

Advanced Programming in MATLAB

Aaron Ponti

September 3, 2010

Abstract

In the first session of this course, we introduced some of the basic concepts of MATLAB programming. In this lecture, we want to explore some more advanced topics. Since the material we can cover is necessarily limited, references are given in every chapter for self study.

Contents

1	Visualization	3
1.1	References	3
1.2	A MATLAB graph	3
1.3	Figure toolbars	5
1.4	Types of plots	6
1.4.1	Two-dimensional plotting functions	6
1.4.2	Three-dimensional plotting functions	8
1.5	Interactive plots with the plot tools	9
1.6	Basic plotting commands	9
1.6.1	Creating Figure Windows	9
1.6.2	Displaying Multiple Plots per Figure	10
1.6.3	Specifying the Target Axes	11
1.7	Using High-Level Plotting Functions	12
1.7.1	Functions for Plotting Line Graphs	12
1.7.2	Programmatic plotting	12
1.7.3	Creating line plots	13
1.7.4	Specifying line style	15
1.7.5	Colors, line styles, and markers	16
1.7.6	Specifying the Color and Size of Lines	17
1.7.7	Adding Plots to an Existing Graph	18
1.7.8	Line Plots of Matrix Data	18
1.7.9	Plotting with two y-axes	20
1.7.10	Combining linear and logarithmic axes	21
1.8	Setting axis parameters	22
1.8.1	Axis Scaling and Ticks	22
1.8.2	Axis Limits and Ticks	23
1.8.3	Semiautomatic Limits	23
1.8.4	Axis tick marks	23
1.8.5	Example — Specifying Ticks and Tick Labels	23

1.8.6	Setting aspect ratio	24
1.9	Printing and exporting	26
1.9.1	Graphical user interfaces	26
1.9.2	Command line interface	31
2	Creating Graphical User Interfaces	33
2.1	References	33
2.2	What is a GUI?	33
2.3	Ways to build MATLAB GUIs	34
2.4	Creating a simple GUI with GUIDE	35
2.4.1	Laying out the GUI with GUIDE	35
2.4.2	Adding code to the GUI	44
2.5	Creating a simple GUI programmatically	46
2.5.1	Creating a GUI code file	47
2.5.2	Laying out a simple GUI	47
3	Object-oriented programming	54
3.1	References	54
3.2	Introduction to OOP	54
3.3	Classes in MATLAB	55
3.4	User-defined classes	55
3.5	MATLAB classes - key terms	55
3.6	Handle vs. value classes	56
3.7	Class folders	57
3.8	Class building blocks	57
3.8.1	The classdef block	57
3.8.2	The properties block	58
3.8.3	The methods block	60
3.8.4	The events block	62
3.8.5	Specifying attributes	63
3.9	Example: a polynomial class	64
3.9.1	Creating the needed folder and file	64
3.9.2	Using the DocPolynom Class	64
3.9.3	The DocPolynom Constructor Method	65
3.9.4	Converting DocPolynom Objects to Other Types	66
3.9.5	The DocPolynom disp method	69
3.10	Defining the + Operator	69
3.11	Overloading MATLAB Functions roots and polyval for the DocPolynom Class	70
3.11.1	Defining the roots function	70
3.11.2	Defining the polyval function	71
3.11.3	Complete DocPolynom example	71
4	Interfacing with Java	71
4.1	References	71
4.2	Java Virtual Machine (JVM)	72
4.3	The Java classpath	72
4.3.1	Static classpath	72
4.3.2	Dynamic classpath	73
4.3.3	Adding JAR packages	73

4.4	Simplifying Java Class Names	73
4.5	Creating and using Java objects	74
4.5.1	Constructing Java objects	74
4.5.2	Invoking methods on Java objects	74
4.5.3	Obtaining information about methods	74
4.6	Passing arguments to and from a Java method	76
4.6.1	Conversion of MATLAB data types	76
4.6.2	Conversion of Java return data types	77
5	Interfacing with C/C++	78
5.1	References	78
5.2	MEX files	78
5.3	Overview of Creating a C/C++ Binary MEX-File	78
5.4	Configuring your environment	79
5.5	Using MEX-files to call a C program	80
5.5.1	Create a source MEX file	80
5.5.2	Create a <i>gateway routine</i>	80
5.5.3	Use preprocessor macros	81
5.5.4	Verify Input and Output Parameters	82
5.5.5	Read input data	83
5.5.6	Prepare output data	83
5.5.7	Perform Calculation	83
5.5.8	Build the Binary MEX-File	83

1 Visualization

1.1 References

- http://www.mathworks.com/access/helpdesk/help/techdoc/creating_plots/bqrw9tj.html
- http://www.mathworks.com/access/helpdesk/help/pdf_doc/matlab/graphg.pdf

1.2 A MATLAB graph

The MATLAB environment offers a variety of data plotting functions plus a set of Graphical User Interface (GUI) tools to create, and modify graphic displays.

A *figure* is a MATLAB window that contains graphic displays (usually data plots) and UI components. You create figures explicitly with the *figure* function, and implicitly whenever you plot graphics and no figure is active.

```
h = figure;
```

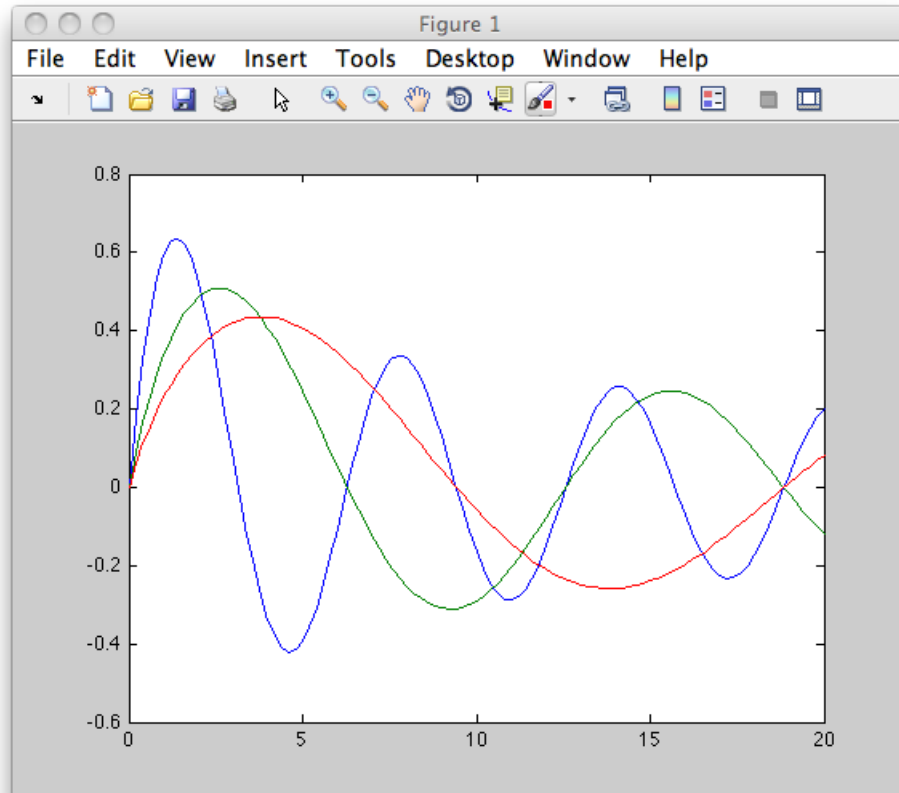
By default, figure windows are resizable and include pull-down menus and tool-bars.

A *plot* is any graphic display you can create within a figure window. Plots can display tabular data, geometric objects, surface and image objects, and annotations such as titles, legends, and colorbars. Figures can contain any

number of plots. Each plot is created within a 2-D or a 3-D data space called an *axes*. You can explicitly create axes with the *axes* or *subplot* functions.

A *graph* is a plot of data within a 2-D or 3-D axes. Most plots made with MATLAB functions and GUIs are therefore graphs. When you graph a one-dimensional variable (e.g., *rand(100,1)*), the indices of the data vector (in this case *1:100*) become assigned as *x* values, and plots the data vector as *y* values. Some types of graphs can display more than one variable at a time, others cannot.

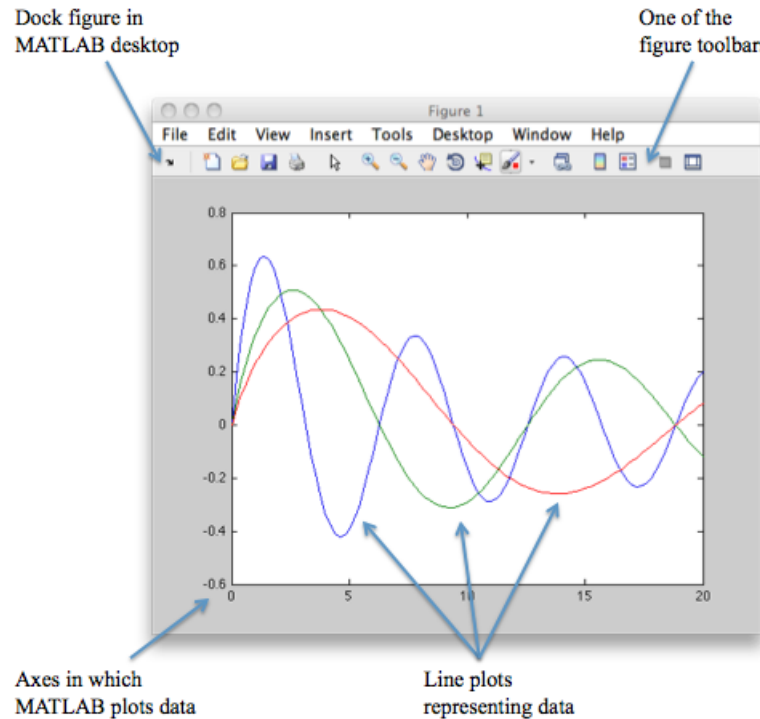
```
x = 0:.2:20;  
y = sin(x)./sqrt(x+1);  
y(2,:) = sin(x/2)./sqrt(x+1);  
y(3,:) = sin(x/3)./sqrt(x+1);  
plot(x,y)
```



The resulting figure contains a 2-D set of axes.

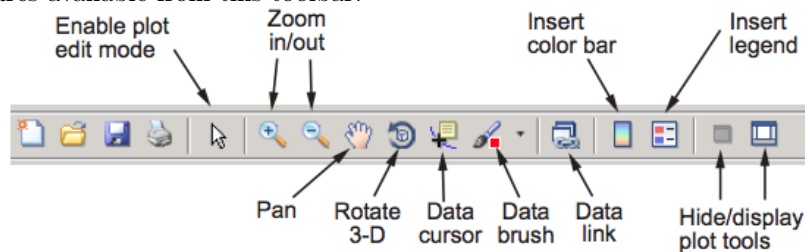
The plot function uses a default line style and color to distinguish the data sets plotted in the graph. You can change the appearance of these graphic components or add annotations to the graph to present your data in a particular way.

This graphic identifies the components and tools of a figure window.



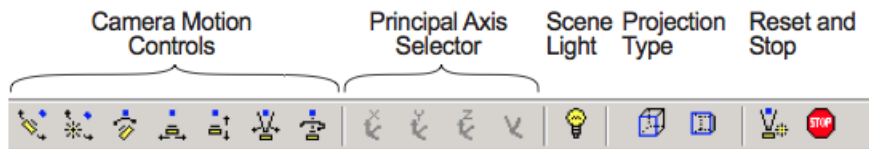
1.3 Figure toolbars

Figure toolbars provide shortcuts to access commonly used features. These include operations such as saving and printing, plus tools for interactive zooming, panning, rotating, querying, and editing plots. The following picture shows the features available from this toolbar.

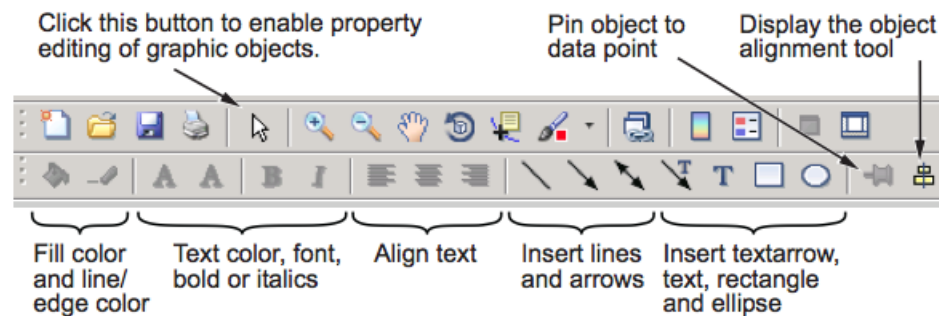


You can enable two other toolbars from the **View** menu:

- **Camera toolbar.** Use for manipulating 3-D views, with camera, light, and projection controls.



- **Plot Edit toolbar.** Use for annotation and setting plot and object properties.



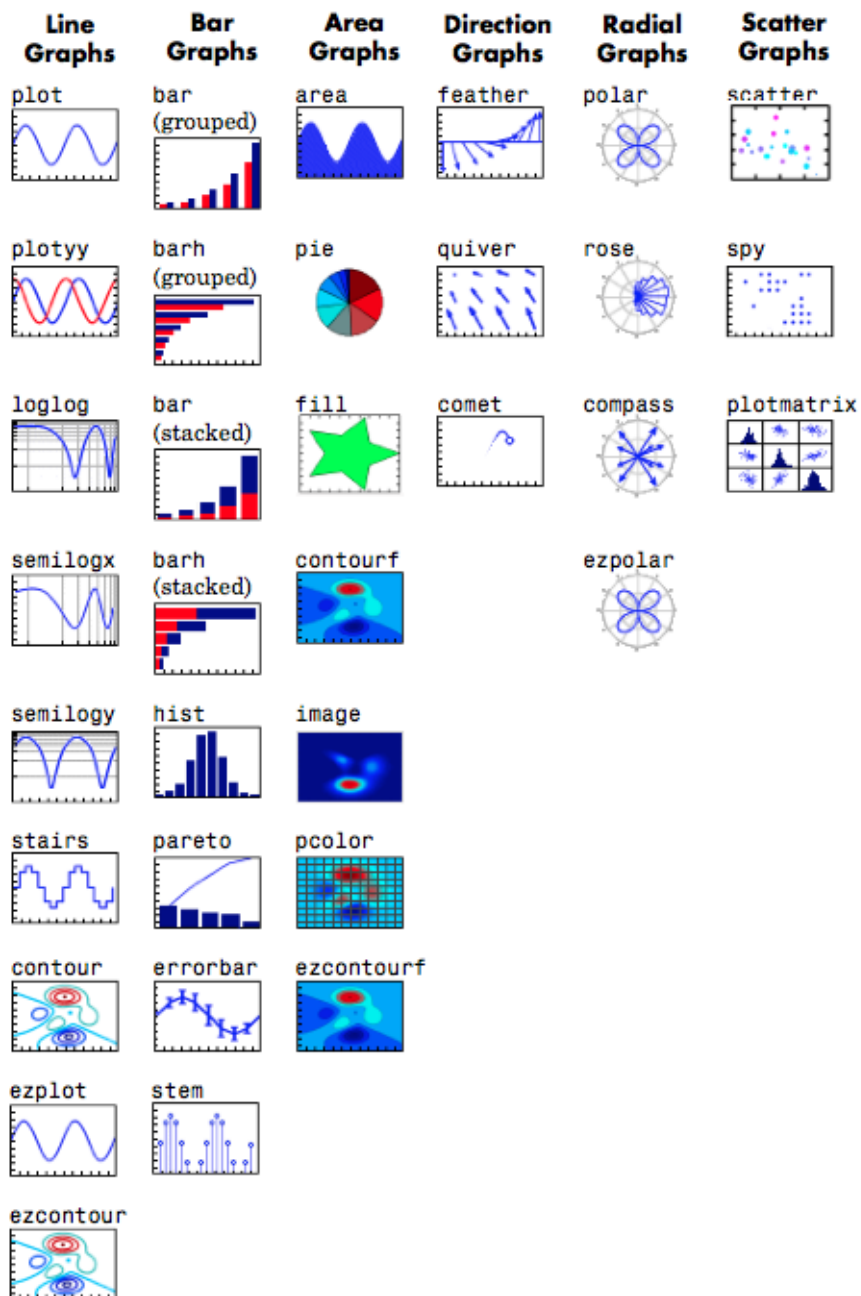
1.4 Types of plots

You can construct a wide variety of 2-D and 3-D MATLAB plots with very little, if any, programming required on your part. The following two tables classify and illustrate most of the kinds of plots you can create. They include line, bar, area, direction and vector field, radial, and scatter graphs. They also include 2-D and 3-D functions that generate and plot geometric shapes and objects. Most 2-D plots have 3-D analogs, and there are a variety of volumetric displays for 3-D solids and vector fields. Plot types that begin with “ez” (such as *ezsurf*) are convenience functions that can plot arguments given as functions.

1.4.1 Two-dimensional plotting functions


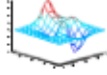

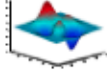







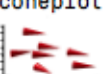




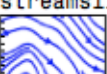
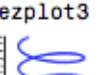



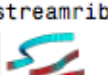




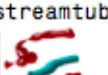



The table below shows all available MATLAB 2-D plot functions. You can get documentation for each of the functions by typing *help function_name* in the MATLAB console, for example:

```
help plotyy
```



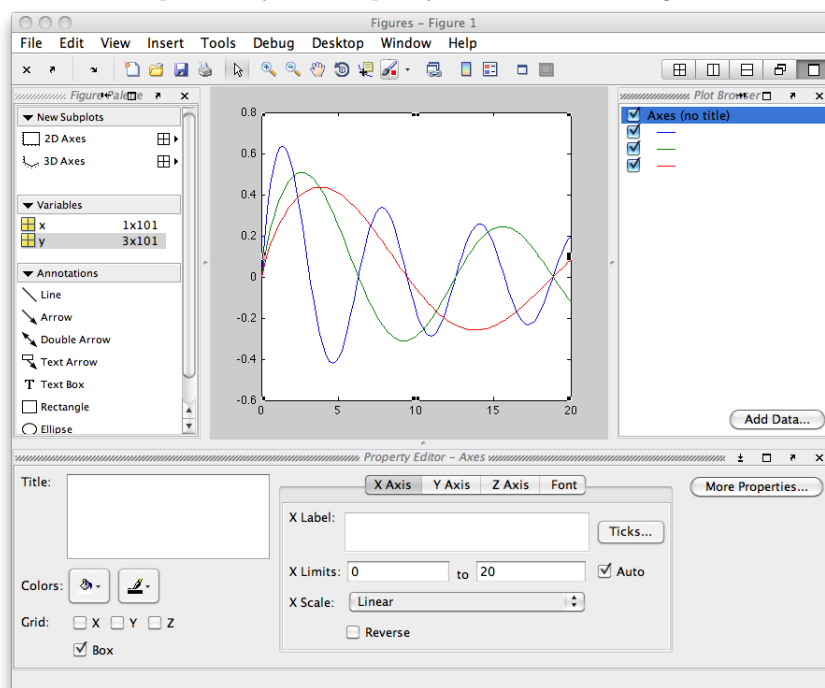
1.4.2 Three-dimensional plotting functions

The table below shows all available MATLAB 3-D and volumetric plot functions. It includes functions that generate 3-D data (*cylinder*, *ellipsoid*, *sphere*), but most plot either arrays of data or functions.

Line Graphs	Mesh Graphs and Bar Graphs	Area Graphs and Constructive Objects	Surface Graphs	Direction Graphs	Volumetric Graphs
plot3 	mesh 	pie3 	surf 	quiver3 	scatter3 
contour3 	meshc 	fill3 	surf1 	comet3 	coneplot 
contourslice 	meshz 	patch 	surfc 	streamslicestreamline 	
ezplot3 	ezmesh 	cylinder 	ezsurf 		streamribbon 
waterfall 	stem3 	ellipsoid 	ezsurfc 		streamtube 
	bar3 	sphere 			
	bar3h 				

1.5 Interactive plots with the plot tools

Most of the plotting functions shown in the previous tables are accessible through the **Figure Palette**, one of the **Plot Tools** you can access via the figure window **View** menu. When the Figure Palette is active and you select one, two or more variables listed within it, you can generate a plot of any appropriate type by right-clicking and selecting a plot type from the context menu that appears. The lowest item on that menu is **More Plots**. When you select **More Plots**, the *Plot Catalog* opens for you to browse through all plot types and generate one of them, either to display the variables you selected in the Figure Palette or a MATLAB expression you can specify in the Plot Catalog window.



We won't discuss the plot tools in this course.

1.6 Basic plotting commands

1.6.1 Creating Figure Windows

MATLAB graphics are directed to a window that is separate from the Command Window. This window is referred to as a **figure**. The characteristics of this window are controlled by your computer's windowing system and MATLAB figure properties.

Graphics functions automatically create new MATLAB figure windows if none currently exist. If a figure already exists, that window is used. If multiple figures exist, one is designated as the current figure and is used (this is generally

the last figure used or the last figure you clicked the mouse in). The figure function creates figure windows. For example,

```
figure
```

creates a new window and makes it the current figure. You can make an existing figure current by clicking it with the mouse or by passing its handle (the number indicated in the window title bar), as an argument to figure.

```
figure(2)
```

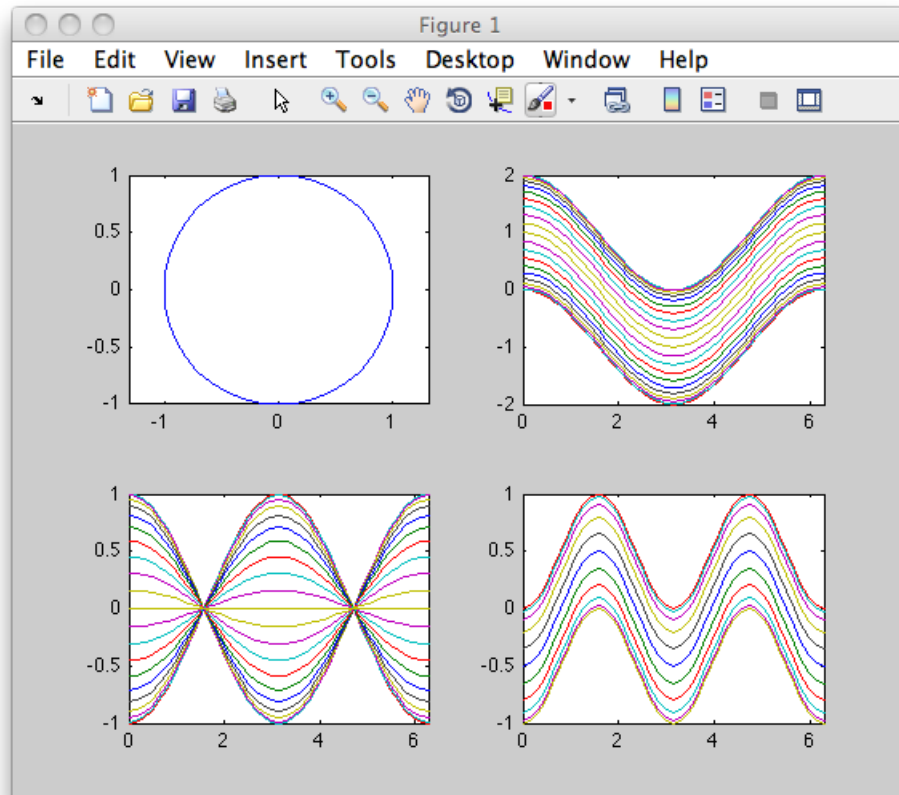
The handle is returned when the figure is created.

```
h = figure  
h =  
    3
```

1.6.2 Displaying Multiple Plots per Figure

You can display multiple plots in the same figure window and print them on the same piece of paper with the *subplot* function. *subplot(m,n,i)* breaks the figure window into an m-by-n matrix of small subplots and selects the *i*th subplot for the current plot. The plots are numbered along the top row of the figure window, then the second row, and so forth. For example, the following statements plot data in four different subregions of the figure window.

```
t = 0:pi/20:2*pi;  
[x,y] = meshgrid(t);  
subplot(2,2,1); plot(sin(t),cos(t)); axis equal;  
subplot(2,2,2); z = sin(x)+cos(y); plot(t,z); axis([0 2*pi -2 2]);  
subplot(2,2,3); z = sin(x).*cos(y); plot(t,z); axis([0 2*pi -1 1]);  
subplot(2,2,4); z = (sin(x).^2)-(cos(y).^2); plot(t,z); axis([0 2*pi -1 1]);
```



Each subregion contains its own axes with characteristics you can control independently of the other subregions. This example uses the *axis* function to set limits and change the shape of the subplots. See the *axes*, *axis*, and *subplot* functions for more information.

1.6.3 Specifying the Target Axes

The current axes is the last one defined by *subplot*. If you want to access a previously defined subplot, for example to add a title, you must first make that axes current. You can make an axes current in three ways:

- Click on the subplot with the mouse.
- Call *subplot* with the *m*, *n*, *i* specifiers.

```
subplot(2,2,2); title('Top Right Plot');
```

- Call *subplot* with the handle (identifier) of the axes.

```
h = get(gcf,'Children');
```

```
subplot(h(1)); title('Most recently created axes');
subplot(gca); title('Current active axes');
```

The call

```
get(gcf, 'Children');
```

returns the handles of all the axes, with the most recently created one first. The function *gca* returns the handle of the current active axes.

1.7 Using High-Level Plotting Functions

1.7.1 Functions for Plotting Line Graphs

Many types of MATLAB functions are available for displaying vector data as line plots, as well as functions for annotating and printing these graphs. The following table summarizes the functions that produce basic line plots. These functions differ in the way they scale the plot's axes. Each accepts input in the form of vectors or matrices and automatically scales the axes to accommodate the data.

Function	Description
<i>plot</i>	Graph 2-D data with linear scales for both axes
<i>plot3</i>	Graph 3-D data with linear scales for both axes
<i>loglog</i>	Graph with logarithmic scales for both axes
<i>semilogx</i>	Graph with a logarithmic scale for the x-axis and a linear scale for the y-axis
<i>semilogy</i>	Graph with a logarithmic scale for the y-axis and a linear scale for the x-axis
<i>plotyy</i>	Graph with y-tick labels on the left and right side

1.7.2 Programmatic plotting

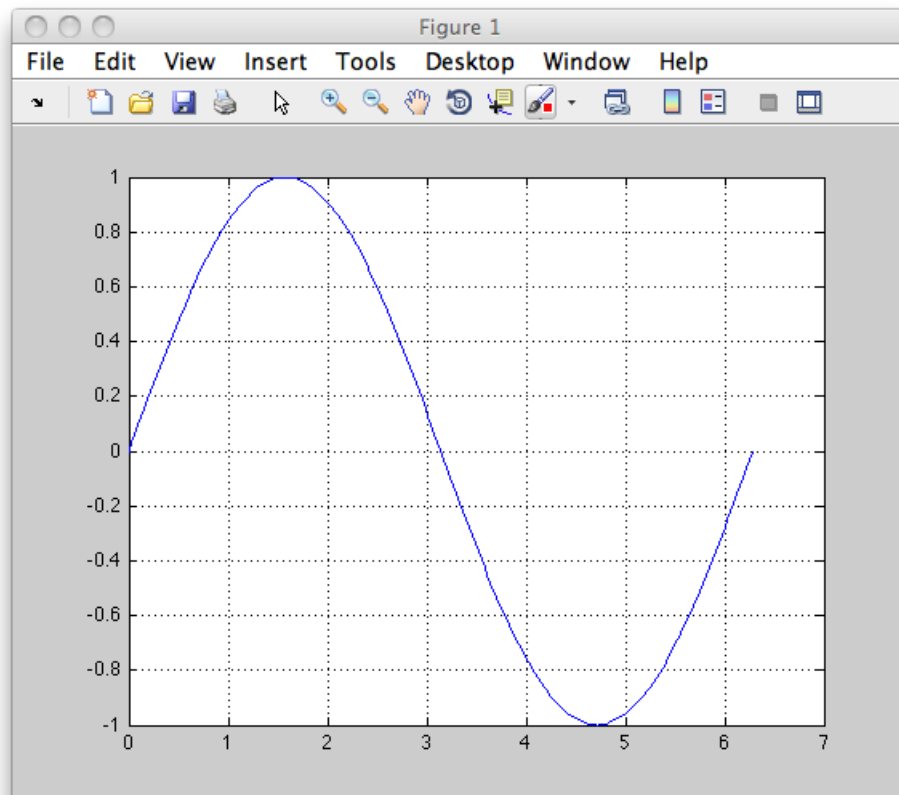
The process of constructing a basic graph to meet your presentation graphics requirements is outlined in the following table. The table shows seven typical steps and some example code for each. If you are performing analysis only, you may want to view various graphs just to explore your data. In this case, steps 1 and 3 may be all you need. If you are creating presentation graphics, you may want to fine-tune your graph by positioning it on the page, setting line styles and colors, adding annotations, and making other such improvements.

Step	Typical code
1 Prepare your data	<code>x = 0:0.2:12; y1 = bessell(1,x); y2 = bessell(2,x); y3 = bessell(3,x);</code>
2 Select a window and position a plot region within the window	<code>figure(1); subplot(2,2,1);</code>
3 Call elementary plotting function	<code>h = plot(x,y1,x,y2,x,y3);</code>
4 Select line and marker characteristics	<code>set(h,'LineWidth',2, ... { 'LineStyle' }, { '-' ; ':' ; '-' }); set(h, { 'Marker' }, { 'none' ; 'o' ; 'x' }); set(h, { 'Color' }, { 'r' ; 'g' ; 'b' });</code>
5 Set axis limits, tick marks, and grid lines	<code>axis([0 12 -0.5 1]); grid on;</code>
6 Annotate the graph with axis labels, legend, and text	<code>xlabel('Time'); ylabel('Amplitude'); legend(h,'First','Second','Third'); title('Bessel Functions'); [y,ix] = min(y1); text(x(ix),y,'First Min \rightarrow', ... 'HorizontalAlignment','right');</code>
7 Export graph	<code>print -depsc -tiff -r200 myplot</code>

1.7.3 Creating line plots

The *plot* function has different forms depending on the input arguments. For example, if *y* is a vector, *plot(y)* produces a linear graph of the elements of *y* versus the index of the elements of *y*. If you specify two vectors as arguments, *plot(x,y)* produces a graph of *y* versus *x*. For example, the following statements create a vector of values in the range $[0, 2\pi]$ in increments of $\pi/100$ and then use this vector to evaluate the sine function over that range. MATLAB plots the vector on the x-axis and the value of the sine function on the y-axis.

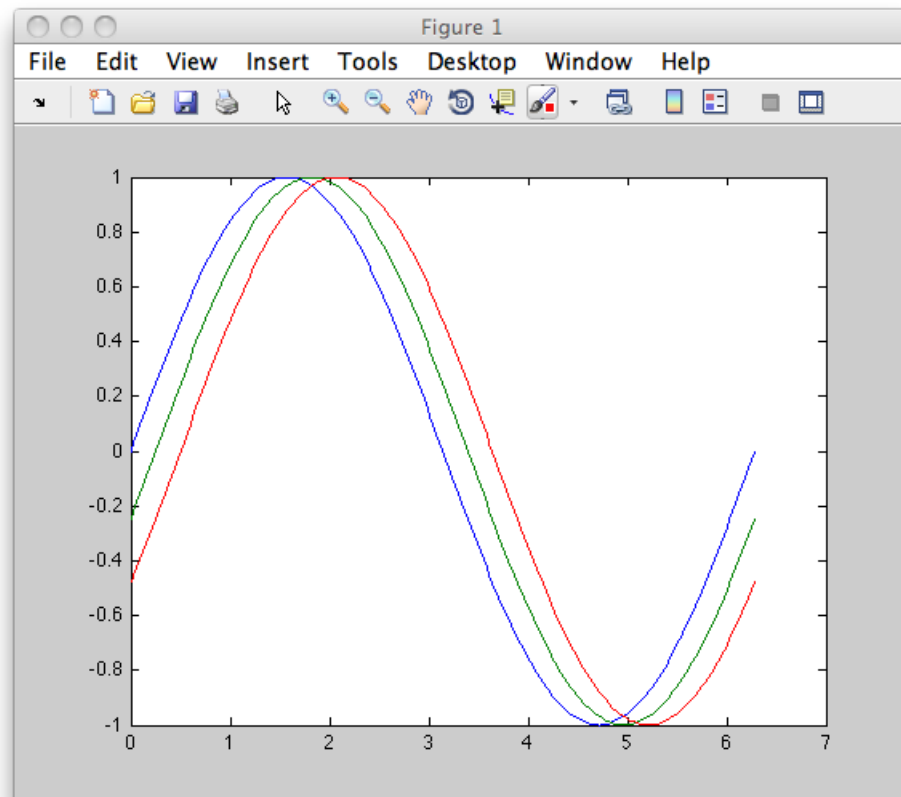
```
t = 0:pi/100:2*pi;
y = sin(t);
plot(t,y);
grid on % Turn on grid lines for this plot
```



Appropriate axis ranges and tick mark locations are automatically selected.

You can plot multiple graphs in one call to plot using x - y pairs. MATLAB automatically cycles through a predefined list of colors (determined by the axes *ColorOrder* property) to allow discrimination between sets of data. Plotting three curves as a function of t produces

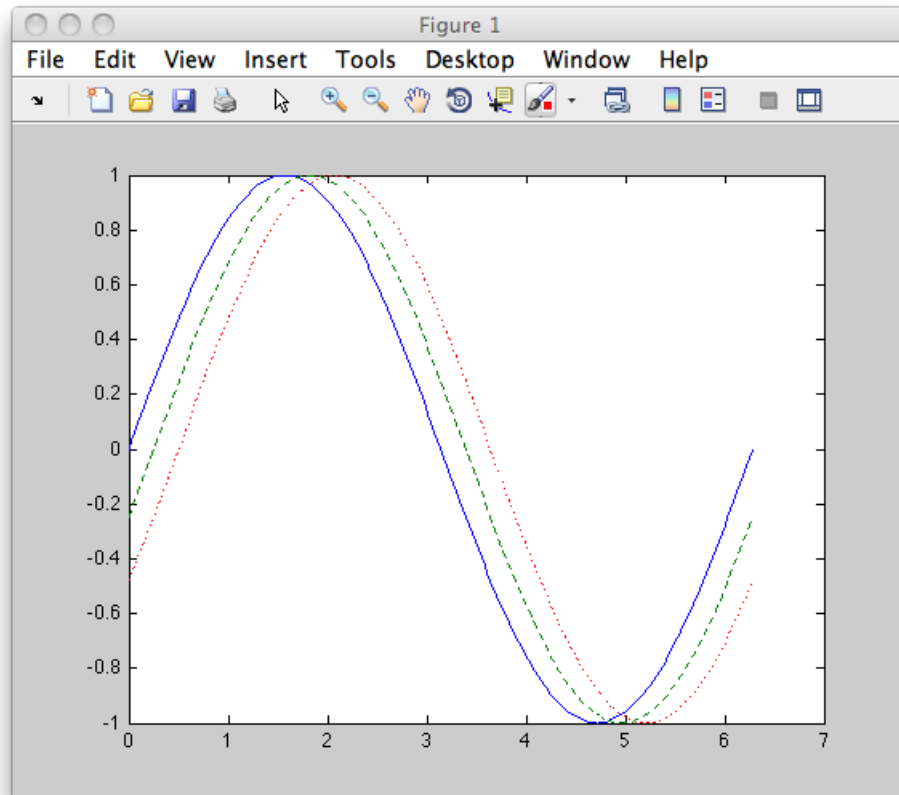
```
y = sin(t);  
y2 = sin(t-0.25);  
y3 = sin(t-0.5);  
plot(t,y,t,y2,t,y3)
```



1.7.4 Specifying line style

You can assign different line styles to each data set by passing line style identifier strings to `plot`. For example,

```
plot(t,y,'- ',t,y2,'-- ',t,y3,': ')
```



The graph shows three lines of different colors and lines styles representing the value of the sine function with a small phase shift between each line, as defined by y , $y2$, and $y3$. The lines are blue solid, green dashed, and red dotted.

1.7.5 Colors, line styles, and markers

The basic plotting functions accepts character-string arguments that specify various line styles, marker symbols, and colors for each vector plotted. In the general form,

```
plot(x,y,'linestyle_marker_color')
```

linestyle_marker_color is a character string (delineated by single quotation marks) constructed from:

- A line style (e.g., dashed, dotted, etc.)
- A marker type (e.g., x , $*$, o , etc.)
- A predefined color specifier (c , m , y , k , r , g , b , w)

For example,

```
plot(x,y,':squarey')
```

plots a yellow dotted line and places square markers at each data point. If you specify a marker type, but not a line style, only the marker is plotted. The specification can consist of one or none of each specifier in any order. For example, the string

```
'go--'
```

defines a dashed line with circular markers, both colored green. You can also specify the size of the marker and, for markers that are closed shapes, you can specify separately the colors of the edges and the face.

1.7.6 Specifying the Color and Size of Lines

You can control a number of line style characteristics by specifying values for line properties:

- *LineWidth* — Width of the line in units of points
- *MarkerEdgeColor* — Color of the marker or the edge color for filled markers (circle, square, diamond, pentagram, hexagram, and the four triangles)
- *MarkerFaceColor* — Color of the face of filled markers
- *MarkerSize* — Size of the marker in units of points

For example, these statements,

```
x = -pi:pi/10:pi;  
y = tan(sin(x)) - sin(tan(x));  
plot(x,y,'--rs','LineWidth',2,...  
     'MarkerEdgeColor','k',...  
     'MarkerFaceColor','g',...  
     'MarkerSize',10)
```

produce a graph with:

- A red dashed line with square markers
- A line width of two points
- The edge of the marker colored black
- The face of the marker colored green
- The size of the marker set to 10 points

Try!

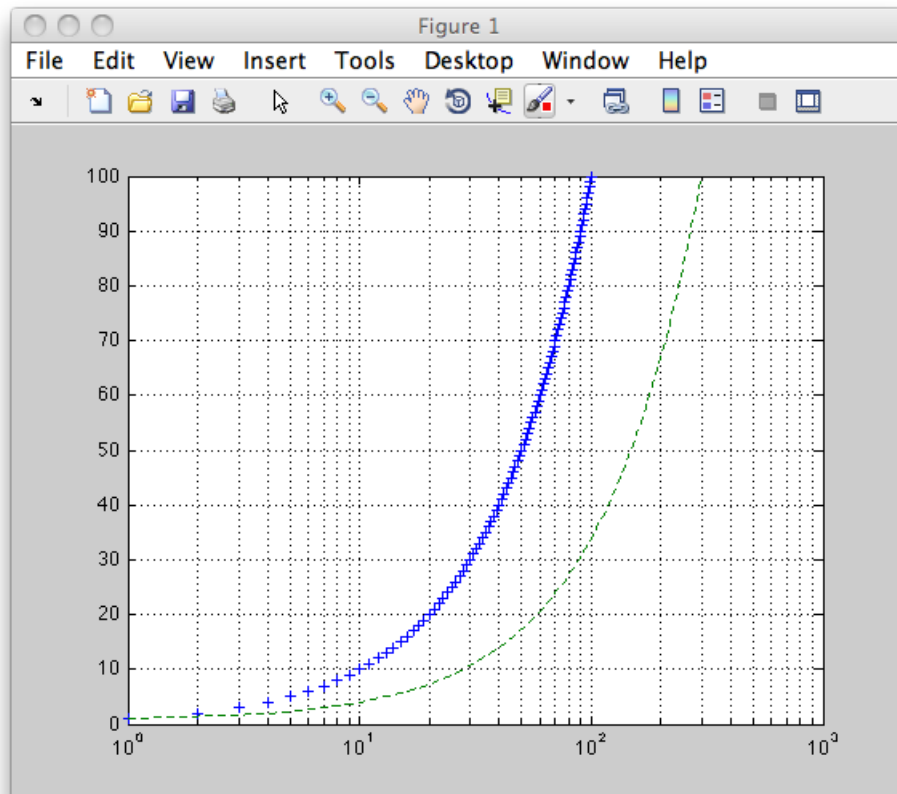
1.7.7 Adding Plots to an Existing Graph

You can add plots to an existing graph using the *hold* command. When you set *hold* to *on*, MATLAB does not remove the existing graph; it adds the new data to the current graph, rescaling if the new data falls outside the range of the previous axis limits.

For example, these statements first create a semilogarithmic plot, then add a linear plot.

```
semilogx(1:100,'+')
hold all % hold plot and cycle line colors
plot(1:3:300,1:100,'--')
hold off
grid on % Turn on grid lines for this plot
```

The x-axis limits are reset to accommodate the new data, but the scaling from logarithmic to linear does not change.



1.7.8 Line Plots of Matrix Data

When you call the *plot* function with a single matrix argument

```
plot(Y)
```

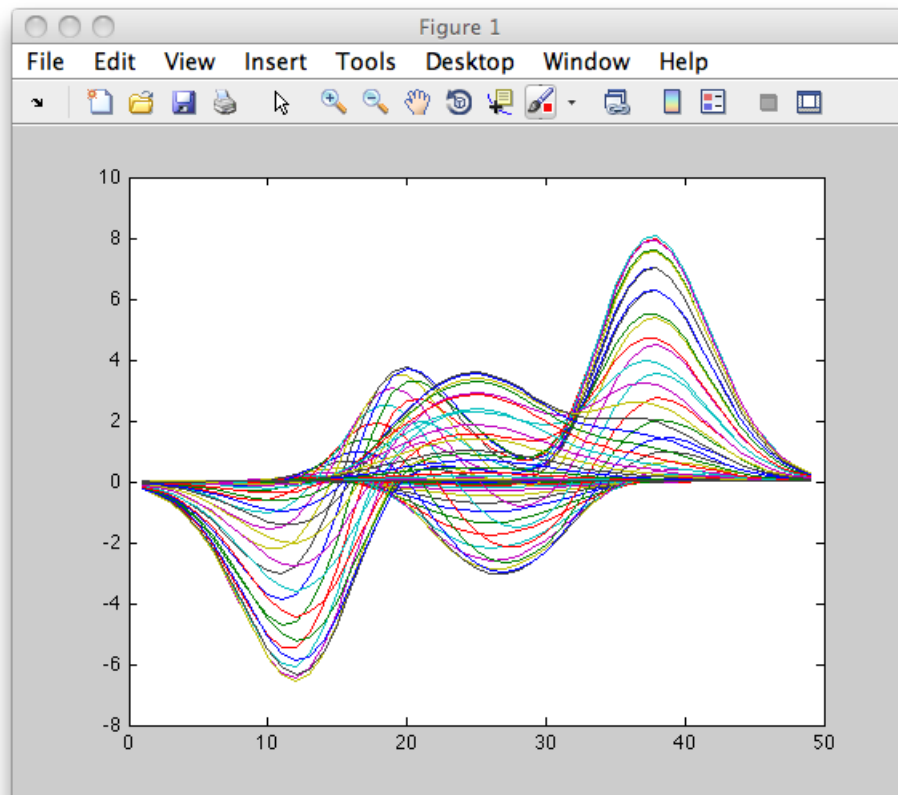
one line is plotted for each column of the matrix. The x-axis is labeled with the row index vector $1:m$, where m is the number of rows in Y . For example,

```
Z = peaks;
```

returns a 49-by-49 matrix obtained by evaluating a function of two variables. Plotting this matrix

```
plot(Z)
```

produces a graph with 49 lines.



In general, if *plot* is used with two arguments and if either X or Y has more than one row or column, then:

- If Y is a matrix, and x is a vector, *plot*(x, Y) successively plots the rows or columns of Y versus vector x , using different colors or line types for each. The row or column orientation varies depending on whether the number of elements in x matches the number of rows in Y or the number of columns. If Y is square, its columns are used.

- If X is a matrix and y is a vector, `plot(X,y)` plots each row or column of X versus vector y . For example, plotting the peaks matrix versus the vector `1:length(peaks)` rotates the previous plot.
- If X and Y are both matrices of the same size, `plot(X,Y)` plots the columns of X versus the columns of Y . You can also use the plot function with multiple pairs of matrix arguments.

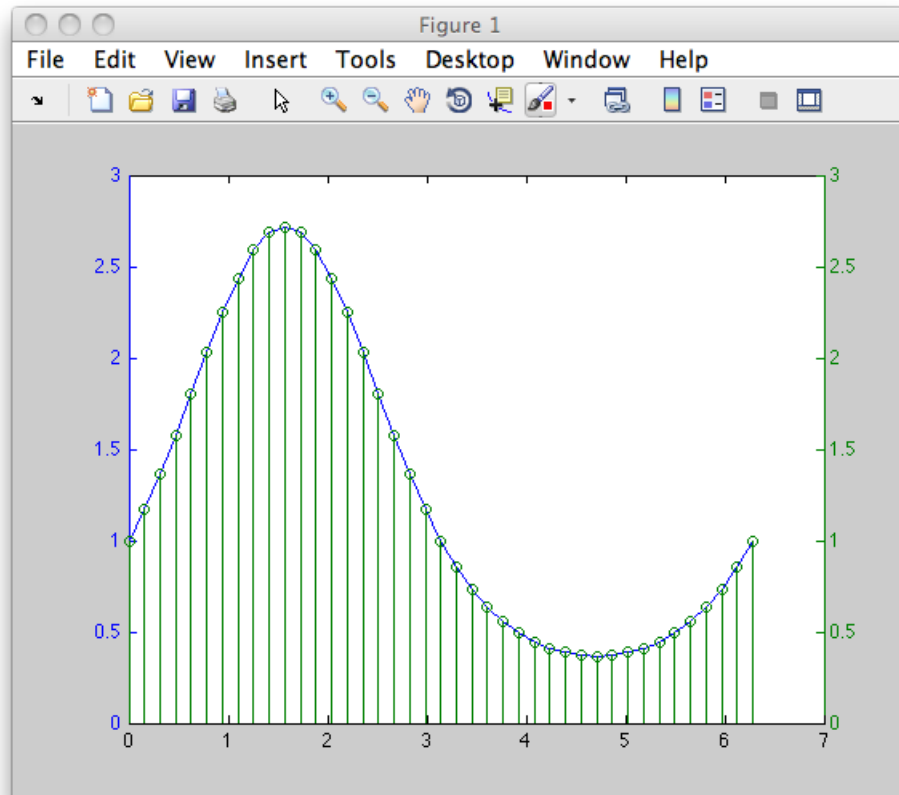
```
plot(X1,Y1,X2,Y2,...)
```

This statement graphs each X-Y pair, generating multiple lines. The different pairs can be of different dimensions.

1.7.9 Plotting with two y-axes

The `plotyy` function enables you to create plots of two data sets and use both left and right side y-axes. You can also apply different plotting functions to each data set. For example, you can combine a line plot with a stem plot of the same data.

```
t = 0:pi/20:2*pi;
y = exp(sin(t));
plotyy(t,y,t,y,'plot','stem')
```



1.7.10 Combining linear and logarithmic axes

You can use *plotyy* to apply linear and logarithmic scaling to compare two data sets having different ranges of values.

```
t = 0:900; A = 1000; a = 0.005; b = 0.005;
z1 = A*exp(-a*t);
z2 = sin(b*t);
[haxes,hline1,hline2] = plotyy(t,z1,t,z2,'semilogy','plot');
```

This example saves the handles of the lines and axes created to adjust and label the graph. First, label the axes whose y value ranges from 10 to 1000. This is the first handle in *haxes* because it was specified first in the call to *plotyy*. Use the *axes* function to make *haxes(1)* the current axes, which is then the target for the *ylabel* function.

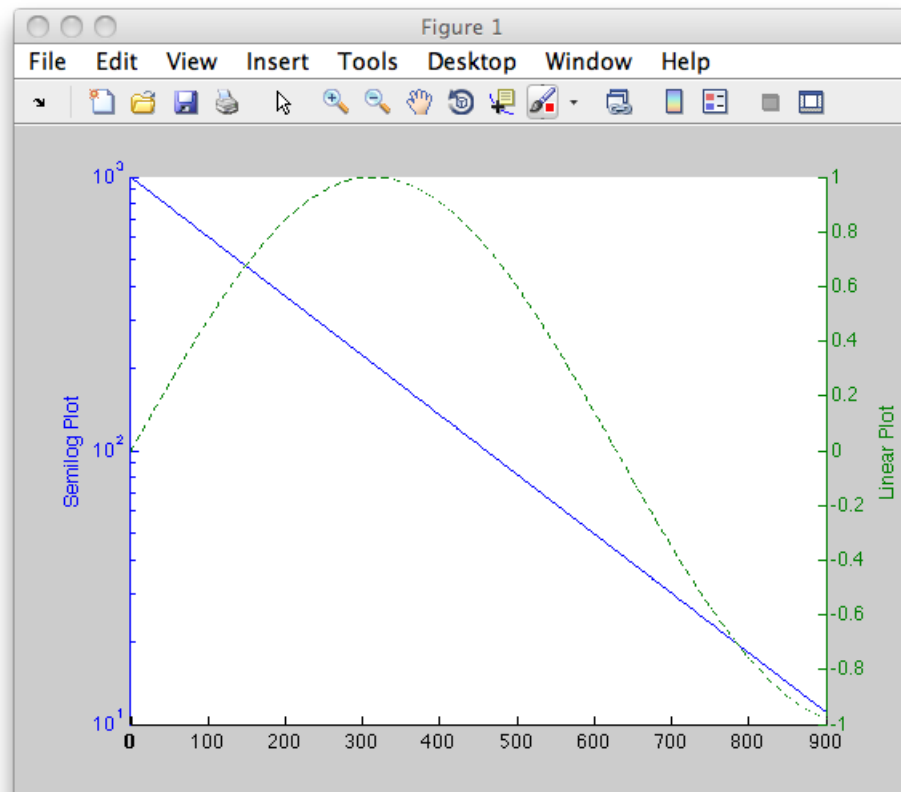
```
axes(haxes(1))
ylabel('Semilog Plot')
```

Now make the second axes current and call *ylabel* again.

```
axes(haxes(2))
ylabel('Linear Plot')
```

You can modify the characteristics of the plotted lines in a similar way. For example, to change the line style of the second line plotted to a dashed line, use the statement

```
set(hline2,'LineStyle','--')
```



1.8 Setting axis parameters

1.8.1 Axis Scaling and Ticks

When you create a MATLAB graph, the axis limits and tick-mark spacing are automatically selected based on the data plotted. However, you can also specify your own values for axis limits and tick marks with the following functions:

- *axis* — Sets values that affect the current axes object (the most recently created or the last clicked on).

- *axes* — Creates a new axes object with the specified characteristics.
- *get* and *set* — Enable you to query and set a wide variety of properties of existing axes.
- *gca* — Returns the handle (identifier) of the current axes. If there are multiple axes in the figure window, the current axes is the last graph created or the last graph you clicked on with the mouse. The following two sections provide more information and examples

1.8.2 Axis Limits and Ticks

By default, axis limits are chosen to encompass the range of the plotted data. You can specify the limits manually using the *axis* function. Call *axis* with the new limits defined as a four-element vector.

```
axis([xmin,xmax,ymin,ymax]);
```

The minimum values must be less than the maximum values.

1.8.3 Semiautomatic Limits

If you want to autoscale only one of a min/max set of axis limits, but you want to specify the other, use the MATLAB variable *Inf* or *-Inf* for the autoscaled limit.

```
axis([-Inf 5 2 2.5]);
```

1.8.4 Axis tick marks

The tick-mark locations are based on the range of data so as to produce equally spaced ticks (for linear graphs). You can specify different tick marks by setting the axes *XTick* and *YTick* properties. Define tick marks as a vector of increasing values. The values do not need to be equally spaced. For example:

```
set(gca,'YTick',[2 2.1 2.2 2.3 2.4 2.5]);
```

produces a graph with only the specified ticks on the y-axis.

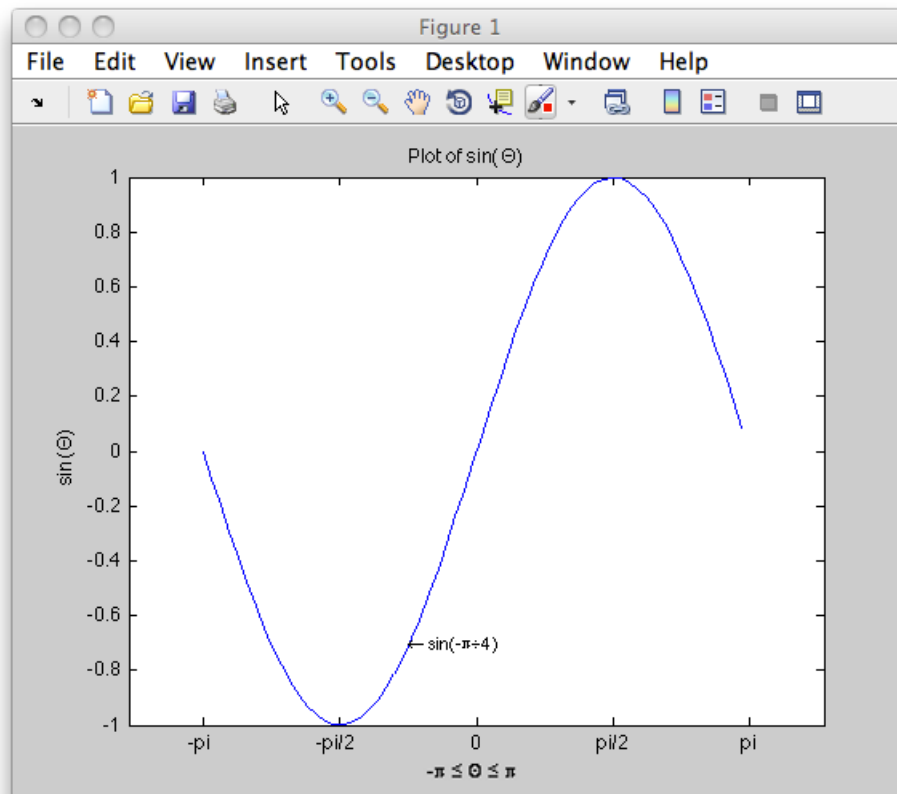
1.8.5 Example — Specifying Ticks and Tick Labels

You can adjust the axis tick-mark locations and the labels appearing at each tick mark. For example, this plot of the sine function relabels the x-axis with more meaningful values.

```
x = -pi:.1:pi;
y = sin(x);
plot(x,y);
set(gca,'XTick',-pi:pi/2:pi);
set(gca,'XTickLabel',{'-pi','-pi/2','0','pi/2','pi'})
```

These functions (*xlabel*, *ylabel*, *title*, *text*) add axis labels and draw an arrow that points to the location on the graph where $y = \sin(-\pi/4)$.

```
xlabel('-\pi \leq \Theta \leq \pi');
ylabel('sin(\Theta)');
title('Plot of sin(\Theta)');
text(-pi/4,sin(-pi/4),'\leftarrow sin(-\pi\div4)',...
    'HorizontalAlignment','left');
```



The Greek symbols are created using \TeX character sequences.

1.8.6 Setting aspect ratio

By default, graphs display in a rectangular axes that has the same aspect ratio as the figure window. This makes optimum use of space available for plotting. You exercise control over the aspect ratio with the `axis` function. For example,

```
t = 0:pi/20:2*pi;
plot(sin(t),2*cos(t));
grid on
```


produces a graph with the default aspect ratio. The command

```
axis square
```

makes the x- and y-axes equal in length.

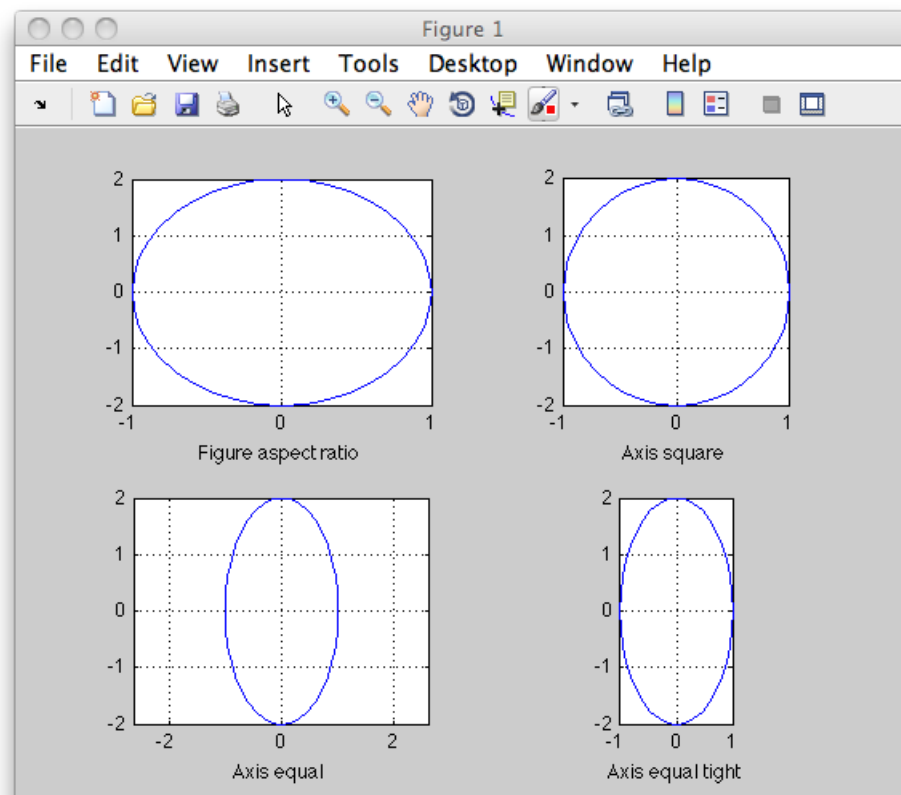
The square axes has one data unit in x to equal two data units in y. If you want the x- and y-data units to be equal, use the command

```
axis equal
```

This produces an axes that is rectangular in shape, but has equal scaling along each axis. If you want the axes shape to conform to the plotted data, use the tight option in conjunction with equal.

```
axis equal tight
```

The generated plots are displayed in the following figure.



1.9 Printing and exporting

There are four basic operations that you can perform in printing or transferring figures you've created with MATLAB graphics to specific file formats for other applications to use.

Opeation	Description
Print	Send a figure from the screen directly to the printer.
Print to File	Write a figure to a PostScript® file to be printed later.
Export to File	Export a figure in graphics format to a file, so that you can import it into an application.
Export to Clipboard	Copy a figure to the Microsoft Windows clipboard, so that you can paster it into an application.

1.9.1 Graphical user interfaces

In addition to typing MATLAB commands, you can use interactive tools for either Microsoft Windows or UNIX® to print and export graphics. The table below lists the GUIs available for doing this and explains how to open them from figure windows.

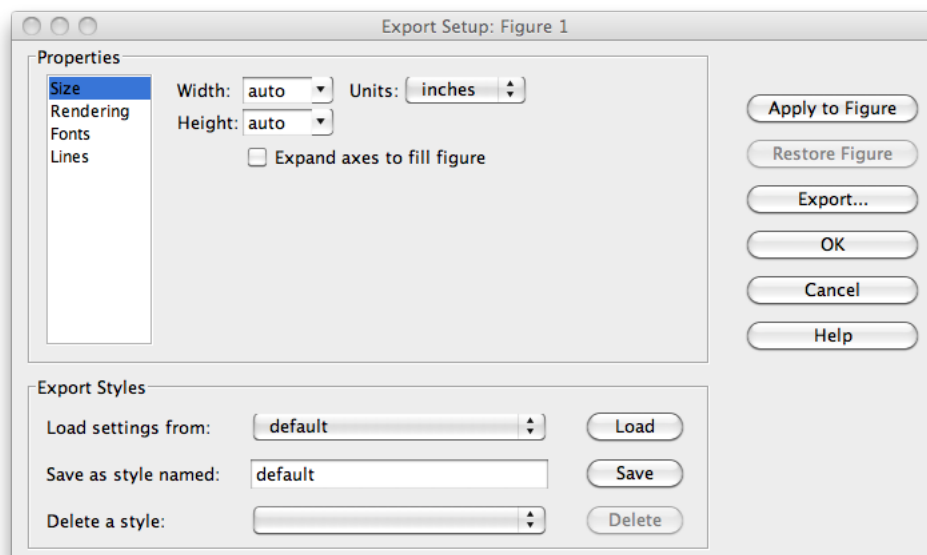
Dialog Box	How to open	Description
Print (Windows and Unix)	File > Print or <i>printdlg</i> function	Send figure to the printer, select the printer, print to file, and several other options
Print Preview	File > Print Preview or <i>printpreview</i> function	View and adjust the final output
Export	File > Export	Export the figure in graphics format to a file
Copy Options	Edit > Copy Options	Set format, figure size, and background color for Copy to Clipboard
Figure Copy Template	File > Preferences	Change text, line, axis, and UI control properties

Before you print or export a figure, preview the image by selecting **Print Preview** from the figure window's File menu. If necessary, you can use the *set* function (see below) to adjust specific characteristics of the printed or exported figure. Adjustments that you make in the **Print Preview** dialog also set figure properties; these changes can affect the output you get should you print the figure later with the *print* command.

The Export Setup GUI appears when you select **Export Setup** from the **File** menu of a figure window. This GUI has four dialog boxes that enable you to adjust the size, rendering, font, and line appearance of your figure prior to

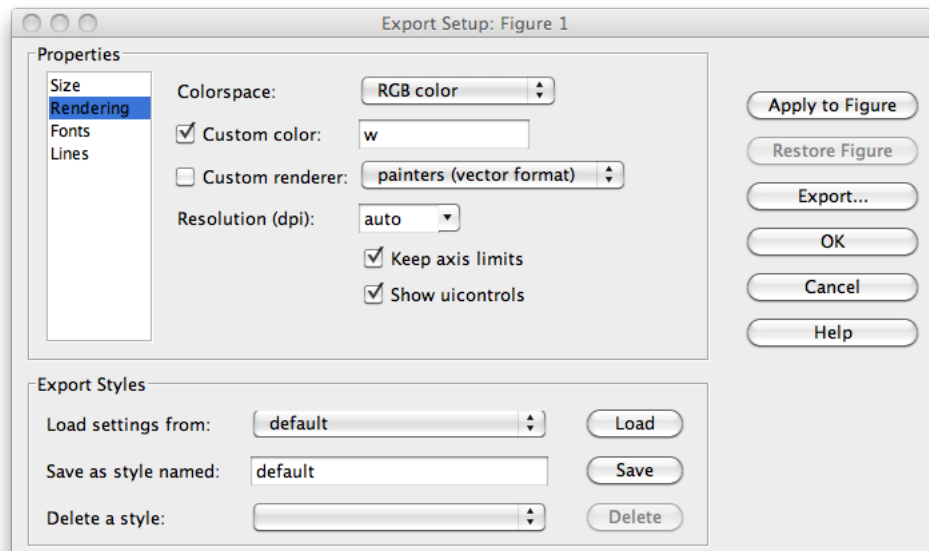
exporting it. You select each of these dialog boxes by clicking **Size**, **Rendering**, **Fonts**, or **Lines** from the **Properties** list.

Adjusting the figure size Click **Size** in the Export Setup dialog box to display this dialog box.



The **Size** dialog box modifies the size of the figure as it will appear when imported from the export file into your application. If you leave the **Width** and **Height** settings on *auto*, the figure remains the same size as it appears on your screen. You can change the size of the figure by entering new values in the **Width** and **Height** text boxes and then clicking **Apply to Figure**. To go back to the original settings, click **Restore Figure**.

Adjusting the rendering Click **Rendering** in the Export Setup dialog box to display this dialog box.

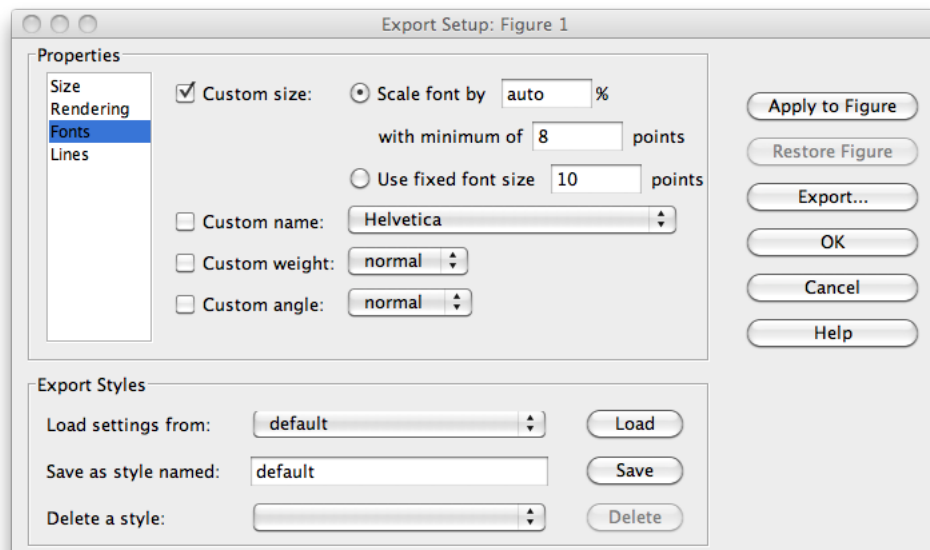


You can change the settings in this dialog box as follows.

- **Colorspace.** Use the drop-down list to select a colorspace. Your choices are:
 - Black and white
 - Grayscale
 - RGB color
 - CMYK color
- **Custom Color.** Click the check box and enter a color to be used for the figure background. Valid entries are:
 - white, yellow, magenta, red, cyan, green, blue, or black
 - Abbreviated name for the same colors — w, y, m, r, c, g, b, k
 - Three-element RGB value Examples: $[1\ 0\ 1]$ is magenta. $[0\ .5\ .4]$ is a dark shade of green.
- **Custom Renderer.** Click the check box and select a renderer from the drop-down list:
 - painters (vector format)
 - OpenGL (bitmap format)
 - Z-buffer (bitmap format)
- **Resolution.** You can select one of the following from the drop-down list:

- Screen — The same resolution as used on your screen display
 - A specific numeric setting — 150, 300, or 600 dpi
 - auto — UNIX selects a suitable setting
- **Keep axis limits.** Click the check box to keep axis tick marks and limits as shown. If unchecked, automatically adjust depending on figure size.
 - **Show uicontrols.** Click the check box to show all user interface controls in the figure. If unchecked, hide user interface controls.

Changing font characteristics Click **Fonts** in the Export Setup dialog box to display this dialog box.

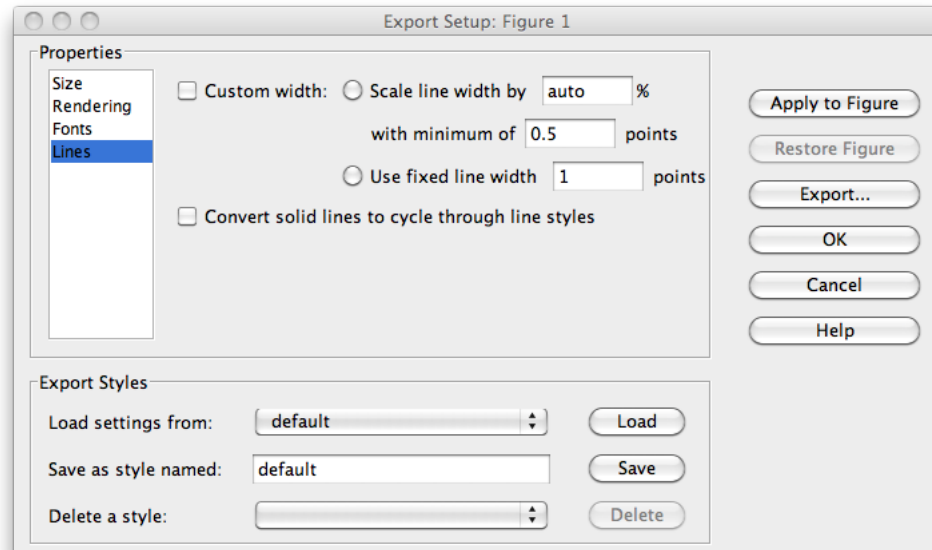


You can change the settings in this dialog box as follows.

- **Custom Size.** Click the check box and use the radio buttons to select a relative or absolute font size for text in the figure.
 - Scale font by N % — Increases or decreases the size of all fonts by a relative amount, N percent. Enter the word *auto* to automatically select the appropriate font size.
 - With minimum of N points — You can specify a minimum font size when scaling the font by a percentage.
 - Use fixed font size N points — Sets the size of all fonts to an absolute value, N points.
- **Custom Name.** Click the check box and use the drop-down list to select a font name from those offered in the drop-down list.

- **Custom Weight.** Click the check box and use the drop-down list to select the weight or thickness to be applied to text in the figure. Choose from *normal*, *light*, *demi*, or *bold*.
- **Custom Angle.** Click the check box and use the drop-down list to select the angle to be applied to text in the figure. Choose from *normal*, *italic*, or *oblique*.

Changing line characteristics Click **Lines** in the Export Setup dialog box to display this dialog box.



You can change the settings in this dialog box as follows.

- **Custom width.** Click the check box and use the radio buttons to select a relative or absolute line size for the figure.
 - Scale line width by N % — Increases or decreases the width of all lines by a relative amount, N percent. Enter the word *auto* to automatically select the appropriate line width.
 - With minimum of N points — Specify a minimum line width when scaling the font by a percentage.
 - Use fixed line width N points — Sets the width of all lines to an absolute value, N points.

Convert solid lines to cycle through line styles. When colored graphics are imported into an application that does not support color, lines that could formerly be distinguished by unique color are likely to appear the same. For example, a red line that shows an input level and a blue line showing output

both appear as black when imported into an application that does not support colored graphics. Clicking this check box causes exported lines to have different line styles, such as solid, dotted, or dashed lines rather than differentiating between lines based on color.

Saving and Loading Settings If you think you might use these export settings at another time, you can save them now and reload them later. At the bottom of each Export Setup dialog box, there is a panel labeled **Export Styles**. To save your current export styles, type a name into the **Save as style named** text box, and then click **Save**. If you then click the **Load** settings from drop-down list, the name of the style you just saved appears among the choices of export styles you can load. To load a style, select one of the choices from this list and then click **Load**. To delete any style you no longer have use for, select that style name from the **Delete a style** drop-down list and click **Delete**.

Exporting the figure When you finish setting the export style for your figure, you can export the figure to a file by clicking the **Export** button on the right side of any of the four Export Setup dialog boxes. A new window labeled **Save As** opens. Select a folder to save the file in from the Save in list at the top. Select a file type for your file from the **Save as type** drop-down list at the bottom, and then enter a file name in the **File name** text box. Click the **Save** button to export the file.

1.9.2 Command line interface

You can print a MATLAB figure from the command line or from a MATLAB file. Use the *set* function to set the properties that control how the printed figure looks. Use the *print* function to specify the output format and start the print or export operation.

The *set* function changes the values of properties that control the look of a figure and objects within it. These properties are stored with the figure; some are also properties of children such as axes or annotations. When you change one of the properties, the new value is saved with the figure and affects the look of the figure each time you print it until you change the setting again.

To change the print properties of the current figure, the *set* command has the form

```
set(gcf, 'Property1', value1, 'Property2', value2, ...)
```

where *gcf* is a function call that returns the handle of the current figure, and each property value pair consists of a named property followed by the value to which the property is set. For example,

```
set(gcf, 'PaperUnits', 'centimeters', 'PaperType', 'A4', ...)
```

sets the units of measure and the paper size.

The *print* function performs any of the four actions shown in the table below. You control what action is taken, depending on the presence or absence of certain arguments.

Action	Print command
Print a figure to a printer	<code>print</code>
Print a figure to a file for later printing	<code>print filename</code>
Copy a figure in graphics format to the clipboard	<code>print -dfileformat</code>
Export a figure to a graphics format file that you can later import into an application	<code>print -dfileformat filename</code>

You can also include optional arguments with the *print* command. For example, to export Figure No. 2 to file `spline2d.eps`, with 600 dpi resolution, and using the EPS color graphics format, use

```
print -f2 -r600 -depsc spline2d
```

The functional form of this command is

```
print('-f2', '-r600', '-depsc', 'spline2d');
```

Changing figure properties for printing or exporting The table below shows parameters that you can set before submitting your figure to the printer. The print Command or set Property column shows how to set the parameter using the MATLAB *print* or *set* function. When using *print*, the table shows the appropriate command option (for example, *print -loose*). When using *set*, it shows the property name to set along with the type of object (for example, (Line) for line objects).

Parameter	<i>Default</i>	<i>print</i> Command or <i>set</i> Property
Select figure	Last active window	<code>print -fhandle</code>
Select printer	System default	<code>print -pprinter</code>
Figure size	8-by-6 inches	PaperSize (Figure), PaperUnits (Figure)
Position on page	0.25 in. from left, 2.5 in. from bottom	PaperPosition (Figure), PaperUnits (Figure)
Position mode	Manual	PaperPositionMode (Figure)
Paper type	Letter	PaperType (Figure)
Paper orientation	Portrait	PaperOrientation (Figure)
Renderer	Selected automatically	<code>print -zbuffer</code> <code>-painters</code> <code>-opengl</code>
Renderer mode	Auto	RendererMode (Figure)
Resolution	Depends on driver or graphics format	<code>print -rresolution</code>
Axes tick marks	Recompute	XTickMode, etc. (Axes)
Background color	Force to white	Color (Figure), InvertHardCopy (Figure)
Font size	As in the figure	FontSize (Text)
Bold font	Regular font	FontWeight (Text)
Line width	As in the figure	LineWidth (Line)
Line style	Black or white	LineStyle (Line)
Line and text color	Black and white	Color (Line, Text)
CMYK color	RGB color	<code>print -cmyk</code>
UI controls	Printed	<code>print -noui</code>
Bounding box	Tight	<code>print -loose</code>
Copy background	Transparent	See “Background color”
Copy size	Same as screen size	See “Figure Size”

2 Creating Graphical User Interfaces

2.1 References

- http://www.mathworks.com/access/helpdesk/help/techdoc/creating_guis/bqz79mu.html
- http://www.mathworks.com/access/helpdesk/help/pdf_doc/matlab/buildgui.pdf

2.2 What is a GUI?

A graphical user interface (GUI) is a graphical display in one or more windows containing controls, called components, that enable a user to perform interactive tasks. The user of the GUI does not have to create a script or type commands at the command line to accomplish the tasks. Unlike coding programs to accomplish tasks, the user of a GUI need not understand the details of how the tasks are performed. GUI components can include menus, toolbars, push buttons,

radio buttons, list boxes, and sliders—just to name a few. GUIs created using MATLAB tools can also perform any type of computation, read and write data files, communicate with other GUIs, and display data as tables or as plots.

Most GUIs wait for their user to manipulate a control, and then respond to each action in turn. Each control, and the GUI itself, has one or more user-written routines (executable MATLAB code) known as *callbacks*, named for the fact that they “call back” to MATLAB to ask it to do things. The execution of each callback is triggered by a particular user action such as pressing a screen button, clicking a mouse button, selecting a menu item, typing a string or a numeric value, or passing the cursor over a component. The GUI then responds to these events. You, as the creator of the GUI, provide callbacks which define what the components do to handle events. This kind of programming is often referred to as *event-driven programming*. In the example, a button click is one such event. In event-driven programming, callback execution is *asynchronous*, that is, it is triggered by events external to the software. In the case of MATLAB GUIs, most events are user interactions with the GUI, but the GUI can respond to other kinds of events as well, for example, the creation of a file or connecting a device to the computer.

2.3 Ways to build MATLAB GUIs

A MATLAB GUI is a figure window to which you add user-operated controls. You can select, size, and position these components as you like. Using callbacks you can make the components do what you want when the user clicks or manipulates them with keystrokes. You can build MATLAB GUIs in two ways:

- Use GUIDE (GUI Development Environment), an interactive GUI construction kit.
- Create code files that generate GUIs as functions or scripts (programmatic GUI construction).

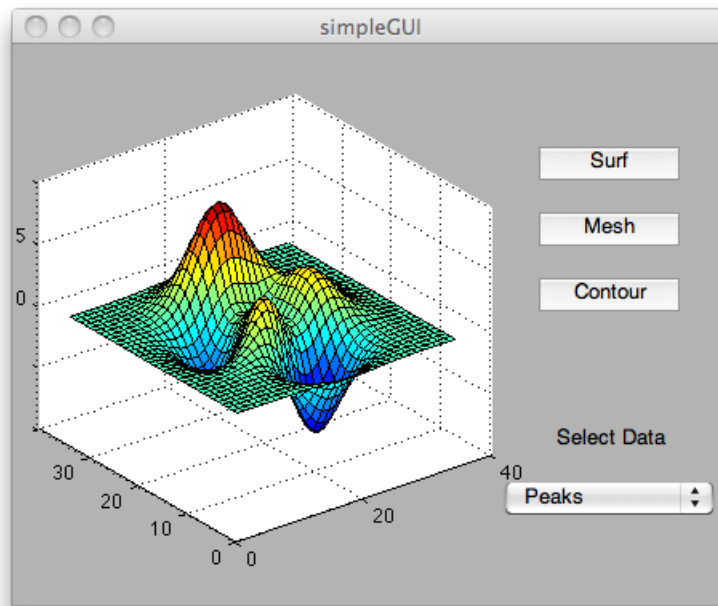
The first approach starts with a figure that you populate with components from within a graphic layout editor. GUIDE creates an associated code file containing callbacks for the GUI and its components. GUIDE saves both the figure (as a FIG-file) and the code file. Opening either one also opens the other to run the GUI.

In the second, *programmatic*, GUI-building approach, you create a code file that defines all component properties and behaviors; when a user executes the file, it creates a figure, populates it with components, and handles user interactions. The figure is not normally saved between sessions because the code in the file creates a new one each time it runs. As a result, the code files of the two approaches look different.

Programmatic GUI files are generally longer, because they explicitly define every property of the figure and its controls, as well as the callbacks. GUIDE GUIs define most of the properties within the figure itself. They store the definitions in its FIG-file rather than in its code file. The code file contains callbacks and other functions that initialize the GUI when it opens.

2.4 Creating a simple GUI with GUIDE

This section shows you how to create the graphical user interface (GUI) shown in the following figure. using GUIDE.



The GUI contains:

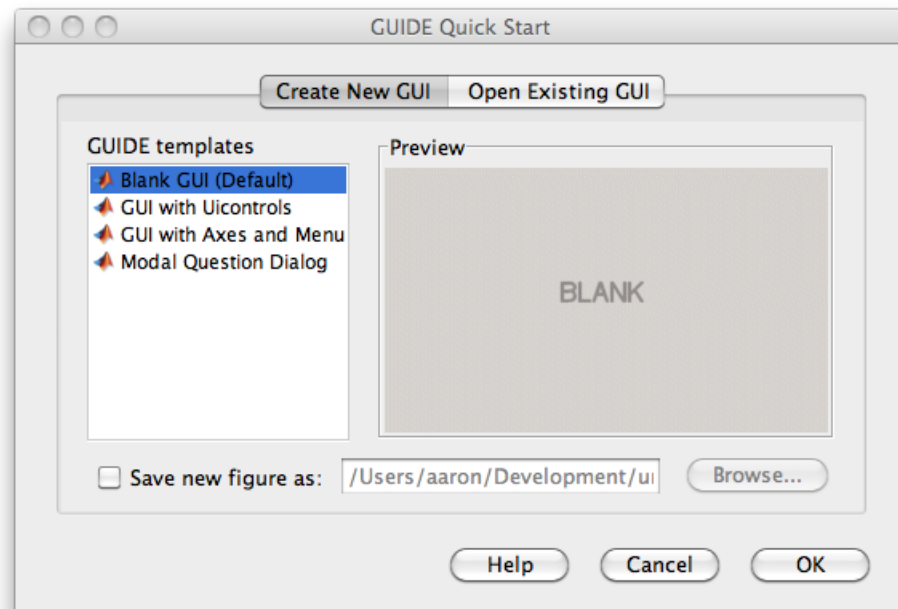
- An axes component
- A pop-up menu listing three different data sets that correspond to MATLAB functions: *peaks*, *membrane*, and *sinc*
- A static text component to label the pop-up menu
- Three push buttons, each of which displays a different type of plot: *surface*, *mesh*, and *contour*

2.4.1 Laying out the GUI with GUIDE

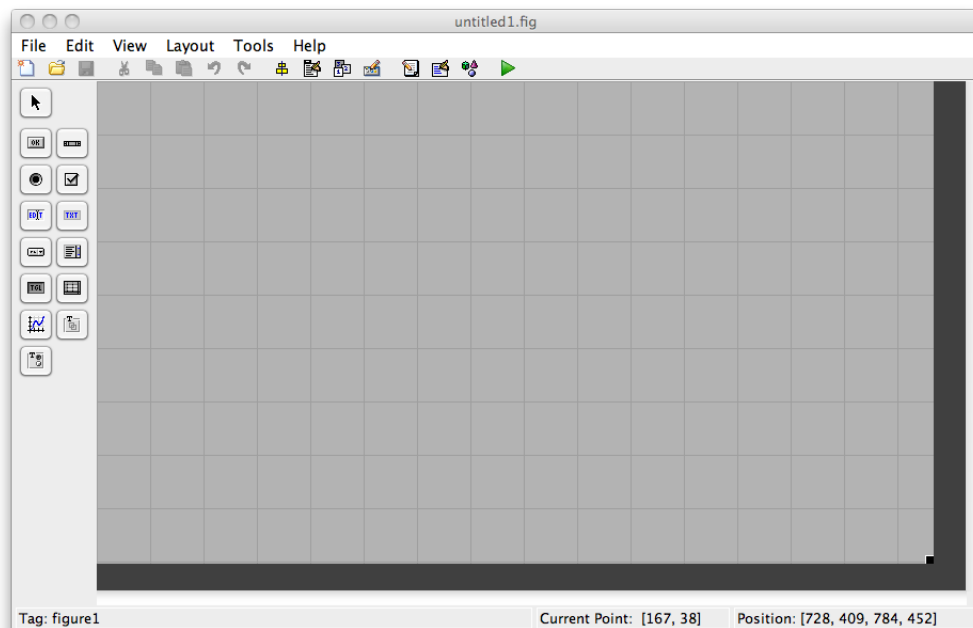
- Start GUIDE by typing:

```
>> guide
```

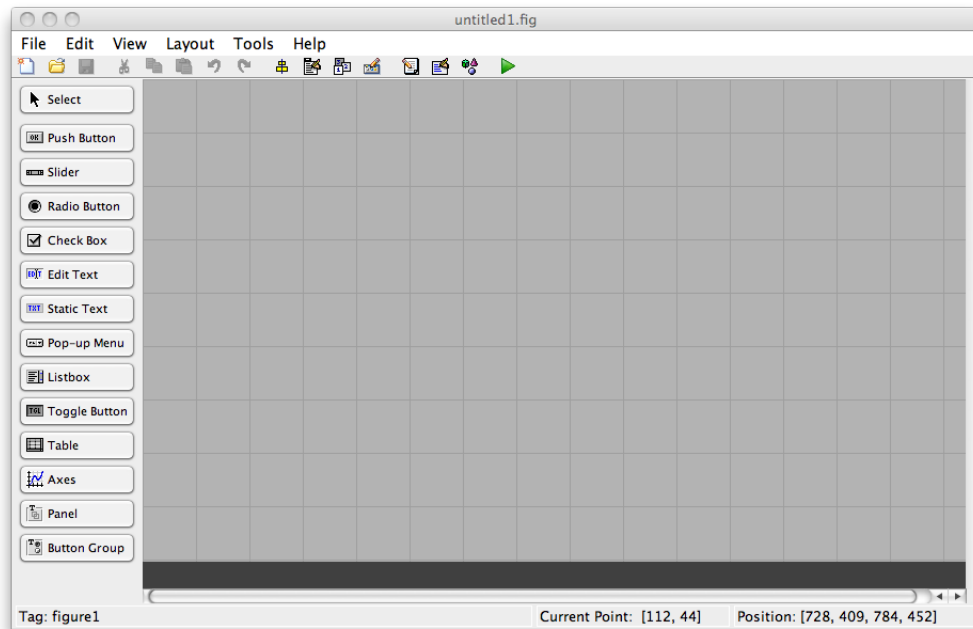
at the MATLAB prompt. The GUIDE Quick Start dialog displays, as shown in the following figure.



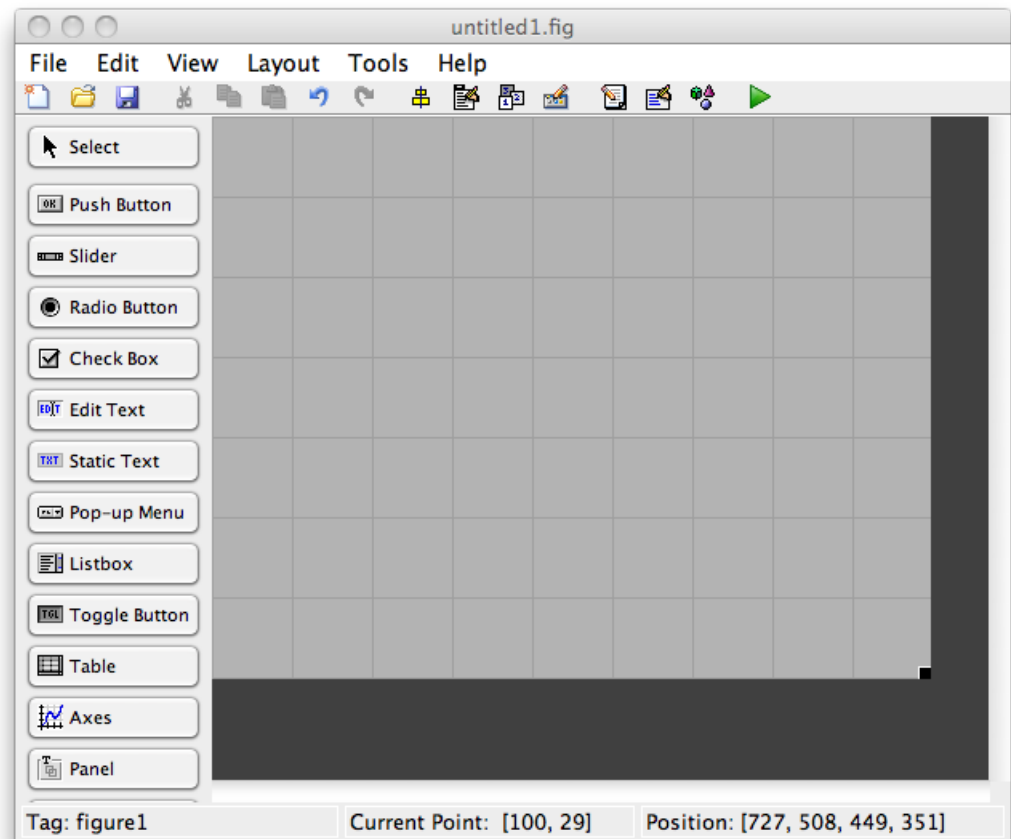
- In the Quick Start dialog, select the *Blank GUI (Default)* template. Click OK to display the blank GUI in the Layout Editor, as shown in the following figure.



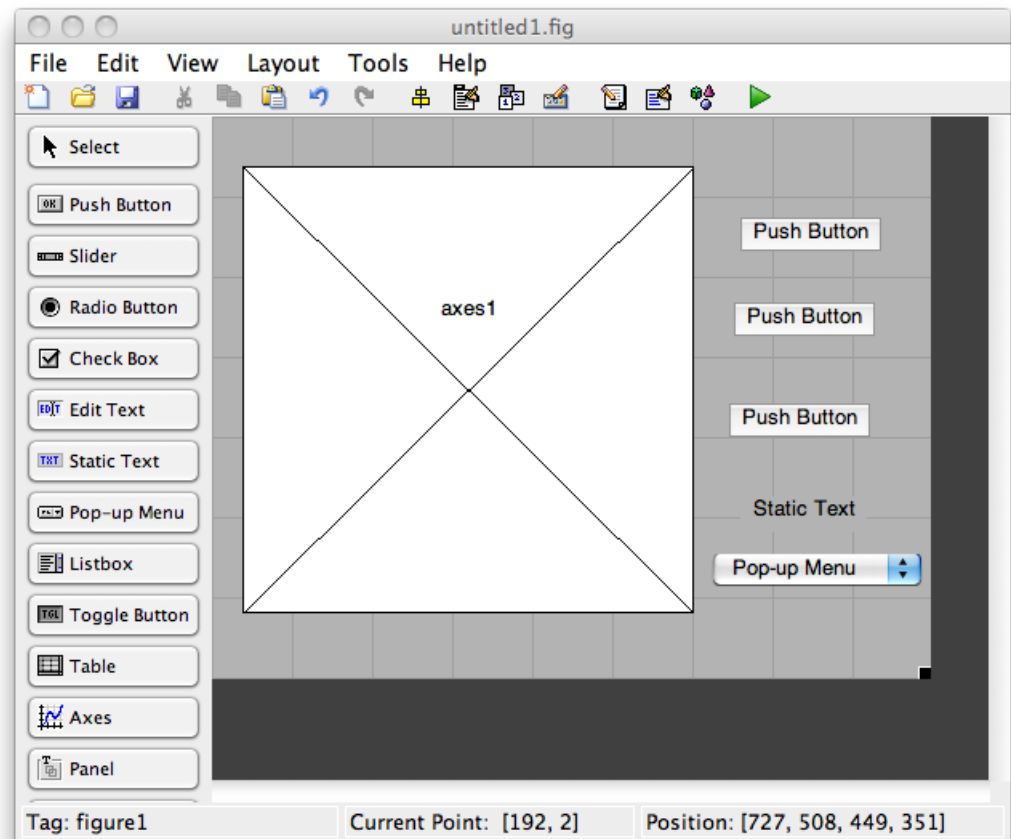
- Display the names of the GUI components in the component palette. Select **Preferences** from the MATLAB File menu. Then select **GUIDE** > **Show names in component palette**, and click OK. The Layout Editor then appears as shown in the following figure.



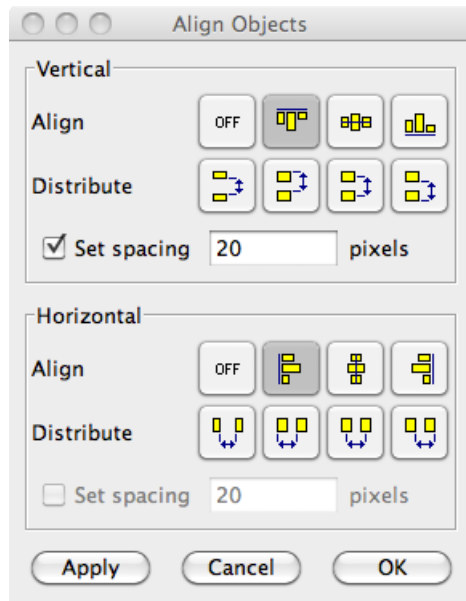
- Set the size of the GUI by resizing the grid area in the Layout Editor. Click the lower-right corner and drag it until the GUI is approximately 3 inches high and 4 inches wide. If necessary, make the window larger.



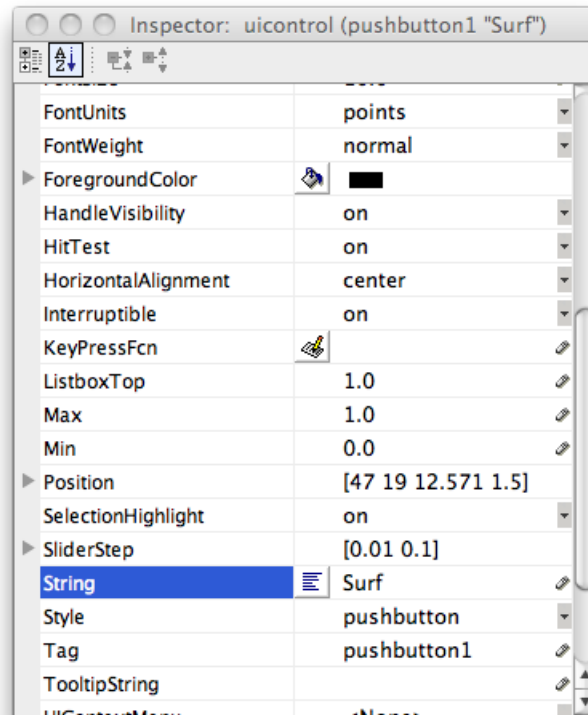
- Add **three push buttons**, a **static text area**, a **pop-up menu**, and an **axes** to the GUI. Select the corresponding entries from the component palette at the left side of the Layout Editor and drag them into the layout area. Position all controls approximately as shown in the following figure.



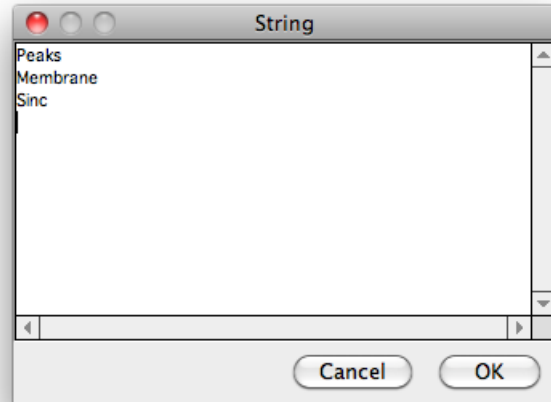
- If several components have the same parent, you can use the **Alignment Tool** to align them to one another. To align the three push buttons:
 1. Select all three push buttons by pressing **Ctrl** and clicking them.
 2. Select **Align Objects** from the **Tools** menu to display the **Alignment Tool**.
 3. Make these settings in the Alignment Tool, as shown in the following figure:
 - 20 pixels spacing between push buttons in the vertical direction.
 - Left-aligned in the horizontal direction.



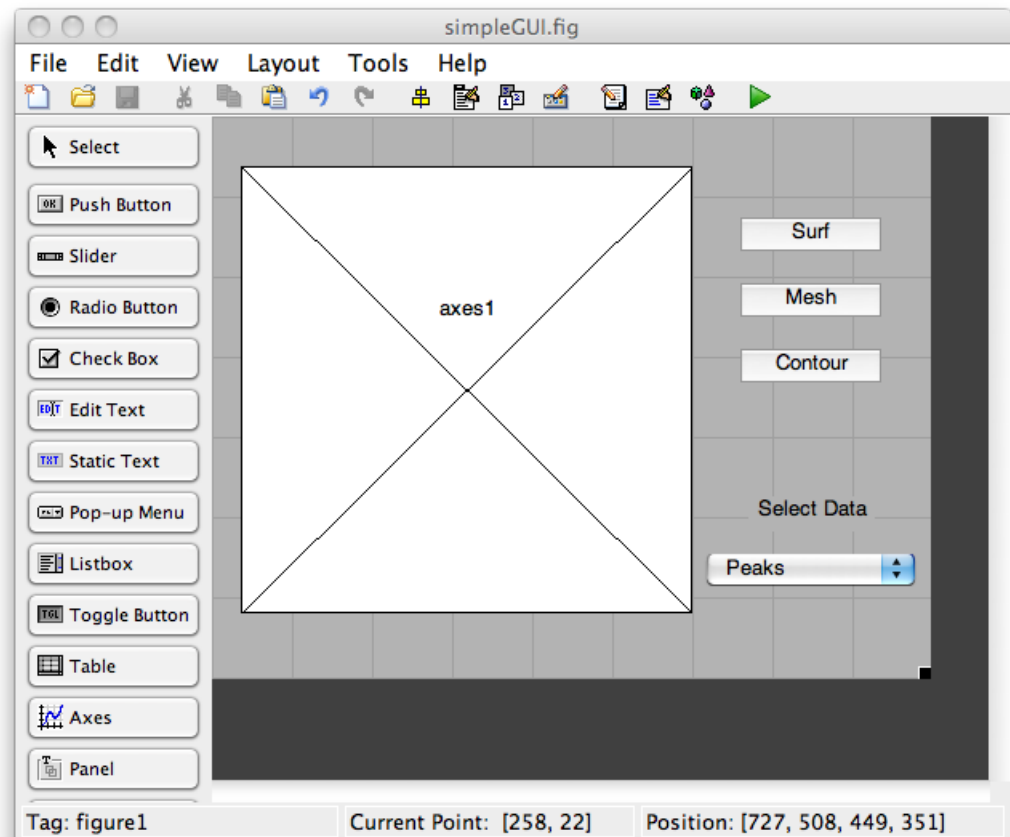
- Use the Alignment Tool to align and distribute all controls in the figure.
- The push buttons, pop-up menu, and static text have default labels when you create them. Their text is generic, for example **Push Button 1**. Change the text to be specific to your GUI, so that it explains what the component is for.
- To change the text of the first Push Button:
 1. Select **Property Inspector** from the **View** menu.
 2. In the layout area, click the Push Button you want to modify.
 3. In the Property Inspector, select the *String* property and then replace the existing value with the word *Surf*.
 4. Select each of the remaining push buttons in turn and repeat steps 2 and 3. Label the middle push button *Mesh*, and the bottom button *Contour*.



- The pop-up menu provides a choice of three data sets: *peaks*, *membrane*, and *sinc*. These data sets correspond to MATLAB functions of the same name. To list those data sets as choices in the pop-menu:
 1. In the layout area, select the pop-up menu by clicking it.
 2. In the Property Inspector, click the button next to String. The String dialog box displays.
 3. Replace the existing text with the names of the three data sets: *Peaks*, *Membrane*, and *Sinc*. Press Enter to move to the next line.
 4. When you have finished editing the items, click OK. The first item in your list, Peaks, appears in the pop-up menu in the layout area.



- In this GUI, the static text serves as a label for the pop-up menu. The user cannot change this text. This topic shows you how to change the static text to read *Select Data*.
 1. In the layout area, select the static text by clicking it.
 2. In the Property Inspector, click the button next to String. In the String dialog box that displays, replace the existing text with the phrase *Select Data*.
 3. Click OK. The phrase *Select Data* appears in the static text component above the pop-up menu.
- In the Layout Editor, your GUI now looks like in the following figure, and the next step is to save the layout.



- When you save a GUI, GUIDE creates two files, a FIG-file and a code file. The FIG-file, with extension .fig, is a binary file that contains a description of the layout. The code file, with extension .m, contains MATLAB functions that control the GUI.
- To save a GUI you can either choose **Save as...** from the **File** menu or **Run** it (from the **Tools** menu). If you try to run an unsaved Figure, GUIDE will ask you to save it first. Save the Figure as **simpleGUI.fig** (in the current path): GUIDE saves the files simpleGUI.fig and simpleGUI.m. The latter file is opened in the Editor.
- Run the figure by choosing Tools > Run or from the MATLAB console, by typing:

```
>> simpleGUI
```
- Now the GUI is designed, but it does not do anything!

2.4.2 Adding code to the GUI

When you saved your GUI in the previous section, GUIDE created two files: a FIG-file `simpleGUI.fig` that contains the GUI layout and a file, `simpleGUI.m`, that contains the code that controls how the GUI behaves. The code consists of a set of MATLAB functions (that is, it is not a script). But the GUI did not respond because the functions contain no statements that perform actions yet.

Here we will see how to add code to the file to make the GUI do things.

Generating data to plot In this section we will learn how to generate the data to be plotted when the GUI user clicks a button. The opening function generates this data by calling MATLAB functions. The **opening function**, which initializes a GUI when it opens, is the first callback in every GUIDE-generated GUI code file. In this example, you add code that creates three data sets to the opening function. The code uses the MATLAB functions `peaks`, `membrane`, and `sinc`. Open `simpleGUI.m` in the Editor:

```
>> edit simpleGUI
```

Scroll to the `simpleGUI_OpeningFcn` function (or choose it in the Show functions `f()` tool in the editor and add this code to create data for the GUI immediately after the comment that begins with

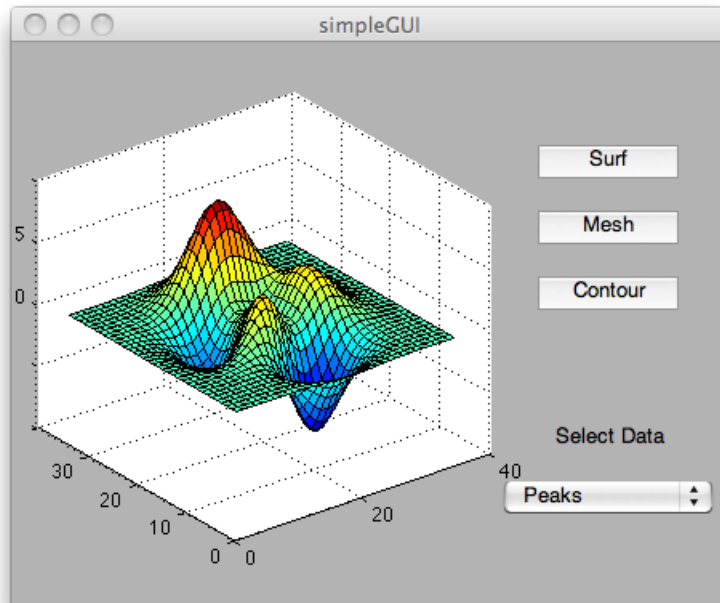
```
% varargin    command line arguments to simpleGUI (see VARARGIN)

% Create the data to plot.
handles.peaks=peaks(35);
handles.membrane=membrane;
[x,y] = meshgrid(-8:.5:8);
r = sqrt(x.^2+y.^2) + eps;
sinc = sin(r)./r;
handles.sinc = sinc;
% Set the current data value.
handles.current_data = handles.peaks;
surf(handles.current_data)
```

Make sure to leave the already existing code after the lines you added.

The first six executable lines create the data using the MATLAB functions `peaks`, `membrane`, and `sinc`. They store the data in the *handles* structure, an argument provided to all callbacks. Callbacks for the push buttons can retrieve the data from the handles structure.

The last two lines create a current data value and set it to `peaks`, and then display the surf plot for `peaks`. The following figure shows how the GUI now looks when it first displays.



Programming the pop-up menu The pop-up menu enables the user to select the data to plot. When the GUI user selects one of the three plots, MATLAB sets the pop-up menu *Value* property to the index of the selected string. The pop-up menu callback reads the pop-up menu *Value* property to determine the item that the menu currently displays, and sets *handles.current_data* accordingly.

- To program the pop-up menu callback, right-click on the pop-up menu in GUIDE and select **View Callbacks > Callback**.
- This will display the *popupmenu1_Callback()* function in the Editor.
- Add the following code to the *popupmenu1_Callback()* after the comments. This code first retrieves two pop-up menu properties:

- String — a cell array that contains the menu contents
- Value — the index into the menu contents of the selected data set

It then uses a switch statement to make the selected data set the current data. The last statement saves the changes to the handles structure.

```
% Determine the selected data set.
str = get(hObject, 'String');
val = get(hObject, 'Value');
% Set current data to the selected data set.
```

```

switch str{val};
    case 'Peaks' % User selects peaks.
        handles.current_data = handles.peaks;
    case 'Membrane' % User selects membrane.
        handles.current_data = handles.membrane;
    case 'Sinc' % User selects sinc.
        handles.current_data = handles.sinc;
end
% Save the handles structure.
guidata(hObject,handles)

```

Programming the push buttons Each of the push buttons creates a different type of plot using the data specified by the current selection in the pop-up menu. The push button callbacks get data from the handles structure and then plot it.

- Right-click the *Surf* push button in the Layout Editor to display a context menu. From that menu, select **View Callbacks > Callback**.
- This will display the *pushbutton1_Callback()* function in the Editor.
- Add the following code to the *pushbutton1_Callback()* after the comments:

```

% Display surf plot of the currently selected data.
surf(handles.current_data);

```

- Repeat previous steps for the *Mesh* push button and add this code to the *pushbutton2_Callback()*:

```

% Display mesh plot of the currently selected data.
mesh(handles.current_data);

```

- Repeat previous steps for the *Contour* push button and add this code to the *pushbutton3_Callback()*:

```

% Display contour plot of the currently selected data.
contour(handles.current_data);

```

- Save the simpleGUI.m file.

Running the GUI Run the GUI from the MATLAB console.

```
>> simpleGUI;
```

2.5 Creating a simple GUI programmatically

In this section we will create the same GUI as in the previous one, but without using GUIDE.

2.5.1 Creating a GUI code file

We will start by creating the backbone of our M-file. In the MATLAB console type:

```
>> edit simpleGUI2
```

and add the following code in the file:

```
function simpleGUI2
% SIMPLEGUI2 Select a data set from the pop-up menu, then
% click one of the plot-type push buttons. Clicking the button
% plots the selected data in the axes.
end
```

Save the file in the current path.

2.5.2 Laying out a simple GUI

Creating the figure In MATLAB, a GUI is a figure. This first step creates the figure and positions it on the screen. It also makes the GUI invisible so that the GUI user cannot see the components being added or initialized. When the GUI has all its components and is initialized, the example makes it visible.

```
% Initialize and hide the GUI as it is being constructed.
f = figure('Visible','off','MenuBar','None','Position',[360,500,450,285]);
```

The call to the figure function uses two *property/value* pairs. The *Position* property is a four-element vector that specifies the location of the GUI on the screen and its size: *[distance from left, distance from bottom, width, height]*. Add the previous code right after the initial comments in simpleGUI2.m.

Adding the components The example GUI has six components: three push buttons, one static text, one pop-up menu, and one axes. Start by writing statements that add these components to the GUI. Create the push buttons, static text, and pop-up menu with the `uicontrol` function. Use the `axes` function to create the axes.

- Add the three push buttons to your GUI by adding these statements to your code file following the call to figure.

```
% Construct the components.
hsurf = uicontrol('Style','pushbutton',...
    'String','Surf','Position',[315,220,70,25]);
hmesh = uicontrol('Style','pushbutton',...
    'String','Mesh','Position',[315,180,70,25]);
hcontour = uicontrol('Style','pushbutton',...
    'String','Countour','Position',[315,135,70,25]);
```

These statements use the *uicontrol* function to create the push buttons. Each statement uses a series of *property/value* pairs to define a push button:

- Style — In the example, pushbutton specifies the component as a push button.
- String — Specifies the label that appears on each push button. Here, there are three types of plots: *Surf*, *Mesh*, *Contour*.
- Position — Uses a four-element vector to specify the location of each push button within the GUI and its size: *[distance from left, distance from bottom, width, height]*. Default units for push buttons are pixels.

Each call returns the handle of the component that is created (*hsurf*, *hmesh*, *hcontour*).

- Add the pop-up menu and its label to your GUI by adding these statements to the code file following the push button definitions.

```
hpopup = uicontrol('Style','popupmenu',...
    'String',{'Peaks','Membrane','Sinc'},...
    'Position',[300,50,100,25]);
htext = uicontrol('Style','text','String','Select Data',...
    'Position',[325,90,60,15]);
```

For the pop-up menu, the *String* property uses a cell array to specify the three items in the pop-up menu: *Peaks*, *Membrane*, *Sinc*. The static text component serves as a label for the pop-up menu. Its *String* property tells the GUI user to *Select Data*. Default units for these components are pixels.

- Add the axes to the GUI by adding this statement to the code file. Set the Units property to pixels so that it has the same units as the other components.

```
ha = axes('Units','pixels','Position',[50,60,200,185]);
```

- Align all components except the axes along their centers with the following statement. Add it to the code file following all the component definitions.

```
align([hsurf,hmesh,hcontour,htext,hpopup],'Center','None');
```

- Make your GUI visible by adding this command following the align command.

```
set(f,'Visible','on')
```


- This is what your code file should now look like:

```
function simpleGUI2
% SIMPLEGUI2 Select a data set from the pop-up menu, then
% click one of the plot-type push buttons. Clicking the button
% plots the selected data in the axes.

% Create and hide the GUI as it is being constructed.
f = figure('Visible','off','MenuBar','None','Position',[360,500,450,285]);

% Construct the components.
hsurf = uicontrol('Style','pushbutton','String','Surf',...
    'Position',[315,220,70,25]);
hmesh = uicontrol('Style','pushbutton','String','Mesh',...
    'Position',[315,180,70,25]);
hcontour = uicontrol('Style','pushbutton',...
    'String','Countour',...
    'Position',[315,135,70,25]);
htext = uicontrol('Style','text','String','Select Data',...
    'Position',[325,90,60,15]);
hpopup = uicontrol('Style','popupmenu',...
    'String',{'Peaks','Membrane','Sinc'},...
    'Position',[300,50,100,25]);
ha = axes('Units','Pixels','Position',[50,60,200,185]);
align([hsurf,hmesh,hcontour,htext,hpopup],'Center','None');

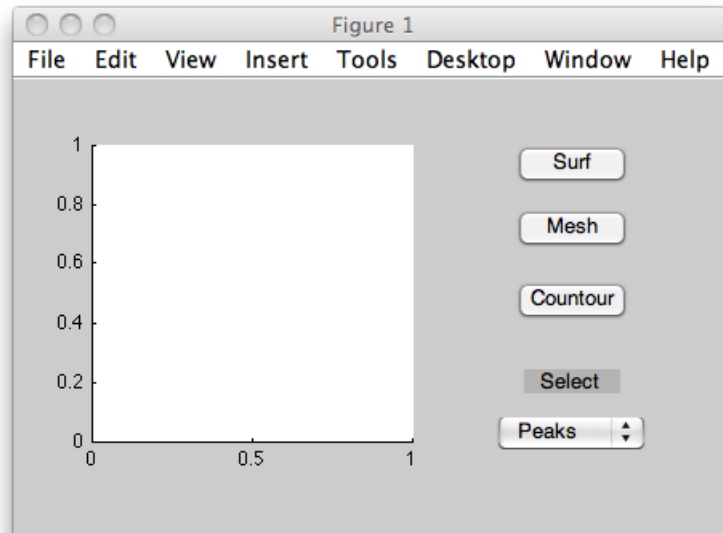
% Make the GUI visible.
set(f,'Visible','on')

end
```

- Run your code by typing

```
>> simpleGUI2
```

at the command line. This is what your GUI now looks like. Note that you can select a data set in the pop-up menu and click the push buttons. But nothing happens. This is because there is no code in the file to service the pop-up menu or the buttons.



Initializing the GUI When you make the GUI visible, it should be initialized so that it is ready for the user. This section shows you how to

1. Make the GUI behave properly when it is resized by changing the component and figure units to normalized. This causes the components to resize when the GUI is resized. Normalized units map the lower-left corner of the figure window to (0,0) and the upper-right corner to (1.0, 1.0).
2. Generate the data to plot. The example needs three sets of data: `peaks_data`, `membrane_data`, and `sinc_data`. Each set corresponds to one of the items in the pop-up menu.
3. Create an initial plot in the axes
4. Assign the GUI a name that appears in the window title
5. Move the GUI to the center of the screen
6. Make the GUI visible

Replace this code in editor:

```
% Make the GUI visible.
set(f,'Visible','on')
```

with this code:

```
% Initialize the GUI.
```

```
% Change units to normalized so components resize automatically.
set([f,hsurf,hmesh,hcontour,htext,hpopup],'Units','normalized');

% Generate the data to plot.
peaks_data = peaks(35);
membrane_data = membrane;
[x,y] = meshgrid(-8:.5:8);
r = sqrt(x.^2+y.^2) + eps;
sinc_data = sin(r)./r;

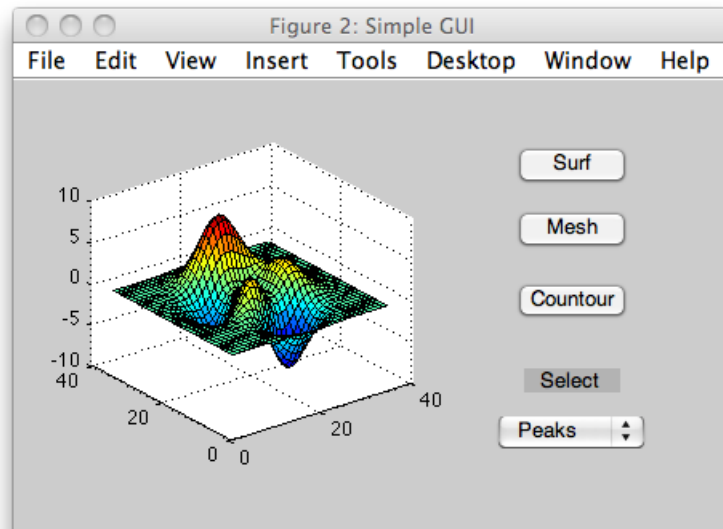
% Create a plot in the axes.
current_data = peaks_data;
surf(current_data);

% Assign the GUI a name to appear in the window title.
set(f,'Name','Simple GUI')

% Move the GUI to the center of the screen.
movegui(f,'center')

% Make the GUI visible.
set(f,'Visible','on');
```

- Run your code by typing simpleGUI2 at the command line. The initialization above cause it to display the default peaks data with the surf function, making the GUI look like this.



Programming the pop-up menu The pop-up menu enables users to select the data to plot. When a GUI user selects one of the three data sets, MATLAB sets the pop-up menu *Value* property to the index of the selected string. The pop-up menu callback reads the pop-up menu *Value* property to determine which item is currently displayed and sets *current_data* accordingly. Add the following callback to your file following the initialization code and before the final end statement.

```
% Pop-up menu callback. Read the pop-up menu Value property to
% determine which item is currently displayed and make it the
% current data. This callback automatically has access to
% current_data because this function is nested at a lower level.
function popup_menu_Callback(source,eventdata)
    % Determine the selected data set.
    str = get(source, 'String');
    val = get(source, 'Value');
    % Set current data to the selected data set.
    switch str{val};
        case 'Peaks' % User selects Peaks.
            current_data = peaks_data;
        case 'Membrane' % User selects Membrane.
            current_data = membrane_data;
        case 'Sinc' % User selects Sinc.
            current_data = sinc_data;
    end
end
```

Programming the push buttons Each of the three push buttons creates a different type of plot using the data specified by the current selection in the pop-up menu. The push button callbacks plot the data in *current_data*. They automatically have access to *current_data* because they are nested at a lower level. Add the following callbacks to your file following the pop-up menu callback and before the final end statement.

```
% Push button callbacks. Each callback plots current_data in the
% specified plot type.
function surfbutton_Callback(source,eventdata)
    % Display surf plot of the currently selected data.
    surf(current_data);
end
function meshbutton_Callback(source,eventdata)
    % Display mesh plot of the currently selected data.
    mesh(current_data);
end
```

```
function contourbutton_Callback(source,eventdata)
    % Display contour plot of the currently selected data.
    contour(current_data);
end
```

Associating callbacks to their components When the GUI user selects a data set from the pop-up menu or clicks one of the push buttons, MATLAB software executes the callback associated with that particular event. But how does the software know which callback to execute? You must use each component's *Callback* property to specify the name of the callback with which it is associated.

- To the *uicontrol* statement that defines the *Surf* push button, add the *property/value* pair

```
'Callback',{@surfbutton_Callback}
```

so that the statement looks like this:

```
hsurf = uicontrol('Style','pushbutton','String','Surf',...
    'Position',[315,220,70,25],...
    'Callback',{@surfbutton_Callback});
```

Callback is the name of the *property*. *surfbutton_Callback* is the name of the callback that services the Surf push button.

- Similarly, to the *uicontrol* statement that defines the *Mesh* push button, add the *property/value* pair

```
'Callback',{@meshbutton_Callback}
```

- To the *uicontrol* statement that defines the *Contour* push button, add the *property/value* pair

```
'Callback',{@contourbutton_Callback}
```

- To the *uicontrol* statement that defines the pop-up menu, add the *property/value* pair

```
'Callback',{@popup_menu_Callback}
```

Running the GUI Run the GUI by typing the name of the code file at the command line.

```
>> simpleGUI2
```

3 Object-oriented programming

3.1 References

- http://www.mathworks.com/access/helpdesk/help/techdoc/matlab_oop/ug_intropage.html
- http://www.mathworks.com/access/helpdesk/help/pdf_doc/matlab/matlab_oop.pdf

3.2 Introduction to OOP

Creating software applications typically involves designing how to represent the application data and determining how to implement operations performed on that data. **Procedural programs** pass data to functions, which perform the necessary operations on the data. **Object-oriented software** encapsulates data and operations in objects that interact with each other via the object's interface.

The MATLAB language enables you to create programs using both procedural and object-oriented techniques and to use objects and ordinary functions in your programs.

Procedural Program Design In procedural programming, your design focuses on steps that must be executed to achieve a desired state. You typically represent data as individual variables or fields of a structure and implement operations as functions that take the variables as arguments. Programs usually call a sequence of functions, each one of which is passed data, and then returns modified data. Each function performs an operation or perhaps many operations on the data.

Object-Oriented Program Design The object-oriented program design involves:

- Identifying the components of the system or application that you want to build
- Analyzing and identifying patterns to determine what components are used repeatedly or share characteristics
- Classifying components based on similarities and differences

After performing this analysis, you define classes that describe the objects your application uses.

Classes and Objects A class describes a set of objects with common characteristics. Objects are specific instances of a class. The values contained in an object's properties are what make an object different from other objects of the same class (an object of class *double* might have a value of 5). The functions defined by the class (called *methods*) are what implement object behaviors that are common to all objects of a class (you can add two doubles regardless of their values).

3.3 Classes in MATLAB

In the MATLAB language, every value is assigned to a **class**. For example, creating a variable with an assignment statement constructs a variable of the appropriate class:

```
>> a = 7;
>> b = 'some string';
>> whos
```

Name	Size	Bytes	Class
a	1x1	8	double
b	1x11	22	char

Basic commands like *whos* display the class of each value in the workspace. This information helps MATLAB users recognize that some values are characters and display as text while other values might be double, single, or other types of numbers. Some variables can contain different classes of values like *cells*.

3.4 User-defined classes

You can create your own MATLAB classes. For example, you could define a class to represent polynomials. This class could define the operations typically associated with MATLAB classes, like addition, subtraction, indexing, displaying in the command window, and so on. However, these operations would need to perform the equivalent of polynomial addition, polynomial subtraction, and so on. For example, when you add two polynomial objects:

$p1 + p2$

the *plus* operation would know how to add polynomial objects because the polynomial class defines this operation. When you define a class, you overload special MATLAB functions (*plus.m* for the addition operator) that are called by the MATLAB runtime when those operations are applied to an object of your class.

3.5 MATLAB classes - key terms

MATLAB classes use the following words to describe different parts of a class definition and related concepts.

- Class definition — Description of what is common to every instance of a class.
- Properties — Data storage for class instances
- Methods — Special functions that implement operations that are usually performed only on instances of the class

- Events — Messages that are defined by classes and broadcast by class instances when some specific action occurs
- Attributes — Values that modify the behavior of properties, methods, events, and classes
- Listeners — Objects that respond to a specific event by executing a callback function when the event notice is broadcast
- Objects — Instances of classes, which contain actual data values stored in the objects' properties
- Subclasses — Classes that are derived from other classes and that inherit the methods, properties, and events from those classes (subclasses facilitate the reuse of code defined in the superclass from which they are derived).
- Superclasses — Classes that are used as a basis for the creation of more specifically defined classes (i.e., subclasses).
- Packages — Folders that define a scope for class and function naming

3.6 Handle vs. value classes

There are two kinds of MATLAB classes—handle classes and value classes.

- Handle classes create objects that reference the data contained. When you copy a handle object, MATLAB copies the handle, but not the data stored in the object properties. The copy refers to the same data as the original handle. If you change a property value on the original object, the copied object reflects the same change.
- Value classes make copies of the data whenever the object is copied or passed to a function. MATLAB numeric types are value classes.

The kind of class you use depends on the desired behavior of the class instances and what features you want to use. Use a handle class when you want to create a reference to the data contained in an object of the class, and do not want copies of the object to make copies of the object data. For example, use a handle class to implement an object that contains information for a phone book entry. Multiple application programs can access a particular phone book entry, but there can be only one set of underlying data. The reference behavior of handles enables these classes to support features like events, listeners, and dynamic properties. Use value classes to represent entities that do not need to be unique, like numeric values. For example, use a value class to implement a polynomial data type. You can copy a polynomial object and then modify its coefficients to make a different polynomial without affecting the original polynomial.

3.7 Class folders

There are two basic ways to specify classes with respect to folders:

- Creating a **single, self-contained class definition file** in a folder on the MATLAB path. The name of the file must match the class (and constructor) name and must have the .m extension. The class is defined entirely in this file.
- Distributing a class definition to multiple files in an **@ folder** inside a path folder. Only one class can be defined in a @ folder.

In addition, **package folders** (which always begin with a “+” character) can contain multiple class definitions, package-scoped functions, and other packages. A package folder defines a new name space in which you can reuse class names. Use the package name to refer to classes and functions defined in package folders (for example, *packagefld1.ClassNameA()*, *packagefld2.packageFunction()*).

3.8 Class building blocks

The basic components in the class definition are blocks describing the whole class and specific aspects of its definition.

3.8.1 The classdef block

The classdef block contains the class definition within a file that starts with the classdef keyword and terminates with the end keyword.

```
classdef className
    ...
end
```

The classdef block contains the class definition. The classdef line is where you specify:

- Class attributes
- Class attributes modify class behavior in some way. Assign values to class attributes only when you want to change their default value. No change to default attribute values:

```
classdef class_name
    ...
end
```

One or more attribute values assigned:

```
classdef (attribute1 = value,...) class_name
    ...
end
```

- Superclasses

To define a class in terms of one or more other classes by specifying the superclasses on the `classdef` line:

```
classdef class_name < superclass1_name & superclass2_name
    ...
end
```

A handle class inherits from the class *handle* (i.e. it has the class *handle* as superclass):

```
classdef handle_class_name < handle
    ...
end
```

The `classdef` block contains the properties, methods, and events subblocks.

3.8.2 The properties block

The properties block (one for each unique set of attribute specifications) contains property definitions, including optional initial values. The properties block starts with the `properties` keyword and terminates with the `end` keyword.

```
classdef className
    properties
        ...
    end
    ...
end
```

You can control aspects of property definitions in the following ways:

- Specifying a default value for each property individually
- Assigning attribute values on a per block basis
- Defining methods that execute when the property is set or queried

There are two basic approaches to initializing property values:

- In the property definition — MATLAB evaluates the expression only once and assigns the same value to the property of every instance.

```
classdef class_name
    properties
        PropertyName % No default value assigned
        PropertyName = 'some text';
        PropertyName = sin(pi/12); % Expression returns default value
    end
end
```

Evaluation of property default values occurs only when the value is first needed, and only once when MATLAB first initializes the class. MATLAB does not reevaluate the expression each time you create a class instance.

- In the class constructor — MATLAB evaluates the assignment expression for each instance, which ensures that each instance has a unique value.

To assign values to a property from within the class constructor, reference the object that the constructor returns (the output variable *obj*):

```
classdef MyClass
    properties
        PropertyOne
    end
    methods
        function obj = MyClass(intval)
            obj.PropertyOne = intval;
        end
    end
end
```

All properties have **attributes** that modify certain aspects of the property's behavior. Specified attributes apply to all properties in a particular properties block. For example:

```
classdef class_name
    properties
        PropertyName % No default value assigned
        PropertyName = sin(pi/12); % Expression returns default value
    end
    properties (SetAccess = private, GetAccess = private)
        Stress
        Strain
    end
end
```

In this case, only methods in the same class definition can modify and query the *Stress* and *Strain* properties. This restriction exists because the class defines these properties in a properties block with *SetAccess* and *GetAccess* attributes set to private.

You can define methods that MATLAB calls whenever setting or querying a property value. Define property set access or get access methods in *methods* blocks that specify no attributes and have the following syntax:

```
methods
    function value = get.PropertyName(object)
        ...
    end
```

```

        function obj = set.PropertyName(obj,value)
        ...
    end
end

```

MATLAB does not call the property set access method when assigning the default value specified in the property's definition block. If a handle class defines the property, the set access method does not need to return the modified object.

3.8.3 The methods block

The methods block (one for each unique set of attribute specifications) contains function definitions for the class methods. The methods block starts with the methods keyword and terminates with the end keyword.

```

classdef className
    methods
        ...
    end
    ...
end

```

Methods are functions that implement the operations performed on objects of a class. Methods, along with other class members support the concept of **encapsulation**—class instances contain data in properties and class methods operate on that data. This allows the internal workings of classes to be hidden from code outside of the class, and thereby enabling the class implementation to change without affecting code that is external to the class. Methods have access to private members of their class including other methods and properties. This enables you to hide data and create special interfaces that must be used to access the data stored in objects.

There are specialized kinds of methods that perform certain functions or behave in particular ways:

- *Ordinary* methods are functions that act on one or more objects and return some new object or some computed value. These methods are like ordinary MATLAB functions that cannot modify input arguments. Ordinary methods enable classes to implement arithmetic operators and computational functions. These methods require an object of the class on which to operate.
- *Constructor* methods are specialized methods that create objects of the class. A constructor method must have the same name as the class and typically initializes property values with data obtained from input arguments. The class constructor method must return the object it creates.
- *Destructor* methods are called automatically when the object is destroyed, for example if you call `delete(object)` or there are no longer any references to the object.

- *Property* access methods enable a class to define code to execute whenever a property value is queried or set.
- *Static* methods are functions that are associated with a class, but do not necessarily operate on class objects. These methods do not require an instance of the class to be referenced during invocation of the method, but typically perform operations in a way specific to the class.
- *Conversion* methods are overloaded constructor methods from other classes that enable your class to convert its own objects to the class of the overloaded constructor. For example, if your class implements a double method, then this method is called instead of the double class constructor to convert your class object to a MATLAB double object.
- *Abstract* methods serve to define a class that cannot be instantiated itself, but serves as a way to define a common interface used by a number of subclasses. Classes that contain abstract methods are often referred to as interfaces.

The constructor method has the same name as the class and returns an object. You can assign values to properties in the class constructor. Terminate all method functions with an end statement.

```
classdef ClassName
    methods
        function obj = ClassName(arg1,arg2,...)
            obj.Prop1 = arg1;
            ...
        end
        function normal_method(obj,arg1,...)
            ...
        end
    end
    methods (Static = true)
        function static_method(arg1,...)
            ...
        end
    end
end
```

MATLAB differs from languages like C++ and Java in that there is no special hidden class instance (e.g. the *this* object) passed to all methods. You must pass an object of the class explicitly to the method. The left most argument does not need to be the class instance, and the argument list can have multiple objects.

You can define class methods in files that are separate from the class definition file. To use multiple files for a class definition, put the class files in a folder having a name beginning with the @ character followed by the name of

the class. Ensure that the parent folder of the @-folder is on the MATLAB path. To define a method in a separate file in the class @-folder, create the function in a separate file, but do not use a method block in that file. Name the file with the function name, as with any function.

You must put the following methods in the classdef file, not in separate files:

- Class constructor
- Delete method
- All functions that use dots in their names, including:
 - Converter methods that convert to classes contained in packages, which must use the package name as part of the class name
 - Property set and get access methods

If you specify method attributes for a method that you define in a separate file, include the method signature in a methods block in the classdef block. For example, the following code shows a method with Access set to private in the methods block. The method implementation resides in a separate file. Do not include the function or end keywords in the methods block, just the function signature showing input and output arguments.

```
classdef ClassName
    % In a methods block, set the method attributes
    % and add the function signature
    methods (Access = private)
        output = myFunc(obj,arg1,arg2)
    end
end
```

In a file named myFunc.m, in the @ClassName folder, define the function:

```
function output = myFunc(obj,arg1,arg2)
    ...
end
```

3.8.4 The events block

The events block (one for each unique set of attribute specifications) contains the names of events that this class declares. The events blocks starts with the events keyword and terminates with the end keyword.

```
classdef className
    events
    ...
end
...
end
```

To define an event, you declare a name for the event in the events block. Then one of the class methods triggers the event using the notify method, which is method inherited from the *handle* class. Only classes derived from the *handle* class can define events. For example, the following class:

- Defines an event named StateChange
 - Triggers the event using the inherited *notify* method.
- ```
classdef class_name < handle % Subclass handle
 events % Define an event called StateChange
 StateChange
 end
 ...
 methods
 function upDateGUI(obj)
 ...
 % Broadcast notice that StateChange event has occurred
 notify(obj,'StateChange');
 end
 end
end
```

Any number of objects can be listening for the StateChange event to occur. When notify executes, MATLAB calls all registered listener callbacks and passes the handle of the object generating the event and an event structure to these functions. To register a listener callback, use the addlistener method of the handle class.

```
addlistener(event_obj,'StateChange',@myCallback)
```

### 3.8.5 Specifying attributes

Attributes modify the behavior of classes and class components (*properties*, *methods*, and *events*). Attributes enable you to define useful behaviors without writing complicated code. For example, you can create a read-only property by setting its SetAccess attribute to private, but leaving its GetAccess attribute set to public (the default):

```
properties (SetAccess = private)
 ScreenSize = getScreenSize;
end
```

All class definition blocks (*classdef*<sup>1</sup>, *properties*<sup>2</sup>, *methods*<sup>3</sup>, and *events*<sup>4</sup>) support specific attributes and all attributes have default values. Specific attribute

<sup>1</sup>Possible attributes for classes are *Hidden*, *InferiorClasses*, *ConstructOnLoad*, *Sealed*.

<sup>2</sup>Possible attributes for properties are *AbortSet*, *Abstract*, *Access*, *Constant*, *Dependent*, *GetAccess*, *GetObservable*, *Hidden*, *SetAccess*, *SetObservable*, *Transient*.

<sup>3</sup>Possible attributes for methods are *Abstract*, *Access*, *Hidden*, *Sealed*, *Static*.

<sup>4</sup>Possible attributes for events are *Hidden*, *ListenAccess*, *NotifyAccess*.

values only in cases where you want to change from the default value to another predefined value.

### 3.9 Example: a polynomial class

This example implements a class to represent polynomials in the MATLAB language. A value class is used because the behavior of a polynomial object within the MATLAB environment should follow the copy semantics of other MATLAB variables. In contrast to the example on the official MATLAB documentation, this example only implements basic functionalities for the polynomial class.

This class overloads a number of MATLAB functions, such as *roots*, *polyval*, *diff*, and *plot* so that these function can be used with the new polynomial object.

#### 3.9.1 Creating the needed folder and file

To use the class, create a folder named *@DocPolynom* and save *DocPolynom.m* to this folder. The parent folder of *@DocPolynom* must be on the MATLAB path.

#### 3.9.2 Using the DocPolynom Class

The following examples illustrate basic use of the *DocPolynom* class.

Create *DocPolynom* objects to represent the following polynomials. Note that the argument to the constructor function contains the polynomial coefficients  $f(x) = x^3 - 2x - 5$  and  $f(x) = 2x^4 + 3x^2 + 2x - 7$ .

```
p1 = DocPolynom([1 0 -2 -5])
p1 =
 x^3 - 2*x - 5
p2 = DocPolynom([2 0 3 2 -7])
p2 =
 2*x^4 + 3*x^2 + 2*x - 7
```

The *DocPolynom disp* method displays the polynomial in MATLAB syntax.

Find the roots of the polynomial using the overloaded *roots* method.

```
roots(p1)
ans =
 2.0946
 -1.0473 + 1.1359i
 -1.0473 - 1.1359i
```

Add the two polynomials p1 and p2.

The MATLAB runtime calls the *plus* method defined for the *DocPolynom* class when you add two *DocPolynom* objects.

```
p1 + p2
ans =
 2*x^4 + x^3 + 3*x^2 - 12
```



The sections that follow describe the implementation of the methods illustrated here, as well as some implementation details.

### 3.9.3 The DocPolynom Constructor Method

The following function is the *DocPolynom* class constructor, which is in the file *@DocPolynom/DocPolynom.m*:

```
function obj = DocPolynom(c)
% Construct a DocPolynom object using the coefficients supplied
if isa(c,'DocPolynom')
 obj.coef = c.coef;
else
 obj.coef = c(:).';
end
end
```

The coefficients are stored in the *coef* property of the *DocPolynom* class:

```
classdef DocPolynom

 properties
 coef
 end

 methods
 function obj = DocPolynom(c)
 % Construct a DocPolynom object using the coefficients supplied
 ...
 end
 end
end
```

**Constructor Calling Syntax** You can call the *DocPolynom* constructor method with two different arguments:

- Input argument is a *DocPolynom* object — If you call the constructor function with an input argument that is already a *DocPolynom* object, the constructor returns a new *DocPolynom* object with the same coefficients as the input argument. The *isa* function checks for this situation.
- Input argument is a coefficient vector — If the input argument is not a *DocPolynom* object, the constructor attempts to reshape the values into a vector and assign them to the *coef* property.

The *coef* property set method restricts property values to doubles. See next section for a discussion of the property set method.

**Removing Irrelevant Coefficients** MATLAB software represents polynomials as row vectors containing coefficients ordered by descending powers. Zeros in the coefficient vector represent terms that drop out of the polynomial. Leading zeros, therefore, can be ignored when forming the polynomial.

Some *DocPolynom* class methods use the length of the coefficient vector to determine the degree of the polynomial. It is useful, therefore, to remove leading zeros from the coefficient vector so that its length represents the true value.

The *DocPolynom* class stores the coefficient vector in a property that uses a *set* method to remove leading zeros from the specified coefficients before setting the property value.

```
function obj = set.coef(obj,val)
% coef set method
if ~isa(val,'double')
 error('Coefficients must be of class double')
end
ind = find(val(:). ~=0);
if ~isempty(ind)
 obj.coef = val(ind(1):end);
else
 obj.coef = val;
end
end
```

**Example use** An example use of the *DocPolynom* constructor is the statement:

```
p = DocPolynom([1 0 -2 -5])
p =
 x^3 - 2*x -5
```

This statement creates an instance of the *DocPolynom* class with the specified coefficients. Note how class methods display the equivalent polynomial using MATLAB language syntax. The *DocPolynom* class implements this display using the *disp* and *char* class methods.

### 3.9.4 Converting DocPolynom Objects to Other Types

The *DocPolynom* class defines two methods to convert DocPolynom objects to other classes:

- **double** — Converts to standard MATLAB numeric type so you can perform mathematical operations on the coefficients.
- **char** — Converts to string; used to format output for display in the command window

**The DocPolynom to Double Converter** The *double* converter method for the *DocPolynom* class simply returns the coefficient vector, which is a double by definition:

```
function c = double(obj)
 % DocPolynom/Double Converter
 c = obj.coef;
end
```

For the *DocPolynom* object *p*:

```
p = DocPolynom([1 0 -2 -5])
```

the statement:

```
c = double(p)
```

returns:

```
c =
 1 0 -2 -5
```

which is of class *double*:

```
class(c)
ans =
 double
```

**The DocPolynom to Character Converter** The *char* method produces a character string that represents the polynomial displayed as powers of an independent variable, *x*. Therefore, after you have specified a value for *x*, the string returned is a syntactically correct MATLAB expression, which you can evaluate.

The *char* method uses a cell array to collect the string components that make up the displayed polynomial.

The *disp* method uses *char* to format the *DocPolynom* object for display. Class users are not likely to call the *char* or *disp* methods directly, but these methods enable the *DocPolynom* class to behave like other data classes in MATLAB.

Here is the *char* method.

```
function str = char(obj)
 % Created a formatted display of the polynom
 % as powers of x
 if all(obj.coef == 0)
 s = '0';
 else
 d = length(obj.coef)-1;
```

```

s = cell(1,d);
ind = 1;
for a = obj.coef;
 if a ~= 0;
 if ind ~= 1
 if a > 0
 s(ind) = {' + '};
 ind = ind + 1;
 else
 s(ind) = {' - '};
 a = -a;
 ind = ind + 1;
 end
 end
 if a ~= 1 || d == 0
 if a == -1
 s(ind) = {'-'};
 ind = ind + 1;
 else
 s(ind) = {num2str(a)};
 ind = ind + 1;
 if d > 0
 s(ind) = {'*'};
 ind = ind + 1;
 end
 end
 end
 if d >= 2
 s(ind) = {'x^' int2str(d)};
 ind = ind + 1;
 elseif d == 1
 s(ind) = {'x'};
 ind = ind + 1;
 end
 d = d - 1;
 end
end
str = [s{:}];
end

```

**Evaluating the Output** If you create the *DocPolynom* object *p*:

```
p = DocPolynom([1 0 -2 -5]);
```

and then call the *char* method on *p*:

```
char(p)
```

the result is:

```
ans =
x^3 - 2*x - 5
```

The value returned by *char* is a string that you can pass to *eval* after you have defined a scalar value for *x*. For example: *x* = 3;

```
eval(char(p))
ans =
16
```

### 3.9.5 The DocPolynom disp method

To provide a more useful display of *DocPolynom* objects, this class overloads *disp* in the class definition.

This *disp* method relies on the *char* method to produce a string representation of the polynomial, which it then displays on the screen.

```
function disp(obj)
% DISP Display object in MATLAB syntax
c = char(obj);
if iscell(c)
 disp([' ', c{:}])
else
 disp(c)
end
end
```

The statement:

```
p = DocPolynom([1 0 -2 -5])
```

creates a *DocPolynom* object. Since the statement is not terminated with a semicolon, MATLAB calls the *disp* method of the *DocPolynom* object to display the output on the command line:

```
p =
x^3 - 2*x - 5
```

## 3.10 Defining the + Operator

If either *p* or *q* is a *DocPolynom* object, the expression

```
p + q
```

generates a call to a function *@DocPolynom/plus*.

The following function redefines the *plus* (+) operator for the *DocPolynom* class:

```
function r = plus(obj1,obj2)
 % Plus Implement obj1 + obj2 for DocPolynom
 obj1 = DocPolynom(obj1);
 obj2 = DocPolynom(obj2);
 k = length(obj2.coef) - length(obj1.coef);
 r = DocPolynom([zeros(1,k) obj1.coef]+[zeros(1,-k) obj2.coef]);
end
```

Here is how the function works:

- Ensure that both input arguments are *DocPolynom* objects so that expressions such as

$$p + 1$$

that involve both a *DocPolynom* and a *double*, work correctly.

- Access the two coefficient vectors and, if necessary, pad one of them with zeros to make both the same length. The actual addition is simply the vector sum of the two coefficient vectors.
- Call the *DocPolynom* constructor to create a properly typed result.

### 3.11 Overloading MATLAB Functions roots and polyval for the DocPolynom Class

The MATLAB language already has several functions for working with polynomials that are represented by coefficient vectors. You can overload these functions to work with the new *DocPolynom* class.

In the case of *DocPolynom* objects, the overloaded methods can simply apply the original MATLAB function to the coefficients (i.e., the values returned by the *coef* property).

This section shows how to implement the following MATLAB functions.

#### 3.11.1 Defining the roots function

The *DocPolynom* *roots* method finds the roots of *DocPolynom* objects by passing the coefficients to the overloaded roots function:

```
function r = roots(obj)
 % roots(obj) returns a vector containing the roots of obj
 r = roots(obj.coef);
end
```

If *p* is the following *DocPolynom* object:

```
p = DocPolynom([1 0 -2 -5]);
```

then the statement:

```
roots(p)
```

gives the following answer:

```
ans =
 2.0946
 -1.0473 + 1.1359i
 -1.0473 - 1.1359i
```

### 3.11.2 Defining the polyval function

The MATLAB *polyval* function evaluates a polynomial at a given set of points. The *DocPolynom polyval* method simply extracts the coefficients from the *coef* property and then calls the MATLAB version to compute the various powers of *x*:

```
function y = polyval(obj,x)
 % polyval(obj,x) evaluates obj at the points x
 y = polyval(obj.coef,x);
end
```

The following code evaluates the polynomial  $p = x^3$  for  $x = -2 : 0.1 : 2$  and plots it.

```
p = DocPolynom([1 0 0 0])
p =
 x^3
x = -2:0.1:2;
y = polyval(p, x);
plot(x, y);
```

### 3.11.3 Complete DocPolynom example

The complete DocPolynom example class can be found in the MATLAB documentation code:

```
edit ([docroot 'techdoc/matlab_oop/examples/@DocPolynom/DocPolynom.m']);
```

## 4 Interfacing with Java

### 4.1 References

- [http://www.mathworks.com/access/helpdesk/help/techdoc/matlab\\_external/f44062.html](http://www.mathworks.com/access/helpdesk/help/techdoc/matlab_external/f44062.html)
- [http://www.mathworks.com/access/helpdesk/help/pdf\\_doc/matlab/apiext.pdf](http://www.mathworks.com/access/helpdesk/help/pdf_doc/matlab/apiext.pdf)

## 4.2 Java Virtual Machine (JVM)

Every installation of MATLAB software includes Java Virtual Machine (JVM) software, so that you can use the Java interpreter via MATLAB commands, and you can create and run programs that create and access Java objects<sup>5</sup>.

The MATLAB Java interface enables you to:

- Access Java API (application programming interface) class packages that support essential activities such as I/O and networking. For example, the URL class provides convenient access to resources on the Internet.
- Access third-party Java classes
- Easily construct Java objects in MATLAB workspace
- Call Java object methods, using either Java or MATLAB syntax
- Pass data between MATLAB variables and Java objects

MATLAB can run Java code in the form of *.class* or *.jar* files.

## 4.3 The Java classpath

MATLAB loads Java class definitions from files that are on the Java class path. The class path is a series of file and directory specifications that MATLAB software uses to locate class definitions. When loading a particular Java class, MATLAB searches files and directories in the order they occur on the class path until a file is found that contains that class definition. The search ends when the first definition is found. The Java class path consists of two segments: the static path and the dynamic path. MATLAB loads the static path at startup. If you change the path you must restart MATLAB. You can load and modify the dynamic path at any time using MATLAB functions. MATLAB always searches the static path before the dynamic path. You can view these two path segments using the *javaclasspath* function.

### 4.3.1 Static classpath

To see which *classpath.txt* file is currently being used by your MATLAB environment, use the *which* function:

```
which classpath.txt
```

To edit either the default file or the copy in your own directory, type:

```
edit classpath.txt
```

---

<sup>5</sup>To use a different JVM than the one that comes with MATLAB, you now can set the MATLAB\_JAVA system environment variable to the path of your JVM software.



### 4.3.2 Dynamic classpath

The dynamic class path can be loaded any time during a MATLAB software session using the *javaclasspath* function. You can define the dynamic path (using *javaclasspath*), modify the path (using *javaaddpath* and *javarmppath*), and refresh the Java class definitions for all classes on the dynamic path (using *clear* with the keyword *java*) without restarting MATLAB.

### 4.3.3 Adding JAR packages

In contrast to the static and dynamic classpaths, where the **directory** containing the *.class* file is added to the path, to make the contents of a JAR file available for use in MATLAB, specify the full path, including full file name, for the JAR file. You can also put the JAR file on the MATLAB path. For example, to make available the JAR file *e:\java\classes\utilpkg.jar*, add the following file specification to your static or dynamic class path:

```
e:\java\classes\utilpkg.jar
```

**Exercise** Add ImageJ (ij.jar) to the MATLAB path.

## 4.4 Simplifying Java Class Names

Your MATLAB commands can refer to any Java class by its fully qualified name, which includes its package name. For example, the following are fully qualified names:

- java.lang.String
- java.util.Enumera**tion**

A fully qualified name can be rather long, making commands and functions, such as constructors, cumbersome to edit and to read. You can refer to classes by the class name alone (without a package name) if you first import the fully qualified name into MATLAB. The *import* command has the following forms:

```
import pkg_name.* % Import all classes in package
import pkg_name1.* pkg_name2.* % Import multiple packages
import class_name % Import one class
import % Display current import list
L = import % Return current import list
```

MATLAB adds all classes that you import to a list called the import list. You can see what classes are on that list by typing *import*, without any arguments. Your code can refer to any class on the list by class name alone. When called from a function, *import* adds the specified classes to the import list in effect for that function.

## 4.5 Creating and using Java objects

You create a Java object in the MATLAB workspace by calling one of the constructors of that class. You then use commands and programming statements to perform operations on these objects. You can also save your Java objects to a MAT-file and, in subsequent sessions, reload them into MATLAB.

### 4.5.1 Constructing Java objects

You construct Java objects in the MATLAB workspace by calling the Java class constructor, which has the same name as the class. For example, the following constructor creates a *myDate* object:

```
myDate = java.util.Date
```

MATLAB displays information similar to:

```
myDate =
Thu Aug 23 12:58:54 EDT 2007
```

### 4.5.2 Invoking methods on Java objects

To call methods on Java objects, you can use the Java syntax:

```
object.method(arg1,...,argn)
```

or the MATLAB syntax<sup>6</sup>:

```
method(object, arg1,...,argn)
```

In the following example, *myDate* is a *java.util.Date* object, and *getHours* and *setHours* are methods of that object.

```
myDate = java.util.Date;
myDate.setHours(3)
myDate.getHours // Java syntax
ans =
 3
getHours(myDate) // MATLAB syntax
ans =
 3
```

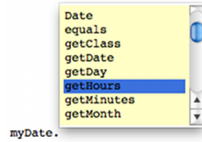
### 4.5.3 Obtaining information about methods

MATLAB offers several ways to help obtain information related to the Java methods you are working with. You can request a list of all of the methods that are implemented by any class. The list might be accompanied by other method information such as argument types and exceptions. You can also request a listing of every Java class that you loaded into MATLAB that implements a specified method.

---

<sup>6</sup>There is an alternative syntax, which makes use of the `javaObject()` and `javaMethod()` functions: `myDate = javaObject( 'java.util.Date' ); h = javaMethod( 'getHours', myDate );`

**Tab key** The simplest way is to type the name of a Java object on the MATLAB console followed by `.'` and press the *Tab* key.



**The `methodsview` function** A more complete way is by using the *methodsview* function:

```
methodsview java.util.Date // Syntax 1
methodsview('java.util.Date') // Syntax 2
myDate = java.util.Date;
methodsview(myDate) // Syntax 3
```

A new window appears, listing one row of information for each method in the class.

| Qualifiers | Return Type      | Name              | Arguments                 | Other |
|------------|------------------|-------------------|---------------------------|-------|
|            | Date             | Date              | ()                        |       |
|            | Date             | Date              | (long)                    |       |
|            | Date             | Date              | (int,int,int)             |       |
|            | Date             | Date              | (int,int,int,int,int)     |       |
|            | Date             | Date              | (int,int,int,int,int,int) |       |
|            | Date             | Date              | (java.lang.String)        |       |
| static     | long             | UTC               | (int,int,int,int,int)     |       |
|            | boolean          | after             | (java.util.Date)          |       |
|            | boolean          | before            | (java.util.Date)          |       |
|            | java.lang.Object | clone             | ()                        |       |
|            | int              | compareTo         | (java.util.Date)          |       |
|            | int              | compareTo         | (java.lang.Object)        |       |
|            | boolean          | equals            | (java.lang.Object)        |       |
|            | java.lang.Class  | getClass          | ()                        |       |
|            | int              | getDate           | ()                        |       |
|            | int              | getDay            | ()                        |       |
|            | int              | getHours          | ()                        |       |
|            | int              | getMinutes        | ()                        |       |
|            | int              | getMonth          | ()                        |       |
|            | int              | getSeconds        | ()                        |       |
|            | long             | getTime           | ()                        |       |
|            | int              | getTimezoneOffset | ()                        |       |
|            | int              | getYear           | ()                        |       |
|            | int              | hashCode          | ()                        |       |
|            | void             | notify            | ()                        |       |
|            | void             | notifyAll         | ()                        |       |

Each row in the window displays up to six fields of information describing the method:

- Qualifiers (i.e. method type qualifiers): abstract, synchronized, ...
- Return Type (i.e. type returned by the method): void, java.lang.String, ...
- Name (i.e. method name): getHours, addActionListener, ...
- Arguments (i.e. types of arguments passed to method): boolean, java.lang.Object, ...

- Other (i.e. other relevant information): throws java.io.IOException, ...
- Inherited From (i.e. parent of the specified class): java.awt.MenuComponent, ...

**The `methods` function** The `methods` function returns information on methods of MATLAB and Java classes. You can use any of the following forms of this command.

```
methods class_name
methods class_name -full
n = methods('class_name')
n = methods('class_name', '-full')
```

Use `methods` without the `'-full'` qualifier to return the names of all the methods (including inherited methods) of the class. Names of overloaded methods are listed only once. With the `'-full'` qualifier, `methods` returns a listing of the method names (including inherited methods) along with attributes, argument lists, and inheritance information on each. Each overloaded method is listed separately.

For example, display a full description of all methods of the `java.awt.Dimension` object.

```
methods java.awt.Dimension -full

Methods for class java.awt.Dimension:
Dimension(int,int)
Dimension()
Dimension(java.awt.Dimension)
java.lang.Object clone() % Inherited from java.awt.geom.Dimension2D
...
```

## 4.6 Passing arguments to and from a Java method

When you make a call in the MATLAB to Java code, any MATLAB types you pass in the call are converted to types native to the Java language. MATLAB performs this conversion on each argument that is passed, except for those arguments that are already Java objects. If data is to be returned by the method being called, MATLAB receives this data and converts it to the appropriate MATLAB format wherever necessary.

### 4.6.1 Conversion of MATLAB data types

MATLAB data, passed as arguments to Java methods, are converted by MATLAB into types that best represent the data to the Java language. The table below shows all of the MATLAB base types for passed arguments and the Java

base types defined for input arguments. Each row shows a MATLAB type followed by the possible Java argument matches, from left to right in order of closeness of the match. The MATLAB types (except cell arrays) can all be scalar (1-by-1) arrays or matrices. All of the Java types can be scalar values or arrays.

| <b>MATLAB Argument</b> | <b>Closest Type (7)</b> | <b>Java Input Argument (Scalar or Array)</b> |       |        |        |        |         | <b>Least Close Type (1)</b> |
|------------------------|-------------------------|----------------------------------------------|-------|--------|--------|--------|---------|-----------------------------|
| logical                | boolean                 | byte                                         | short | int    | long   | float  | double  |                             |
| double                 | double                  | float                                        | long  | int    | short  | byte   | boolean |                             |
| single                 | float                   | double                                       | N/A   | N/A    | N/A    | N/A    | N/A     |                             |
| char                   | String                  | char                                         | N/A   | N/A    | N/A    | N/A    | N/A     |                             |
| uint8                  | byte                    | short                                        | int   | long   | float  | double | N/A     |                             |
| uint16                 | short                   | int                                          | long  | float  | double | N/A    | N/A     |                             |
| uint32                 | int                     | long                                         | float | double | N/A    | N/A    | N/A     |                             |
| int8                   | byte                    | short                                        | int   | long   | float  | double | N/A     |                             |
| int16                  | short                   | int                                          | long  | float  | double | N/A    | N/A     |                             |
| int32                  | int                     | long                                         | float | double | N/A    | N/A    | N/A     |                             |
| cell array of strings  | array of String         | N/A                                          | N/A   | N/A    | N/A    | N/A    | N/A     |                             |
| Java object            | Object                  | N/A                                          | N/A   | N/A    | N/A    | N/A    | N/A     |                             |
| cell array of object   | array of Object         | N/A                                          | N/A   | N/A    | N/A    | N/A    | N/A     |                             |
| MATLAB object          | N/A                     | N/A                                          | N/A   | N/A    | N/A    | N/A    | N/A     |                             |

#### 4.6.2 Conversion of Java return data types

In many cases, data returned from a Java method is incompatible with the types operated on in the MATLAB environment. When this is the case, MATLAB converts the returned value to a type native to the MATLAB language. The following table lists Java return types and the resulting MATLAB types. For some Java base return types, MATLAB treats scalar and array returns differently, as described following the table.

| Java Return Type | If Scalar Return,<br>Resulting MATLAB<br>Type | If Array Return,<br>Resulting MATLAB<br>Type |
|------------------|-----------------------------------------------|----------------------------------------------|
| boolean          | logical                                       | logical                                      |
| byte             | double                                        | int8                                         |
| short            | double                                        | int16                                        |
| int              | double                                        | int32                                        |
| long             | double                                        | double                                       |
| float            | double                                        | single                                       |
| double           | double                                        | double                                       |
| char             | char                                          | char                                         |

## 5 Interfacing with C/C++

### 5.1 References

- [http://www.mathworks.com/access/helpdesk/help/techdoc/matlab\\_external/f7667.html](http://www.mathworks.com/access/helpdesk/help/techdoc/matlab_external/f7667.html)
- [http://www.mathworks.com/access/helpdesk/help/pdf\\_doc/matlab/apiext.pdf](http://www.mathworks.com/access/helpdesk/help/pdf_doc/matlab/apiext.pdf)

### 5.2 MEX files

You can call your own C, C++, or Fortran subroutines from the MATLAB command line as if they were built-in functions. These programs, called binary MEX-files, are dynamically-linked subroutines that the MATLAB interpreter loads and executes. MEX stands for “MATLAB executable.” In this course we won’t discuss Fortran MEX files.

MEX-files have several applications:

- Calling large pre-existing C/C++ and Fortran programs from MATLAB without rewriting them as MATLAB functions
- Replacing performance-critical routines with C/C++ implementations

The second point has become less critical over the years, with the MATLAB becoming faster and faster.

A computational routine is the source code that performs functionality you want to use with MATLAB. For example, if you created a standalone C program for this functionality, it would have a *main()* function. MATLAB communicates with your MEX-file using a *gateway routine*. The MATLAB function that creates the gateway routine is *mexfunction*. You use *mexfunction* instead of *main()* in your source file.

### 5.3 Overview of Creating a C/C++ Binary MEX-File

To create a binary MEX-file:

- Assemble your functions and the MATLAB API functions into one or more C/C++ source files.
- Write a gateway function in one of your C/C++ source files.
- Use the MATLAB *mex* function, called a build script, to build a binary MEX-file.
- Use your binary MEX-file like any MATLAB function.

## 5.4 Configuring your environment

Before you start building binary MEX-files, select your default compiler. In the MATLAB console, type:

```
>> mex -setup
```

```
Options files control which compiler to use, the compiler and link command
options, and the runtime libraries to link against.
```

```
Using the 'mex -setup' command selects an options file that is
placed in ~/.matlab/R2010a and used by default for 'mex'. An options
file in the current working directory or specified on the command line
overrides the default options file in ~/.matlab/R2010a.
```

```
To override the default options file, use the 'mex -f' command
(see 'mex -help' for more information).
```

```
The options files available for mex are:
```

```
1: /Applications/MATLAB_R2010a.app/bin/gccopts.sh :
```

```
 Template Options file for building gcc MEX-files
```

```
2: /Applications/MATLAB_R2010a.app/bin/mexopts.sh :
```

```
 Template Options file for building MEX-files via the system ANSI compiler
```

```
0: Exit with no changes
```

```
Enter the number of the compiler (0-2):
```

```
1
```

```
/Applications/MATLAB_R2010a.app/bin/gccopts.sh is being copied to
/Users/aaron/.matlab/R2010a/mexopts.sh
```

```

```

```
Warning: The MATLAB C and Fortran API has changed to support MATLAB
variables with more than 2^32-1 elements. In the near future
you will be required to update your code to utilize the new
API. You can find more information about this at:
```

```
http://www.mathworks.com/support/solutions/en/data/1-5C27B9/?
solution=1-5C27B9
```

```
Building with the -largeArrayDims option enables the new API.
```

```

```

Now, MATLAB's *mex* is configured to use the selected compiler to build your MEX-files. We will discuss the Warning message from newer MATLAB versions below.

## 5.5 Using MEX-files to call a C program

Suppose you have some C code, called *arrayProduct*, that multiplies an 1-dimensional array *y* with *n* elements by a scalar value *x* and returns the results in array *z*. It might look something like the following:

```
void arrayProduct(double x, double *y, double *z, int n)
{
 int i;
 for (i=0; i<n; i++)
 {
 z[i] = x * y[i];
 }
}
```

If  $x = 5$  and *y* is an array with values 1.5, 2, and 9, then calling:

```
arrayProduct(x,y,z,n)
```

from within your C program creates an array *z* with the values 7.5, 10, and 45.

The following steps show how to call this function in MATLAB, using a MATLAB matrix, by creating the MEX-file *arrayProduct*.

### 5.5.1 Create a source MEX file

Open MATLAB Editor and copy your code into a new file. Save the file on your MATLAB path and name it *arrayProduct.c*. This file is your *computational routine*, and the name of your MEX-file is *arrayProduct*. We will now modify the code to turn it into a valid (and usable) MEX-file.

### 5.5.2 Create a *gateway routine*

At the beginning of the file, add the C/C++ header file:

```
#include "mex.h"
```

After the computational routine, add the gateway routine *mexFunction*:

```
/* The gateway function */
void mexFunction(int nlhs, mxArray *plhs[], int nrhs, const mxArray *prhs[])
{
 /* variable declarations here */

 /* code here */
}
```



}<sup>7</sup>

`mexFunction` is the entry point for the MEX-file, i.e. what MATLAB will call when launching your MEX-file. This is currently just a placeholder. We will now add the content of the *gateway function*, but before that we will spend a couple of words on the signature of *mexFunction*.

```
void mexFunction(int nlhs, // Number of input parameters
 mxArray *plhs[], // Array of pointers to inputs
 int nrhs, // Number of output parameters
 const mxArray *prhs[]) // Array of pointers to const
 // outputs
{ ... }
```

Input parameters (found in the `prhs` array) are read-only; do not modify them in your MEX-file. Changing data in an input parameter can produce undesired side effects.

The MATLAB language works with only a single object type: the MATLAB array. All MATLAB variables, including scalars, vectors, matrices, strings, cell arrays, structures, and objects, are stored as MATLAB arrays. In C/C++, the MATLAB array is declared to be of type **`mxArray`**. The `mxArray` structure contains, among other things:

- Its type
- Its dimensions
- The data associated with this array
- If numeric, whether the variable is real or complex
- If sparse, its indices and nonzero maximum elements
- If a structure or object, the number of fields and field names.

### 5.5.3 Use preprocessor macros

The MX Matrix Library and MEX Library functions use MATLAB preprocessor macros for cross-platform flexibility. Edit your computational routine to use *mwSize* for **`mxArray`** size *n* and index *i*.

```
void arrayProduct(double x, double *y, double *z, mwSize n)
{
 mwSize i;
 for (i=0; i<n; i++)
```

---

<sup>7</sup>This example is written in C and in C all variable declarations MUST be at the beginning of the function. This constraint does not exist in C++.

```

 {
 z[i] = x * y[i];
 }
}

```

*mwSize* replaces *int* to ensure that **mxArrays** with more than  $2^{32} - 1$  elements can be addressed correctly.

#### 5.5.4 Verify Input and Output Parameters

In this example, there are two input arguments (a matrix and a scalar) and one output argument (the product). To check that the number of input arguments *nrhs* is two and the number of output arguments *nlhs* is one, put the following code inside the *mexFunction* routine:

```

/* check for proper number of arguments */
if(nrhs!=2)
{
 mexErrMsgIdAndTxt("MyToolbox:arrayProduct:nrhs",
 "Two inputs required.");
}
if(nlhs!=1)
{
 mexErrMsgIdAndTxt("MyToolbox:arrayProduct:nlhs",
 "One output required.");
}

```

The following code validates the input values:

```

/* make sure the first input argument is scalar */
if(!mxIsDouble(prhs[0]) ||
 mxIsComplex(prhs[0]) ||
 mxGetNumberOfElements(prhs[0])!=1)
{
 mexErrMsgIdAndTxt("MyToolbox:arrayProduct:notScalar",
 "Input multiplier must be a scalar.");
}

```

The second input argument must be a row vector.

```

/* check that number of rows in second input argument is 1 */
if(mxGetM(prhs[1])!=1) {
 mexErrMsgIdAndTxt("MyToolbox:arrayProduct:notRowVector",
 "Input must be a row vector.");
}

```

### 5.5.5 Read input data

Put the following declaration statements at the beginning of your mexFunction:

```
double multiplier; /* input scalar */
double *inMatrix; /* 1xN input matrix */
mwSize ncols; /* size of matrix */
```

Add these statements to the code section of mexFunction:

```
/* get the value of the scalar input */
multiplier = mxGetScalar(prhs[0]);
/* create a pointer to the real data in the input matrix */
inMatrix = mxGetPr(prhs[1]);
/* get dimensions of the input matrix */
ncols = mxGetN(prhs[1]);
```

### 5.5.6 Prepare output data

Put the following declaration statement after your input variable declarations:

```
double *outMatrix; /* output matrix */
```

Add these statements to the code section of mexFunction:

```
/* create the output matrix */
plhs[0] = mxCreateDoubleMatrix(1,ncols,mxREAL);
/* get a pointer to the real data in the output matrix */
outMatrix = mxGetPr(plhs[0]);
```

### 5.5.7 Perform Calculation

The following statement executes your function:

```
/* call the computational routine */
arrayProduct(multiplier,inMatrix,outMatrix,ncols);
```

### 5.5.8 Build the Binary MEX-File

You can build your MEX-file by typing:

```
>> mex arrayProduct.c
```

If your source file is correct, mex should compile silently. In case something is wrong, mex will output error messages to the MATLAB console.

**Test the MEX-File** Type:

```
s = 5;
A = [1.5, 2, 9];
B = arrayProduct(s,A)
```

This should output:

```
B =
 7.5000 10.0000 45.0000
```

**Exercise** Try the following:

```
s = 5;
A = [1.5, 2; 9 11];
B = arrayProduct(s,A)
```

What do you get? Where is the mistake?

**Exercise** Try the following:

```
s = 3
A = uint8([1, 5, 9]);
B = arrayProduct(s,A)
```

What do you get? Where is the mistake?