmax` function.

```
x = flintmax
```

```
x =
   9.0072e+15
```

Find the largest consecutive integer in single-precision floating-point format, which is 2^{24}.

```
y = flintmax("single")
```

```
y =
   single
   16777216
```

When you convert an integer data type to a floating-point data type, integers that are not exactly representable in floating-point format lose accuracy. `flintmax`, which is a floating-point number, is less than the greatest integer representable by integer data types using the same number of bits. For example, `flintmax` for double precision is 2^{53}, while the maximum value for type `int64` is $2^{64} - 1$. Therefore, converting an integer greater than 2^{53} to double precision results in a loss of accuracy.

Accuracy of Floating-Point Data

The accuracy of floating-point data can be affected by several factors:

- Limitations of your computer hardware — For example, hardware with insufficient memory truncates the results of floating-point calculations.
- Gaps between each floating-point number and the next larger floating-point number — These gaps are present on any computer and limit precision.

Gaps Between Floating-Point Numbers

You can determine the size of a gap between consecutive floating-point numbers by using the `eps` function. For example, find the distance between `5` and the next larger double-precision number.

```
e = eps(5)
```

```
e =
   8.8818e-16
```

You cannot represent numbers between `5` and `5 + eps(5)` in double-precision format. If a double-precision computation returns the answer `5`, the result is accurate within `eps(5)`. This radius of accuracy is often called machine epsilon.

The gaps between floating-point numbers are not equal. For example, the gap between `1e10` and the next larger double-precision number is larger than the gap between `5` and the next larger double-precision number.

```
e = eps(1e10)

e =
   1.9073e-06
```

Similarly, find the distance between `5` and the next larger single-precision number.

```
x = single(5);
e = eps(x)

e =
   single
   4.7684e-07
```

Gaps between single-precision numbers are larger than the gaps between double-precision numbers because there are fewer single-precision numbers. So, results of single-precision calculations are less precise than results of double-precision calculations.

When you convert a double-precision number to a single-precision number, you can determine the upper bound for the amount the number is rounded by using the `eps` function. For example, when you convert the double-precision number `3.14` to single precision, the number is rounded by at most `eps(single(3.14))`.

Gaps Between Consecutive Floating-Point Integers

The `flintmax` function returns the largest consecutive integer in floating-point format. Above this value, consecutive floating-point integers have a gap greater than `1`.

Find the gap between `flintmax` and the next floating-point number by using `eps`:

```
format long
x = flintmax

x =
   9.007199254740992e+15

e = eps(x)

e =
   2
```

Because `eps(x)` is `2`, the next larger floating-point number that can be represented exactly is `x + 2`.

```
y = x + e

y =
   9.007199254740994e+15
```

If you add `1` to `x`, the result is rounded to `x`.

```
z = x + 1

z =
   9.007199254740992e+15
```

Arithmetic Operations on Floating-Point Numbers

You can use a range of data types in arithmetic operations with floating-point numbers, and the data type of the result depends on the input types. However, when you perform operations with different data types, some calculations may not be exact due to approximations or intermediate conversions.

Double-Precision Operands

You can perform basic arithmetic operations with `double` and any of the following data types. If one or more operands are an integer scalar or array, the `double` operand must be a scalar. The result is of type `double`, except where noted otherwise.

- `single` — The result is of type `single`.
- `double`
- `int8`, `int16`, `int32`, `int64` — The result is of the same data type as the integer operand.
- `uint8`, `uint16`, `uint32`, `uint64` — The result is of the same data type as the integer operand.
- `char`
- `logical`

Single-Precision Operands

You can perform basic arithmetic operations with `single` and any of the following data types. The result is of type `single`.

- `single`
- `double`
- `char`
- `logical`

Unexpected Results with Floating-Point Arithmetic

Almost all operations in MATLAB are performed in double-precision arithmetic conforming to IEEE Standard 754. Because computers represent numbers to a finite precision, some computations can yield mathematically nonintuitive results. Some common issues that can arise while computing with floating-point numbers are round-off error, cancellation, swamping, and intermediate conversions. The unexpected results are not bugs in MATLAB and occur in any software that uses floating-point numbers. For exact rational representations of numbers, consider using the Symbolic Math Toolbox™.

Round-Off Error

Round-off error can occur due to the finite-precision representation of floating-point numbers. For example, the number `4/3` cannot be represented exactly as a binary fraction. As such, this calculation returns the quantity `eps(1)`, rather than `0`.

```
e = 1 - 3*(4/3 - 1)
```

```
e =
   2.2204e-16
```

Similarly, because `pi` is not an exact representation of π, `sin(pi)` is not exactly zero.

```
x = sin(pi)
```

```
x =
   1.2246e-16
```

Round-off error is most noticeable when many operations are performed on floating-point numbers, allowing errors to accumulate and compound. A best practice is to minimize the number of operations whenever possible.

Cancellation

Cancellation can occur when you subtract a number from another number of roughly the same magnitude, as measured by `eps`. For example, `eps(2^53)` is `2`, so the numbers `2^53 + 1` and `2^53` have the same floating-point representation.

```
x = (2^53 + 1) - 2^53
```

```
x =
   0
```

When possible, try rewriting computations in an equivalent form that avoids cancellations.

Swamping

Swamping can occur when you perform operations on floating-point numbers that differ by many orders of magnitude. For example, this calculation shows a loss of precision that makes the addition insignificant.

```
x = 1 + 1e-16
```

```
x =
   1
```

Intermediate Conversions

When you perform arithmetic with different data types, intermediate calculations and conversions can yield unexpected results. For example, although `x` and `y` are both `0.2`, subtracting them yields a nonzero result. The reason is that `y` is first converted to `double` before the subtraction is performed. This subtraction result is then converted to `single`, `z`.

```
format long
x = 0.2
```

```
x =
   0.200000000000000
```

```
y = single(0.2)
```

```
y =
   single
   0.2000000
```

```
z = x - y
```

```
z =
   single
   -2.9802323e-09
```

Linear Algebra

Common issues in floating-point arithmetic, such as the ones described above, can compound when applied to linear algebra problems because the related calculations typically consist of multiple steps. For example, when solving the system of linear equations `Ax = b`, MATLAB warns that the results may be inaccurate because operand matrix `A` is ill conditioned.

```
A = diag([2 eps]);
b = [2; eps];
x = A\b;
```

```
Warning: Matrix is close to singular or badly scaled.
         Results may be inaccurate. RCOND = 1.110223e-16.
```

Integers

Integer Classes

MATLAB has four signed and four unsigned integer classes. Signed types enable you to work with negative integers as well as positive, but cannot represent as wide a range of numbers as the unsigned types because one bit is used to designate a positive or negative sign for the number. Unsigned types give you a wider range of numbers, but these numbers can only be zero or positive.

MATLAB supports 1-, 2-, 4-, and 8-byte storage for integer data. You can save memory and execution time for your programs if you use the smallest integer type that accommodates your data. For example, you do not need a 32-bit integer to store the value `100`.

Here are the eight integer classes, the range of values you can store with each type, and the MATLAB conversion function required to create that type.

Class	Range of Values	Conversion Function
Signed 8-bit integer	-2^7 to 2^7-1	`int8`
Signed 16-bit integer	-2^{15} to $2^{15}-1$	`int16`
Signed 32-bit integer	-2^{31} to $2^{31}-1$	`int32`
Signed 64-bit integer	-2^{63} to $2^{63}-1$	`int64`
Unsigned 8-bit integer	0 to 2^8-1	`uint8`
Unsigned 16-bit integer	0 to $2^{16}-1$	`uint16`
Unsigned 32-bit integer	0 to $2^{32}-1$	`uint32`
Unsigned 64-bit integer	0 to $2^{64}-1$	`uint64`

Creating Integer Data

MATLAB stores numeric data as double-precision floating point (`double`) by default. To store data as an integer, you need to convert from `double` to the desired integer type. Use one of the conversion functions shown in the table above.

For example, to store `325` as a 16-bit signed integer assigned to variable `x`, type

```
x = int16(325);
```

If the number being converted to an integer has a fractional part, MATLAB rounds to the nearest integer. If the fractional part is exactly `0.5`, then MATLAB chooses the nearest integer whose absolute value is larger in magnitude:

```
x = 325.499;
int16(x)
```

```
ans =
```

```
  int16

   325
```

```
x = x + .001;
int16(x)
```

```
ans =

  int16

   326
```

If you need to round a number using a rounding scheme other than the default, MATLAB provides four rounding functions: `round`, `fix`, `floor`, and `ceil`. The `fix` function enables you to override the default and round *towards zero* when there is a nonzero fractional part:

```
x = 325.9;
int16(fix(x))
```

```
ans =

  int16

   325
```

Arithmetic operations that involve both integers and floating-point numbers always result in an integer data type. MATLAB rounds the result, when necessary, according to the default rounding algorithm. The example below yields an exact answer of `1426.75` which MATLAB then rounds to the next highest integer:

```
int16(325)*4.39
```

```
ans =

  int16

   1427
```

The integer conversion functions are also useful when converting other classes, such as character vectors, to integers:

```
chr = 'Hello World';
int8(chr)
```

```
ans =

  1×11 int8 row vector

    72   101   108   108   111    32    87   111   114   108   100
```

If you convert a `NaN` value to an integer class, the result is a value of `0` in that integer class. For example:

```
int32(NaN)
```

```
ans =

  int32
```

```
   0
```

Arithmetic Operations on Integer Classes

MATLAB can perform integer arithmetic on the following types of data:

- Integers or integer arrays of the same integer data type. Arithmetic operations yield a result that has the same data type as the operands:

  ```
  x = uint32([132 347 528]) .* uint32(75);
  class(x)

  ans =
     'uint32'
  ```

- Integers or integer arrays and scalar double-precision floating-point numbers. Arithmetic operations yield a result that has the same data type as the integer operands:

  ```
  x = uint32([132 347 528]) .* 75.49;
  class(x)

  ans =
     'uint32'
  ```

For all binary operations in which one operand is an array of integer data type (except 64-bit integers) and the other is a scalar double, MATLAB computes the operation using element-wise double-precision arithmetic, and then converts the result back to the original integer data type. For binary operations involving a 64-bit integer array and a scalar double, MATLAB computes the operation as if 80-bit extended-precision arithmetic were used, to prevent loss of precision.

Operations involving complex numbers with integer types are not supported.

Largest and Smallest Values for Integer Classes

For each integer data type, there is a largest and smallest number that you can represent with that type. The table shown under “Integer Classes” lists the largest and smallest values for each integer data type in the “Range of Values” column.

You can also obtain these values with the `intmax` and `intmin` functions:

```
intmax("int8")

ans =

  int8

   127

intmin("int8")

ans =

  int8

   -128
```

If you convert a number that is larger than the maximum value of an integer data type to that type, MATLAB sets it to the maximum value. Similarly, if you convert a number that is smaller than the minimum value of the integer data type, MATLAB sets it to the minimum value. For example:

```
x = int8(300)

x =

  int8

   127

x = int8(-300)

x =

  int8

   -128
```

Also, when the result of an arithmetic operation involving integers exceeds the maximum (or minimum) value of the data type, MATLAB sets it to the maximum (or minimum) value:

```
x = int8(100)*3

x =

  int8

   127

x = int8(-100)*3

x =

  int8

   -128
```

Loss of Precision Due to Conversion

When you create a numeric array of large integers (larger than `flintmax`), MATLAB initially represents the input as double precision by default. Precision can be lost when you convert this input to the `int64` or `uint64` data type. To maintain precision, call `int64` or `uint64` with each scalar element of the array instead.

For example, convert a numeric array of large integers to a 64-bit signed integer array by using `int64`. The output array loses precision.

```
Y_inaccurate = int64([-72057594035891654 81997179153022975])

Y_inaccurate = 1×2 int64 row vector

   -72057594035891656    81997179153022976
```

Instead, call `int64` with each scalar element to return an accurate array.

```
Y_accurate = [int64(-72057594035891654) int64(81997179153022975)]
```

```
Y_accurate = 1×2 int64 row vector

   -72057594035891654    81997179153022975
```

You can also create the integer array without loss of precision by using the hexadecimal or binary values of the integers.

```
Y_accurate = [0xFF000000001F123As64 0x1234FFFFFFFFFFs64]

Y_accurate = 1×2 int64 row vector

   -72057594035891654    81997179153022975
```

Mathematics

- “Linear Algebra” on page 3-2
- “Create Arrays of Random Numbers” on page 3-29
- “Operations on Nonlinear Functions” on page 3-32

Linear Algebra

In this section...
“Matrices in the MATLAB Environment” on page 3-2
“Powers and Exponentials” on page 3-10
“Systems of Linear Equations” on page 3-13
“Eigenvalues” on page 3-22
“Singular Values” on page 3-25

Matrices in the MATLAB Environment

This topic contains an introduction to creating matrices and performing basic matrix calculations in MATLAB.

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. The mathematical operations defined on matrices are the subject of linear algebra.

Creating Matrices

MATLAB has many functions that create different kinds of matrices. For example, you can create a symmetric matrix with entries based on Pascal's triangle:

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

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

Or, you can create an unsymmetric *magic square matrix*, which has equal row and column sums:

```
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. In this case the first input to `randi` describes the range of possible values for the integers, and the second two inputs describe the number of rows and columns.

```
C = randi(10,3,2)
```

```
C =

     9    10
    10     7
     2     1
```

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. To define a matrix manually, use square brackets `[ ]` to denote the beginning and end of the array. Within the brackets, use a semicolon `;` to denote the end of a row. In the case of a scalar (1-by-1 matrix), the brackets are not required. For example, these statements produce a column vector, a row vector, and a scalar:

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

v = [2 0 -1]

s = 7
```

```
u =
       3
       1
       4

v =
       2     0    -1

s =
       7
```

For more information about creating and working with matrices, see “Creating, Concatenating, and Expanding Matrices”.

Adding and Subtracting Matrices

Addition and subtraction of matrices and arrays is performed element-by-element, or *element-wise*. For example, adding `A` to `B` and then subtracting `A` from the result recovers `B`:

```
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 compatible dimensions. If the dimensions are incompatible, an error results:

```
X = A + C
```

```
Arrays have incompatible sizes for this operation.
```

For more information, see “Array vs. Matrix Operations”.

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, called the *inner product*, or a matrix, called 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}. For complex matrices, another consideration is whether to take the complex conjugate of complex entries in the array to form the *complex conjugate transpose*. MATLAB uses the apostrophe operator (`'`) to perform a complex conjugate transpose, and 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 = pascal(3)` is *symmetric*, so `A'` is equal to `A`. However, `B = magic(3)` is not symmetric, so `B'` has the elements reflected along the main diagonal:

```
B = magic(3)

B =

     8     1     6
     3     5     7
     4     9     2

X = B'

X =

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

For vectors, transposition turns a row vector into a column vector (and vice-versa):

```
x = v'

x =
       2
       0
      -1
```

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

```
x'*y
```

and

```
y'*x
```

produce the same scalar result. This quantity is used so frequently, it has three different names: *inner* product, *scalar* product, or *dot* product. There is even a dedicated function for dot products named `dot`.

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. For example, consider the complex matrix

```
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   0.0000 + 9.0000i   4.0000 + 7.0000i
```

The complex conjugate transpose of `z` is:

```
z'
```

```
ans =

   1.0000 - 2.0000i   6.0000 + 2.0000i
   7.0000 + 3.0000i   0.0000 - 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   0.0000 + 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 an asterisk to denote matrix multiplication, as in `C = A*B`. Matrix multiplication is not commutative; that is, `A*B` is typically not equal to `B*A`:

```
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. Since `A` is 3-by-3 and `u` is 3-by-1, you can multiply them to get a 3-by-1 result (the common inner dimension cancels):

```
x = A*u

x =

     8
    17
    30
```

However, the multiplication does not work in the reverse order:

```
y = u*A

Error using  *
Incorrect dimensions for matrix multiplication. Check that the number of columns
in the first matrix matches the number of rows in the second matrix. To operate
on each element of the matrix individually, use TIMES (.*) for elementwise
multiplication.
```

You can multiply anything with a scalar:

```
s = 10;
w = s*y
```

```
w =

   120   -70   100
```

When you multiply an array by a scalar, the scalar implicitly expands to be the same size as the other input. This is often referred to as *scalar expansion*.

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.

Matrix Inverse

If a matrix `A` is square and nonsingular (nonzero determinant), then the equations $AX = I$ and $XA = I$ have the same solution X. This solution is called the *inverse* of `A` and is denoted A^{-1}. The `inv` function and the expression `A^-1` both compute the matrix inverse.

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

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

```
X = inv(A)
```

```
X =

    3.0000   -3.0000    1.0000
   -3.0000    5.0000   -2.0000
    1.0000   -2.0000    1.0000
```

```
A*X
```

```
ans =

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

The *determinant* calculated by `det` is a measure of the scaling factor of the linear transformation described by the matrix. When the determinant is exactly zero, the matrix is *singular* and no inverse exists.

```
d = det(A)
```

```
d =

     1
```

Some matrices are *nearly singular*, and despite the fact that an inverse matrix exists, the calculation is susceptible to numerical errors. The `cond` function computes the *condition number for inversion*, which gives an indication of the accuracy of the results from matrix inversion. The condition number ranges from `1` for a numerically stable matrix to `Inf` for a singular matrix.

```
c = cond(A)
```

```
c =

   61.9839
```

It is seldom necessary to form the explicit inverse of a matrix. A frequent misuse of `inv` arises when solving the system of linear equations $Ax = b$. The best way to solve this equation, from the standpoint of both execution time and numerical accuracy, is to use the matrix backslash operator `x = A\b`. See `mldivide` for more information.

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 such that each element of `X` is multiplied by the entire matrix `Y`:

```
[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:

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

```
ans =

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

and

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

```
ans =

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

Aside from `kron`, some other functions that are useful to replicate arrays are `repmat`, `repelem`, and `blkdiag`.

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 operation is defined for 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* or *vector magnitude*:

```
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 $p = 2$:

```
A = pascal(3);
[norm(A,1) norm(A) norm(A,inf)]
```

```
ans =

   10.0000    7.8730   10.0000
```

In cases where you want to calculate the norm of each row or column of a matrix, you can use `vecnorm`:

```
vecnorm(A)
```

```
ans =

    1.7321    3.7417    6.7823
```

Using Multithreaded Computation with Linear Algebra Functions

MATLAB 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.

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.

Powers and Exponentials

This topic shows how to compute matrix powers and exponentials using a variety of methods.

Positive Integer Powers

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

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

```
ans = 3×3

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

Inverse and Fractional Powers

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

```
A^(-3)
```

```
ans = 3×3

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

MATLAB® calculates `inv(A)` and `A^(-1)` with the same algorithm, so the results are exactly the same. Both `inv(A)` and `A^(-1)` produce warnings if the matrix is close to being singular.

```
isequal(inv(A),A^(-1))
```

```
ans = logical
   1
```

Fractional powers, such as `A^(2/3)`, are also permitted. The results using fractional powers depend on the distribution of the eigenvalues of the matrix.

```
A^(2/3)
```

```
ans = 3×3
```

```
    0.8901    0.5882    0.3684
    0.5882    1.2035    1.3799
    0.3684    1.3799    3.1167
```

Element-by-Element Powers

The `.^` operator calculates element-by-element powers. For example, to square each element in a matrix you can use `A.^2`.

```
A.^2
```

```
ans = 3×3

     1     1     1
     1     4     9
     1     9    36
```

Square Roots

The `sqrt` function is a convenient way to calculate the square root of each element in a matrix. An alternate way to do this is `A.^(1/2)`.

```
sqrt(A)
```

```
ans = 3×3

    1.0000    1.0000    1.0000
    1.0000    1.4142    1.7321
    1.0000    1.7321    2.4495
```

For other roots, you can use `nthroot`. For example, calculate `A.^(1/3)`.

```
nthroot(A,3)
```

```
ans = 3×3

    1.0000    1.0000    1.0000
    1.0000    1.2599    1.4422
    1.0000    1.4422    1.8171
```

These element-wise roots differ from the matrix square root, which calculates a second matrix *B* such that *A* = BB. 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.

```
B = sqrtm(A)
```

```
B = 3×3

    0.8775    0.4387    0.1937
    0.4387    1.0099    0.8874
    0.1937    0.8874    2.2749
```

```
B^2
```

```
ans = 3×3

    1.0000    1.0000    1.0000
    1.0000    2.0000    3.0000
    1.0000    3.0000    6.0000
```

Scalar Bases

In addition to raising a matrix to a power, you also can raise a scalar to the power of a matrix.

```
2^A
```

```
ans = 3×3

   10.4630   21.6602   38.5862
   21.6602   53.2807   94.6010
   38.5862   94.6010  173.7734
```

When you raise a scalar to the power of a matrix, MATLAB uses the eigenvalues and eigenvectors of the matrix to calculate the matrix power. If `[V,D] = eig(A)`, then $2^A = V\,2^D\,V^{-1}$.

```
[V,D] = eig(A);
V*2^D*V^(-1)
```

```
ans = 3×3

   10.4630   21.6602   38.5862
   21.6602   53.2807   94.6010
   38.5862   94.6010  173.7734
```

Matrix Exponentials

The matrix exponential is a special case of raising a scalar to a matrix power. The base for a matrix exponential is Euler's number `e = exp(1)`.

```
e = exp(1);
e^A
```

```
ans = 3×3
10^3 ×

    0.1008    0.2407    0.4368
    0.2407    0.5867    1.0654
    0.4368    1.0654    1.9418
```

The `expm` function is a more convenient way to calculate matrix exponentials.

```
expm(A)
```

```
ans = 3×3
10^3 ×

    0.1008    0.2407    0.4368
    0.2407    0.5867    1.0654
```

```
    0.4368    1.0654    1.9418
```

The matrix exponential can be calculated in a number of ways. See “Matrix Exponentials” for more information.

Dealing with Small Numbers

The MATLAB functions `log1p` and `expm1` calculate $\log(1 + x)$ and $e^x - 1$ accurately for very small values of x. For example, if you try to add a number smaller than machine precision to 1, then the result gets rounded to 1.

```
log(1+eps/2)
```

```
ans =
0
```

However, `log1p` is able to return a more accurate answer.

```
log1p(eps/2)
```

```
ans =
1.1102e-16
```

Likewise for $e^x - 1$, if x is very small then it is rounded to zero.

```
exp(eps/2)-1
```

```
ans =
0
```

Again, `expm1` is able to return a more accurate answer.

```
expm1(eps/2)
```

```
ans =
1.1102e-16
```

Systems of Linear Equations

- “Computational Considerations” on page 3-13
- “General Solution” on page 3-15
- “Square Systems” on page 3-15
- “Overdetermined Systems” on page 3-17
- “Underdetermined Systems” on page 3-19
- “Solving for Several Right-Hand Sides” on page 3-20
- “Iterative Methods” on page 3-21
- “Multithreaded Computation” on page 3-22

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; MATLAB 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`. These operators 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 $xA = b$, obtained using `mrdivide`.
`x = A\b`	Denotes the solution to the matrix equation $Ax = b$, obtained using `mldivide`.

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` has size m-by-n, then there are three cases:

`m = n`	Square system. Seek an exact solution.
`m > n`	Overdetermined system, with more equations than unknowns. Find a least-squares solution.
`m < n`	Underdetermined system, with fewer equations than unknowns. 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-hand side column vector `b`.

Nonsingular Coefficient Matrix

If the matrix `A` is nonsingular, then the solution, `x = A\b`, is 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 or 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
```

`pinv(A)` is a pseudoinverse of A. If $Ax = b$ does not have an exact solution, then `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.

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=6×2 table
     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 = 6×2

    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
```

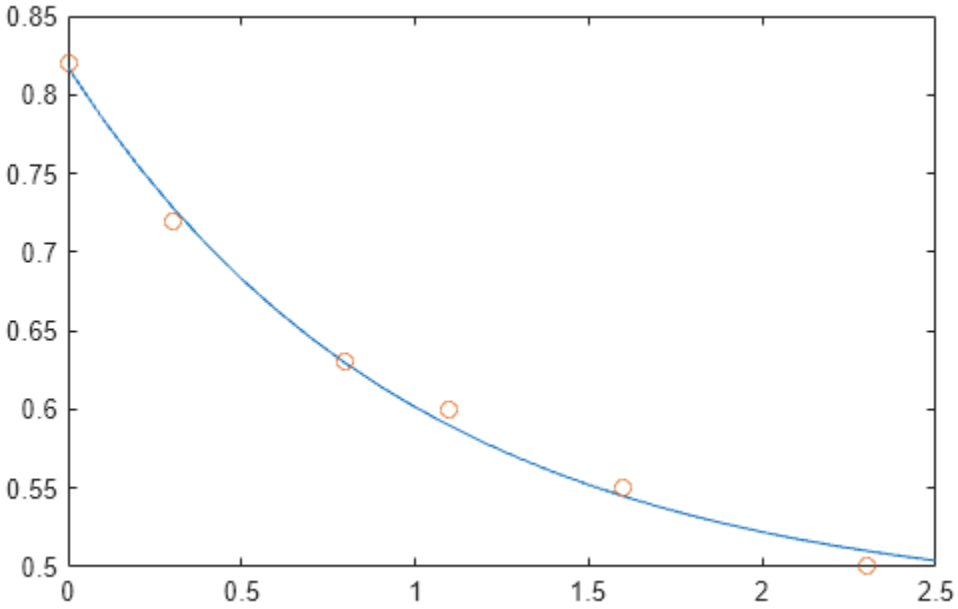
```
c = 2×1

    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, then the least-squares solution to `AX = B` is not unique. `A\B` issues a warning if `A` is rank deficient and produces a least-squares solution. You can use `lsqminnorm` to find the solution `X` that has the minimum norm among all 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 least-squares 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:

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

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
```

When infinitely many solutions are available, the solution with minimum norm is of particular interest. You can use `lsqminnorm` to compute the minimum-norm least-squares solution. This solution has the smallest possible value for `norm(p)`.

```
p = lsqminnorm(R,b)
```

```
p =

    -207/137
     365/137
      79/137
    -424/137
```

Solving for Several Right-Hand Sides

Some problems are concerned with solving linear systems that have the same coefficient matrix A, but different right-hand sides b. When the different values of b are available at the same time, you can construct b as a matrix with several columns and solve all of the systems of equations at the same time using a single backslash command: `X = A\[b1 b2 b3 …]`.

However, sometimes the different values of b are not all available at the same time, which means you need to solve several systems of equations consecutively. When you solve one of these systems of equations using slash (/) or backslash (\), the operator factorizes the coefficient matrix A and uses this matrix decomposition to compute the solution. However, each subsequent time you solve a similar system of equations with a different b, the operator computes the same decomposition of A, which is a redundant computation.

The solution to this problem is to precompute the decomposition of A, and then reuse the factors to solve for the different values of b. In practice, however, precomputing the decomposition in this manner can be difficult since you need to know which decomposition to compute (LU, LDL, Cholesky, and so on) as well as how to multiply the factors to solve the problem. For example, with LU decomposition you need to solve two linear systems to solve the original system $Ax = b$:

```
[L,U] = lu(A);
x = U \ (L \ b);
```

Instead, the recommended method for solving linear systems with several consecutive right-hand sides is to use `decomposition` objects. These objects enable you to leverage the performance

benefits of precomputing the matrix decomposition, but they *do not* require knowledge of how to use the matrix factors. You can replace the previous LU decomposition with:

```
dA = decomposition(A,'lu');
x = dA\b;
```

If you are unsure which decomposition to use, `decomposition(A)` chooses the correct type based on the properties of `A`, similar to what backslash does.

Here is a simple test of the possible performance benefits of this approach. The test solves the same sparse linear system 100 times using both backslash (\) and `decomposition`.

```
n = 1e3;
A = sprand(n,n,0.2) + speye(n);
b = ones(n,1);

% Backslash solution
tic
for k = 1:100
    x = A\b;
end
toc

Elapsed time is 9.006156 seconds.

% decomposition solution
tic
dA = decomposition(A);
for k = 1:100
    x = dA\b;
end
toc

Elapsed time is 0.374347 seconds.
```

For this problem, the `decomposition` solution is much faster than using backslash alone, yet the syntax remains simple.

Iterative Methods

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.

Function	Description
`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

Function	Description
`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

Multithreaded Computation

MATLAB 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.

Eigenvalues

- “Eigenvalue Decomposition” on page 3-22
- “Multiple Eigenvalues” on page 3-23
- “Schur Decomposition” on page 3-24

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 the coefficient matrix of the differential equation $dx/dt = Ax$:

```
A =
     0    -6    -1
```

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

The solution to this equation is expressed in terms of the matrix exponential $x(t) = e^{tA}x(0)$. The statement

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

produces a column vector containing the eigenvalues of `A`. 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

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

$A = USU'$,

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 an orthogonal basis, which has much better numerical properties than a set of eigenvectors.

For example, compare the eigenvalue and Schur decompositions of this defective matrix:

```
A = [ 6    12    19
     -9   -20   -33
      4     9    15 ];

[V,D] = eig(A)

V =

  -0.4741 + 0.0000i  -0.4082 - 0.0000i  -0.4082 + 0.0000i
   0.8127 + 0.0000i   0.8165 + 0.0000i   0.8165 + 0.0000i
  -0.3386 + 0.0000i  -0.4082 + 0.0000i  -0.4082 - 0.0000i


D =

  -1.0000 + 0.0000i   0.0000 + 0.0000i   0.0000 + 0.0000i
   0.0000 + 0.0000i   1.0000 + 0.0000i   0.0000 + 0.0000i
   0.0000 + 0.0000i   0.0000 + 0.0000i   1.0000 - 0.0000i

[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    0.0000    1.0000
```

The matrix `A` is defective since it does not have a full set of linearly independent eigenvectors (the second and third columns of `V` are the same). Since not all columns of `V` are linearly independent, it has a large condition number of about `~1e8`. However, `schur` is able to calculate three different basis vectors in `U`. Since `U` is orthogonal, `cond(U) = 1`.

The matrix `S` has the real eigenvalue as the first entry on the diagonal and the repeated eigenvalue represented by the lower right 2-by-2 block. The eigenvalues of the 2-by-2 block are also eigenvalues of `A`:

```
eig(S(2:3,2:3))
```

```
ans =

   1.0000 + 0.0000i
   1.0000 - 0.0000i
```

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 in a diagonal matrix Σ and the corresponding singular vectors forming the columns of two orthogonal matrices U and V, you obtain the equations

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

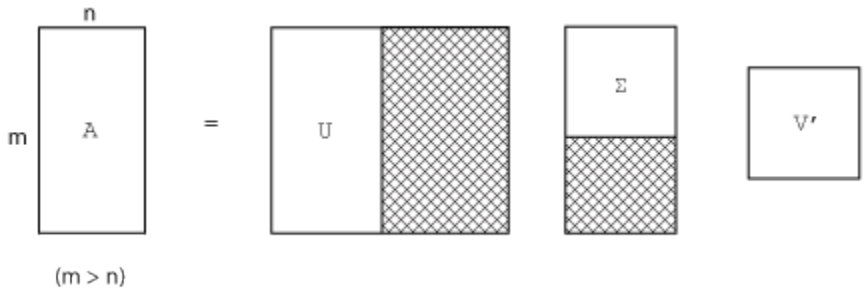
Since U and V are unitary matrices, multiplying the first equation by V^H on the right yields the singular value decomposition equation

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

The full singular value decomposition of an m-by-n matrix involves:

- m-by-m matrix U
- m-by-n matrix Σ
- n-by-n matrix 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 (`m > n`), then the resulting `m`-by-`m` matrix U is large. However, most of the columns in U 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,"econ")
```

```
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.

Batched SVD Calculation

If you need to decompose a large collection of matrices that have the same size, it is inefficient to perform all of the decompositions in a loop with `svd`. Instead, you can concatenate all of the matrices into a multidimensional array and use `pagesvd` to perform singular value decompositions on all of the array pages with a single function call.

Function	Usage
`pagesvd`	Use `pagesvd` to perform singular value decompositions on the pages of a multidimensional array. This is an efficient way to perform SVD on a large collection of matrices that all have the same size.

For example, consider a collection of three 2-by-2 matrices. Concatenate the matrices into a 2-by-2-by-3 array with the `cat` function.

```
A = [0 -1; 1 0];
B = [-1 0; 0 -1];
C = [0 1; -1 0];
X = cat(3,A,B,C);
```

Now, use `pagesvd` to simultaneously perform the three decompositions.

```
[U,S,V] = pagesvd(X);
```

For each page of `X`, there are corresponding pages in the outputs `U`, `S`, and `V`. For example, the matrix `A` is on the first page of `X`, and its decomposition is given by `U(:,:,1)*S(:,:,1)*V(:,:,1)'`.

Low-Rank SVD Approximations

For large sparse matrices, using `svd` to calculate *all* of the singular values and singular 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 extra work.

In cases where only a subset of the singular values and singular vectors are required, the `svds` and `svdsketch` functions are preferred over `svd`.

Function	Usage
`svds`	Use `svds` to calculate a rank-*k* approximation of the SVD. You can specify whether the subset of singular values should be the largest, the smallest, or the closest to a specific number. `svds` generally calculates the best possible rank-*k* approximation.
`svdsketch`	Use `svdsketch` to calculate a partial SVD of the input matrix satisfying a specified tolerance. While `svds` requires that you specify the rank, `svdsketch` adaptively determines the rank of the matrix sketch based on the specified tolerance. The rank-*k* approximation that `svdsketch` ultimately uses satisfies the tolerance, but unlike `svds`, it is not guaranteed to be the best one possible.

For example, consider 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` or `svdsketch`. However, for truly large and sparse matrices, using `svds` or `svdsketch` becomes necessary.

Create Arrays of Random Numbers

In this section...
“Random Number Functions” on page 3-29
“Random Number Generators” on page 3-30
“Random Number Data Types” on page 3-30

MATLAB uses algorithms to generate pseudorandom and pseudoindependent numbers. These numbers are not strictly random and independent in the mathematical sense, but they pass various statistical tests of randomness and independence, and their calculation can be repeated for testing or diagnostic purposes.

The `rand`, `randi`, `randn`, and `randperm` functions are the primary functions for creating arrays of random numbers. The `rng` function allows you to control the seed and algorithm that generates random numbers.

Random Number Functions

There are four fundamental random number functions: `rand`, `randi`, `randn`, and `randperm`. The `rand` function returns floating-point numbers between 0 and 1 that are drawn from a uniform distribution. For example, create a 1000-by-1 column vector containing real floating-point numbers drawn from a uniform distribution.

```
rng("default")
r1 = rand(1000,1);
```

All the values in `r1` are in the open interval (0,1). A histogram of these values is roughly flat, which indicates a fairly uniform sampling of numbers.

The `randi` function returns `double` integer values drawn from a discrete uniform distribution. For example, create a 1000-by-1 column vector containing integer values drawn from a discrete uniform distribution.

```
r2 = randi(10,1000,1);
```

All the values in `r2` are in the close interval [1, 10]. A histogram of these values is roughly flat, which indicates a fairly uniform sampling of integers between 1 and 10.

The `randn` function returns arrays of real floating-point numbers that are drawn from a standard normal distribution. For example, create a 1000-by-1 column vector containing numbers drawn from a standard normal distribution.

```
r3 = randn(1000,1);
```

A histogram of `r3` looks like a roughly normal distribution whose mean is 0 and standard deviation is 1.

You can use the `randperm` function to create a `double` array of random integer values that have no repeated values. For example, create a 1-by-5 array containing integers randomly selected from the range [1, 15].

```
r4 = randperm(15,5);
```

Unlike `randi`, which can return an array containing repeated values, the array returned by `randperm` has no repeated values.

Successive calls to any of these functions return different results. This behavior is useful for creating several different arrays of random values.

Random Number Generators

MATLAB offers several generator algorithm options, which are summarized in the table.

Value	Generator Name	Generator Keyword
`"twister"`	Mersenne Twister	mt19937ar
`"simdTwister"`	SIMD-Oriented Fast Mersenne Twister	dsfmt19937
`"combRecursive"`	Combined Multiple Recursive	mrg32k3a
`"multFibonacci"`	Multiplicative Lagged Fibonacci	mlfg6331_64
`"philox"`	Philox 4x32 generator with 10 rounds	philox4x32_10
`"threefry"`	Threefry 4x64 generator with 20 rounds	threefry4x64_20
`"v4"`	Legacy MATLAB version 4.0 generator	mcg16807
`"v5uniform"`	Legacy MATLAB version 5.0 uniform generator	swb2712
`"v5normal"`	Legacy MATLAB version 5.0 normal generator	shr3cong

Use the `rng` function to set the seed and generator used by the `rand`, `randi`, `randn`, and `randperm` functions.

For example, `rng(0,"twister")` sets the seed to 0 and the generator algorithm to Mersenne Twister. To avoid repetition of random number arrays when MATLAB restarts, see “Why Do Random Numbers Repeat After Startup?”

For more information about controlling the random number generator's state to repeat calculations using the same random numbers, or to guarantee that different random numbers are used in repeated calculations, see “Controlling Random Number Generation”.

You can set the default algorithm and seed in MATLAB settings. If you do not change these settings, then `rng` uses the factory value of `"twister"` for the Mersenne Twister generator with seed 0, as in previous releases. For more information, see “Default Settings for Random Number Generator” and “Reproducibility for Random Number Generator”.

Random Number Data Types

`rand` and `randn` functions generate values in double precision by default.

```
rng("default")
A = rand(1,5);
class(A)
```

```
ans = 'double'
```

To specify the class as double explicitly:

```
rng("default")
B = rand(1,5,"double");
class(B)
```

```
ans = 'double'
```

```
isequal(A,B)
```

```
ans =
1
```

`rand` and `randn` can also generate values in single precision.

```
rng("default")
A = rand(1,5,"single");
class(A)
```

```
ans = 'single'
```

The values are the same as if you had cast the double precision values from the previous example. The random stream that the functions draw from advances the same way regardless of what class of values is returned.

```
A,B
```

```
A =
    0.8147    0.9058    0.1270    0.9134    0.6324


B =
    0.8147    0.9058    0.1270    0.9134    0.6324
```

`randi` supports both integer types and single or double precision.

```
A = randi([1 10],1,5,"double");
class(A)
```

```
ans = 'double'
```

```
B = randi([1 10],1,5,"uint8");
class(B)
```

```
ans = 'uint8'
```

Operations on Nonlinear Functions

In this section...
“Create Function Handle” on page 3-32 “Pass Function to Another Function” on page 3-34

Create Function Handle

You can create function handles to named and anonymous functions. You can store multiple function handles in an array, and save and load them, as you would any other variable.

What Is a Function Handle?

A function handle is a MATLAB® data type that stores an association to a function. Indirectly calling a function enables you to invoke the function regardless of where you call it from. Typical uses of function handles include:

- Passing a function to another function (often called *function functions*). For example, passing a function to integration and optimization functions, such as `integral` and `fzero`.
- Specifying callback functions (for example, a callback that responds to a UI event or interacts with data acquisition hardware).
- Constructing handles to functions defined inline instead of stored in a program file (anonymous functions).
- Calling local functions from outside the main function.

You can see if a variable, `h`, is a function handle using `isa(h,'function_handle')`.

Creating Function Handles

To create a handle for a function, precede the function name with an `@` sign. For example, if you have a function called `myfunction`, create a handle named `f` as follows:

```
f = @myfunction;
```

You call a function using a handle the same way you call the function directly. For example, suppose that you have a function named `computeSquare`, defined as:

```
function y = computeSquare(x)
y = x.^2;
end
```

Create a handle and call the function to compute the square of four.

```
f = @computeSquare;
a = 4;
b = f(a)
```

```
b =
16
```

If the function does not require any inputs, then you can call the function with empty parentheses, such as

```
h = @ones;
a = h()

a =
1
```

Without the parentheses, the assignment creates another function handle.

```
a = h

a = function_handle with value:
    @ones
```

Function handles are variables that you can pass to other functions. For example, calculate the integral of x^2 on the range [0,1].

```
q = integral(f,0,1);
```

Function handles store their absolute path, so when you have a valid handle, you can invoke the function from any location. You do not have to specify the path to the function when creating the handle, only the function name.

Keep the following in mind when creating handles to functions:

- Name length — Each part of the function name (including package and class names) must be less than the number specified by `namelengthmax`. Otherwise, MATLAB truncates the latter part of the name.
- Scope — The function must be in scope at the time you create the handle. Therefore, the function must be on the MATLAB path or in the current folder. Or, for handles to local or nested functions, the function must be in the current file.
- Precedence — When there are multiple functions with the same name, MATLAB uses the same precedence rules to define function handles as it does to call functions. For more information, see “Function Precedence Order”.
- Overloading — When a function handle is invoked with one or more arguments, MATLAB determines the dominant argument. If the dominant argument is an object, MATLAB determines if the object’s class has a method which overloads the same name as the function handle’s associated function. If it does, then the object’s method is invoked instead of the associated function.

Anonymous Functions

You can create handles to anonymous functions. An anonymous function is a one-line expression-based MATLAB function that does not require a program file. Construct a handle to an anonymous function by defining the body of the function, `anonymous_function`, and a comma-separated list of input arguments to the anonymous function, `arglist`. The syntax is:

```
h = @(arglist)anonymous_function
```

For example, create a handle, `sqr`, to an anonymous function that computes the square of a number, and call the anonymous function using its handle.

```
sqr = @(n) n.^2;
x = sqr(3)
```

```
x =
9
```

For more information, see “Anonymous Functions”.

Arrays of Function Handles

You can create an array of function handles by collecting them into a cell or structure array. For example, use a cell array:

```
C = {@sin, @cos, @tan};
C{2}(pi)
```

```
ans =
-1
```

Or use a structure array:

```
S.a = @sin;  S.b = @cos;  S.c = @tan;
S.a(pi/2)
```

```
ans =
1
```

Saving and Loading Function Handles

You can save and load function handles in MATLAB, as you would any other variable. In other words, use the `save` and `load` functions. If you save a function handle, MATLAB saves the absolute path information. You can invoke the function from any location that MATLAB is able to reach, as long as the file for the function still exists at this location. An invalid handle occurs if the file location or file name has changed since you created the handle. If a handle is invalid, MATLAB might display a warning when you load the file. When you invoke an invalid handle, MATLAB issues an error.

Pass Function to Another Function

You can use function handles as input arguments to other functions, which are called *function functions*. These functions evaluate mathematical expressions over a range of values. Typical function functions include `integral`, `quad2d`, `fzero`, and `fminbnd`.

For example, to find the integral of the natural log from 0 through 5, pass a handle to the `log` function to `integral`.

```
a = 0;
b = 5;
q1 = integral(@log,a,b)
```

```
q1 =
3.0472
```

Similarly, to find the integral of the `sin` function and the `exp` function, pass handles to those functions to `integral`.

```
q2 = integral(@sin,a,b)
```

```
q2 =
0.7163
```

```
q3 = integral(@exp,a,b)
```

```
q3 =
147.4132
```

Also, you can pass a handle to an anonymous function to function functions. An anonymous function is a one-line expression-based MATLAB® function that does not require a program file. For example, evaluate the integral of $x/(e^x - 1)$ on the range `[0,Inf]`:

```
fun = @(x)x./(exp(x)-1);
q4 = integral(fun,0,Inf)
```

```
q4 =
1.6449
```

Functions that take a function as an input (called *function functions*) expect that the function associated with the function handle has a certain number of input variables. For example, if you call `integral` or `fzero`, the function associated with the function handle must have exactly one input variable. If you call `integral3`, the function associated with the function handle must have three input variables. For information on calling function functions with more variables, see "Parameterizing Functions".

You can write functions that accept function handles the same way as writing functions to accept other types of inputs. Write a function that doubles the output of the input function handle for a given input.

```
function x = doubleFunction(funHandle,funInput)
    x = 2*funHandle(funInput);
end
```

Test this function by providing a function handle as the input.

```
x = doubleFunction(fun,4)
```

```
x =
0.1493
```

Graphics

- “Create 2-D Line Plot” on page 4-2
- “Format and Annotate Charts” on page 4-7
- “Combine Multiple Plots” on page 4-29
- “Create Chart with Two y-Axes” on page 4-36
- “Surface and Mesh Plots” on page 4-43

Create 2-D Line Plot

Create a simple line plot and label the axes. Customize the appearance of plotted lines by changing the line color, the line style, and adding markers.

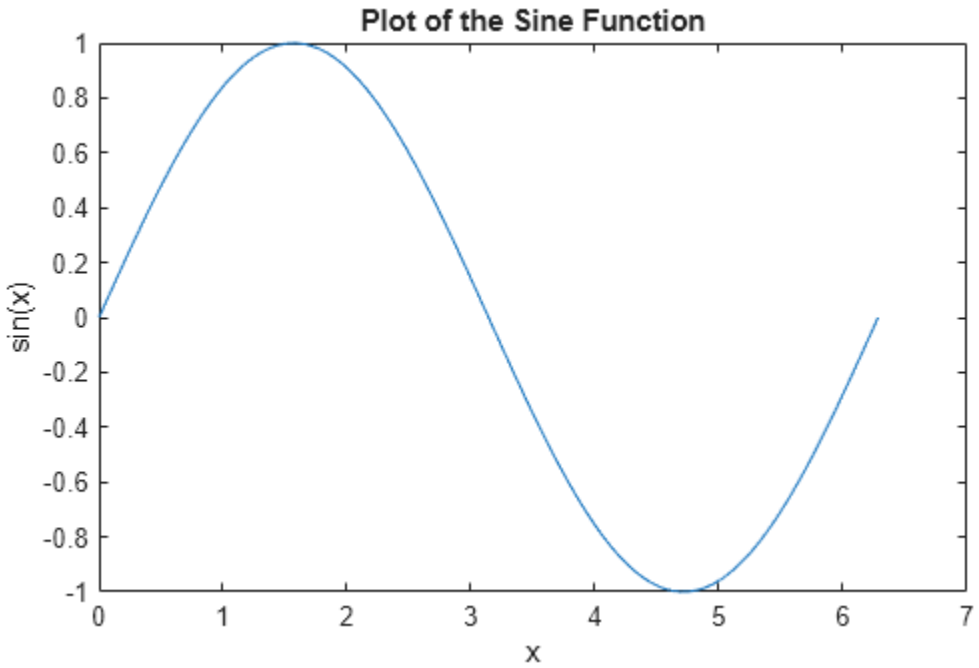
Create Line Plot

Create a two-dimensional line plot using the `plot` function. For example, plot the value of the sine function from 0 to 2π.

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

Label the axes and add a title.

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

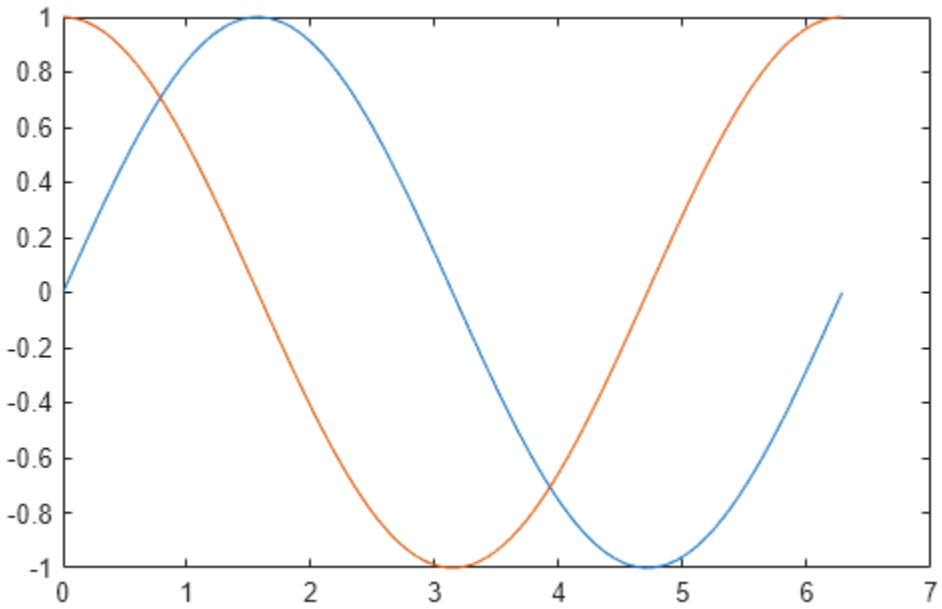


Plot Multiple Lines

By default, MATLAB clears the figure before each plotting command. Use the `figure` command to open a new figure window. You can plot multiple lines using the `hold on` command. Until you use `hold off` or close the window, all plots appear in the current figure window.

```
figure
x = linspace(0,2*pi,100);
y = sin(x);
plot(x,y)
```

```
hold on
y2 = cos(x);
plot(x,y2)
hold off
```



Change Line Appearance

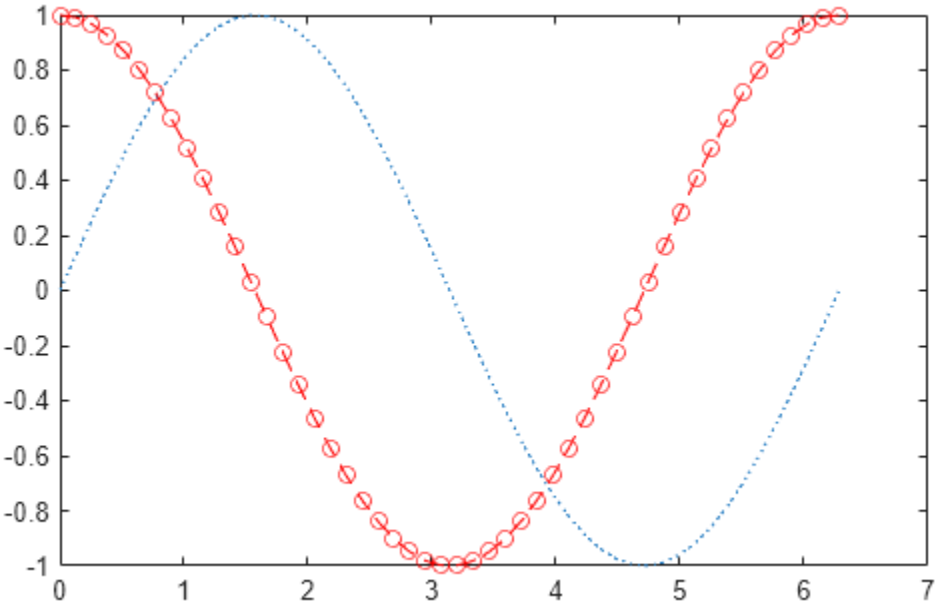
You can change the line color, line style, or add markers by including an optional line specification when calling the `plot` function. For example:

- `':'` plots a dotted line.
- `'g:'` plots a green, dotted line.
- `'g:*'` plots a green, dotted line with star markers.
- `'*'` plots star markers with no line.

The symbols can appear in any order. You do not need to specify all three characteristics (line color, style, and marker). For more information about the different style options, see the `plot` function page.

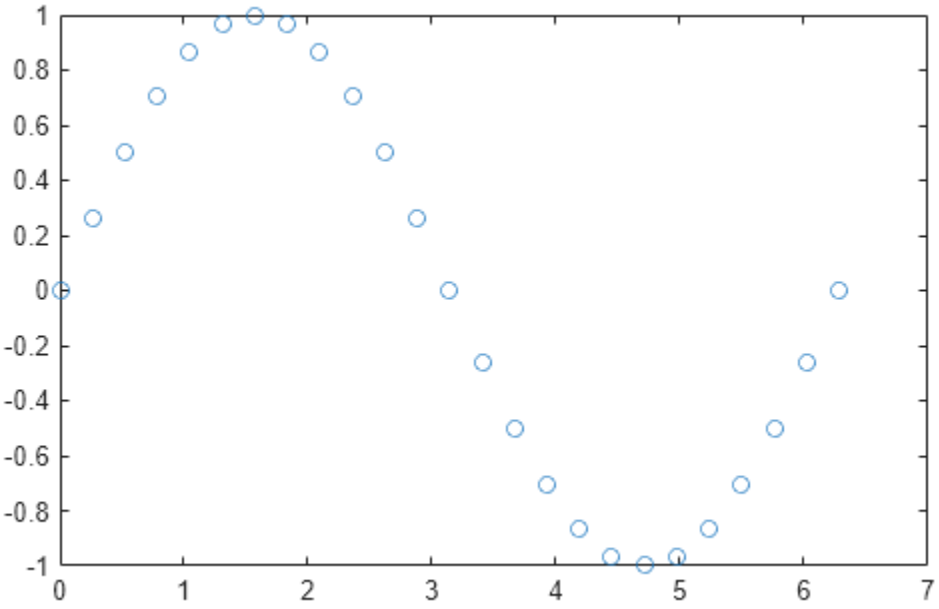
For example, plot a dotted line. Add a second plot that uses a dashed, red line with circle markers.

```
x = linspace(0,2*pi,50);
y = sin(x);
plot(x,y,':')

hold on
y2 = cos(x);
plot(x,y2,'--ro')
hold off
```



Plot only the data points by omitting the line style option from the line specification.

```
x = linspace(0,2*pi,25);
y = sin(x);
plot(x,y,'o')
```



Change Line Object Properties

You also can customize the appearance of the plot by changing properties of the `Line` object used to create the plot.

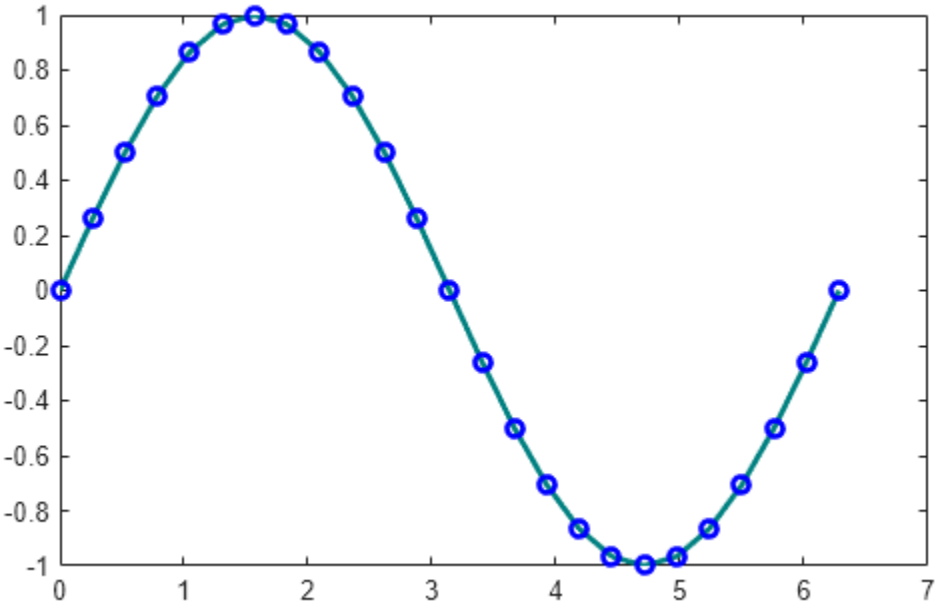
Create a line plot. Assign the `Line` object created to the variable `ln`. The display shows commonly used properties, such as `Color`, `LineStyle`, and `LineWidth`.

```
x = linspace(0,2*pi,25);
y = sin(x);
ln = plot(x,y)
```

```
ln =
  Line with properties:

              Color: [0.0660 0.4430 0.7450]
          LineStyle: '-'
          LineWidth: 0.5000
             Marker: 'none'
         MarkerSize: 6
    MarkerFaceColor: 'none'
              XData: [0 0.2618 0.5236 0.7854 1.0472 1.3090 1.5708 1.8326 2.0944 2.3562 2.6180 2.8
              YData: [0 0.2588 0.5000 0.7071 0.8660 0.9659 1 0.9659 0.8660 0.7071 0.5000 0.2588 1

  Show all properties
```

To access individual properties, use dot notation. For example, change the line width to 2 points and set the line color to an RGB triplet color value, in this case `[0 0.5 0.5]`. Add blue, circle markers.

```
ln.LineWidth = 2;
ln.Color = [0 0.5 0.5];
ln.Marker = 'o';
ln.MarkerEdgeColor = 'b';
```


1
0.8
0.6
0.4
0.2
0
-0.2
-0.4
-0.6
-0.8
-1
0
1
2
3
4
5
6
7

Format and Annotate Charts

In this section...
“Add Title and Axis Labels to Chart” on page 4-7
“Specify Axis Limits” on page 4-11
“Specify Axis Tick Values and Labels” on page 4-16
“Add Legend to Graph” on page 4-22

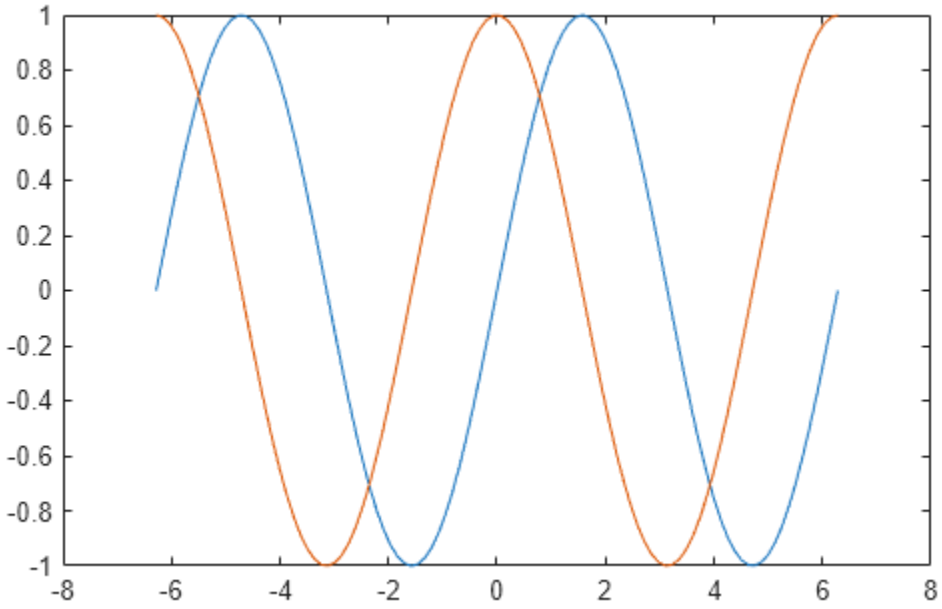
Add Title and Axis Labels to Chart

This example shows how to add a title and axis labels to a chart by using the `title`, `xlabel`, and `ylabel` functions. It also shows how to customize the appearance of the axes text by changing the font size.

Create Simple Line Plot

Create `x` as 100 linearly spaced values between -2π and 2π. Create `y1` and `y2` as sine and cosine values of `x`. Plot both sets of data.

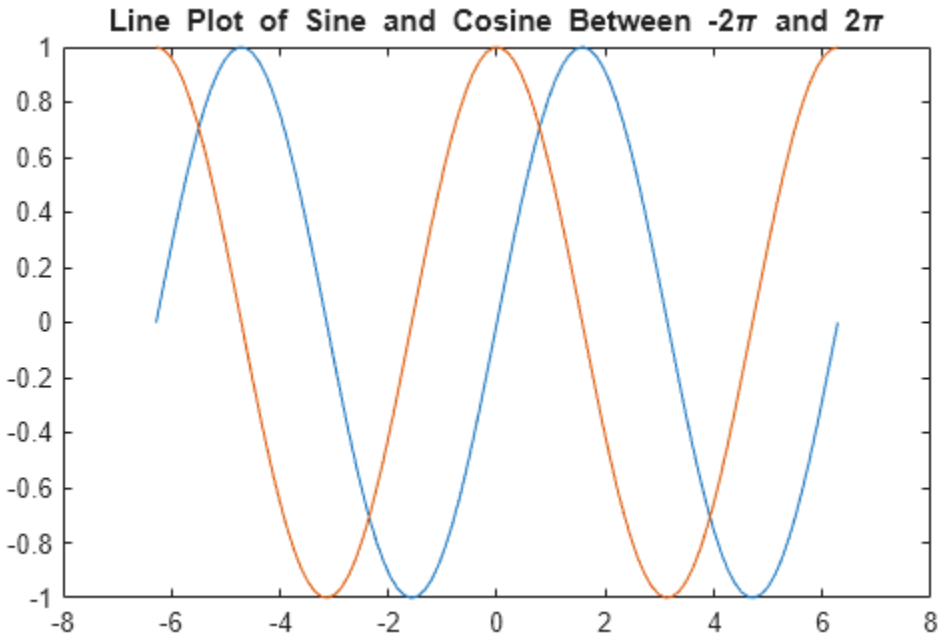
```
x = linspace(-2*pi,2*pi,100);
y1 = sin(x);
y2 = cos(x);
figure
plot(x,y1,x,y2)
```



Add Title

Add a title to the chart by using the `title` function. To display the Greek symbol π, use the TeX markup, `\pi`.

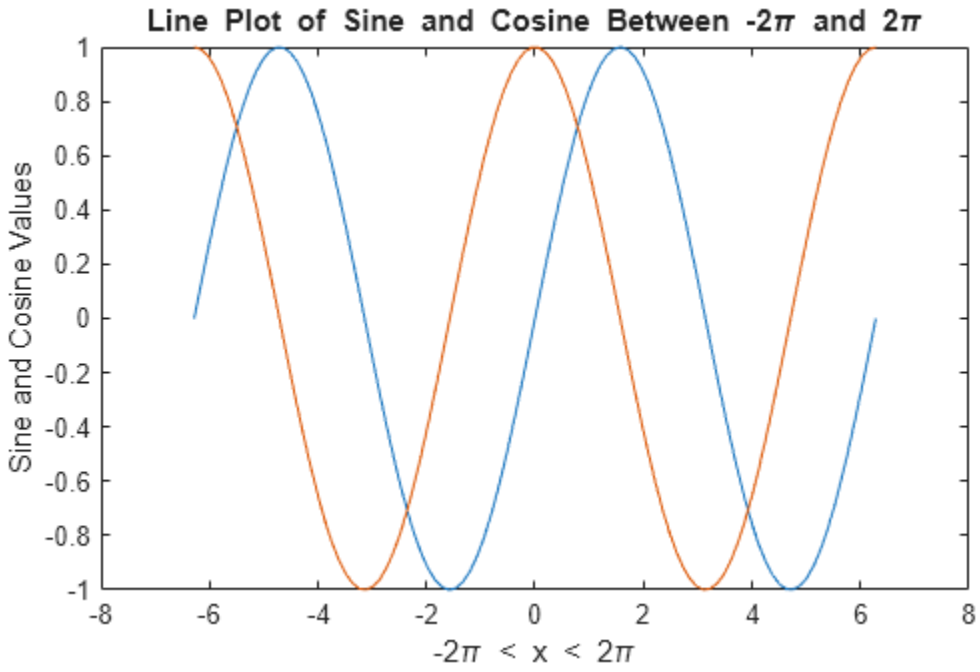
```
title('Line Plot of Sine and Cosine Between -2\pi and 2\pi')
```



Add Axis Labels

Add axis labels to the chart by using the `xlabel` and `ylabel` functions.

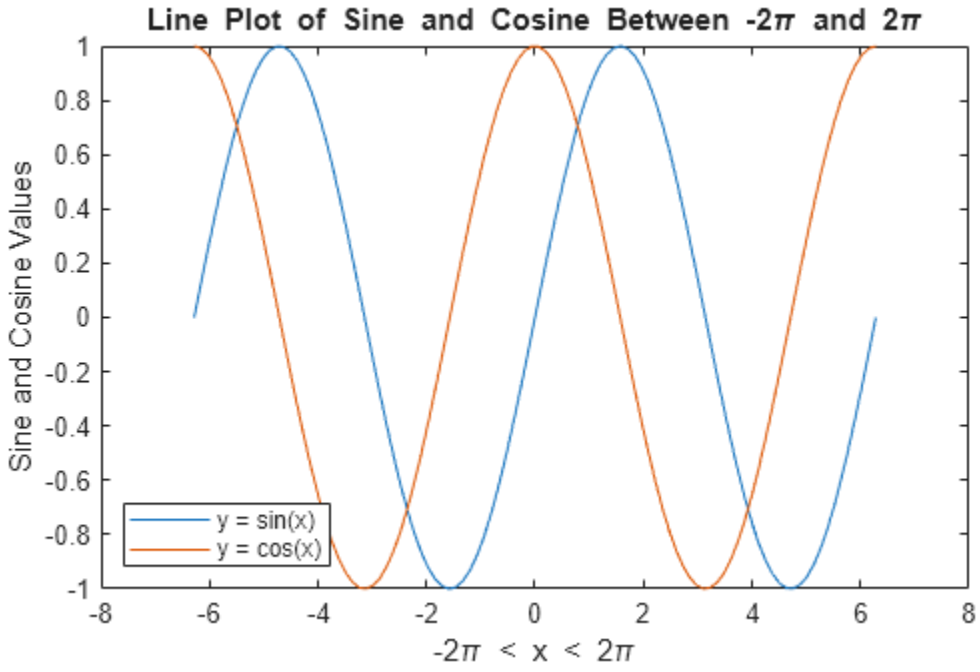
```
xlabel('-2\pi < x < 2\pi')
ylabel('Sine and Cosine Values')
```



Add Legend

Add a legend to the graph that identifies each data set using the `legend` function. Specify the legend descriptions in the order that you plot the lines. Optionally, specify the legend location using one of the eight cardinal or intercardinal directions, in this case, `'southwest'`.

```
legend({'y = sin(x)','y = cos(x)'},'Location','southwest')
```

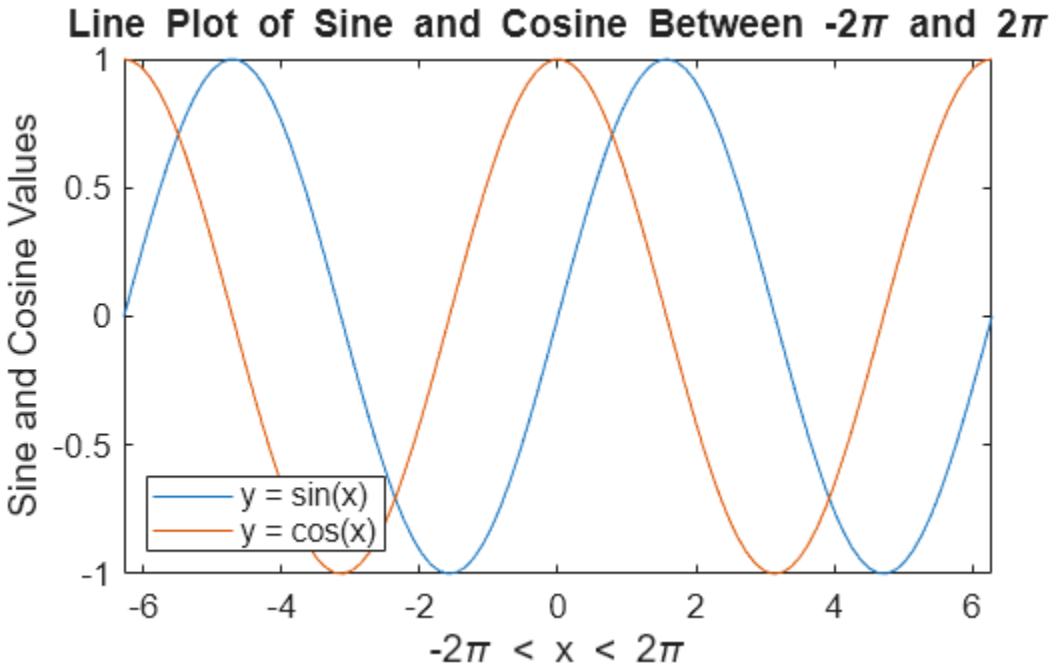


Change Font Size

`Axes` objects have properties that you can use to customize the appearance of the axes. For example, the `FontSize` property controls the font size of the title, labels, and legend.

Access the current `Axes` object using the `gca` function. Then use dot notation to set the `FontSize` property.

```
ax = gca;
ax.FontSize = 13;
```



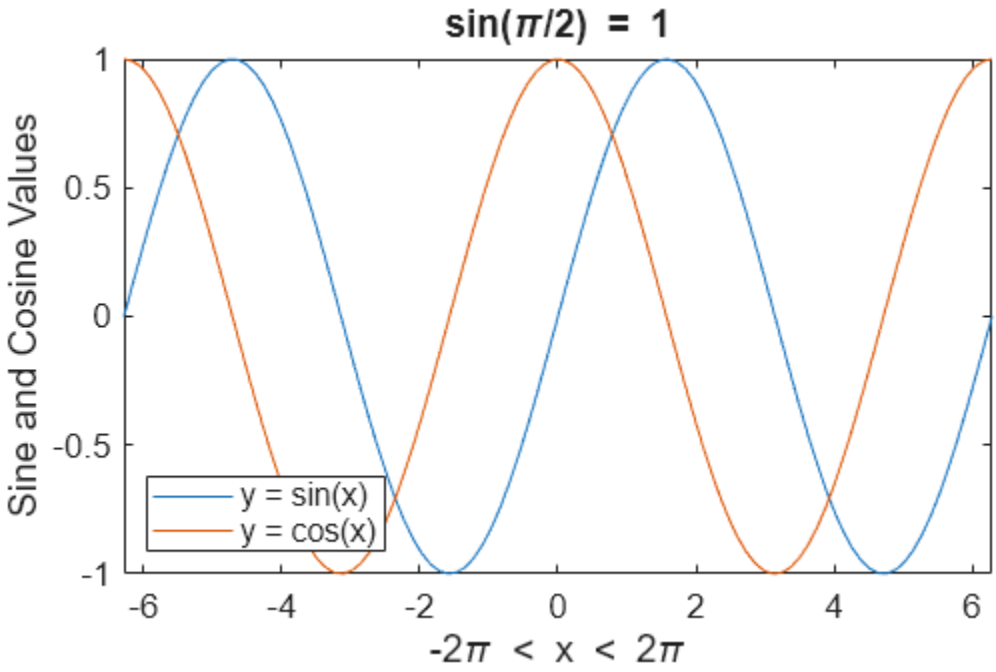
Alternatively, starting in R2022a, you can change the font size of the axes text by using the `fontsize` function.

Title with Variable Value

Include a variable value in the title text by using the `num2str` function to convert the value to text. You can use a similar approach to add variable values to axis labels or legend entries.

Add a title with the value of sin(π)/2.

```
k = sin(pi/2);
title(['sin(\pi/2) = ' num2str(k)])
```



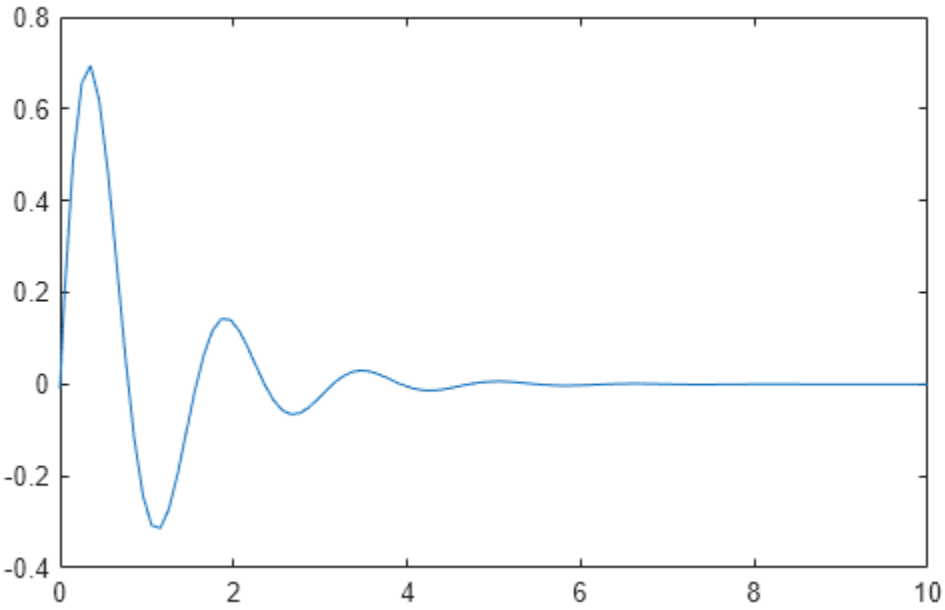
Specify Axis Limits

You can control where data appears in the axes by setting the *x*-axis, *y*-axis, and *z*-axis limits. You also can change where the *x*-axis and *y*-axis lines appear (2-D plots only) or reverse the direction of increasing values along each axis.

Change Axis Limits

Create a line plot. Specify the axis limits using the `xlim` and `ylim` functions. For 3-D plots, use the `zlim` function. Pass the functions a two-element vector of the form `[min max]`.

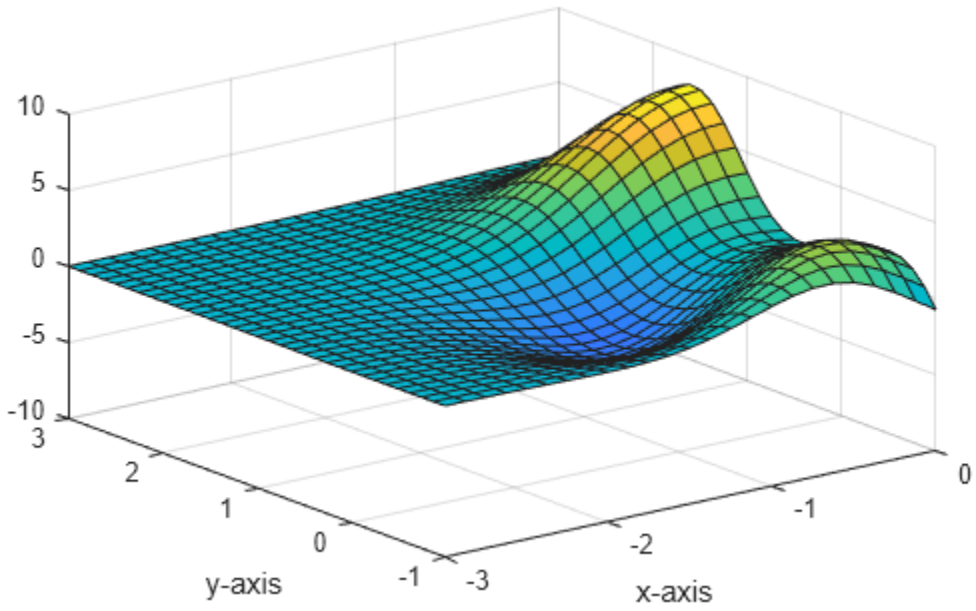
```
x = linspace(-10,10,200);
y = sin(4*x)./exp(x);
plot(x,y)
xlim([0 10])
ylim([-0.4 0.8])
```



Use Semiautomatic Axis Limits

Set the maximum *x*-axis limit to 0 and the minimum *y*-axis limit to -1. Let MATLAB choose the other limits. For an automatically calculated minimum or maximum limit, use `-inf` or `inf`, respectively.

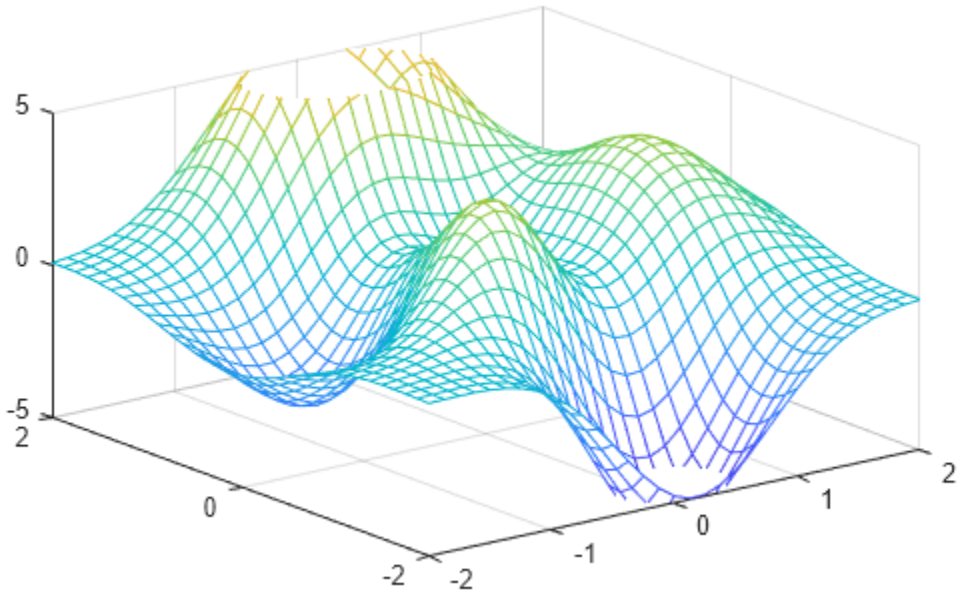
```
[X,Y,Z] = peaks;
surf(X,Y,Z)
xlabel('x-axis')
ylabel('y-axis')
xlim([-inf 0])
ylim([-1 inf])
```



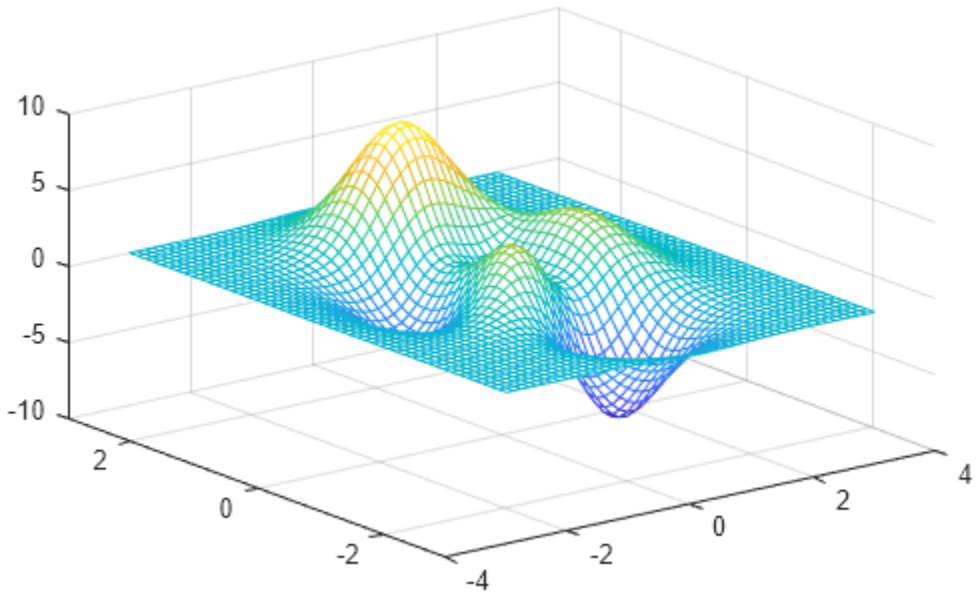
Revert Back to Default Limits

Create a mesh plot and change the axis limits. Then revert back to the default limits.

```
[X,Y,Z] = peaks;
mesh(X,Y,Z)
xlim([-2 2])
ylim([-2 2])
zlim([-5 5])
```



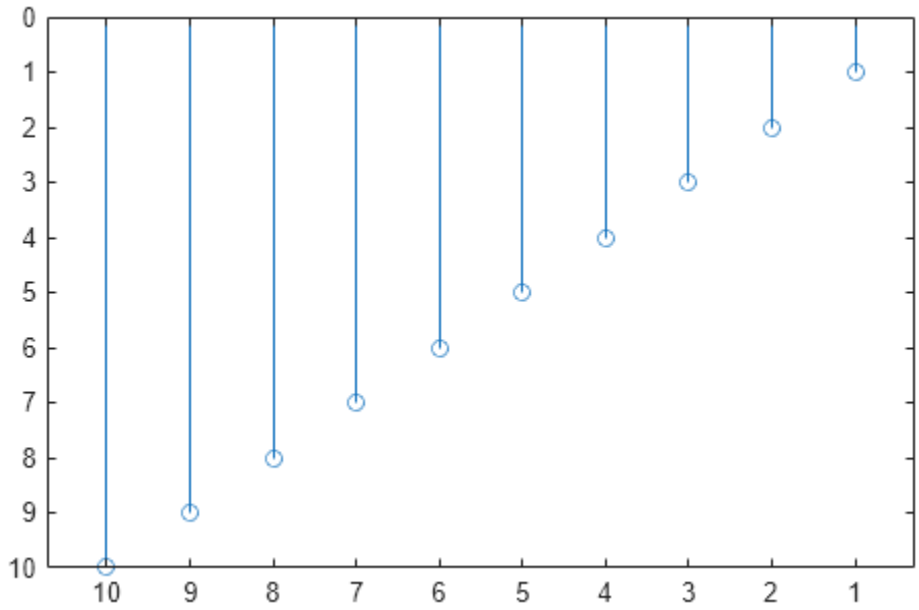
```
xlim auto
ylim auto
zlim auto
```



Reverse Axis Direction

Control the direction of increasing values along the *x*-axis and *y*-axis by setting the `XDir` and `YDir` properties of the `Axes` object. Set these properties to either `'reverse'` or `'normal'` (the default). Use the `gca` command to access the `Axes` object.

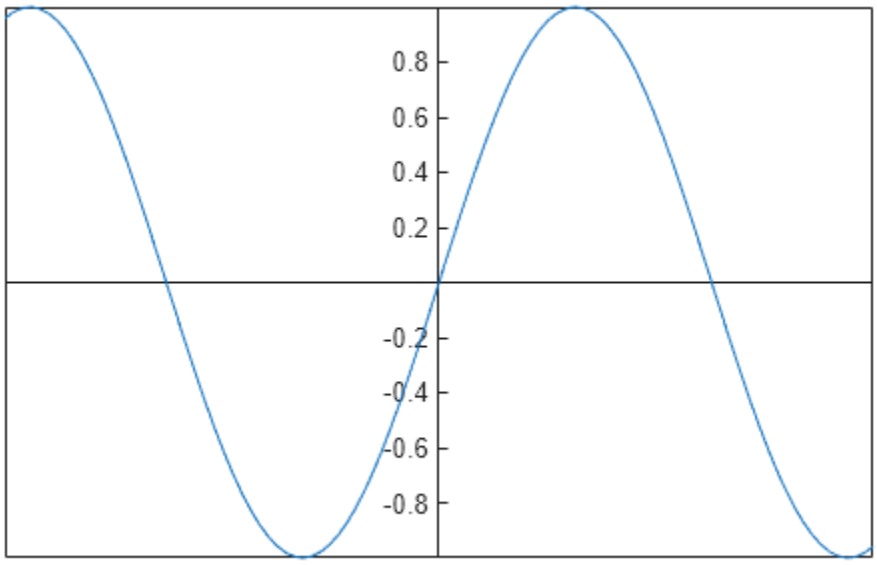
```
stem(1:10)
ax = gca;
ax.XDir = 'reverse';
ax.YDir = 'reverse';
```



Display Axis Lines Through Origin

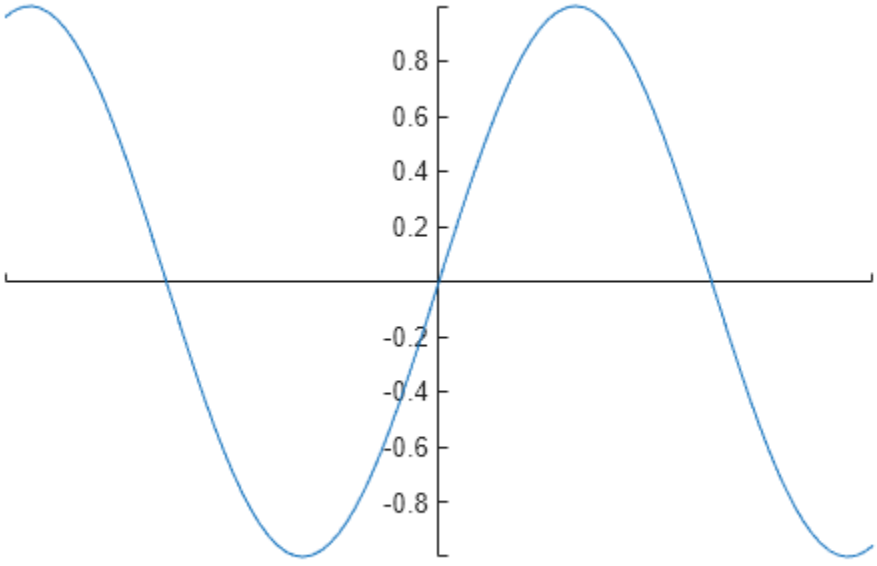
By default, the *x*-axis and *y*-axis appear along the outer bounds of the axes. Change the location of the axis lines so that they cross at the origin point `(0,0)` by setting the `XAxisLocation` and `YAxisLocation` properties of the `Axes` object. Set `XAxisLocation` to either `'top'`, `'bottom'`, or `'origin'`. Set `YAxisLocation` to either `'left'`, `'right'`, or `'origin'`. These properties only apply to axes in a 2-D view.

```
x = linspace(-5,5);
y = sin(x);
plot(x,y)

ax = gca;
ax.XAxisLocation = 'origin';
ax.YAxisLocation = 'origin';
```



Remove the axes box outline.

```
box off
```



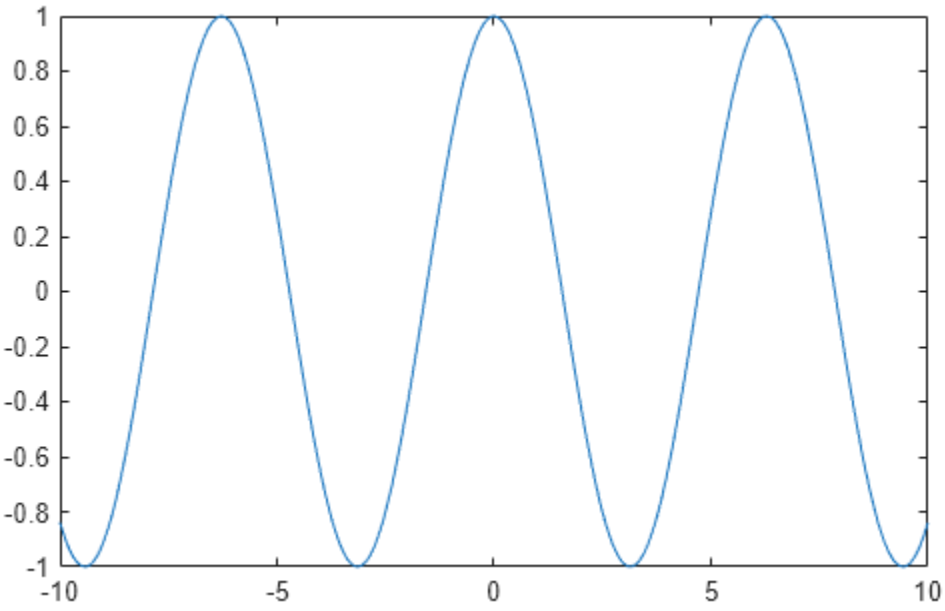
Specify Axis Tick Values and Labels

Customizing the tick values and labels along an axis can help highlight particular aspects of your data. These examples show some common customizations, such as modifying the tick value placement, changing the tick label text and formatting, and rotating the tick labels.

Change Tick Value Locations and Labels

Create x as 200 linearly spaced values between -10 and 10. Create y as the cosine of x. Plot the data.

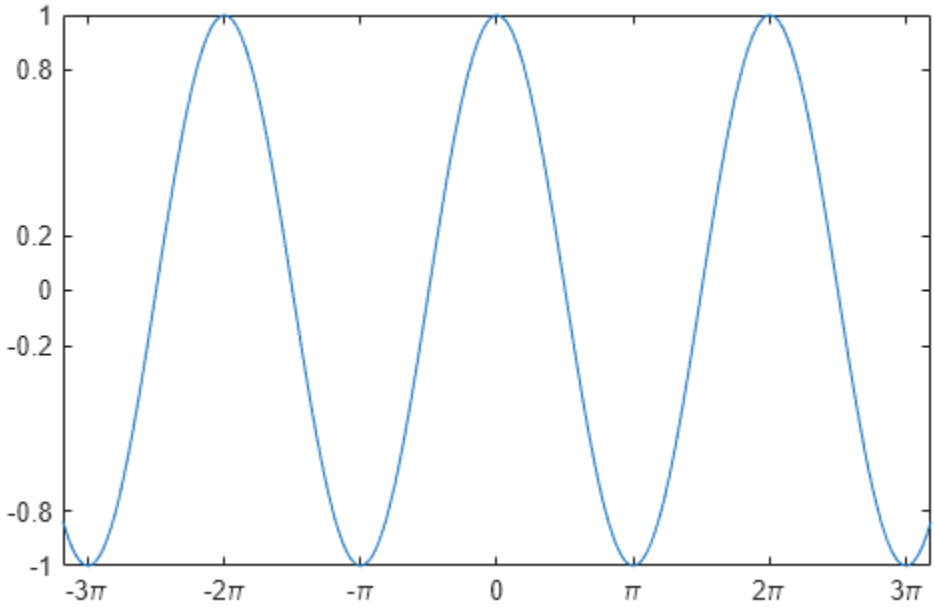
```
x = linspace(-10,10,200);
y = cos(x);
plot(x,y)
```



Change the tick value locations along the *x*-axis and *y*-axis. Specify the locations as a vector of increasing values. The values do not need to be evenly spaced.

Also, change the labels associated with each tick value along the *x*-axis. Specify the labels using a cell array of character vectors. To include special characters or Greek letters in the labels, use TeX markup, such as `\pi` for the π symbol.

```
xticks([-3*pi -2*pi -pi 0 pi 2*pi 3*pi])
xticklabels({'-3\pi','-2\pi','-\pi','0','\pi','2\pi','3\pi'})
yticks([-1 -0.8 -0.2 0 0.2 0.8 1])
```

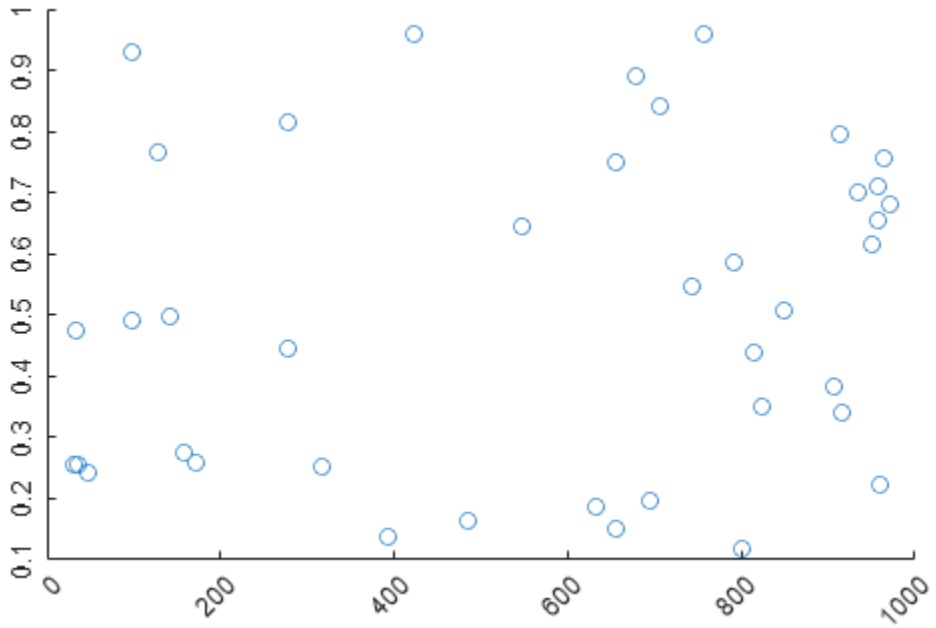


For releases prior to R2016b, instead set the tick values and labels using the `XTick`, `XTickLabel`, `YTick`, and `YTickLabel` properties of the `Axes` object. For example, assign the `Axes` object to a variable, such as `ax = gca`. Then set the `XTick` property using dot notation, such as `ax.XTick = [-3*pi -2*pi -pi 0 pi 2*pi 3*pi]`. For releases prior to R2014b, use the `set` function to set the property instead.

Rotate Tick Labels

Create a scatter plot and rotate the tick labels along each axis. Specify the rotation as a scalar value. Positive values indicate counterclockwise rotation. Negative values indicate clockwise rotation.

```
x = 1000*rand(40,1);
y = rand(40,1);
scatter(x,y)
xtickangle(45)
ytickangle(90)
```

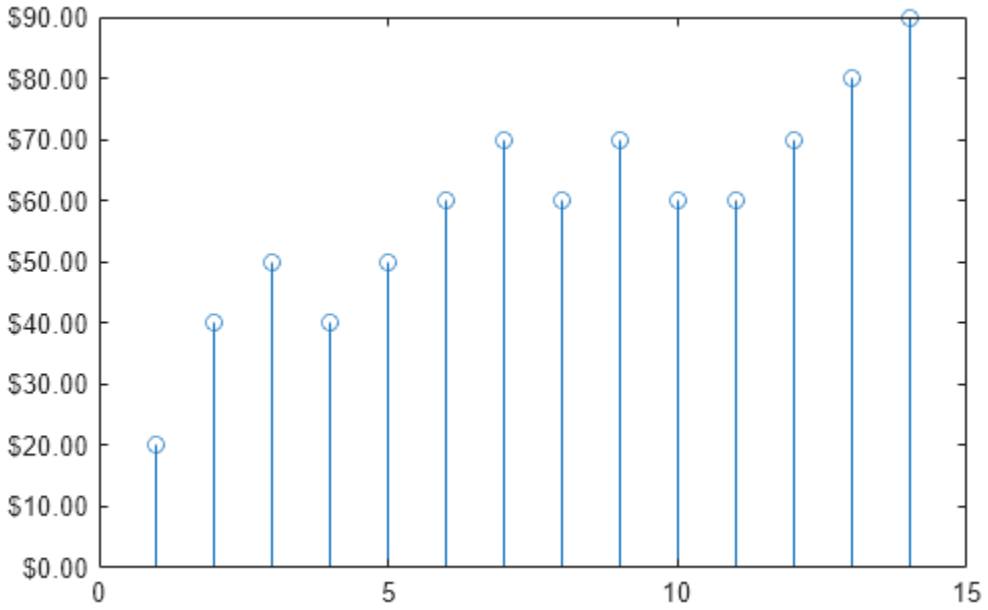


For releases prior to R2016b, specify the rotation using the `XTickLabelRotation` and `YTickLabelRotation` properties of the `Axes` object. For example, assign the `Axes` object to a variable, such as `ax = gca`. Then set the `XTickLabelRotation` property using dot notation, such as `ax.XTickLabelRotation = 45`.

Change Tick Label Formatting

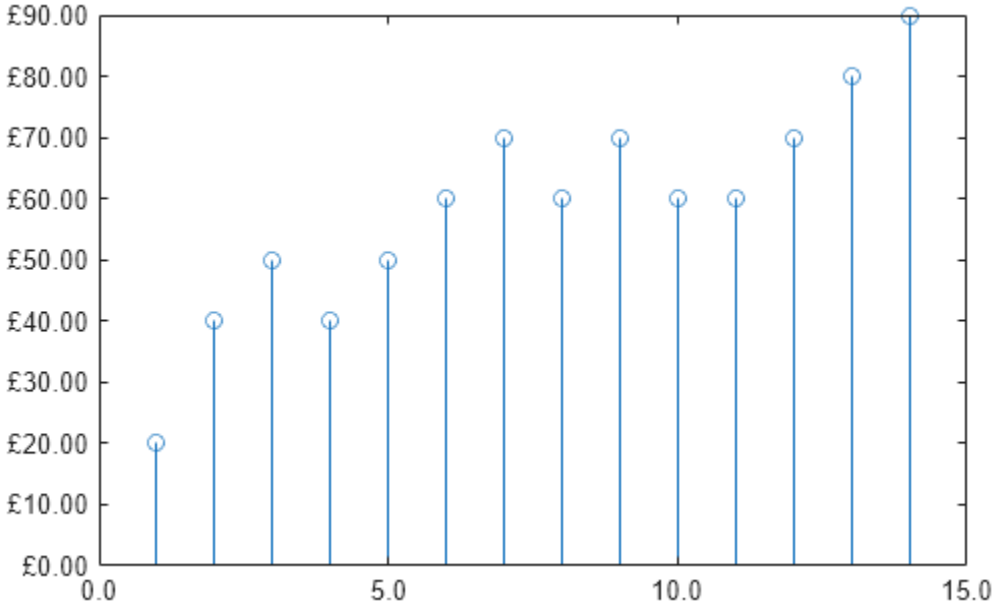
Create a stem chart and display the tick label values along the *y*-axis as US dollar values.

```
profit = [20 40 50 40 50 60 70 60 70 60 60 70 80 90];
stem(profit)
xlim([0 15])
ytickformat('usd')
```



For more control over the formatting, specify a custom format. For example, show one decimal value in the *x*-axis tick labels using `'%.1f'`. Display the *y*-axis tick labels as British Pounds using `'\xA3%.2f'`. The option `\xA3` indicates the Unicode character for the Pound symbol. For more information on specifying a custom format, see the `xtickformat` function.

```
xtickformat('%.1f')
ytickformat('\xA3%.2f')
```



Ruler Objects for Individual Axis Control

MATLAB creates a ruler object for each axis. Like all graphics objects, ruler objects have properties that you can view and modify. Ruler objects allow for more individual control over the formatting of the *x*-axis, *y*-axis, or *z*-axis. Access the ruler object associated with a particular axis through the `XAxis`, `YAxis`, or `ZAxis` property of the `Axes` object. The type of ruler depends on the type of data along the axis. For numeric data, MATLAB creates a `NumericRuler` object.

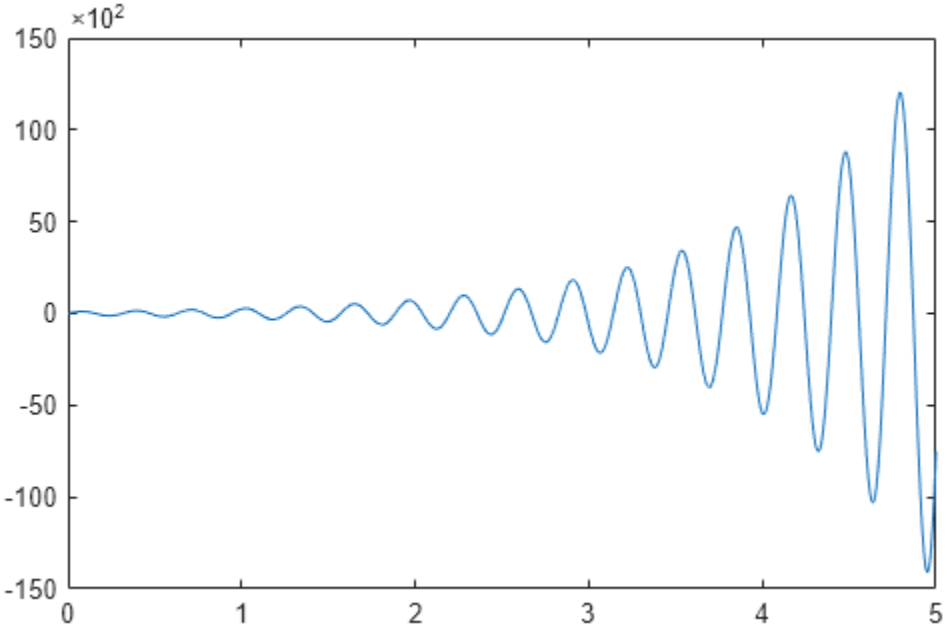
```
ax = gca;
ax.XAxis
```

```
ans =
  NumericRuler with properties:

            Limits: [0 15]
             Scale: 'linear'
          Exponent: 0
        TickValues: [0 5 10 15]
    TickLabelFormat: '%.1f'

  Show all properties
```

Control Value of Exponent in Secondary Label Using Ruler Objects

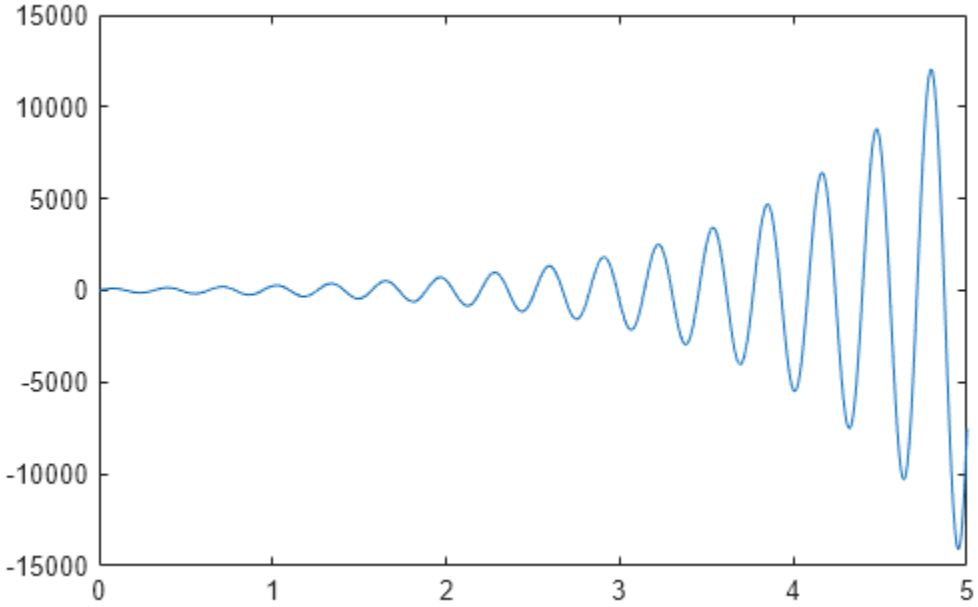
Plot data with *y* values that range between -15,000 and 15,000. By default, the *y*-axis tick labels use exponential notation with an exponent value of 4 and a base of 10. Change the exponent value to 2. Set the `Exponent` property of the ruler object associated with the *y*-axis. Access the ruler object through the `YAxis` property of the `Axes` object. The secondary label and the tick labels change accordingly.

```
x = linspace(0,5,1000);
y = 100*exp(x).*sin(20*x);
plot(x,y)

ax = gca;
ax.YAxis.Exponent = 2;
```



Change the exponent value to 0 so that the tick labels do not use exponential notation.

```
ax.YAxis.Exponent = 0;
```



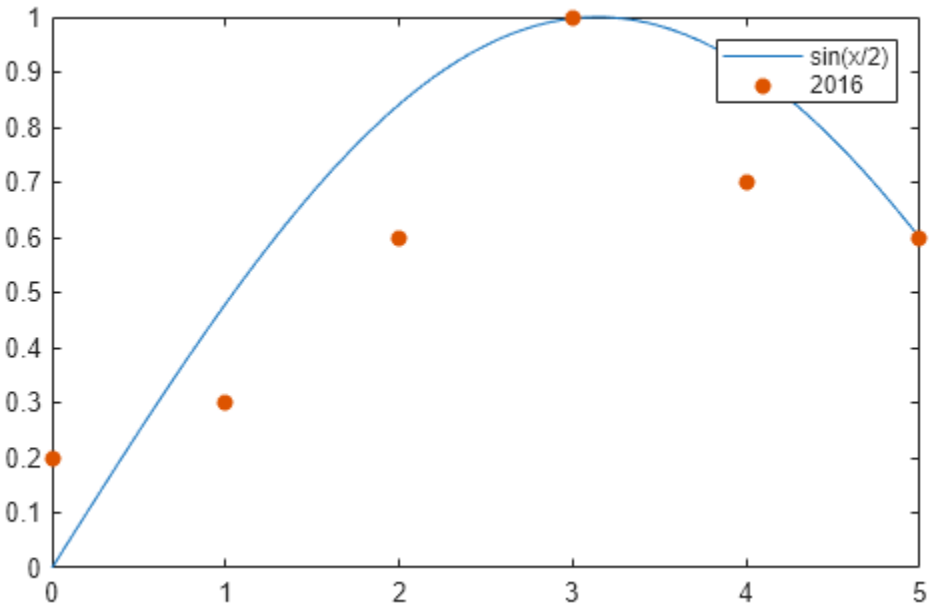
Add Legend to Graph

Legends are a useful way to label data series plotted on a graph. These examples show how to create a legend and make some common modifications, such as changing the location, setting the font size, and adding a title. You also can create a legend with multiple columns or create a legend for a subset of the plotted data.

Create Simple Legend

Create a figure with a line chart and a scatter chart. Add a legend with a description for each chart. Specify the legend labels as inputs to the `legend` function.

```
figure
x1 = linspace(0,5);
y1 = sin(x1/2);
plot(x1,y1)

hold on
x2 = [0 1 2 3 4 5];
y2 = [0.2 0.3 0.6 1 0.7 0.6];
scatter(x2,y2,'filled')
hold off

legend('sin(x/2)','2016')
```



Specify Labels Using DisplayName

Alternatively, you can specify the legend labels using the `DisplayName` property. Set the `DisplayName` property as a name-value pair when calling the plotting functions. Then, call the `legend` command to create the legend.

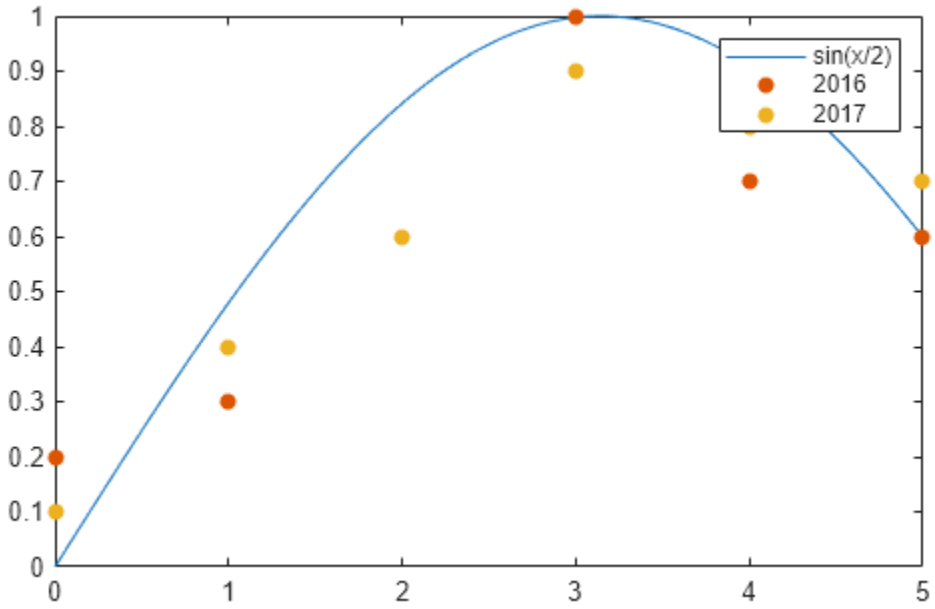
```
x1 = linspace(0,5);
y1 = sin(x1/2);
plot(x1,y1,'DisplayName','sin(x/2)')
```

```
hold on
x2 = [0 1 2 3 4 5];
y2 = [0.2 0.3 0.6 1 0.7 0.6];
scatter(x2,y2,'filled','DisplayName','2016')

legend
```

Legends automatically update when you add or delete a data series. If you add more data to the axes, use the `DisplayName` property to specify the labels. If you do not set the `DisplayName` property, then the legend uses a label of the form `'dataN'`.

Add a scatter chart for 2017 data.

```
x3 = [0 1 2 3 4 5];
y3 = [0.1 0.4 0.6 0.9 0.8 0.7];
scatter(x3,y3,'filled','DisplayName','2017')
drawnow
hold off
```



Customize Legend Appearance

The `legend` function creates a `Legend` object. `Legend` objects have properties that you can use to customize the appearance of the legend, such as the `Location`, `Orientation`, `FontSize`, and `Title` properties. For a full list, see Legend Properties.

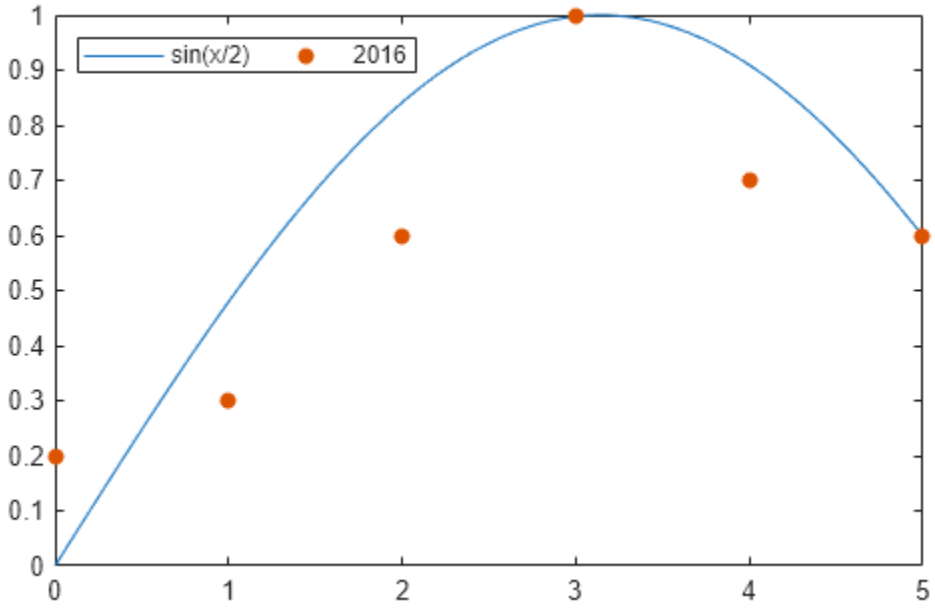
You can set properties in two ways:

- Use name-value pairs in the `legend` command. In most cases, when you use name-value pairs, you must specify the labels in a cell array, such as `legend({'label1','label2'},'FontSize',14)`.
- Use the `Legend` object. You can return the `Legend` object as an output argument from the `legend` function, such as `lgd = legend`. Then, use `lgd` with dot notation to set properties, such as `lgd.FontSize = 14`.

Legend Location and Orientation

Specify the legend location and orientation by setting the `Location` and `Orientation` properties as name-value pairs. Set the location to one of the eight cardinal or intercardinal directions, in this case, `'northwest'`. Set the orientation to `'vertical'` (the default) or `'horizontal'`, as in this case. Specify the labels in a cell array.

```
x1 = linspace(0,5);
y1 = sin(x1/2);
plot(x1,y1)

hold on
x2 = [0 1 2 3 4 5];
y2 = [0.2 0.3 0.6 1 0.7 0.6];
scatter(x2,y2,'filled')
hold off

legend({'sin(x/2)','2016'},'Location','northwest','Orientation','horizontal')
```

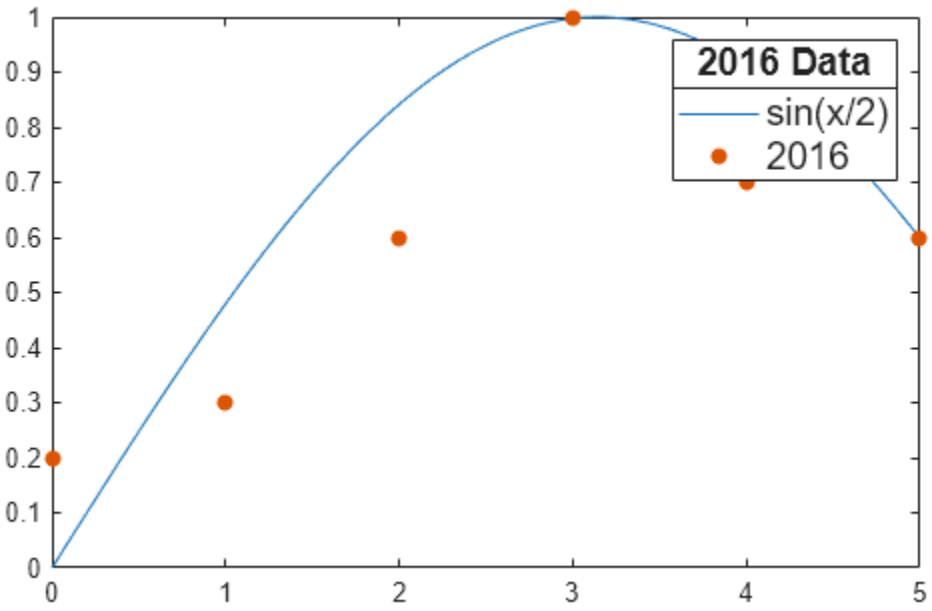


Legend Font Size and Title

Specify the legend font size and title by setting the `FontSize` and `Title` properties. Assign the `Legend` object to the variable `lgd`. Then, use `lgd` to change the properties using dot notation.

```
x1 = linspace(0,5);
y1 = sin(x1/2);
plot(x1,y1,'DisplayName','sin(x/2)')

hold on
x2 = [0 1 2 3 4 5];
y2 = [0.2 0.3 0.6 1 0.7 0.6];
scatter(x2,y2,'filled','DisplayName','2016')
hold off
```

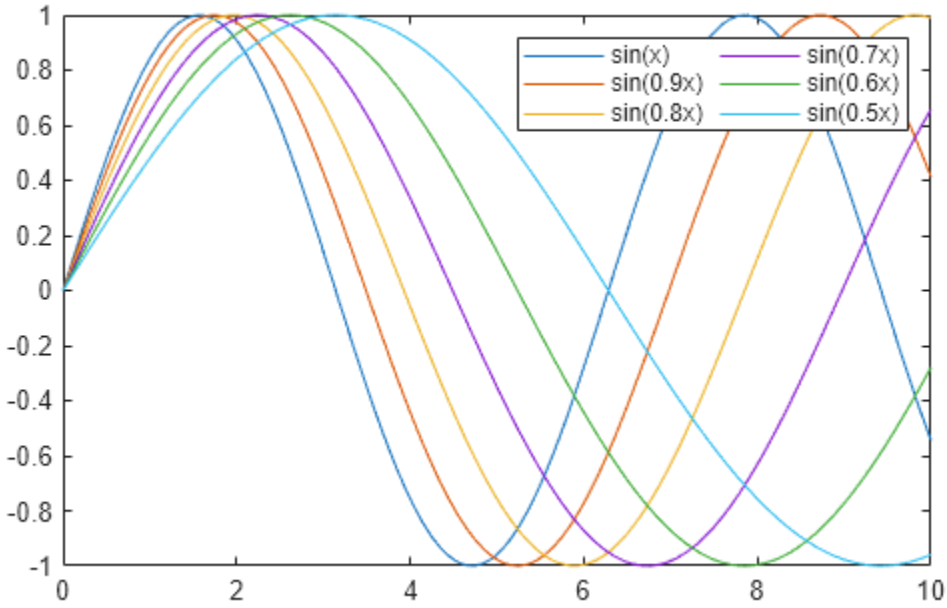
```
lgd = legend;
lgd.FontSize = 14;
lgd.Title.String = '2016 Data';
```



Legend with Multiple Columns

Create a chart with six line plots. Add a legend with two columns by setting the `NumColumns` property to 2.

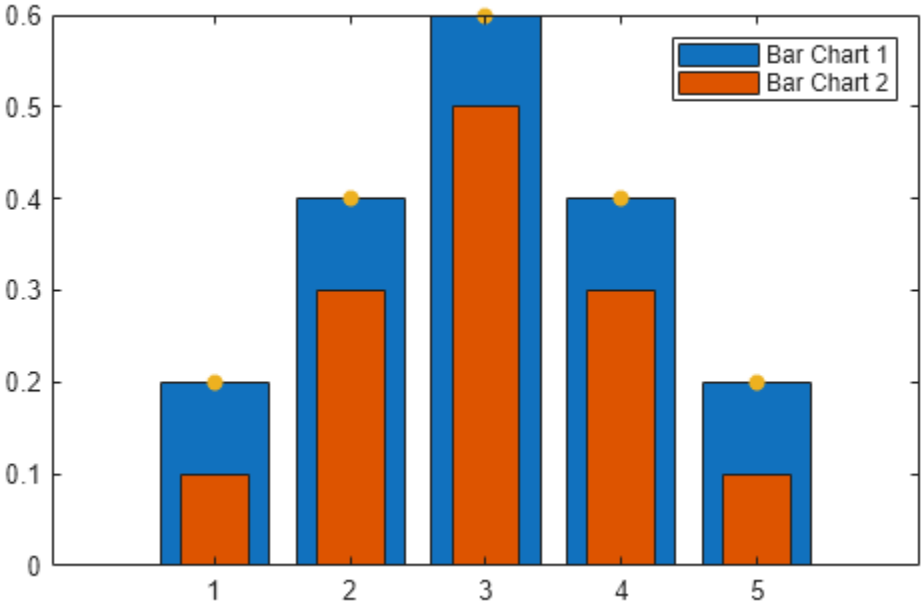
```
x = linspace(0,10);
y1 = sin(x);
y2 = sin(0.9*x);
y3 = sin(0.8*x);
y4 = sin(0.7*x);
y5 = sin(0.6*x);
y6 = sin(0.5*x);

plot(x,y1,'DisplayName','sin(x)')
hold on
plot(x,y2,'DisplayName','sin(0.9x)')
plot(x,y3,'DisplayName','sin(0.8x)')
plot(x,y4,'DisplayName','sin(0.7x)')
plot(x,y5,'DisplayName','sin(0.6x)')
plot(x,y6,'DisplayName','sin(0.5x)')
hold off

lgd = legend;
lgd.NumColumns = 2;
```



Include Subset of Charts in Legend

Combine two bar charts and a scatter chart. Create a legend that includes only the bar charts by specifying the `Bar` objects, `b1` and `b2`, as the first input argument to the `legend` function. Specify the objects in a vector.

```
x = [1 2 3 4 5];
y1 = [.2 .4 .6 .4 .2];
b1 = bar(x,y1);

hold on
y2 = [.1 .3 .5 .3 .1];
b2 = bar(x,y2,'BarWidth',0.5);

y3 = [.2 .4 .6 .4 .2];
s = scatter(x,y3,'filled');
hold off

legend([b1 b2],'Bar Chart 1','Bar Chart 2')
```


0.6
0.5
0.4
0.3
0.2
0.1
0
1
2
3
4
5
Bar Chart 1
Bar Chart 2

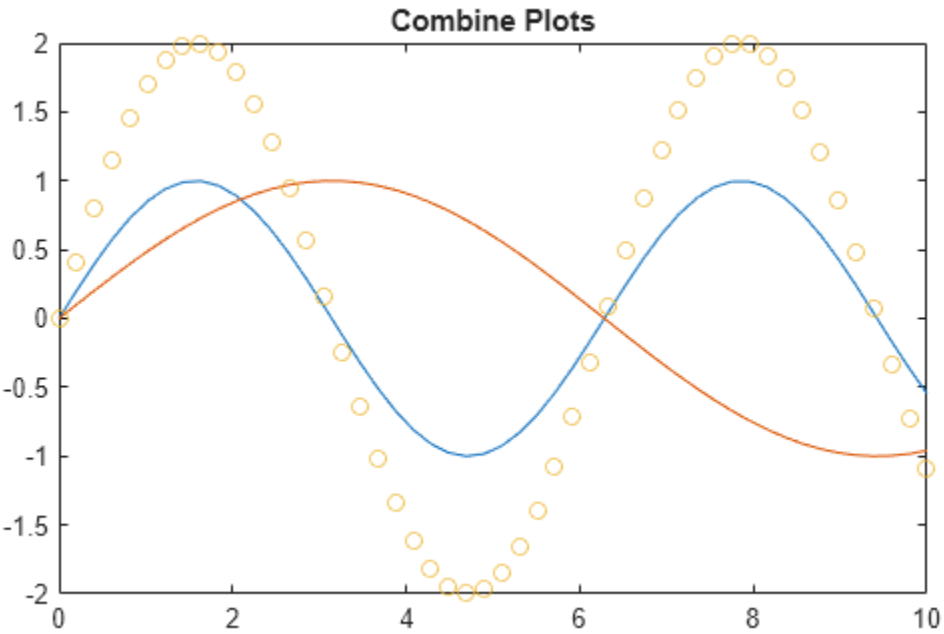
Combine Multiple Plots

This example shows how to combine plots in the same axes using the `hold` function, and how to create multiple axes in a figure using the `tiledlayout` function.

Combine Plots in Same Axes

By default, new plots clear existing plots and reset axes properties, such as the title. However, you can use the `hold on` command to combine multiple plots in the same axes. For example, plot two lines and a scatter plot. Then reset the hold state to off.

```
x = linspace(0,10,50);
y1 = sin(x);
plot(x,y1)
title('Combine Plots')

hold on

y2 = sin(x/2);
plot(x,y2)

y3 = 2*sin(x);
scatter(x,y3)

hold off
```

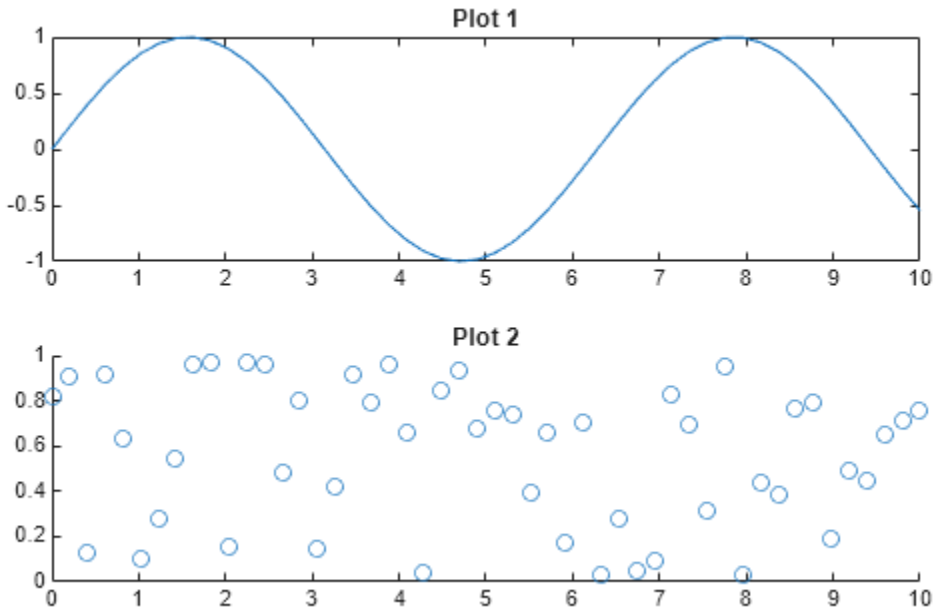


When the hold state is on, new plots do not clear existing plots or reset axes properties, such as the title or axis labels. The plots cycle through colors and line styles based on the `ColorOrder` and `LineStyleOrder` properties of the axes. The axes limits and tick values might adjust to accommodate new data.

Display Multiple Axes in a Figure

You can display multiple axes in a single figure by using the `tiledlayout` function. This function creates a tiled chart layout containing an invisible grid of tiles over the entire figure. Each tile can contain an axes for displaying a plot. After creating a layout, call the `nexttile` function to place an axes object into the layout. Then call a plotting function to plot into the axes. For example, create two plots in a 2-by-1 layout. Add a title to each plot.

```
x = linspace(0,10,50);
y1 = sin(x);
y2 = rand(50,1);
tiledlayout(2,1)

% Top plot
nexttile
plot(x,y1)
title('Plot 1')

% Bottom plot
nexttile
scatter(x,y2)
title('Plot 2')
```

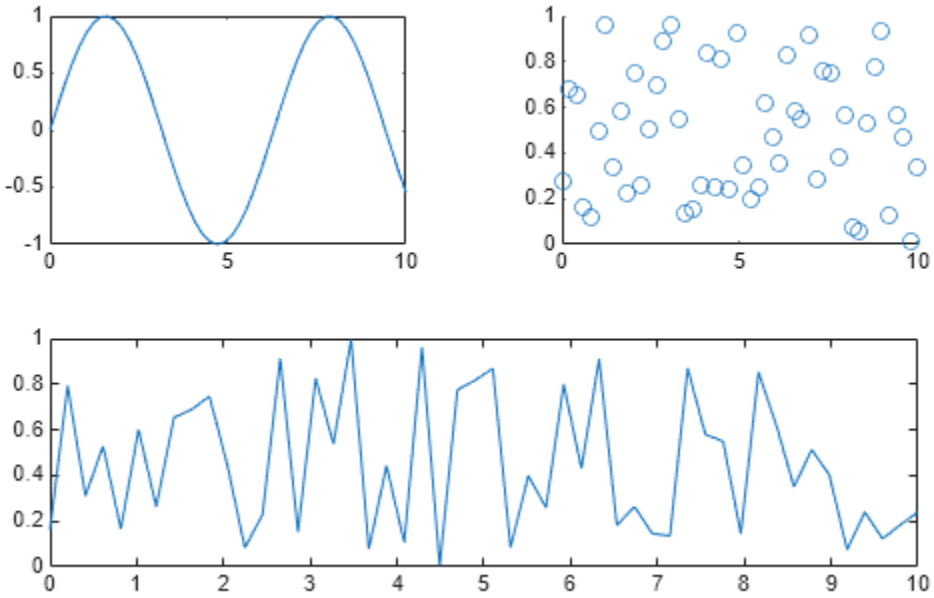


Create Plot Spanning Multiple Rows or Columns

To create a plot that spans multiple rows or columns, specify the `span` argument when you call `nexttile`. For example, create a 2-by-2 layout. Plot into the first two tiles. Then create a plot that spans one row and two columns.

```
x = linspace(0,10,50);
y1 = sin(x);
y2 = rand(50,1);
```

```
% Top two plots
tiledlayout(2,2)
nexttile
plot(x,y1)
nexttile
scatter(x,y2)

% Plot that spans
nexttile([1 2])
y2 = rand(50,1);
plot(x,y2)
```



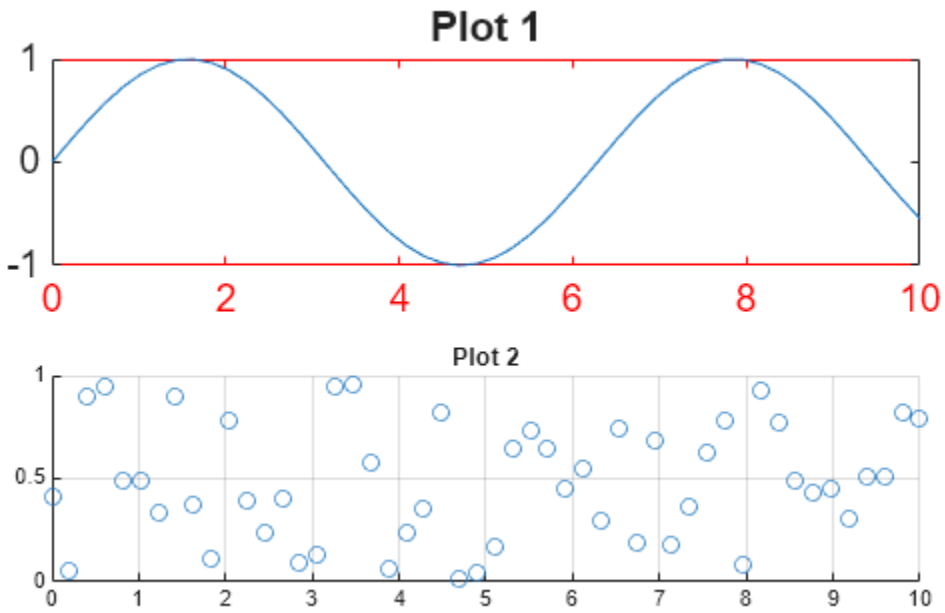
Modify Axes Appearance

Modify the axes appearance by setting properties on each of the axes objects. You can get the axes object by calling the `nexttile` function with an output argument. You also can specify the axes object as the first input argument to a graphics function to ensure that the function targets the correct axes.

For example, create two plots and assign the axes objects to the variables `ax1` and `ax2`. Change the axes font size and *x*-axis color for the first plot. Add grid lines to the second plot.

```
x = linspace(0,10,50);
y1 = sin(x);
y2 = rand(50,1);
tiledlayout(2,1)

% Top plot
ax1 = nexttile;
plot(ax1,x,y1)
title(ax1,'Plot 1')
ax1.FontSize = 14;
ax1.XColor = 'red';
```

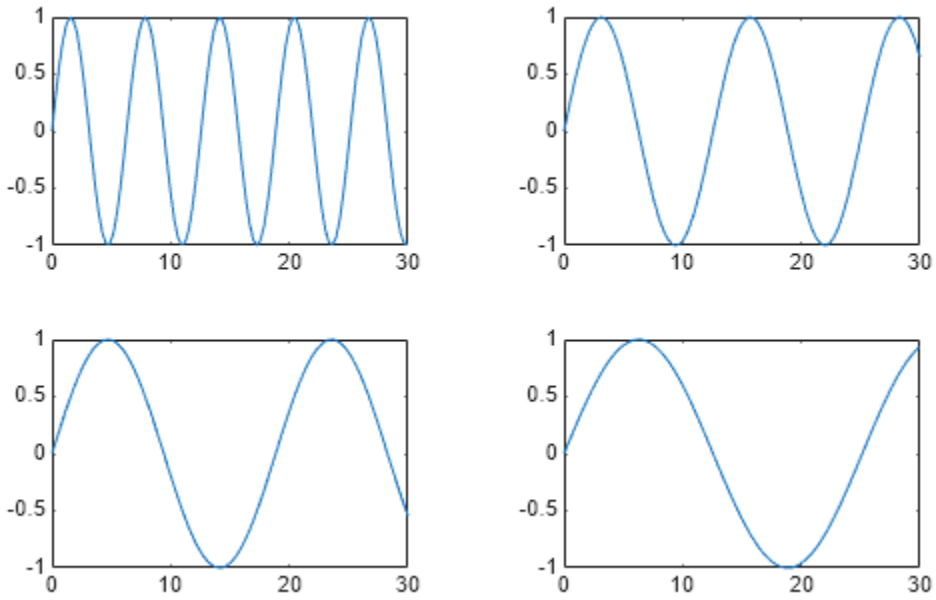
```
% Bottom plot
ax2 = nexttile;
scatter(ax2,x,y2)
title(ax2,'Plot 2')
grid(ax2,'on')
```



Control Spacing Around the Tiles

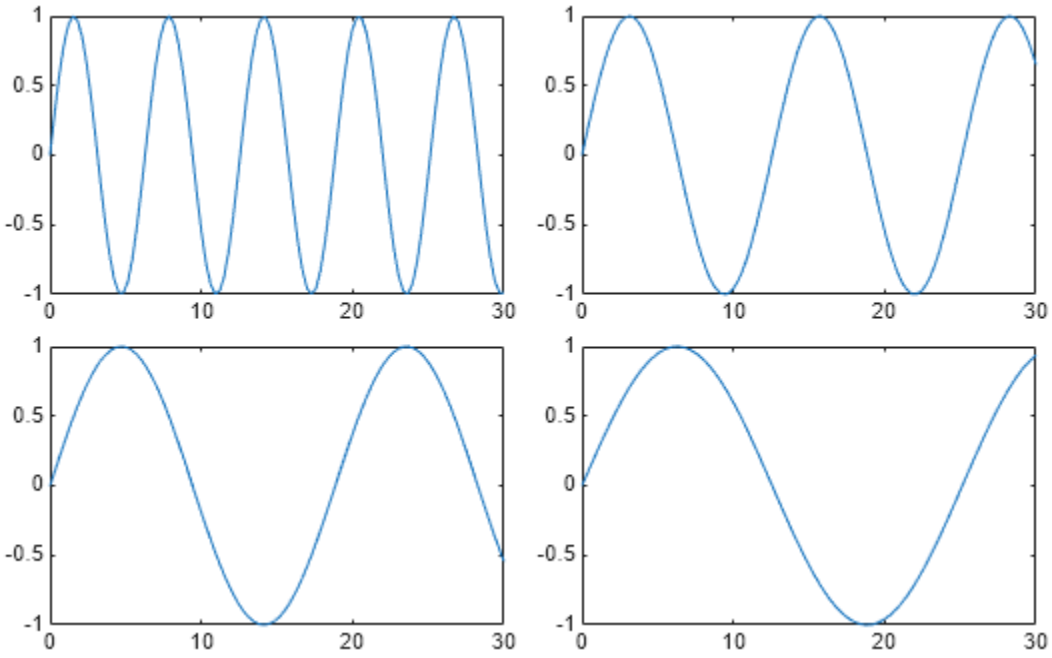
You can control the spacing around the tiles in a layout by specifying the `Padding` and `TileSpacing` properties. For example, display four plots in a 2-by-2 layout.

```
x = linspace(0,30);
y1 = sin(x);
y2 = sin(x/2);
y3 = sin(x/3);
y4 = sin(x/4);

% Create plots
t = tiledlayout(2,2);
nexttile
plot(x,y1)
nexttile
plot(x,y2)
nexttile
plot(x,y3)
nexttile
plot(x,y4)
```



Reduce the spacing around the perimeter of the layout and around each tile by setting the `Padding` and `TileSpacing` properties to `'compact'`.

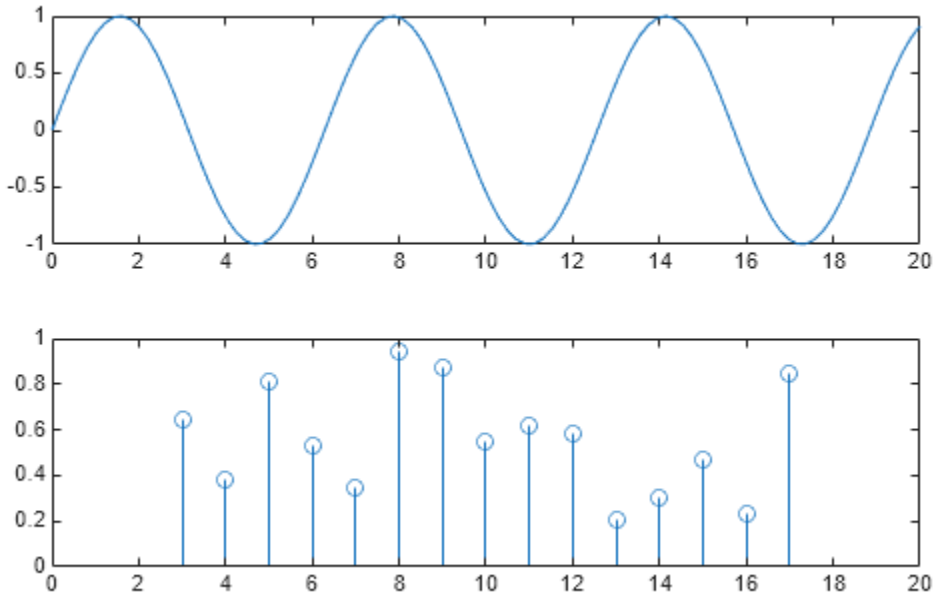
```
t.Padding = 'compact';
t.TileSpacing = 'compact';
```



Display Shared Title and Axis Labels

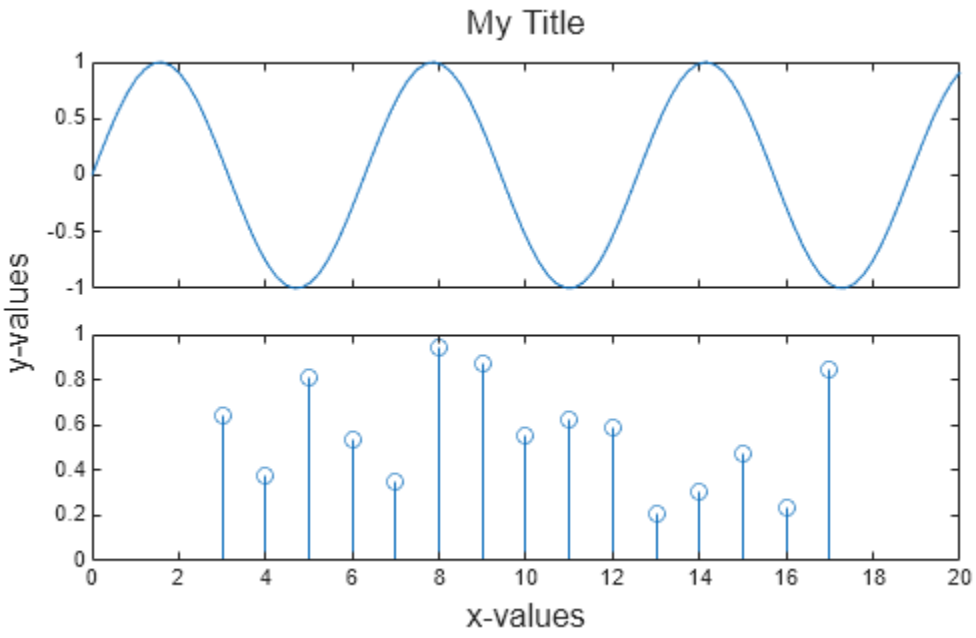
You can display a shared title and shared axis labels in a layout. Create a 2-by-1 layout `t`. Then display a line plot and a stem plot. Synchronize the *x*-axis limits by calling the `linkaxes` function.

```
x1 = linspace(0,20,100);
y1 = sin(x1);
x2 = 3:17;
y2 = rand(1,15);

% Create plots.
t = tiledlayout(2,1);
ax1 = nexttile;
plot(ax1,x1,y1)
ax2 = nexttile;
stem(ax2,x2,y2)

% Link the axes
linkaxes([ax1,ax2],'x');
```



Add a shared title and shared axis labels by passing `t` to the `title`, `xlabel`, and `ylabel` functions. Move the plots closer together by removing the *x*-axis tick labels from the top plot and setting the `TileSpacing` property of `t` to `'compact'`.

```
% Add shared title and axis labels
title(t,'My Title')
xlabel(t,'x-values')
ylabel(t,'y-values')

% Move plots closer together
xticklabels(ax1,{})
t.TileSpacing = 'compact';
```


My Title
y-values
x-values
1
0.5
0
-0.5
-1
0.8
0.6
0.4
0.2
0
2
4
6
8
10
12
14
16
18
20

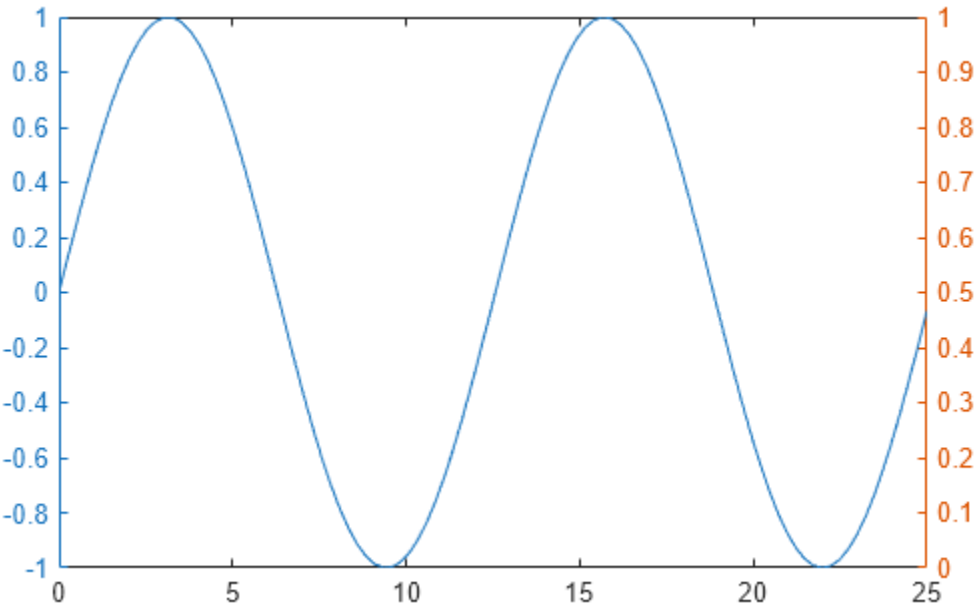
Create Chart with Two y-Axes

This example shows how to create a chart with *y*-axes on the left and right sides using the `yyaxis` function. It also shows how to label each axis, combine multiple plots, and clear the plots associated with one or both of the sides.

Plot Data Against Left *y*-Axis

Create axes with a *y*-axis on the left and right sides. The `yyaxis left` command creates the axes and activates the left side. Subsequent graphics functions, such as `plot`, target the active side. Plot data against the left *y*-axis.

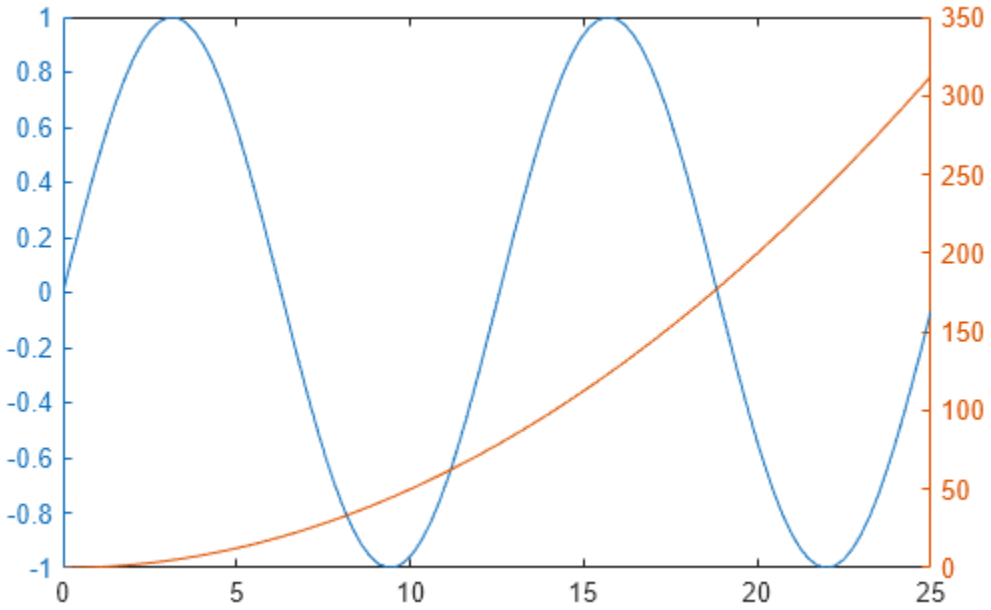
```
x = linspace(0,25);
y = sin(x/2);
yyaxis left
plot(x,y);
```



Plot Data Against Right *y*-Axis

Activate the right side using `yyaxis right`. Then plot a set of data against the right *y*-axis.

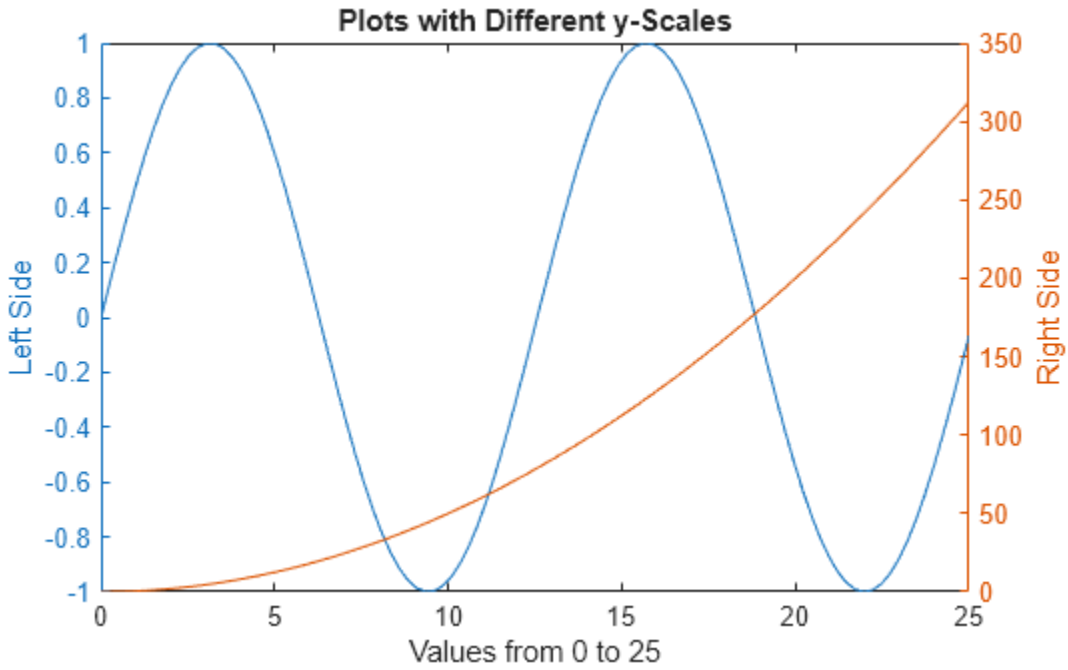
```
r = x.^2/2;
yyaxis right
plot(x,r);
```



Add Title and Axis Labels

Control which side of the axes is active using the `yyaxis left` and `yyaxis right` commands. Then, add a title and axis labels.

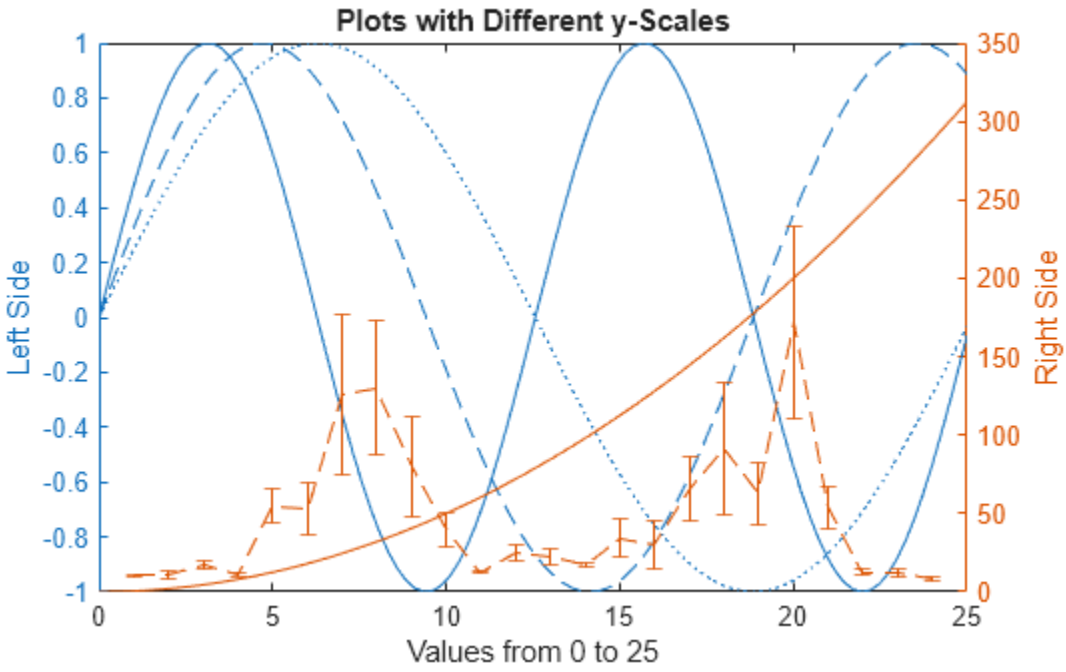
```
yyaxis left
title('Plots with Different y-Scales')
xlabel('Values from 0 to 25')
ylabel('Left Side')

yyaxis right
ylabel('Right Side')
```



Plot Additional Data Against Each Side

Add two more lines to the left side using the `hold on` command. Add an errorbar to the right side. The new plots use the same color as the corresponding *y*-axis and cycle through the line style order. The `hold on` command affects both the left and right sides.

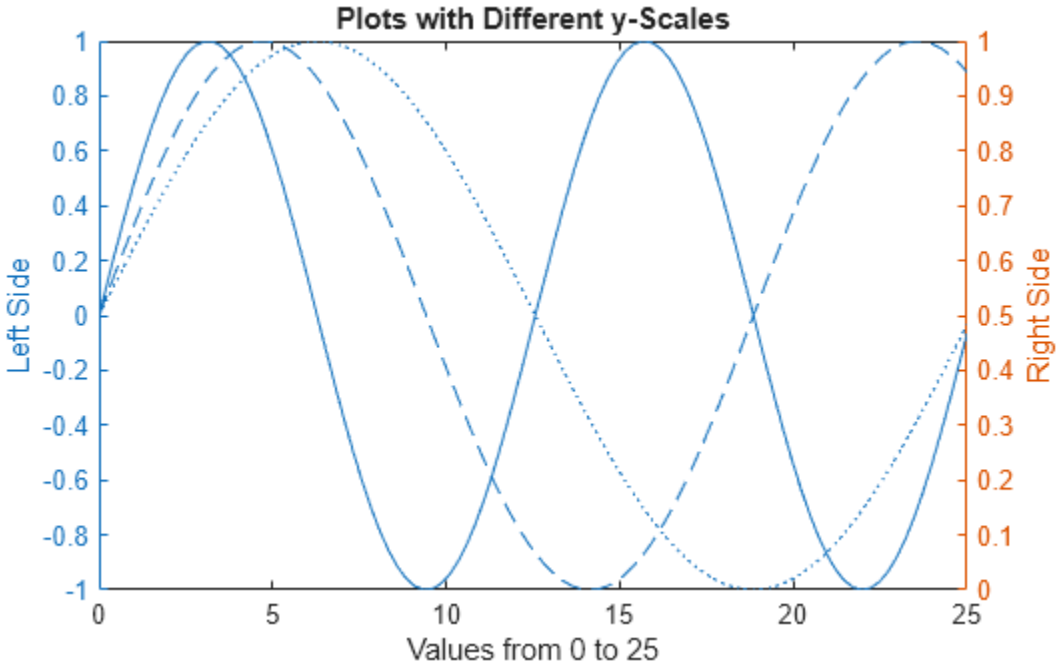
```
hold on

yyaxis left
y2 = sin(x/3);
plot(x,y2);
y3 = sin(x/4);
plot(x,y3);

yyaxis right
load count.dat;
m = mean(count,2);
e = std(count,1,2);
errorbar(m,e)

hold off
```



Clear One Side of Axes

Clear the data from the right side of the axes by first making it active, and then using the `cla` command.

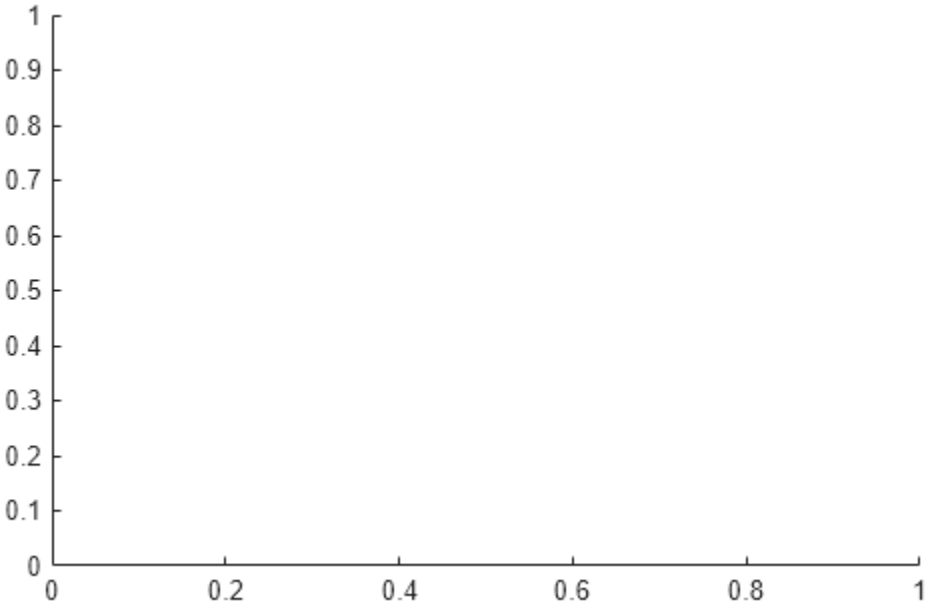
```
yyaxis right
cla
```



Clear Axes and Remove Right *y*-Axis

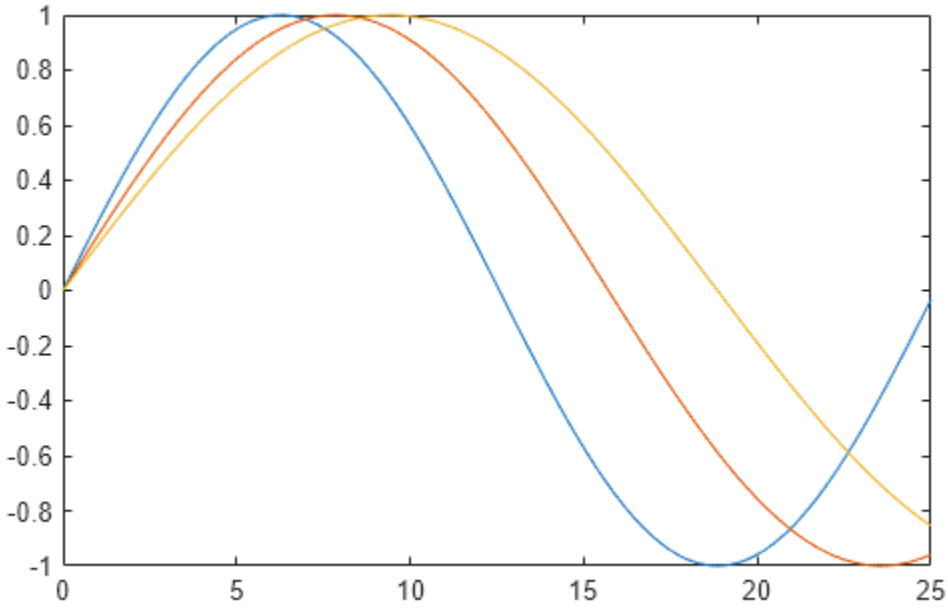
Clear the entire axes and remove the right *y*-axis using `cla reset`.

```
cla reset
```



Now when you create a plot, it only has one *y*-axis. For example, plot three lines against the single y-axis.

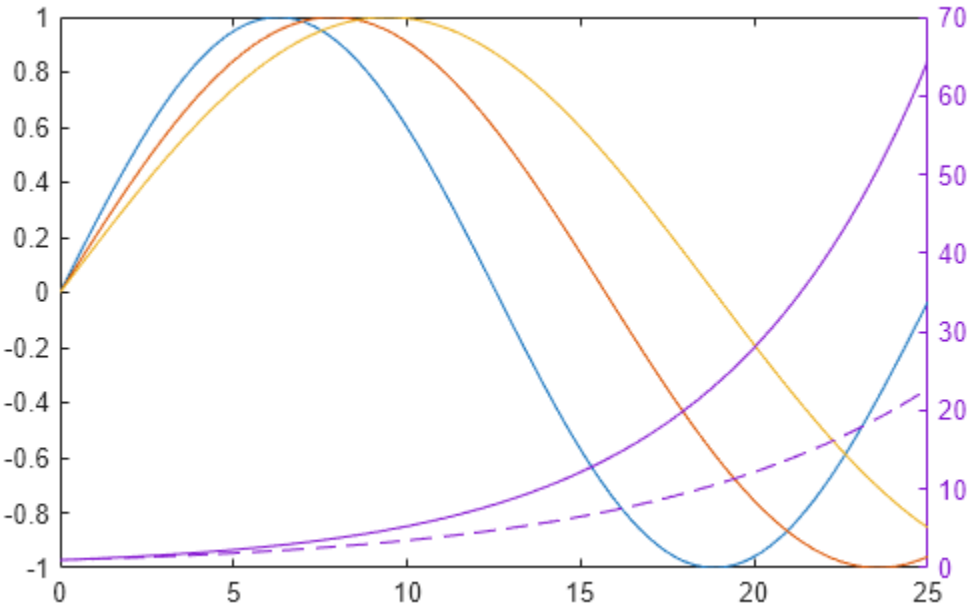
```
xx = linspace(0,25);
yy1 = sin(xx/4);
yy2 = sin(xx/5);
yy3 = sin(xx/6);
plot(xx,yy1,xx,yy2,xx,yy3)
```



Add Second *y*-Axis to Existing Chart

Add a second *y*-axis to an existing chart using `yyaxis`. The existing plots and the left *y*-axis do not change colors. The right *y*-axis uses the next color in the axes color order. New plots added to the axes use the same color as the corresponding *y*-axis.

```
yyaxis right
rr1 = exp(xx/6);
rr2 = exp(xx/8);
plot(xx,rr1,xx,rr2)
```


1
0.8
0.6
0.4
0.2
0
-0.2
-0.4
-0.6
-0.8
-1
0
5
10
15
20
25
70
60
50
40
30
20
10
0

Surface and Mesh Plots

A surface plot is a graphical representation of a three-dimensional data set. Typically, you plot the values of a matrix `Z` as heights above the *xy*-plane. By default, the color of a surface varies according to the heights specified by `Z`.

To create a surface plot, call the `surf` or `mesh` functions. You can modify different aspects of these plots, such as the color, transparency, and lighting by setting properties or calling functions.

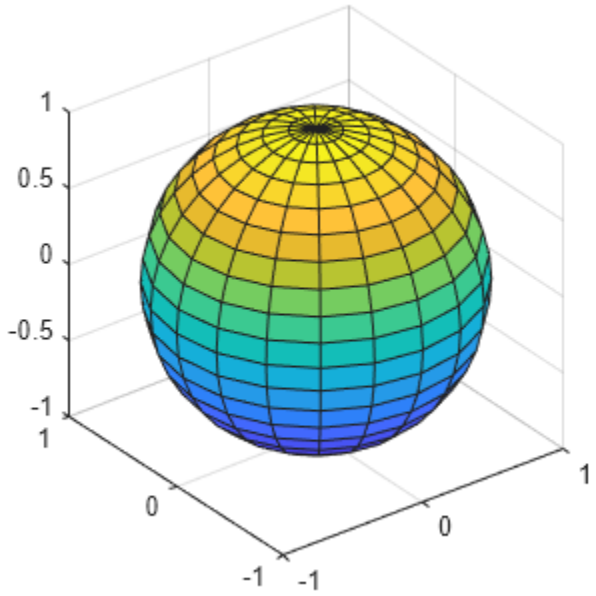
- `surf` displays a solid surface comprised of colored faces and mesh lines. The mesh lines represent the grid of *x*- and *y*-coordinates.
- `mesh` displays the mesh lines without the colored faces.

Other functions for visualizing 3-D data sets include `contour`, `surfc` and `meshc` functions, these functions display the contour lines for a surface.

Visualizing 3-D Data Sets

Built-in functions, such as `sphere`, `ellipsoid`, and `cylinder`, are useful for creating simple shapes that you can plot. For example, create a unit sphere and plot it using the `surf` function. The `axis equal` command preserves the aspect ratio of the sphere.

```
[X,Y,Z] = sphere;
surf(X,Y,Z)
axis equal
```

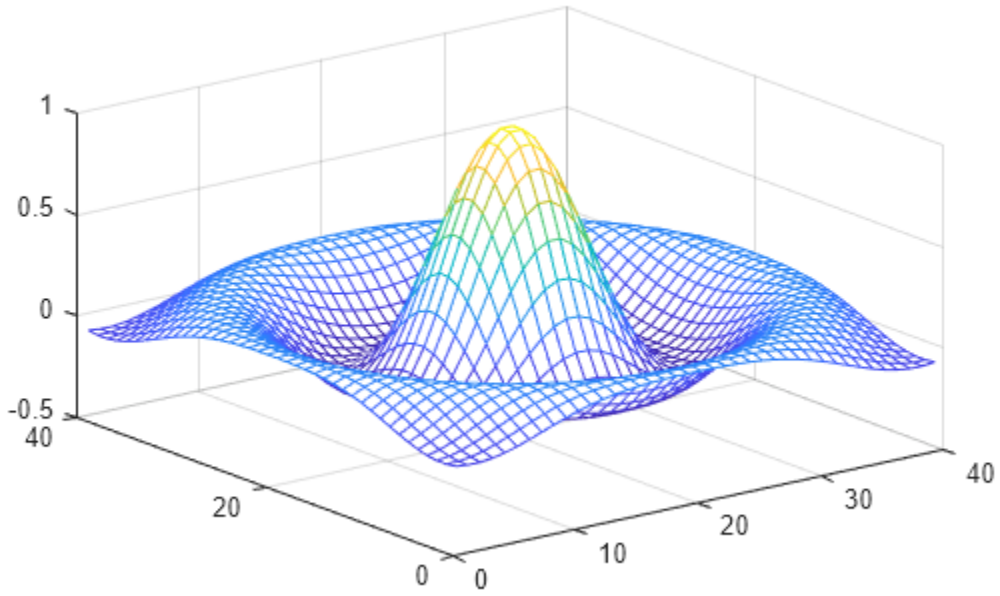


You can also create your own data set. For example, create row and column vectors `X` and `Y` with 40 points each in the range [-8, 8]. Define `R` as the distance from the origin, which is at the center of the matrix. Define matrix `Z` as a function of `R`.

```
X = linspace(-8,8,40);
Y = X';
R = sqrt(X.^2 + Y.^2);
Z = sin(R)./R;
```

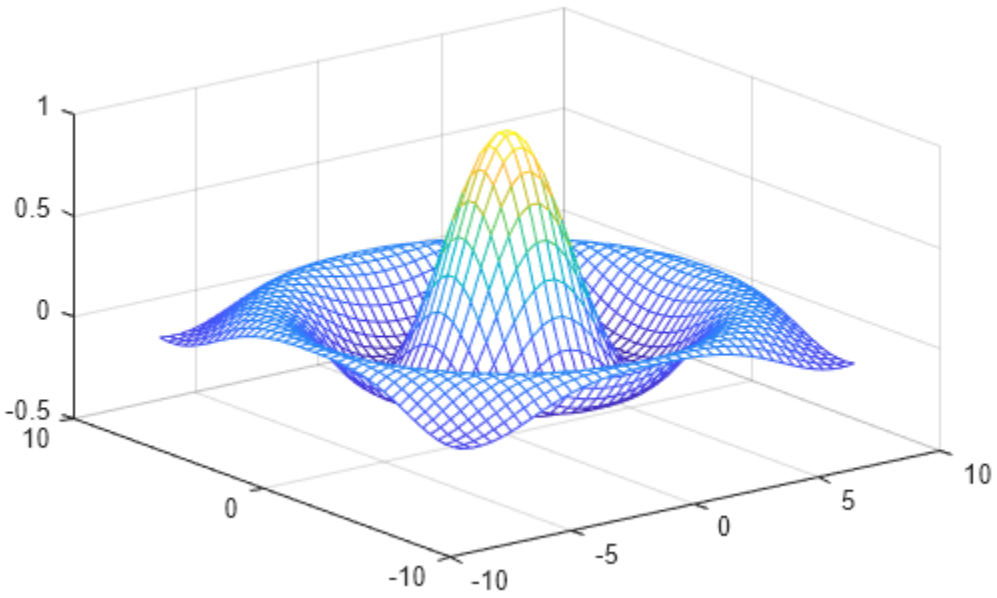
Create a mesh plot of Z. The *x*- and *y*-coordinates range from 1 to 40 because `mesh` uses the row and column indices of Z when you specify Z as the only input argument.

```
mesh(Z)
```



Create another mesh plot, but this time, specify all three sets of coordinates. The *x*- and *y*-coordinates in this plot match the X and Y vectors.

```
mesh(X,Y,Z)
```

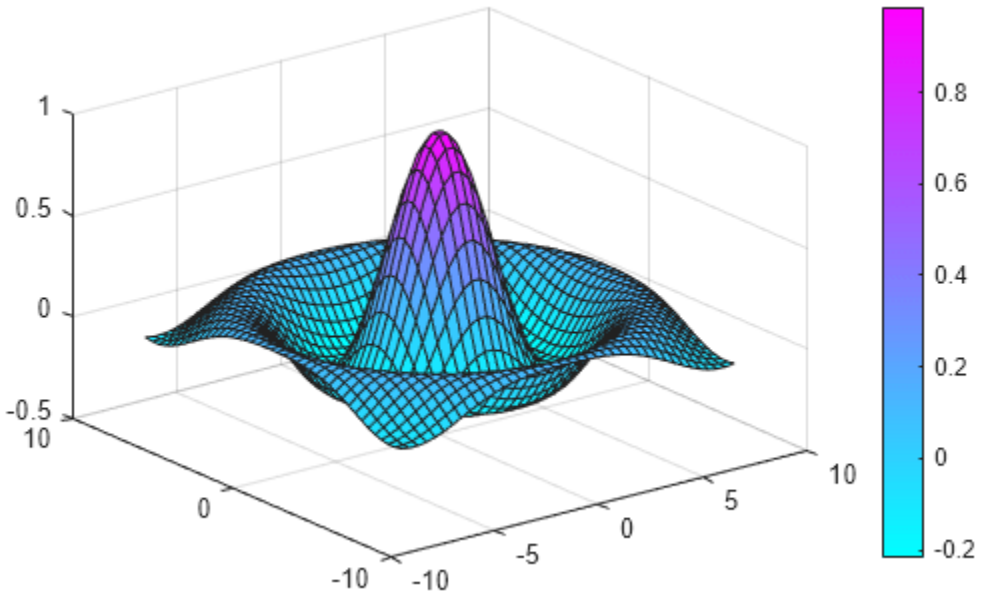


Coloring Surface Plots

The colors of the faces in a surface plot depend on the values of Z and a colormap. A colormap is a matrix in which each row corresponds to a different color. By default, the colors map to the vertices of the *xy*-grid over the range of Z. The smallest value of Z displays the color in the first row of the colormap. The largest value of Z displays the color in the last row of the colormap. The intermediate values of Z map linearly to the intermediate rows in the colormap.

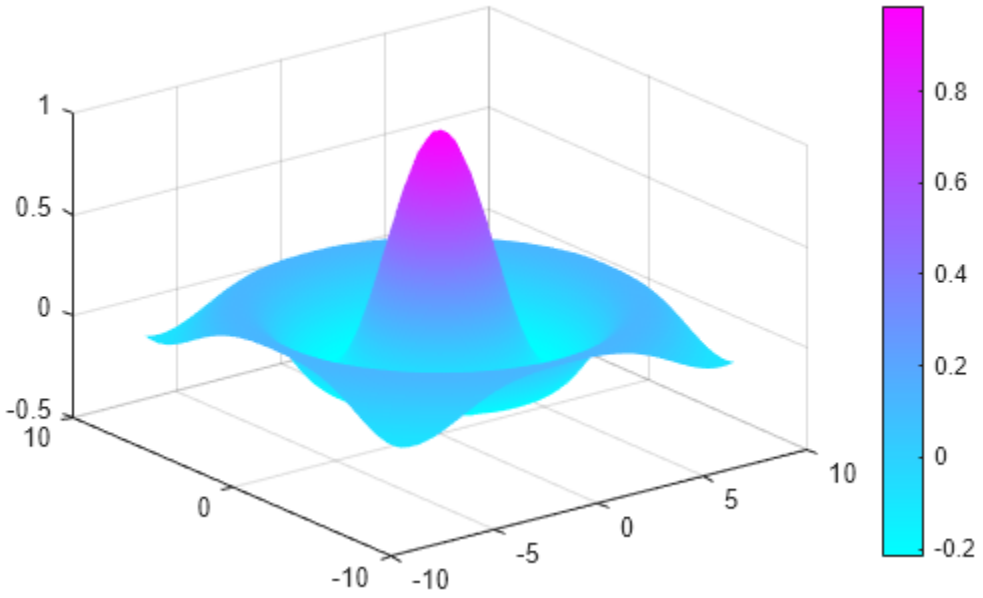
For example, create a surface plot and set the colormap to `cool`. Display a colorbar to the right of the plot to indicate the color scale.

```
surf(X,Y,Z)
colormap cool
colorbar
```



You can adjust how the colors blend across the surface by using the `shading` function. For example, interpolate the colors across the faces and edges of the surface.

```
shading interp
```

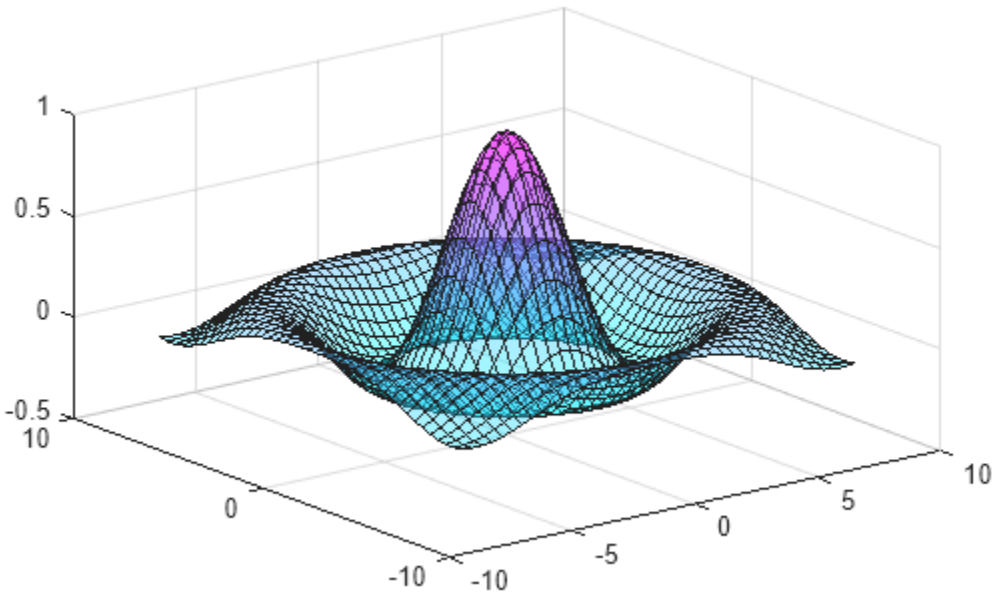


Making Surfaces Transparent

You can specify the transparency (also called the alpha value) as a single value for the whole surface. Alpha values range from 0 (completely transparent) to 1 (opaque).

Plot a surface and apply an alpha value of `0.4` to the entire surface.

```
surf(X,Y,Z)
alpha(0.4)
```

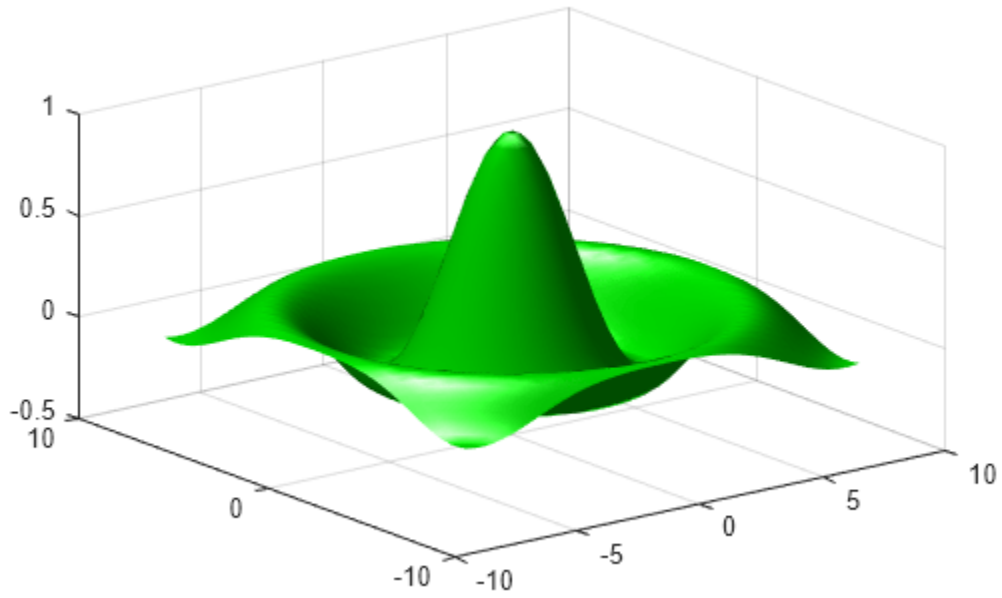


Lighting Surface Plots

Lighting is the technique of illuminating an object with a directional light source. You can also use lighting to add realism to three-dimensional plots.

Plot a solid green surface without displaying the mesh lines. Then add a light and set the lighting method to `gouraud`.

```
surf(X,Y,Z,"FaceColor","green","EdgeColor","none")
camlight left;
lighting gouraud
```


1
0.5
0
-0.5
10
0
-10
-10
-5
0
5
10

Programming

- “Control Flow” on page 5-2
- “Scripts and Functions” on page 5-8

Control Flow

In this section...
“Conditional Control — if, else, switch” on page 5-2
“Loop Control — for, while, continue, break” on page 5-4
“Program Termination — return” on page 5-6
“Vectorization” on page 5-6
“Preallocation” on page 5-6

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.

```
A = magic(4);
B = A;
B(1,1) = 0;

A == B

ans =

  4×4 logical array

   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 =

  logical
```

```
   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` delimits 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` delimits 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 = 0.01;
y = log10(x);
for k = 1:999
  x(k+1) = x(k) + 0.01;
  y(k+1) = log10(x(k+1));
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” on page 5-8
“Scripts” on page 5-8
“Functions” on page 5-9
“Types of Functions” on page 5-10
“Global Variables” on page 5-12
“Command vs. Function Syntax” on page 5-12

Overview

MATLAB 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`.

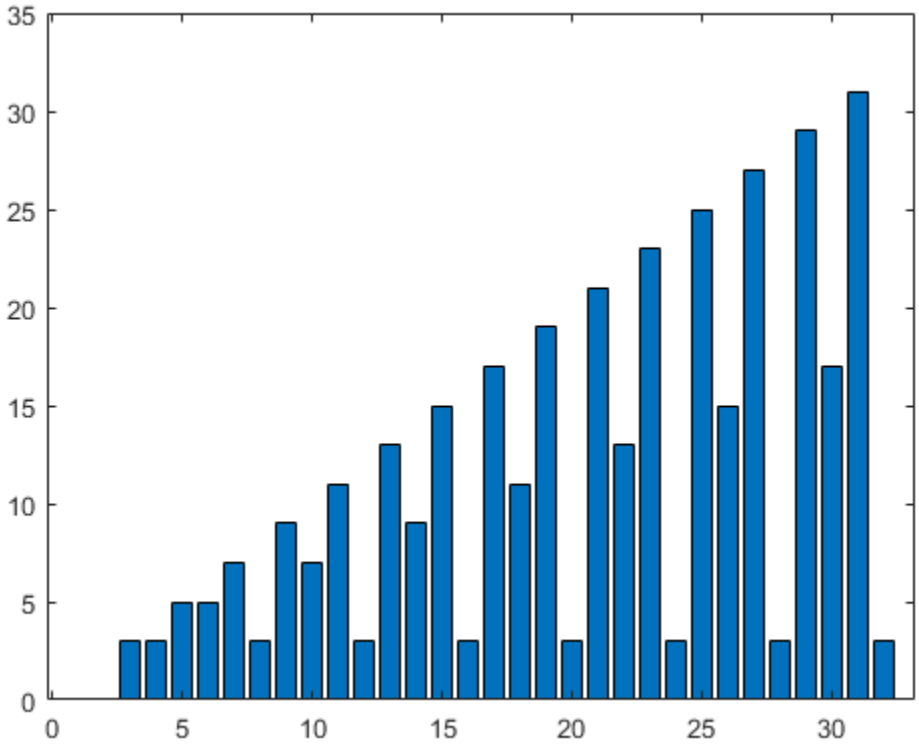
For example, create a file called `magicrank.m` that contains these MATLAB commands:

```
% Investigate the rank of magic squares
r = zeros(1,32);
for n = 3:32
   r(n) = rank(magic(n));
end
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, type

```
a = sqr(5)
a =
   25
```

Main Functions and Local Functions

Any function that is not anonymous must be defined within a file. Each such function file contains a required *main function* that appears first, and any number of *local functions* that can follow the main function. Main functions have a wider scope than local functions. That is, main 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 local functions cannot. Local functions are visible only to the main function and other local functions within their own file.

The `rank` function shown in the section on “Functions” on page 5-9 is an example of a main function.

Private Functions

A private function is a type of main 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 main function can be read or overwritten by a function nested at any level within the main function. 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
```