Chapter 5

Graphics

Graphical display is one of the greatest strengths of the MATLAB software, and one of its most complicated subjects. The basics are quite simple, but you also get complete control over practically every aspect of each graph, and with that power comes some complexity.

Graphical objects are classified by **type**, the most important of which are shown in Figure 5.1. The available types lie in a strict hierarchy: each data-relevant object, such as a **line** or **surface**, must lie in an **axes**, which in turn lies in a **figure** window, descended from a virtual **root**. Of the command names in Figure 5.1, the most useful is `figure`, which by itself opens a new figure window. The others, such as `line` and `surface`, are considered low-level functions and are not usually called directly. Instead you use friendlier functions that create these object types inside the most recently created or clicked-on axes. Similarly, a **group** or **series** graphics object is typically a collection of more primitive objects that are created and bundled together by a convenient routine.

Each rendered graphics object has properties that you can set at creation time or after the fact to control its appearance. You can also add axes labels, titles, and other annotations to the graph. Most graphical modifications can be done either via the command line or by using the GUI (graphical user interface). In particular, the Edit Plot button, which has the icon  , enables you to double-click or right-click virtually any graphical object to change its attributes. While there are examples of making changes in which one method is clearly preferred, for the most part the choice is a matter of personal comfort.

5.1 Data plots versus function plots

Ultimately, any plot comes down to the display of numerical values. The source of those values, however, dictates how you generate or use them. In many

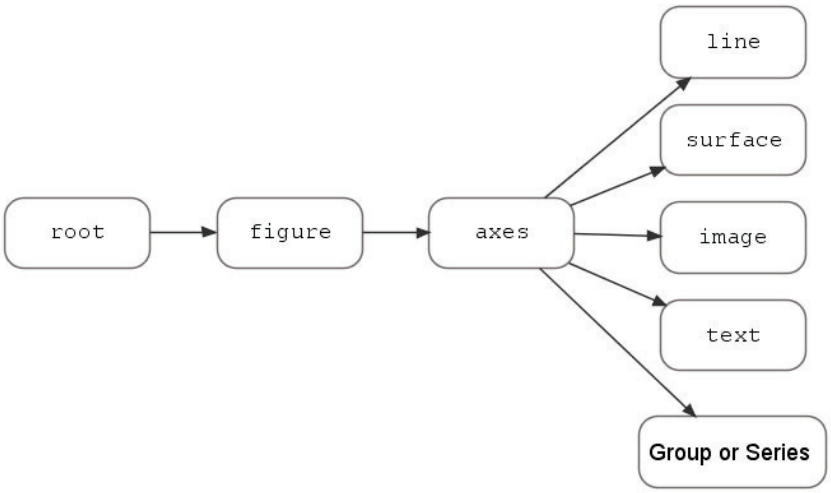

Figure 5.1. *A partial graphics object hierarchy. Except for the group/series case, each name is a graphics object type as well as a low-level command name.*

applications, the source is observed data or discrete values of an unknown function, such as the solution of a differential equation. In this case, you conceptually want to display a fixed set of points in the plane or space, possibly with additional visual information such as colors or connecting lines. The "traditional" and better-known plotting functions in MATLAB follow this model.

Sometimes, though, the source of displayed data is one or more explicitly known mathematical functions. Here one would like to work with the functions themselves and keep the possibility of evaluating them as needed to generate the points that make a nice-looking plot. MATLAB has a number of functions, all starting with the letters `ez`, for plots made in this way. The `ez` functions are the ones that work the most like the default plotting functions in *Mathematica*® and Maple® and a symbolic scientific calculator. However, keep in mind that once points are generated and rendered, the original source of them is forgotten and becomes irrelevant—so, for instance, zooming in on part of a plot will not refine the graph, as it does in many symbolic packages.

5.1.1 `ez` plots

The command `ezplot` makes two-dimensional (2D) plots of explicit, implicit, or parametric functions. These are represented mathematically by $y = f(x)$, $F(x, y) = 0$, or the pair $x = f(t)$, $y = g(t)$, respectively. You must supply `ezplot` with one or more function handles to define the plot, and define a range for the variables, as in these examples:

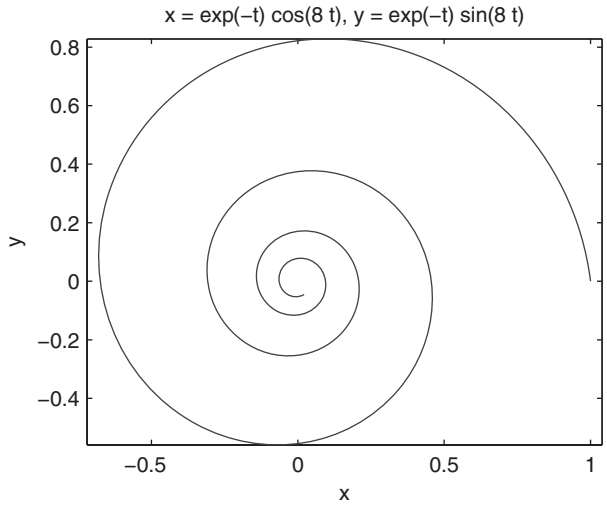

Figure 5.2. *Result of an* `ezplot` *command.*

```
>> ezplot( @sin, [0 2*pi] )
>> ezplot( @(x,y) x.^4+y.^4-1, [-1 1] ), axis equal
>> x= @(t) exp(-t).*cos(8*t);
>> y= @(t) exp(-t).*sin(8*t);
>> ezplot(x,y,[0 3])
```

The results of the last command are shown in Figure 5.2. Note that all of the anonymous function definitions must use array operators such as `.^` and `.*`. The `axis equal` command used in the second example makes the coordinates have a one-to-one aspect ratio, so that circles don't look like ellipses. The `ezpolar` command also makes a curve in the plane, given a function that defines r as a function of θ. The `ezplot3` command plots parametric space curves and can animate "walking" along the curve:

```
>> ezplot3(@cos, @sin, @(t) exp(-t/8), [0 40], 'animate')
```

One way to visualize a function of two variables is to use a contour plot, in which a family of curves $F(x,y) = c$ is plotted in the x-y plane for multiple choices of c. There are two `ez` variants for this:

```
>> ezcontour(@(x,y) 4*x.^2-x.*y+y.^2, [-1 1 -2 2] )
>> ezcontourf(@(x,y) sin(3*x-y).*sin(x+2*y), [-pi pi])
```

The second form fills each intercontour region with a different solid color for a more dramatic plot.

For an explicit surface $z = F(x,y)$, or a parametric surface in which x, y, and z each is a function of parameters (u,v), use `ezsurf`. A wireframe plot is produced if you use `ezmesh`.

```
>> ezsurf( @(x,y) x.^2-y.^2-1, [-1 1 -1 1] )
>> x=@(u,v) cosh(u).*cos(v); y=@(u,v) sinh(u).*cos(v);
>> z=@(u,v) sin(v);  ezmesh( x,y,z, [-1 1 0 2*pi] )
```

In both types of plots, color is used as an additional cue to indicate the value of the z coordinate.

You can find many more details and options in the online documentation than are covered here.

5.1.2 Two-dimensional data plots

For discrete data in two dimensions, use `plot`. For example, MATLAB comes with U.S. census data from 1790–1990:

```
>> load census
>> plot(cdate,pop)
```

The plot simply connects dots at the points defined by the values in the two coordinate vectors. You can change the line style graphically after the fact or at the time of creation, as in

```
>> plot(cdate,pop,'ro')  % red circles
>> plot(cdate,pop,'k:')  % black dotted lines
>> plot(cdate,pop,'ys-') % yellow squares and lines
```

The default is for any new plot to replace the existing one. To add curves to the current axes instead, use `hold on`.

This particular data set looks a bit better when the vertical axis is logarithmically scaled:

```
>> semilogy(cdate,pop,'p-')
```

Here you can see the data very nearly represents exponential growth at least up to 1860. We can add a crude exponential extrapolation from the 1900 data as follows:

```
>> t = 1900:10:2030;
>> hold on
>> semilogy( t, exp(-20.44+0.01306*t), 'k--' )
```

The result is shown in Figure 5.3. Note that the axes limits that frame the data are chosen automatically to be "nice" numbers. If instead you want to minimize the whitespace, use `axis tight` to make the frame as small as possible.

Other useful 2D plotting commands are given in Table 5.1. See a bunch more by typing `help graph2d`. Also explore the figure window buttons and menus, which should be fairly self-explanatory, and try right-clicking on objects to modify their appearance.

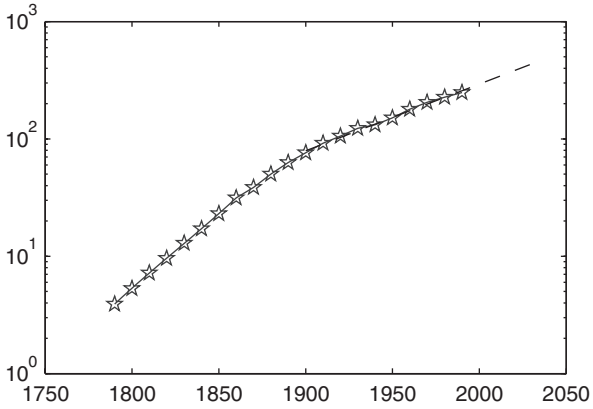

Figure 5.3. *Semilog plot of U.S. census data, and an extrapolation.*

Table 5.1. *2D plotting commands.*

`figure`	Open a new figure window.
`subplot`	Create multiple axes in one figure.
`semilogx`, `semilogy`, `loglog`	Scale axes logarithmically.
`axis`, `xlim`, `ylim`	Set axes limits.
`legend`	Create legend for multiple curves.
`print`	Send to printer.

5.1.3 Three-dimensional data plots

Plots of surfaces and the like for functions $z = f(x, y)$ operate on the connect-the-dots principle, but the details are more difficult. The first step is to create a grid of points in the x-y plane where f is evaluated to obtain the "dots" in three dimensions. This is most easily done using `meshgrid`.

Here is a typical example:

```
>> x = pi*(0:0.02:1);
>> y = 2*x;
>> [X,Y] = meshgrid(x,y);
```

These arrays define the coordinates of a 2D grid, as can be seen by using

```
>> plot(X,Y,'k.')
```

With this grid definition, we can define some height data in the z direction by array operations:

```
>> Z = sin(X.^2+Y);
>> surf(X,Y,Z)
```

Table 5.2. *3D plotting commands.*

`surf`, `mesh`, `waterfall`	Create surfaces in three dimensions.
`colorbar`	Show color scaling.
`plot3`	Plot curves in space.
`pcolor`	Show top view of a colored surface.
`contour`, `contourf`	Create contour plot.

The array `Z` is actually an array of values of $f(x,y) = \sin(x^2 + y)$ evaluated on the grid. Finally, `surf` makes a solid-looking surface in which color and apparent height describe the given values of f. An alternative command `mesh` is similar to `surf` but makes a "wireframe" surface. The most common 3D plotting commands are shown in Table 5.2.

Because of mathematical conventions, the output of `meshgrid` is an endless source of confusion. The problem is that for the graph of a function $f(x,y)$, the first variable is interpreted as horizontal and the second variable as vertical. In an array, A, however, the first index dimension is vertical (row) and the second is horizontal (column).[8] The function `meshgrid` and the 3D plotting functions adopt the *graphical* convention. Hence, using the definitions above, the grid point (`x(i)`,`y(j)`) would be found in the arrays above as (`X(j,i)`,`Y(j,i)`). You might choose to avoid this issue by using `ndgrid`, which adopts the array convention, instead of `meshgrid`, but then all calls to `surf` and its ilk need to have transposes on all the arrays, or the graph itself will have x and y transposed.

It's also useful to realize that, while the data must be given at points on a mesh, the mesh has to be *logically* rectangular, not literally rectangular. For instance, you can plot data on the unit disk as follows:

```
>> [R,T] = meshgrid( 0:.02:1, pi*(-1:.05:1) );
>> X = R.*cos(T);  Y = R.*sin(T);
>> pcolor(X,Y,X.^2-Y.^3), axis equal
```

Any surface parameterized by two variables defined on a rectangle can be visualized in the same way.

5.2 Annotation

Graphs of data usually need labels and maybe a title. For example:

```
>> ezplot(@(t) exp(-t/5).*sin(t), [0 6*pi])
>> xlabel('time')
>> ylabel('amplitude')
>> title('Damped oscillator')
```

[8] An additional complication, ignored by the plotting routines, is that the vertical direction meaning "increasing" is reversed in the two interpretations. For plots that represent a matrix, use `axis ij` to reverse the vertical axis.

These can also be added and edited by enabling the Edit Plot button and double-clicking the axes. You can also add legends, text, arrows, or text/arrow combinations, to help label data. See the Insert menu on the figure window.

MATLAB supports a limited subset of TEX processing in text strings, including axes labels and titles. This is an easy way to get Greek letters, superscripts, and symbols.

```
>> xlabel('time \tau')
>> legend('{\it e}^{-\tau/5} sin(\tau)')
```

You can get even more elaborate mathematical annotation by requiring a LATEX interpreter for your string, as explained in the documentation. This route can be effective for a stand-alone figure; ironically, though, it may not produce results that blend well with a LATEX document, since such documents are usually customized with respect to fonts and macros. Instead, you might consider the excellent psfrag package for LATEX, which allows you to replace strings in an EPS figure with arbitrary boxes of processed text from within your document.

5.3 Handles and properties

Every rendered object has an identifier or **handle**. The functions `gcf`, `gca`, and `gco` return the handles to the active figure, axes, and object (usually the most recently drawn or clicked-upon). The handle of a figure is always the figure number; other handles have arbitrary values.

Properties can be accessed and changed at the command line by the functions `get` and `set`, or graphically (see the Plot Edit Toolbar, Plot Browser, and Property Editor in the figure's View menu). Here is just a taste of what you can do:

```
>> h = plot(t,sin(t))
>> set(h,'color','m','linewidth',2,'marker','s')
>> set(gca,'pos',[0 0 1 1],'visible','off')
```

Handles also make it easy to change entire batches of objects at once. For example, the following code makes all of the blue lines in the current figure have width equal to 3:

```
>> h = findobj(gcf,'type','line','color','b');
>> set(h,'linewidth',3)
```

Because of handles, plots in MATLAB can be created in a basic form and then modified to look exactly how you want. However, it can be useful to change the default property values that are initially used to render an object. You can do this by resetting the defaults at any level above the target object's type in the hierarchy of Figure 5.1. For instance, to make sure that all future text objects in the current figure have font size 10, enter

```
>> set(gcf,'defaulttextfontsize',10)
```

Table 5.3. *Common colors in RGB and as string abbreviations.*

Color	RGB vector	Abbreviation
Black	`[0 0 0]`	`'k'`
Red	`[1 0 0]`	`'r'`
Green	`[0 1 0]`	`'g'`
Blue	`[0 0 1]`	`'b'`
Yellow	`[1 1 0]`	`'y'`
Magenta	`[1 0 1]`	`'m'`
Cyan	`[0 1 1]`	`'c'`
White	`[1 1 1]`	`'w'`

All figures are also considered to be children of the virtual `root` object that has handle `0`, so setting properties to this handle creates global defaults.

5.4 Color

The coloring of lines and text is easy to understand. Each object has a color property that can be assigned an RGB (red, green, blue) vector in which each of the three entries is between zero and one. The primary and secondary colors are shown in Table 5.3, along with their one-letter string abbreviations. Note that green does not show well against a white background; MATLAB often uses a darker green color given by `[0 0.5 0]`.

Surfaces are colored differently. There are two aspects, both modifiable: coloring of the data points, and coloring of the patches or mesh lines between them. The data point colors are specified by a `CData` property of the surface object. This could be a specification of RGB values at each individual data point, which is called the **truecolor model**. It is best for photographs and bitmap images.

The more common model for abstract data, and the default when you specify no explicit color information, is the **indexed color model**. This model involves an interaction between the `CData` property of the surface, the `CLim` property of its parent axes, and the `Colormap` property of the axes' parent figure. The figure's `Colormap` is an $m \times 3$ array, with each row interpreted as RGB values. The `CLim` property is a vector `[a b]` that defines a linear transformation from the interval $[a,b]$ to the interval $[1,m]$. Thus, each value in the `CData` property is mapped according to the linear transformation, then rounded to the nearest integer in $\{1,2,\ldots,m\}$, which serves as a row index into the colormap to determine a color.

As with all graphics object properties, those named above can be changed using `set`. However, there are more commonly used alternatives. By default, the `CData` surface property is equal to the array of z coordinate values (`ZData`) for the surface, but it can be set by giving a fourth argument to the plotting functions `surf` and `mesh`. The `CLim` is set by default to include exactly the full range of data values, and it can be changed after the fact by using `caxis`. The default figure `Colormap` is named `jet` and varies from blue to green to red. It can be

Table 5.4. *Shading models for surfaces.*

`shading flat`	Each face or mesh segment has constant color, determined by one boundary point.
`shading faceted`	Flat shading is used for faces and black for edges.
`shading interp`	Color in each face or segment is found by linear interpolation of corners.

changed by using the `colormap` command. Any changes to these properties, however made, have instant effects on the colors:

```
>> [X,Y,Z] = peaks;           % some built-in data
>> surf(X,Y,Z)
>> colorbar                   % show data->color mapping
>> caxis                      % current CLim value of axes
ans =
-6.5466    8.0752

>> caxis([-8 8]), colorbar    % zero-level symmetry
>> colormap pink              % change figure colormap
>> colormap gray              % change figure colormap
>> colormap(flipud(gray))     % reverse ordering
```

You can specify explicit `CData` for a surface in the indexed model. One natural use of this is for functions of a complex variable:

```
>> [T,R] = meshgrid(2*pi*(0:0.02:1),0:0.05:1);
>> [X,Y] = pol2cart(T,R);
>> Z = X + 1i*Y;
>> W = Z.^2;
>> surf(X,Y,abs(W),angle(W)/pi)  % arg(W) for coloring
>> axis equal, colorbar
>> colormap hsv                  % periodic colormap
```

Between grid points, color is determined by **shading**. You can change the shading of a surface by calling `shading` after the surface is created. The available types are shown in Table 5.4. While interpolated shading makes much smoother colors and prettier pictures, it can be very slow to render, particularly on printers. In fact, it's often faster to interpolate the data yourself to a finer grid and print with flat shading. See `interp2` to get started on this.

5.5 Saving and exporting figures

It often happens that a figure needs to be changed long after its creation. You can simply save the figure-creation commands in a script (section 3.1), but this approach has drawbacks. If the plot data takes a long time to generate, rerunning the script will waste time. Also, any edits made through menus and buttons will be lost.

Instead, you can save a figure in a format that allows it to be recreated in future sessions. Just enter

```
>> saveas(gcf,'myfigure.fig')
```

to save the current figure in a file named myfigure.fig. (This is equivalent to Save as... on the figure's File menu.) Later you can enter `open myfigure.fig` to recreate it. Since graphics objects store the data that defines them (in the `XData`, `YData`, and `ZData` properties), this tactic is also a way of storing the data used to create the graph. You can even leave notes about the data in an object's `UserData` field.

Ultimately, you will want to use some figures in a presentation or publication, a process that involves **exporting** the figure to a graphics file.[9] In this process, what you see on the screen is not necessarily what you get in the file, because MATLAB tweaks graphs differently depending on the output device. The relevant changes are laid out as a GUI if you select Export setup... from a figure's File menu. This dialog also lets you save a collection of preferences as a named style, so the GUI might be the most convenient approach if it meets your needs. But all of the options it offers affect properties that can be set using the handles of the figure and its descendant objects, should you prefer to work with commands.

Here are some significant issues to consider when exporting a figure:

Size. At first the size of the output graphic might seem irrelevant, since any document preparation or presentation program will let you change the image's size in the final product. However, resizing after exporting scales everything in the figure, including text, sometimes leading to graphs with thin lines or comically small font sizes. You are better off using MATLAB to set figure size, line widths, and font sizes independently before exporting.

Format. The most important difference in graphics file formats is between **vector** formats (such as EPS, PDF, and EMF) and **bitmap** formats (such as GIF, JPEG, PNG, and TIFF). Bitmaps, which store a pixel-by-pixel snapshot, are ideal for photographs but less so for most other scientific applications. These formats fix the resolution of your image forever, whereas the resolution of your screen, your printer, and a publication's printer are all very different. Vector formats, which represent lines, text, and surfaces more abstractly, are usually a much better choice for data-driven graphics.

EPS (encapsulated PostScript) files are the best choice for inclusion by LaTeX documents. They contain bounding box information that allows LaTeX to know exactly how large the graphic is. If you wish to export from the command line, you can use

```
>> print -deps myfig
```

[9] Depending on what tools you use, it may also be possible to copy and paste images, which is often adequate, at least on some platforms. However, you still might need to set up the figure first in the same ways discussed here.

or, for color output,

```
>> print -depsc myfig
```

to create `myfig.eps` from the current figure. However, this method uses the `PaperPosition` property of the figure, not the position on the screen, to determine the size of the output graphic. The EPS format also works with Microsoft Word™ if you print the output on a PostScript printer. In this case it's handy to use

```
>> print -deps -tiff myfig
```

in which case Word will be able to show a crude preview version of the graph in the document onscreen.

An additional twist in LaTeX is that if the file is processed directly to PDF format using pdflatex, EPS graphics cannot be used. Instead, you should use PDF for vector graphics. Currently, MATLAB-generated PDFs do not seem to embed graphic bounding box information in a way that the graphicx package for LaTeX understands. To work around this limitation, you can use the widely available pdfcrop script, or save the file as EPS and use the free script epstopdf to convert it to PDF.

Color. Colored lines are automatically converted to black when saved in a non-color format, so you should distinguish them by other features, such as symbol or line style. The colors of surfaces may be converted to grayscale, which presents a problem. The default colormap has colors ranging from deep blue to deep red, and the hues signal real information about the surface. When these colors are converted to grayscale, the distinction between red and blue is not clear, and the shades become meaningless. For a grayscale surface plot, you should consider using `colormap(gray)` or `colormap(flipud(gray))`, whichever gives less total black, before exporting the figure, so that saturation corresponds to data. Finally, the edges of wireframe surfaces created by `mesh` are also converted to gray, often with poor results. Make all the lines black by entering `colormap([0 0 0])`.

5.6 Other common graphics techniques

To create an array of plots in one figure window, use `subplot`. For example:

```
for n = 1:6
  subplot(2,3,n), ezpolar(@(t) sin(n*t), [0 2*pi])
  xlabel(''), title(['r = sin(',int2str(n),'t)'])
end
delete(findobj(gcf,'type','text'))  % generic labels
```

The call `subplot(m,n,k)` selects the kth axes in an $m \times n$ array, counting from left to right, top to bottom.

Here is a way to make a dynamic graph or animation:

```
figure('doublebuffer','on')   % flash-free redraws
t = linspace(0,8*pi,800)';
for s = 0:0.01:1
  x = exp(-s*t).*cos(6*s*t+t);
  y = exp(-s*t).*sin(6*s*t+t);
  plot(x,y), axis ([-1 1 -1 1])
  pause(0.01)
end
```

Graphics commands accumulate in a buffer and do not take effect immediately. This fact allows you to make multiple changes (for instance, data and title) between animation frames. You must use a `pause` or `drawnow` command whenever you want to force all pending graphics commands to be processed.

The animation technique is best when little time is needed to render each frame. An alternative is to make a movie of bitmap renderings of the frames, using `getframe` to store each frame. The movie can be played back with `movie` or converted to a video file using `movie2avi`.

Generally, when a graphical function encounters a `NaN` value in a list of points, it quietly omits drawing the point. This can be useful when combined with the technique of index masking (see section 6.3). For example, these commands plot a surface that is mathematically defined only over an L-shaped region:

```
>> [X,Y] = meshgrid( -1:1/18:1 );
>> L = membrane(1,18,8,8);
>> surf(X,Y,L)
```

For this graph, the z values outside the domain are set to zero, which is not bad since the function happens to be zero at the real boundary. But the graph is improved if we erase that part of the surface using `NaN`:

```
>> outside = (X < 0) & (Y > 0);   % logical index mask
>> L(outside) = NaN;              % note "scalar expansion"
>> surf(X,Y,L)
```

Exercises

5.1. On a single graph, make a plot of the functions sinh, cosh, and tanh for $-1 \le x \le 1$. Give each curve a different color and label them with text and arrows.

5.2. Recall the identity

$$e = \lim_{n\to\infty} r_n, \qquad r_n = \left(1+\frac{1}{n}\right)^n.$$

Make a standard and a log-log plot of $e - r_n$ for $n = 5, 10, 15, \ldots, 500$. What does the log-log plot reveal about the asymptotic behavior of $e - r_n$ as $n \to \infty$?

5.3. Here are two different ways of plotting a sawtooth wave. Explain concisely why they behave differently:

```
>> x = [0:7;1:8];  y = [zeros(1,8);ones(1,8)];
>> subplot(121), plot(x,y,'b'), axis equal
>> subplot(122), plot(x(:),y(:),'b'), axis equal
```

(The first version is more mathematically proper, but the second is more likely to appear in print.)

5.4. Play the "chaos game." Let P_1, P_2, and P_3 be the vertices of an equilateral triangle. Start with a point anywhere inside the triangle. At random, pick one of the three vertices and move halfway toward it. Repeat indefinitely. If you plot all the points obtained, a very clear pattern will emerge. (Hint: This is particularly easy to do if you use complex numbers. If `z` is complex, then `plot(z)` is equivalent to `plot(real(z),imag(z))`.)

5.5. a. Generate 100 random matrices using `randn(100)`, and plot all of their eigenvalues as dots in the complex plane on one graph. (Thus, you should see 10,000 dots.) Use `axis equal` to make the aspect ratio one-to-one. You should see a fairly striking result.

b. Repeat the experiment with 100 random complex matrices of the form `complex(randn(100),randn(100))`. You should be able to see one very clear qualitative difference between the previous case and this one.

5.6. Make surface plots of the following functions over the given ranges:

1. $(x^2+3y^2)e^{-x^2-y^2}$, $-3 \le x \le 3$, $-3 \le y \le 3$.
2. $-3y/(x^2+y^2+1)$, $|x| \le 2$, $|y| \le 4$.
3. $|x|+|y|$, $|x| \le 1$, $|y| \le 1$.

5.7. Make contour plots of the functions in the previous exercise.

5.8. Make a contour plot of

$$f(x,y) = e^{-(4x^2+2y^2)}\cos(8x) + e^{-3((2x+1/2)^2+2y^2)}$$

for $-1.5 < x < 1.5$, $-2.5 < y < 2.5$, showing only the contour at the level $f(x,y) = 0.001$. You should see a friendly message.

5.9. Plot the surface represented by

$$x = u(3+\cos(v))\cos(2u),\ y = u(3+\cos(v))\sin(2u),\ z = u\sin(v) - 3u$$

for $0 \le u \le 2\pi$, $0 \le v \le 2\pi$.