



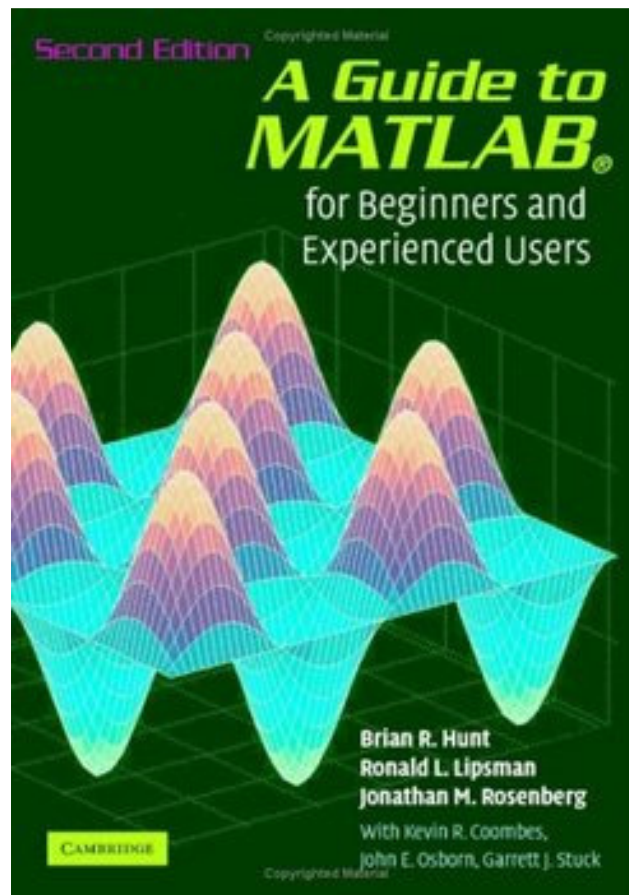
وزارت علوم، تحقیقات و فناوری
مؤسسه آموزش عالی سجاد

راهنمای نگارش فنی:

رسم نمودار توسط نرم افزار MATLAB

این راهنما گزیده ایست از ویرایش دوم کتاب A Guide to MATLAB: For Beginners and Experienced Users

نوشته ی Brian R. Hunt



Chapter 5

MATLAB Graphics

In this chapter we describe more of MATLAB's graphics commands and the most common ways of manipulating and customizing graphics. For an overview of commands, type **help graphics** (for general graphics commands), **help graph2d** (for two-dimensional graphics commands), **help graph3d** (for three-dimensional graphics commands), and **help specgraph** (for specialized graphing commands).

We have already discussed the commands **plot** and **ezplot** in Chapter 2. We will begin this chapter by discussing more uses of these commands, as well as some of the other most commonly used plotting commands. Then we will discuss methods for customizing and manipulating graphics. Finally, we will introduce some commands and techniques for creating and modifying images and sounds.

- ✓ For most types of graphs we describe below, there is a command like **plot** that draws the graph from numerical data, and a command like **ezplot** that graphs functions specified by string or symbolic input. The latter commands may be easier to use at first, but are more limited in their capabilities and less amenable to customization. Thus, we emphasize the commands that plot data, which are likely to be more useful to you in the long run.

Two-Dimensional Plots

Often one wants to draw a curve in the x - y plane, but with y not given explicitly as a function of x . There are two main techniques for plotting such curves: parametric plotting and contour or implicit plotting.

Parametric Plots

Sometimes x and y are both given as functions of some parameter. For example, the circle of radius 1 centered at $(0, 0)$ can be expressed in *parametric* form as $x = \cos(2\pi t)$, $y = \sin(2\pi t)$, where t runs from 0 to 1. Though y is not expressed as a function of x , you can easily graph this curve with **plot**, as follows (see Figure 5.1):

```
>> T = 0:0.01:1;
>> plot(cos(2*pi*T), sin(2*pi*T))
>> axis square
```

If we had used an increment of only 0.1 in the **T** vector, the result would have been a polygon with clearly visible corners. When your graph has corners that shouldn't be there, you should repeat the process with a smaller increment until you get a graph

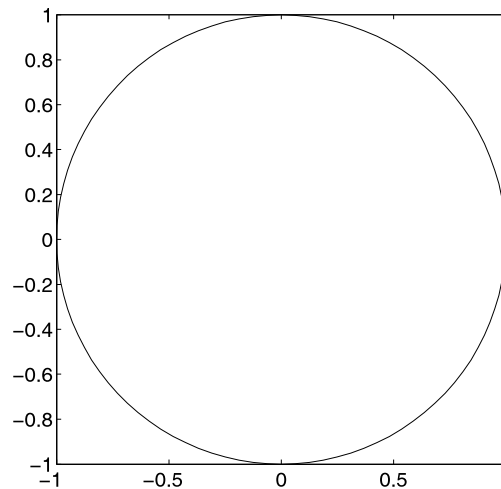


Figure 5.1. The Unit Circle $x^2 + y^2 = 1$.

that looks smooth. Here also **axis square** forces the same scale on both axes; without it the circle would look like an ellipse.

Parametric plotting is also possible with **ezplot**. You can obtain almost the same picture as Figure 5.1 with the command:

```
>> ezplot('cos(t)', 'sin(t)', [0 2*pi]); axis square
```

Notice that we used a semicolon after the **ezplot** command, but it did not prevent the graph from appearing. In general, the semicolon suppresses only text output.

Contour Plots and Implicit Plots

A *contour plot* of a function of two variables is a plot of the *level curves* of the function, i.e., sets of points in the x - y plane where the function assumes a constant value. For example, the level curves of $x^2 + y^2$ are circles centered at the origin, and the *levels* are the squares of the radii of the circles. Contour plots are produced in MATLAB with **meshgrid** and **contour**. The command **meshgrid** produces a grid of points in a rectangular region, with a specified spacing. This grid is used by **contour** to produce a contour plot in the specified region.

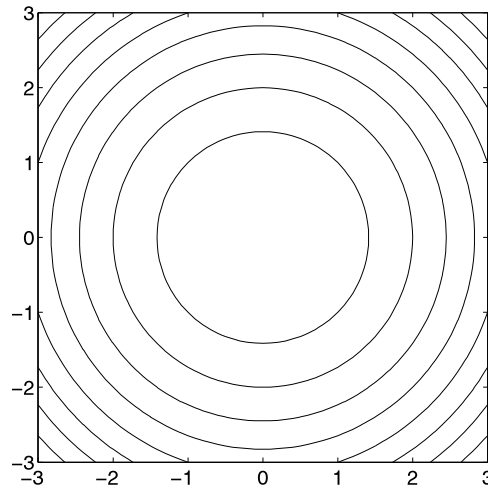
You can make a contour plot of $x^2 + y^2$ as follows (see Figure 5.2):

```
>> [X Y] = meshgrid(-3:0.1:3, -3:0.1:3);
>> contour(X, Y, X.^2 + Y.^2); axis square
```

You can specify particular level sets by including an additional vector argument to **contour**. For example, to plot the circles of radius 1, $\sqrt{2}$, and $\sqrt{3}$, type

```
>> contour(X, Y, X.^2 + Y.^2, [1 2 3])
```

The vector argument must contain at least two elements, so, if you want to plot a

Figure 5.2. Contour Plot of $x^2 + y^2$.

single level set, you must specify the same level twice. This is quite useful for implicit plotting of a curve given by an equation in x and y . For example, to plot the circle of radius 1 about the origin, type


```
>> contour(X, Y, X.^2 + Y.^2, [1 1])
```

Or to plot the lemniscate $x^2 - y^2 = (x^2 + y^2)^2$, rewrite the equation as

$$(x^2 + y^2)^2 - x^2 + y^2 = 0$$

and type (see Figure 5.3)

```
>> [X Y] = meshgrid(-1.1:0.01:1.1, -1.1:0.01:1.1);
>> contour(X, Y, (X.^2 + Y.^2).^2 - X.^2 + Y.^2, [0 0])
>> axis square
>> title('The lemniscate x^2-y^2=(x^2+y^2)^2')
```

 In this case, we used `^` to produce exponents in the title. You can also use `^` for subscripts, and to produce a Greek letter, precede its name with a backslash – for example, `\theta`. Type `doc title` and look under “Examples” for other tricks with titles; these features also apply to labeling commands like `xlabel` and `ylabel`. For more on annotating graphs, see the section Customizing Graphics later in this chapter.

You can also do contour plotting with the command `ezcontour`, and implicit plotting of a curve $f(x, y) = 0$ with `ezplot`. In particular, you can obtain almost the same picture as Figure 5.2 with the command:

```
>> ezcontour('x^2 + y^2', [-3, 3], [-3, 3]); axis square
```

and almost the same picture as Figure 5.3 with the command:

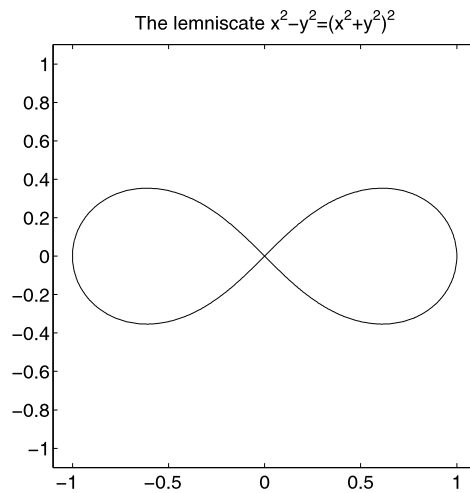


Figure 5.3. A Lemniscate.

```
>> ezplot('(x^2 + y^2)^2 - x^2 + y^2', ...
[-1.1, 1.1], [-1.1, 1.1]); axis square
```

Field Plots

The MATLAB routine **quiver** is used to plot vector fields or arrays of arrows. The arrows can either be located at equally spaced points in the plane (if x - and y -coordinates are not given explicitly), or they can be placed at specified locations. Sometimes some fiddling is required to scale the arrows so that they don't come out looking too big or too small. For this purpose, **quiver** takes an optional scale-factor argument. The following code, for example, plots a vector field with a “saddle point,” corresponding to a combination of an attractive force pointing toward the x -axis and a repulsive force pointing away from the y -axis. The output is shown in Figure 5.4.

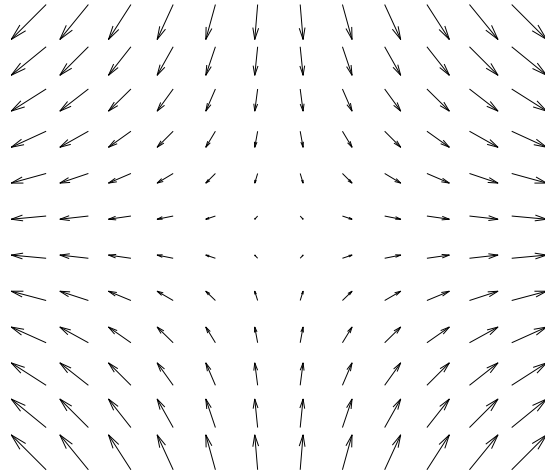
```
>> [x, y] = meshgrid(-1.1:0.2:1.1, -1.1:0.2:1.1);
>> quiver(x, -y); axis equal; axis off
```

Three-Dimensional Plots

MATLAB has several routines for producing three-dimensional plots.

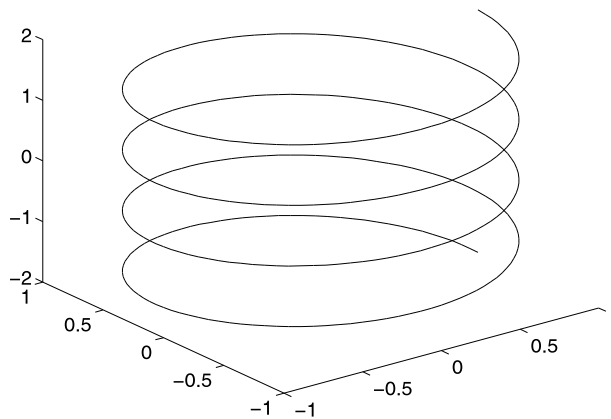
Curves in Three-Dimensional Space

For plotting curves in 3-space, the basic command is **plot3**. It works like **plot**, except that it takes three vectors instead of two, one for the x -coordinates, one for the

Figure 5.4. A Vector-Field Plot of $(x, -y)$.

y -coordinates, and one for the z -coordinates. For example, we can plot a helix with

```
>> T = -2:0.01:2;
>> plot3(cos(2*pi*T), sin(2*pi*T), T)
```

Figure 5.5. The Helix $x = \cos(2\pi z)$, $y = \sin(2\pi z)$.

There is also a three-dimensional analog to `ezplot` called `ezplot3`; you can instead plot the helix in Figure 5.5 with

```
>> ezplot3('cos(2*pi*t)', 'sin(2*pi*t)', 't', [-2, 2])
```

Surfaces in Three-Dimensional Space

There are two basic commands for plotting surfaces in 3-space: **mesh** and **surf**. The former produces a transparent “mesh” surface, the latter an opaque shaded one. There are two different ways of using each command, one for plotting surfaces in which the z -coordinate is given as a function of x and y , and one for *parametric surfaces* in which x , y , and z are all given as functions of two other parameters. Let us illustrate the former with **mesh** and the latter with **surf**.

To plot $z = f(x, y)$, one begins with a **meshgrid** command as in the case of **contour**. For example, the “saddle surface” $z = x^2 - y^2$ can be plotted with

```
>> [X,Y] = meshgrid(-2:0.1:2, -2:0.1:2);
>> Z = X.^2 - Y.^2; mesh(X, Y, Z)
```

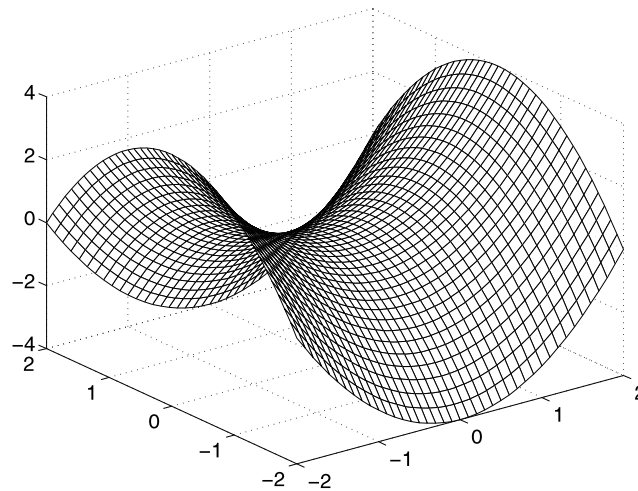


Figure 5.6. The Surface $z = x^2 - y^2$.

The resulting graph looks better on the computer screen since MATLAB shades the surface with a color scheme depending on the z -coordinate. We could have produced an opaque surface instead by replacing **mesh** with **surf**.

There are also shortcut commands **ezmesh** and **ezsurf**; you can obtain a result very similar to Figure 5.6 with

```
>> ezmesh('x^2 - y^2', [-2, 2, -2, 2])
```

If one wants to plot a surface that cannot be represented by an equation of the form $z = f(x, y)$, for example the sphere $x^2 + y^2 + z^2 = 1$, then it is better to parameterize the surface using a suitable coordinate system, in this case cylindrical or spherical coordinates. For example, we can take as parameters the vertical coordinate z and the polar coordinate θ in the x - y plane. If r denotes the distance to the z -axis, then the equation of the sphere becomes $r^2 + z^2 = 1$, or $r = \sqrt{1 - z^2}$, and so

$x = \sqrt{1 - z^2} \cos \theta$, $y = \sqrt{1 - z^2} \sin \theta$. Thus we can produce our plot with

```
>> [Z, Theta] = meshgrid(-1:0.1:1, (0:0.1:2)*pi);
>> X = sqrt(1 - Z.^2).*cos(Theta);
>> Y = sqrt(1 - Z.^2).*sin(Theta);
>> surf(X, Y, Z); axis square
```

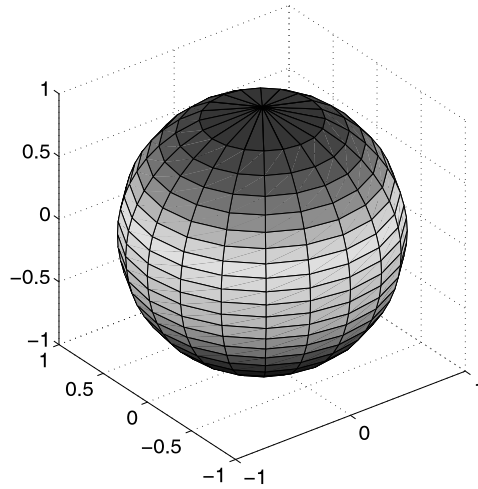


Figure 5.7. The Unit Sphere $x^2 + y^2 + z^2 = 1$.

You can also plot a surface parametrically with **ezmesh** or **ezsurf**; you can obtain a result very similar to Figure 5.7 with

```
>> ezsurf('sqrt(1-z^2)*cos(t)', ...
'sqrt(1-z^2)*sin(t)', 'z', [0, 2*pi, -1, 1]); axis equal
```

Notice that we had to specify the **t** range first because it is alphabetically before **z**, even though **z** occurs before **t** in the strings we entered. Rather than remember such a rule, in MATLAB 7 you can enter functions of more than one variable as anonymous functions, for example `@(z,t) sqrt(1-z.^2).*cos(t)`; then, since you used **z** as the first input variable, you would specify its range first.

Figure Windows

So far we have discussed only graphics commands that produce or modify a single plot. But MATLAB is also capable of opening multiple figure windows or combining several plots in one figure window. Here we discuss some basic methods for managing and manipulating figure windows.

Multiple Figure Windows

When you execute the first plotting command in a given MATLAB session, the graph appears in a new window labeled “Figure 1”. Subsequent graphics commands either modify or replace the graph in this window. You have seen that **hold on** directs that new plotting commands should add to, rather than replace, the current graph. If instead you want to create a new graph in a separate window while keeping the old graph on your screen, type **figure** to open a new window, labeled “Figure 2”. Alternatively, you can select a New Figure from the **File** menu in either your main MATLAB window or the first figure window. Subsequent graphics commands will affect only this window, until you change your *current figure* again with another **figure** command (e.g., type **figure(1)** to switch the current figure back to “Figure 1”), or by bringing another figure to the foreground with the mouse. When multiple figure windows are open, you can find out which is the current one by typing **gcf**, short for “get current figure.” Finally, you can delete figure windows in the usual manner with the mouse, or with the command **close**; see the online help for details.

The Figure Toolbar

Each figure window has a tool bar underneath its menu bar with shortcuts for several menu items, including on the left icons for opening, saving, and printing figures. Near the middle, there are several icons that correspond to items in the **Tools** menu.

The two icons with plus and minus signs control zooming in and out. Click on the icon with a plus sign, and then click on a point in your graph to zoom in near that point. You can click and zoom multiple times; each zoom changes the scale on both axes by a factor of roughly 2. However, don’t click too fast, because double-clicking resets the graph to its original state. Clicking on the icon with the minus sign allows you to zoom out gradually. MATLAB 7 also has an icon in the shape of a hand that allows you to click and drag the graph to pan both horizontally and vertically within the current axes. More zooming options are available by right-clicking in the figure window while zooming, by using the **Options** submenu of the **Tools** menu, or by using the command **zoom** (see its online help).

Clicking the next icon to the right with the circular arrow allows you to rotate three-dimensional (3D) graphics. For more 3D effects, select the Camera Toolbar from the **View** menu (in MATLAB 6 and higher). You can also change the view-point with the command **view**. In particular, the command **view(2)** projects a figure into the x - y plane (by looking down on it from the positive z -axis), and the command **view(3)** views it from the default direction in 3-space, which is in the direction looking toward the origin from a point far out on the ray $x = -0.5272t$, $y = -0.6871t$, $z = 0.5t$, $t > 0$.

- ✓ In MATLAB, any two-dimensional plot can be “viewed in 3D,” and any three-dimensional plot can be projected into the plane. Thus Figure 5.5 above (the helix), if followed by the command **view(2)**, produces a circle.

In MATLAB 7, clicking the next icon to the right of the rotate icon enables the Data Cursor, which allows you to display the coordinates of a point on a curve by

clicking on or near the curve. Right-clicking in the figure window gives several options; changing Selection Style to Mouse Position will allow you to click on an arbitrary point on the curve rather than just a data point. (Remember that the curves plotted by MATLAB are piecewise-linear curves connecting a finite number of data points.) This can be useful especially after zooming, because data points may then be spaced far apart. Another way to get coordinates of a point (in earlier versions as well as MATLAB 7) is by typing `ginput(1)` in the Command Window; this allows you to click on any point in the figure window, not just on a curve. While this gives you more flexibility, if you want the precise coordinates of a point on a curve, it is best to use the Data Cursor, because it will always highlight a point on the curve even if you don't click exactly on the curve.

Also in MATLAB 7 only, the rightmost icon on the Figure Toolbar opens three windows surrounding the figure; these are collectively known as Plot Tools, and you can also open them with the command `plottools`. You can also control the display of these three windows – the Figure Palette, the Plot Browser, and the Property Editor – individually from the **View** menu. These windows enable various means of editing figures. Many of these capabilities are also available in the **Insert** and **Tools** menus, and from the Plot Edit Toolbar in the **View** menu. We will briefly discuss some editing options in the *Customizing Graphics* section below, but there are many more possibilities and we encourage you to experiment with these tools.

- ✓ In earlier versions of MATLAB, as well as MATLAB 7, more limited editing capabilities are available by clicking the arrow icon to the right of the print icon on the Figure Toolbar and then right-clicking in the figure.

Combining Plots in One Window

The command `subplot` divides the figure window into an array of smaller plots. The first two arguments give the dimensions of the array of subplots, and the last argument gives the number of the subplot (counting from left to right across the first row, then from left to right across the next row, and so on) in which to put the output of the next graphing command. The following example, whose output appears in Figure 5.8, produces a 2×2 array of plots of the first four Bessel functions J_n , $0 \leq n \leq 3$.

```
>> X = 0:0.05:40;
>> for n = 1:4
    subplot(2,2,n)
    plot(X, besselj(n - 1, X))
end
```

- ✓ In MATLAB you can also create subplots using the Figure Palette, which you can enable from the **View** menu or as part of the Plot Tools described above.

☆ Customizing Graphics

- ☞ *This is a more advanced topic; if you wish you can skip it on a first reading.*

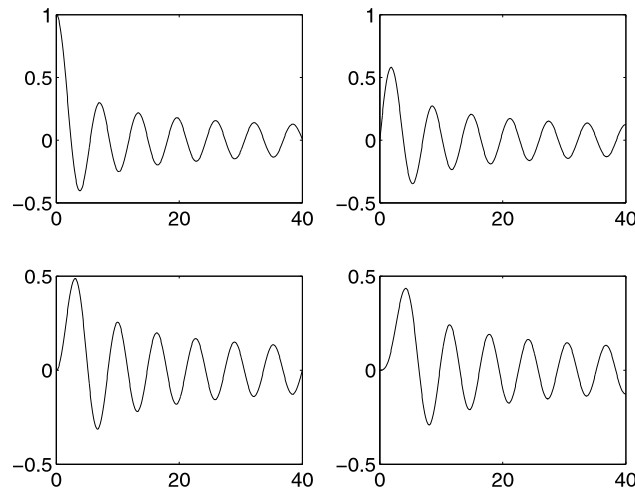


Figure 5.8. Bessel Functions $J_0(x)$ (upper left), $J_1(x)$ (upper right), $J_2(x)$ (lower left), and $J_3(x)$ (lower right).

So far in this chapter, we have discussed some commonly used MATLAB routines for generating and manipulating plots. But often, to get a more precise result, you need to customize or edit the graphics these commands produce. In order to do this, you must understand a few basic principles concerning the way MATLAB stores and displays graphics. For most purposes, the discussion here will be sufficient. But if you need more information, you may want to consult one of the books devoted exclusively to MATLAB graphics, such as *Using MATLAB Graphics*, which comes free (in PDF format) with the software and can be accessed in the “Printable Documentation” section in the Help Browser (or under “Full Documentation Set” from the **helpdesk** in MATLAB 5.3), or P. Marchand & O. Holland, *Graphics and GUIs with MATLAB*, 3rd ed., Chapman & Hall/CRC, London, 2002.

Once you have created a figure, there are two basic ways to manipulate it. The current figure can be modified by typing MATLAB commands in the Command Window, such as the commands **title** and **axis square** that we have already encountered. Or you can modify the figure with the mouse, using the menus and icons in the figure window itself. Almost all of the text commands have counterparts that can be executed directly in the figure window. So why bother learning both techniques? The reason is that editing in the figure window is often more convenient, especially when one wishes to “experiment” with multiple changes, while editing a figure with MATLAB commands in an M-file makes your customizations reproducible. So the true MATLAB expert uses both techniques. While the text commands generally remain the same from one version of MATLAB to the next, the figure-window menus and tools are significantly different in MATLAB 5.3, 6, and 7. All of these versions have a Property Editor, but it is accessed in different ways. In MATLAB 7, you can open it with Plot Tools as described above, or from the **View** menu. In MATLAB 6, select

Edit:Current Object Properties.... In MATLAB 5.3, select **File:Property Editor...**

To modify objects in the figure window with the mouse, editing must be enabled in that window. In MATLAB 6 and later, you can enable or disable editing by selecting **Tools>Edit Plot** or by clicking the arrow icon to the right of the print icon. When editing is enabled, this arrow icon is highlighted, and there is a check mark next to **Edit Plot** in the **Tools** menu. In several places below we will tell you to click on an object in the figure window in order to edit it. When you click on the object, it should be highlighted with small black squares. If this doesn't happen, then you need to enable editing.

Annotation

In order to insert labels or text into a plot, you can use the commands **text**, **xlabel**, **ylabel**, **zlabel**, and **legend**, in addition to **title**. As the names suggest, **xlabel**, **ylabel**, and **zlabel** add text next to the coordinate axes, **legend** puts a "legend" on the plot, and **text** adds text at a specific point. These commands take various optional arguments that can be used to change the font family and font size of the text. As an example, let's illustrate how to modify our plot of the lemniscate (Figure 5.3) by adding and modifying text:

```
>> title('The lemniscate  $x^2-y^2=(x^2+y^2)^2$ ', 'FontSize', ...
20, 'FontName', 'Helvetica', 'FontWeight', 'bold')
>> text(0, 0, ' \leftarrow a node, also an inflection', ...
'FontSize', 12)
>> text(0.2, -0.1, 'point for each branch', 'FontSize', 12)
>> xlabel x, ylabel y
```

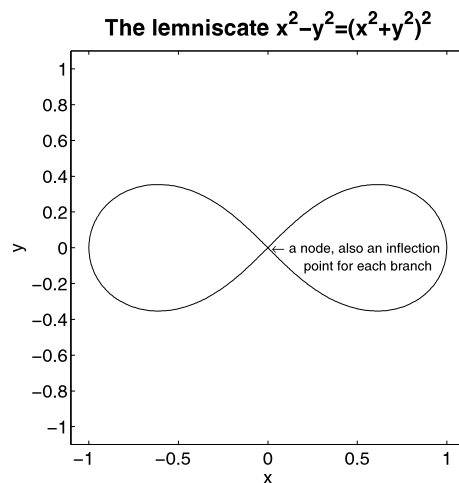


Figure 5.9. The Lemniscate from Figure 5.3 with Annotation and a Larger Title.

Notice that many symbols (such as the arrow pointing to the left in Figure 5.9) can be inserted into a text string by calling them with names starting with `\`. (If you've used the scientific typesetting program \TeX , you'll recognize the convention here.) In most cases the names are self-explanatory. For example, you get a Greek π by typing `\pi`, a summation sign \sum by typing either `\Sigma` (for a capital 'sigma') or `\sum`, and arrows pointing in various directions with `\leftarrow`, `\rightarrow`, and so on. For more details and a complete list of available symbols, see the listing for "Text Properties" in the Help Browser (you can find this listing from the "Search" tab in the Help Browser, or type `doc text` and click on "Text Properties" at the bottom of its page).

- ✓ In MATLAB 6 and later, you can insert the same types of annotation using the **Insert** menu in the Figure Window. In MATLAB 7, many more annotations are available by enabling the Plot Edit Toolbar, the Figure Palette, and/or the Property Editor. You can also use the Property Editor to change the font of a text label; click on the text you want to change, then go to the Property Editor.

Change of Plot Style

Another important way to change the style of graphics is to modify the color or line style in a plot or to change the tick marks and labeling on the axes. Within a `plot` command, you can change the color of a graph, or plot with a dashed or dotted line, or mark the plotted points with special symbols, simply by adding a string third argument for every x - y pair. Symbols for colors are '`y`' for yellow, '`m`' for magenta, '`c`' for cyan, '`r`' for red, '`g`' for green, '`b`' for blue, '`w`' for white, and '`k`' for black. Symbols for point markers include '`o`' for a circle, '`x`' for a cross, '`+`' for a plus sign, and '`*`' for a star. Symbols for line styles include '`-`' for a solid line, '`:`' for a dotted line, '`--`' for a dashed line. If a point style is given but no line style, then the points are plotted but no curve is drawn connecting them. (The same methods work with `plot3` in place of `plot`.) For example, you can produce a solid red sine curve together with a dotted blue cosine curve, marking all the local maximum points on each curve with a distinctive symbol of the same color as the plot, as follows:

```
>> X = (-2:0.02:2)*pi; Y1 = sin(X); Y2 = cos(X);
>> plot(X, Y1, 'r-', X, Y2, 'b:'); hold on
>> X1 = [-3*pi/2 pi/2]; Y3 = [1 1]; plot(X1, Y3, 'r*')
>> X2 = [-2*pi 0 2*pi]; Y4 = [1 1 1]; plot(X2, Y4, 'b+')
>> axis([-7 7 -1.1 1.1])
```

Here you may want the tick marks on the x -axis located at multiples of π . This can be done with the command `set`, which is used to change various properties of graphics. To apply it to the axes, it has to be combined with the command `gca`, which stands for "get current axes." The code

```
>> set(gca, 'XTick', (-2:2)*pi, 'XTickLabel', ...
'-2pi|-pi|0|pi|2pi', 'FontSize', 16)
```

in combination with the code above gets the current axes, sets the ticks on the x -axis to go from -2π to 2π in multiples of π , and then labels these ticks symbolically

(rather than in decimal notation, which is ugly here). It also increases the size of the labels to a 16-point font. The result is shown in Figure 5.10.

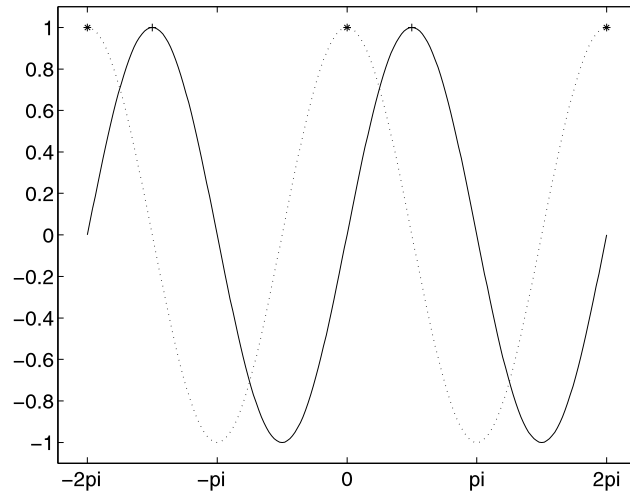


Figure 5.10. Two Periods of $\sin x$ and $\cos x$.

Incidentally, you might wonder how to label the ticks as -2π , $-\pi$, etc., instead of $-2\mathbf{p}\mathbf{i}$, $-\mathbf{p}\mathbf{i}$, and so on. This is trickier but you can do it by typing

```
>> set(gca, 'FontName', 'Symbol')
>> set(gca, 'XTickLabel', '-2p|-p|0|p|2p')
```

since in the Symbol font, π occupies the slot held by \mathbf{p} in text fonts.

- ✓ In MATLAB 7, you can again use the Property Editor to make the same types of stylistic changes. Click on a curve and go to the Property Editor to change its style, color, width, etc. In a 3D plot, you can click on a surface and see options for changing its coloring and other properties. To change the tick marks, labeling, etc., click on the axes or a blank area inside them to focus the Property Editor on the axes. In MATLAB 6, click on a curve and select **Edit:Current Object Properties...** to modify its properties, or select **Edit:Axes Properties...** to change the font for the tick labels.

Full-Fledged Customization

What about changes to other aspects of a plot? The commands `get` and `set` can be used to obtain a complete list of the properties of the objects in a figure window and then to modify them. These objects and properties are arranged in a hierarchical structure, with each object identified by a floating-point number called a *handle*. If you type `get(gcf)`, you will “get” a (rather long) list of properties of the current figure (whose handle is returned by the function `gcf`). Some of these might read

```
Color = [0.8 0.8 0.8]
CurrentAxes = [151.001]
Children = [151.001]
```

Here **Color** gives the background color of the plot in red-green-blue (RGB) coordinates, where [0 0 0] is black and [1 1 1] is white; [0.8 0.8 0.8] is light gray. Notice that **CurrentAxes** and **Children** in this example have the same value, the one-element vector containing the funny-looking number 151.001. This number is the handle of the current axes, which would also be returned by the command **gca** (“get current axes”). The fact that this handle also shows up under **Children** indicates that the axes are “children” of the figure, i.e., they lie one level down in the hierarchical structure. Typing **get(gca)** would then give you a list of axis properties, including the handles of further **Children** such as **Line** objects, within which you would find the **XData** and **YData** encoding the actual plot.

- ⇒ **In the example above, 151.001 is not the exact value of the axes handle, just its first few decimal places. So, typing `get(151.001)` would yield an error message. To retrieve the exact value of **Children** in the example above, type `get(gcf, 'Children')`. In many cases, a figure will have multiple children, in which case this command will return a vector of handles.**

Once you have located the properties you’re interested in, they can be changed with **set**. For example,

```
>> set(gcf, 'Color', [1 0 0])
```

changes the background color of the border of the figure window to red, and

```
>> set(gca, 'Color', [1 1 0])
```

changes the background color of the plot itself (a child of the figure window) to yellow (which in the RGB scheme is half red, half green).

This “one at a time” method for locating and modifying figure properties can be speeded up using the command **findobj** to locate the handles of all the descendents (the main figure window, its children, children of children, etc.) of the current figure. One can also limit the search to handles containing elements of a specific type. For example, **findobj('type', 'line')** hunts for all handles of objects containing a **Line** element. Once you have located these, you can use **set** to change the **LineStyle** from solid to dashed, etc. In addition, the low-level graphics commands **line**, **rectangle**, **fill**, **surface**, and **image** can be used to create new graphics elements within a figure window.

- ✓ In MATLAB 7, you can also see and modify a full list of properties for the figure, axes, or other object using the Property Editor. Click on the object and then click on the “Inspector...” button in the Property Editor. To select the figure itself, click on the border of the figure, outside the axes.

As an example of these techniques, the following code creates a chessboard on a white background, as shown in Figure 5.11.

```

>> white = [1 1 1]; gray = 0.7*white;
>> a = [0 1 1 0]; b = [0 0 1 1]; c = [1 1 1 1];
>> figure; hold on
>> for k = 0:1, for j = 0:2:6
    fill(a'*c + c'*(0:2:6) + k, b'*c + j + k, gray)
end, end
>> plot(8*a', 8*b', 'k')
>> set(gca, 'XTickLabel', [], 'YTickLabel', [])
>> set(gcf, 'Color', white); axis square

```

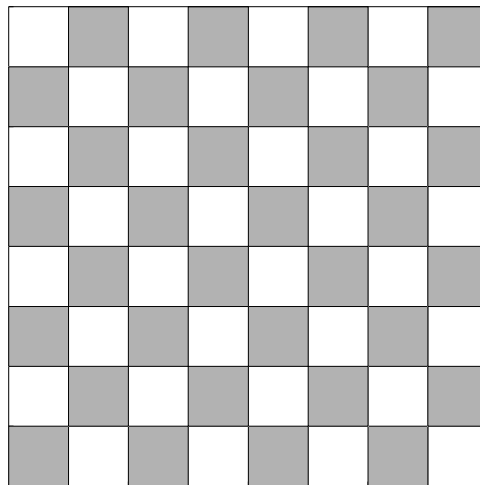


Figure 5.11. A Chessboard.

Here **white** and **gray** are the RGB codings for white and gray. The double **for** loop draws the 32 dark squares on the chessboard, using **fill**, with **j** indexing the dark squares in a single vertical column, with **k = 0** giving the odd-numbered rows, and with **k = 1** giving the even-numbered rows. Notice that **fill** here takes three arguments: a matrix, each of whose columns gives the x -coordinates of the vertices of a polygon to be filled (in this case a square), a second matrix whose corresponding columns give the y -coordinates of the vertices, and a color. We've constructed the matrices with four columns, one for each of the solid squares in a single horizontal row. The **plot** command draws the solid black line around the outside of the board. Finally, the first **set** command removes the printed labels on the axes, and the second **set** command resets the background color to white.

☆ Images, Animations, and Sound

MATLAB is also able to create and manipulate full-color images, animations, and sound files. In addition to the command-line methods described below, you can open

a media file in a format that MATLAB supports by double-clicking on it in the Current Directory Browser or by selecting **File:Import Data...**

Images

MATLAB can read, write, and edit images, such as those created by a digital camera, found on the World Wide Web, or created from within MATLAB. An image is simply a two-dimensional array of tiny colored squares called “pixels.” The image may be stored in a file in a variety of formats, including `png`, `jpeg`, and `gif`. In MATLAB, a color image with height h pixels and width w pixels is generally stored in one of two ways: as an RGB image or an indexed image. An RGB image is represented as an $h \times w \times 3$ array, so that the color of each pixel is specified by three values: a red intensity, a green intensity, and a blue intensity. (A similar type of image is grayscale, which is represented as an $h \times w$ array of pixel intensities.) An indexed image consists of an $h \times w$ array together with an auxiliary $c \times 3$ array called a “colormap”; each element in the first array represents the index of a row in the colormap, and this row gives the RGB values for the corresponding pixel. (What MATLAB calls RGB format is often called “true color” in graphics programming; an indexed image is often called “pseudocolor.”)

The command `imread` will read an image from any of a large variety of image-file formats; see the online help for supported formats. Some formats, such as `png`, can store RGB or indexed images. Other formats can store only one type or the other; `gif` images are always indexed, whereas `jpeg` images are never indexed. Most images you find on the World Wide Web will be stored in RGB format unless they are `gif` files. To read an RGB image from the file `picture.png` and store it in the array `rgbpic`, type

```
>> rgbpic = imread('picture.png');
```

You can read an indexed image with `imread` by assigning its output to two variables, one for the image array and one for the colormap. You can then convert these arrays to a single RGB array with `ind2rgb`. For example:

```
>> [indpic, map] = imread('picture.gif');
>> rgbpic = ind2rgb(indpic, map);
```

- ✓ Converting from RGB to indexed format is harder because generally the number of different colors in the image must be reduced; colormaps often have 256 or fewer colors. MATLAB’s Image Processing Toolbox has a command `rgb2ind` that offers several algorithms for making this conversion.

The command `image` displays an image in a figure window. For an RGB image `rgbpic`, simply type

```
>> image(rgbpic)
>> axis equal tight
```

The second command is not necessary, but ensures that the image is displayed in its intended aspect ratio. For an indexed image `indpic` with colormap `map`, type

```
>> image(indpic)
>> colormap(map)
```

The command `colormap` changes the colormap of the current axes or, with no input arguments, outputs the current colormap.

You can edit an image by changing the values in the image array. Notice that when you display an image, the axes are labeled with the indices of the image array. You can use the zoom feature of the figure window (see the *Figure Windows* section earlier in this chapter) to locate more precisely the indices of a particular feature of the image, or even an individual pixel. Theoretically at least, you can then edit the image in any way you want by changing the numbers in the array. Here we describe how to carry out several common manipulations in a practical manner.

- ⇒ **We will describe mainly how to edit RGB images, since this format allows you more freedom and you can always convert from indexed format to RGB as described above. However, RGB images are stored in three-dimensional arrays, which may take some time to get used to. Previously in this book we have discussed only two-dimensional arrays, and some MATLAB commands that manipulate two-dimensional arrays do not work for three-dimensional arrays.**

To reverse an image up-to-down or left-to-right, you simply need to reverse the array with `flipdim`. For example,

```
>> image(flipdim(rgbpic, 2))
```

will display a left-to-right mirror image, while `flipdim(rgbpic, 1)` reverses the array up-to-down. (For indexed images and other two-dimensional arrays, you can use the more mnemonic commands `fliplr` and `flipud` instead.)

To crop an image, select the appropriate subarray. For example, to remove 50 pixels from the top and bottom of the image and 100 pixels from the left and right, type

```
>> newpic = rgbpic(51:end-50, 101:end-100, :);
```

You can then display the cropped image `newpic` as described above, or save it as described below.

To examine the color of an individual pixel, you can display its RGB values in the Command Window and/or display it in a figure window. For example, to examine the pixel of `rgbpic` in the lower left-hand corner, type:

```
>> rgbpic(end, 1, :)
ans(:,:,1) =
    240
ans(:,:,2) =
    114
ans(:,:,3) =
    14
```

The output above gives (hypothetical) red, green, and blue values for the pixel. To see this color in the current figure window, type `image(rgbpic(end, 1, :))`. You

can then adjust the color as desired; for instance, typing `rgbpic(end, 1, 2) = 180` will increase the green intensity, making the color lighter and more yellow.

Of course, changing a single pixel will not change the appearance of the figure much, but you can also change the color of a whole block of pixels or the entire image in a similar manner. For example, to black out a rectangle within the picture, set all of the values in the corresponding subarray to 0. Thus, `rgbpic(40:60, 90:110, :) = 0` will change to black all the pixels in a 21-by-21 square centered 50 pixels from the top and 100 pixels from the left of the image.

- ✓ Changing the numbers in the array as we have just described will not change the image displayed in the figure window until you issue a new `image` command.

In addition to manipulating images that you read into MATLAB, you can create your own images to visualize numerical data. Suppose, for example, that you have an array `temp` that contains temperatures for some geographical region. You can display the temperatures as an indexed image by typing

```
>> imagesc(temp)
```

The command `imagesc` works like `image`, except that it rescales the values in a two-dimensional array so that the highest number corresponds to the highest numbered color in the current colormap and the lowest number corresponds to the lowest numbered color. With the default colormap in effect, hot regions will be colored red and cold regions will be colored blue, with other colors representing intermediate temperatures.

To display the colormap next to the image, type `colorbar`. Notice that the numbers on this bar correspond to the numbers in your original array, and are independent of the rescaling done by `imagesc`. If you want a different colormap, you can create your own or use one of the other colormaps built into MATLAB; type `doc colormap` for a selection. For example, to transform the temperature map described above into a black-and-white image where white represents hot, black represents cold, and intermediate temperatures are in shades of gray, type `colormap(gray)`.

Finally, you can save an image in one of the standard formats like `png` with the command `imwrite`. For example, to save `newpic` to the file `newpict.png`, type

```
>> imwrite(newpic, 'newpict.png')
```

- ⇒ In Chapter 3, we discussed how to save a figure in a format such as `png` using either `print` or `File:Save As...` from the figure window menu. This will save the entire figure, including the border and axis labels, into the image file. If you only want to save the image inside the axes, use `imwrite`.

Animations

The simplest way to produce an animated picture is with `comet`, which produces a parametric plot of a curve (the way `plot` does), except that you can see the curve being traced out in time. For example,

```
>> T = (0:0.01:2)*pi;
>> figure, axis equal, axis([-1 1 -1 1]), hold on
>> comet(cos(T), sin(T))
```

displays uniform circular motion.

- ✓ We used **hold on** here not to save a previous graph, but to preserve the axis properties we had just set. Without **hold on**, MATLAB would revert to its default axes, and the curve would look elliptical rather than circular as it is being traced.

For more complicated animations, you can use **getframe** and **movieview**. The command **getframe** captures the active figure window for one frame of the movie, and **movieview** (available in MATLAB 6 and later) then plays back the result in a separate window. For example, the following commands produce a movie of a vibrating string.

```
>> X = 0:0.01:1;
>> for n = 0:50
    plot(x, sin(n*pi/5)*sin(pi*X)), axis([0, 1, -2, 2])
    M(n+1) = getframe;
end
>> movieview(M)
```

The **axis** command here is important, to ensure that each frame of the movie is drawn with the same coordinate axes. (Otherwise, the scale of the axes will be different in each frame, and the resulting movie will be totally misleading.) The semicolon after **getframe** is also important, in order to avoid the spewing forth of a lot of numerical data with each frame of the movie.

- ⇒ **Make sure that while MATLAB executes the loop that generates the frames, you do not cover the active figure window with another window (such as the Command Window). If you do, the contents of the other window will be stored in the frames of the movie.**

- ✓ You can also use **movie** (which, unlike **movieview**, is available in MATLAB 5.3) to play back the movie in the current figure window. This command allows additional options, such as varying the frame rate; see the online help for details.

Once you have created a movie, you can use **movie2avi** (in MATLAB 6 and later) to save it as an AVI file, which is a standard format that can be used in other movie-viewing programs, such as Windows Media Player and QuickTime. For example, to save the movie created above to the file `string.avi`, type **movie2avi(M, 'string.avi')**.

Sound

You can use **sound** to generate sound on your computer (provided that your computer is suitably equipped). This command takes a vector, views it as the waveform of a

sound, and “plays” it. A “sinusoidal” vector corresponds to a pure tone, and the frequency of the sinusoidal signal determines the pitch. Thus the following example plays the motto from Beethoven’s Fifth Symphony:

```
>> x = (0:0.1:250)*pi; y = zeros(1,200); z = (0:0.1:1000)*pi;  
>> sound([sin(x), y, sin(x), y, sin(x), y, sin(z*4/5), y, ...  
sin(8/9*x), y, sin(8/9*x), y, sin(8/9*x), y, sin(z*3/4)]);
```

Notice that the zero vector **y** in this example creates a very short pause between successive notes.

For **sound**, the values in the input vector should be between -1 and 1 . Within that range, the amplitude of the vector determines the volume of the sound; to play the sound above at half volume, you can multiply the input vector by 0.5 . For a vector with an amplitude greater than 1 , you can use **soundsc** to rescale the vector to the range -1 to 1 before playing it.

By default, the sound is played at 8192 samples per second, so the length of the vector, divided by 8192, is the length of the sound in seconds. You can change the sample rate with an optional second argument to **sound**; this will change both the pitch and the duration of the sound you hear.

Finally, you can read and write sound files in MATLAB, but only in two formats: **wav** and **au**. More popular formats such as **mp3** are not available, but you may have software that converts other formats to and from **wav**. The commands **wavread** and **wavwrite** read and write this format; see the online help for details.