Roger A. Green

Getting a handle on MATLAB graphics

AMONG engineers and scientists, MATLAB is one of the most popular computational packages. Part of MATLAB's popularity stems from its simple, yet sophisticated, graphics capabilities. While basic plots are relatively easy to obtain, specialized plots require a little more effort to produce.

This article introduces MATLAB's Handle Graphics, which provide a mechanism to fully control and customize graphics objects in MATLAB. With a basic understanding of Handle Graphics, users can produce plots that meet the unique needs and quality standards commonly required by the profession. Several examples are presented to illustrate the concepts.

Preliminaries

Before diving headfirst into MATLAB plotting, a few preliminaries are in order. Many readers may safely skip this section and move straight to the section "Generic plotting."

First, due to its ubiquity in engineering environments, its powerful features, and its refined interface, this article discusses techniques that are specific to MATLAB. However, excellent alternatives exist that possess similar functionality. In particular, Scilab is an open-source platform that is very similar to MATLAB, including its use of object-oriented graphics. Many of the topics discussed in this article can be achieved in similar fashion with Scilab. Furthermore, Scilab is available free of charge over the Internet at <http://www.scilab.org/>.

Second, while some MATLAB programming experience is helpful to fully understand this article, novice users can appreciate much of the discussion if a few basics of MATLAB programming are covered. While a comprehensive introduction is not intended or feasible, the following overview covers the essential MATLAB structures found in the upcoming examples.

To begin, MATLAB is a high-performance language for technical computing that integrates computation, visualization, and programming into a single, easy-to-use environment. In the most basic operation, commands are issued at the MATLAB command prompt (`>>`), and MATLAB responds with an action. Help on any MATLAB command is easy to obtain by simply typing `help` followed by the command name. Integrated, fully searchable, browser-based help provides more comprehensive assistance when needed.

MATLAB offers a wide range of built-in functions. Enter these function names with the appropriate arguments, and MATLAB evaluates them. Arguments can be scalars, vectors, and in some

0278-6648/07/$25.00 © 2007 IEEE

Table 1. Example MATLAB functions.

`besselj`	Bessel function of the first kind
`cos`	Cosine function, radian measure
`length`	Determine the length of a vector
`max`	Largest elements in an array
`num2str`	Convert number to a string
`randn`	Standard normal random numbers
`sin`	Sine function, radian measure

Table 2. Example MATLAB plot-related commands.

`grid`	Add grid to current axes
`hist`	Compute and/or plot histogram
`legend`	Graph legend for lines/patches
`plot`	Generate 2-D line plot
`xlabel`	Label *x*-axis
`ylabel`	Label *y*-axis

Table 3. MATLAB special characters.

`:`	Create vectors, subscript arrays
`()`	Pass arguments, prioritize operators
`[]`	Construct array, concatenate elements
`...`	Continue statement to next line
`,`	Separate rows/function arguments
`;`	Separate columns, suppress output
`' '`	Construct string/character array

cases matrices. Table 1 provides some example functions that are utilized in the upcoming examples.

Once suitable data is generated, MATLAB provides a variety of commands to easily generate and annotate plots. Table 2 details some plot-related commands that are utilized in the upcoming examples.

To effectively utilize MATLAB functions and commands, several special characters and constructs are indispensable, including those listed in Table 3. One particularly important application of the `:` notation is the generation of a vector of equally spaced numbers. This is accomplished by typing `(a:b:c)`, where `a` is the start value, `b` is step size, and `c` is the termination condition. For example, `(0:0.5:1.75)` generates the length-4 vector [0, 0.5, 1.0, 1.5].

As a programming language, MATLAB supports a wide range of general-purpose structures such as for-loops, while-loops, and switch statements. These structures, which are accessed with the `for`, `while`, and `switch` commands, are terminated in each case with the `end` command.

Now acquainted with these essential MATLAB commands and structures, we are ready to proceed to the primary topic: improved MATLAB plotting.

Generic plotting

Pick up the proceedings from almost any engineering conference, and you are bound to find it chock-full of MATLAB plots. Chances are also good that many of those plots are quite difficult to read. Generic commands tend to produce generic plots.

To highlight some of the inadequacies typical of generic plotting, let us consider three examples in roughly increasing order of complexity. Each is easy to produce with standard MATLAB commands, and, in each case, standard commands produce less than satisfying results.

The first example is a histogram of 1,000 observations of a standard normal random variable $Z \sim N(\mu = 0, \sigma^2 = 1)$, the plotting for which is as follows:

```
01    z = randn(1000,1);
02    BinCenters = (−2.5 : 2.5);
03    hist(z,BinCenters);
04    xlabel('z'); ylabel('Count');
```

Looking at the code, the first line creates a 1000×1 vector `z` of observations from a standard normal distribution using the built-in function `randn`. Actually, `randn` is a pseudorandom number generator. Pseudo-random numbers are deterministic sequences that mimic the behavior of random variables. The same exact 1,000 observations presented in this article can be recreated by preceding line 1 with the command `randn`('state',0). The second line creates a length-6 vector that specifies the histogram bin centers, [−2.5, −1.5, −0.5, 0.5, 1.5, 2.5]. In this case, these centers ensure the `hist` command in line 3 sorts the 1,000 elements of vector `z` into bins that span one standard deviation each, starting at zero. Line 3 is also responsible for generating the histogram figure itself. Line 4 adds appropriate axis labels to the plot.

Although straightforward, the resulting plot that is shown in Fig. 1 is not particularly good. The standard two-column format of most conference proceedings causes plot features such as font size to shrink severely, which compromises readability. The default bar color, which cannot be changed in the `hist` command, is a dark blue that shows up nearly black when printed with black-and-white printers. Although the general trends of the histogram are clear, it is nearly impossible to determine the exact count in any particular bin. We cannot tell, for example, exactly how many observations are within one standard deviation of the mean.

The second plot is of a pair of quadrature sinusoids with normalized radian frequency taken over a full period, as can be seen in the following:

```
01    t = (0:0.01:2*pi);
02    x = cos(t); y = sin(t);
03    plot(t,x,'k-',t,y,'k--'); grid;
04    xlabel('t'); ylabel('Amplitude');
05    legend('cos(t)','sin(t)');
```

In this example, the first line of code creates a time vector for a single period, $(0 \le t < 2\pi)$, using a step size of $\Delta t = 0.01$. Line 2 creates the sinusoids $x(t) = \cos(t)$ and $y(t) = \sin(t)$. The `plot` command plots `x` as a black `(k)` solid `(-)` line and `y` as a black `(k)` dashed `(--)` line, both as functions of the time vector `t`. A grid is added as well as axis labels and a legend. The resulting plot is shown in Fig. 2.

As in the first example, plot features shrink severely when sized for a two-column format. Line weights are too light. The grid lines in particular are barely visible on an original print, let alone a photocopy. The horizontal axis grid lines are not spaced to help visualize the $\pi/2$ lag between waveforms, and the vertical axis grid lines are unnecessarily dense. The sinu-

soids touch the upper and lower portions of the plot box, giving a crowded appearance. The horizontal axis extends beyond the computed data, leaving wasted blank space. The plot legend is not only difficult to read, but it obscures the data. Overall, the plot is pretty miserable.

The third example attempts to reproduce a Bessel function plot from Chapter 5 of the communications systems text by Carlson et al. Unlike the histogram in the first example or the sinusoids in the second example, Bessel functions are difficult to accurately sketch by hand. Fortunately, MATLAB makes plotting them simple, as shown in the following example:

```
01 beta = (0:0.1:15); n = [0:3,10];
02 for i=1:length(n),
03       J(i,:) = besselj(n(i),beta);
04 end
05 plot(beta,J);
06 xlabel('beta'); ylabel('J(n,beta)');
07 legend('n=0','n=1','n=2','n=3','n=10')
```

Here, the first line establishes the argument of the Bessel functions, $(0 \leq \beta \leq 15)$, as well as the Bessel function orders to be plotted, $n = [0, 1, 2, 3, 10]$. Iterating over n, the for-loop uses the built-in MATLAB function `besselj` to evaluate $J_n(\beta)$, the desired Bessel functions of the first kind. It is tedious to manually plot each curve, so the `plot` command passes a matrix argument `J` so as to produce each curve simultaneously. Finally, axis labels and a legend are added.

As shown in Fig. 3, the resulting plot again suffers from being crammed into a two-column format: lines are too thin and fonts are too small. A more serious problem occurs when trying to identify particular curves. When plotting a family of curves simultaneously, as done in this case, MATLAB distinguishes the curves using a default color sequence. When such plots are exported to a black-and-white document, one of two things generally happens: 1) the colors are replaced with black lines, as happened in the current case, or 2) the colors, when printed in black and white, produce various difficult-to-distinguish shades of gray; in some cases, these gray lines are so light that they hardly appear on paper at all. Either case is unacceptable as it is impossible or very difficult to distinguish individual curves.

There is no escaping that Figs. 1, 2, and 3 are generic plots with significant deficiencies.

Handling plot customization

We know our generic plots need customization. The question, then, is "How do we handle plot customization effectively?" In MATLAB, the answer is exactly that: a handle.

Every MATLAB graphic you produce is comprised of various objects. These objects possess properties that can be easily modified, if only you know how to access them. Handles are numbers that uniquely identify every graphic object you create. By knowing an object's handle, you can easily access that object's properties and also modify those properties. Figure windows are objects with particularly easy-to-know handles: the handle for Figure X is just the integer X.

Objects are often contained within another object. An axes object, for example, is contained within a figure object. In this case, the axes is considered a child object of the parent figure object. If you delete the parent, all of its children disappear too. It is important to understand object hierarchy and basic object types.

The parent of everything is MATLAB's root, which is identified by handle 0. Figures come next, each with integer-valued handles. As expected, any figure is a child of the root. Axes are found in figures, which, in turn, are comprised of objects such as line, patch, surface, rectangle, image, light, and text objects. Other objects exist, such as those used for graphical

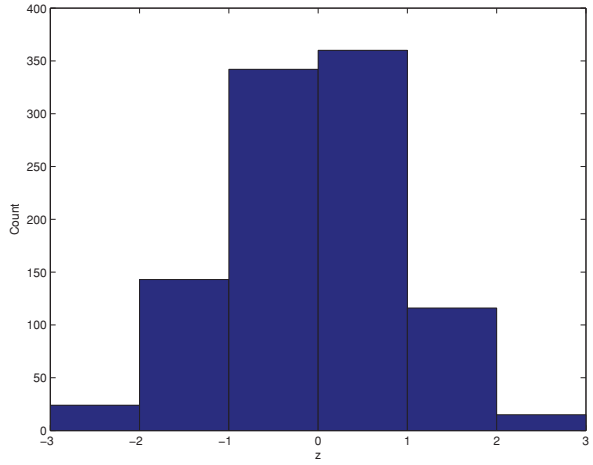

Fig. 1 The generic histogram plot is not particularly good.

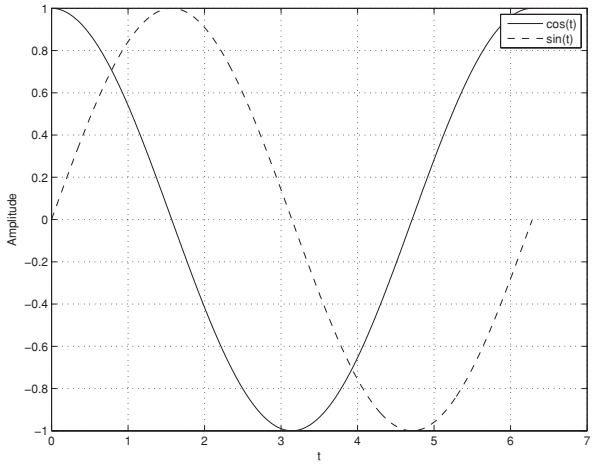

Fig. 2 The generic plot of two quadrature sinusoids shrinks severely when sized for a two-column format.

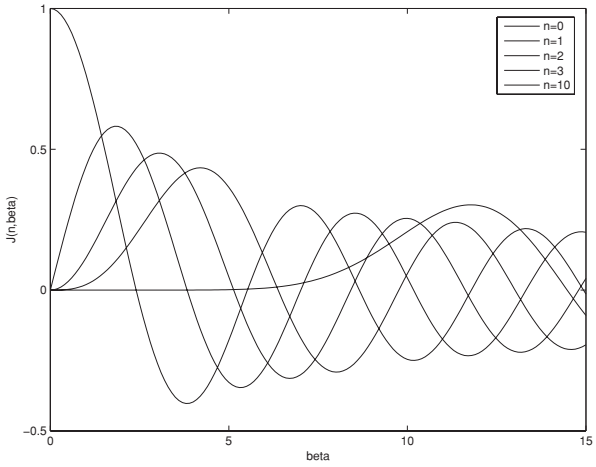

Fig. 3 The generic plot of $J_n(\beta)$ for various n suffers from the two-column format:lines are too thin and fonts are too small.

user interface (GUI) applications, which will not be covered in this article. For typical plotting applications,

$$\text{root} \supset \text{figure} \supset \text{axes} \supset \left\{ \begin{array}{c} \text{line} \\ \text{patch} \\ \text{surface} \\ \text{rectangle} \\ \text{image} \\ \text{light} \\ \text{text.} \end{array} \right.$$

For most plots, understanding the root, figure, axes, line, and text objects are all that is needed.

Knowing what objects exist is one thing; knowing how to create them and how to modify their properties is another. Fortunately, basic creation is simple. Except for root, which always exists when MATLAB is running, you can always create an object by simply typing the object name at the MATLAB command prompt. Type `figure` and a figure object is created. Type `line` and a line object is created in the current plot. In a twist on biology, if you try to create a child when no suitable parent exists, MATLAB just creates the parents needed by the child. For example, typing `line` when no graphics objects exist will not only create the desired line but also the parent axes and figure objects. Perhaps we can now answer the age-old question, "Which came first, the chicken or the egg?"

An object's handle is returned if you simply assign a variable during creation of the basic object types. For example, typing `LineHandle = line` creates a line object and assigns the line's handle to the variable `LineHandle`.

MATLAB help provides detailed descriptions and syntax for each of these commands. For example, typing `help axes` tells you about the `axes` command as well as axes objects. Of course, many generic MATLAB commands, such as `plot` and `hist`, create graphics objects during the course of their execution. Some of these commands can return a handle (e.g., `Handle = plot(0)` creates a plot with a dot at (1,0). The dot is a line object whose handle is assigned to the variable `Handle`), others cannot (e.g., `hist` cannot output a handle).

The `findobj` command can find handles if you know what you are looking for. For example, typing `findobj ('type','line')` returns handles for all visible line objects in root. Actually, you can hide an object's handle so that the `findobj` command will not find it, a topic we will not cover here.

Of course, you can change `'line'` into whatever object you desire. For those desiring the power to destroy, passing a handle to the `delete` command destroys the corresponding object and all of its children. Like real life, you can really wreak havoc by deleting objects. Unlike real life, your power of destruction in MATLAB is somewhat limited because you cannot delete the root.

Ready? GET, SET, go!

Now that we know about graphics objects and their handles, we need just a few commands to effectively use them. There are two really important commands, so important, in fact, that they deserve their own line

`get` and `set`.

The `get` command is used to get information. The `set` command is used to set properties, although it can also be used to get information on how to set something.

Let us begin with the root. By typing `get(0)`, MATLAB returns a list of root properties, most of which can be changed with the set command. One of these properties is called Format, which determines how to display data to the screen. If you do not know the options, type `set(0,'Format')` and MATLAB will return the options of how to set Format

```
[ short | long | shortE | longE | shortG |
    longG | hex | bank | + | rational ].
```

Typing `set(0,'Format','rational')` causes MATLAB to display numbers as rational numbers. For example, typing `pi` now returns 355/113, an approximation of π that is good to about six decimal places. There are a couple of variations to the `get` command that are quite useful: `gcf` gets the current figure handle, `gca` gets the current axes handle, and `gco` gets the current object handle.

Custom plotting

We now have the knowledge we need to transform generic plots into something much more special. To demonstrate how, we revisit our three original examples. For each plot, font sizes are increased to improve readability. In the histogram plot, shading is lightened and bin counts are added to the top of each shaded bar. In the cosine plot, grid spacing, axis and curve labeling, and line weights are all improved. In the Bessel function plot, improvements are made in line weights and styles, curve and axis labeling, and equation display. The code for each case is described in turn, preserving as much of the original code as possible.

To begin, we address the major deficiencies of the original histogram shown in Fig. 1 as follows:

```
01   z = randn(1000,1);
02   BinCenters = (-2.5:2.5);
03   hist(z,BinCenters);
04   set(gca,'FontSize',24);
05   xlabel('z'); ylabel('Count');
06   set(gca,'XLim',[-3,3],...
07      'YTick',(0:100:400));
08   PatchHandle = findobj('Type','Patch');
09   set(PatchHandle,'FaceColor',[0.8 0.8 0.8]);
10   BinCounts = hist(z,BinCenters);
11   for i=1:length(BinCounts),
12     text(BinCenters(i),BinCounts(i),...
13       num2str(BinCounts(i)),...
14       'FontSize',24,...
15       'HorizontalAlignment','center',...
16       'VerticalAlignment','bottom');
17    end
```

The first three lines are identical to the original. The `hist` command on line 3 creates the figure, axes, and other objects that comprise the histogram. Not knowing the axes handle, line 4 uses the `gca` command to fetch the current axes handle, which the `set` command then uses to adjust the font size to 24 point. This is done prior to the line 5 axis labeling commands since they happen to inherit the axes font size property. The larger font size helps ensure the text is readable when the graphic is reduced to fit into tight spaces such as a two-column conference format.

Lines 6–7 set two other axes properties: one to ensure the x-axis limits are ± 3 and another to force the y-axis tick labels to step from 0 to 400 in increments of 100. The `findobj` com-

mand in line 8 locates the handle of the patch object that creates the blue-colored bars in the original histogram, and line 9 changes the histogram bar color to a more printer-friendly light-gray using the RGB triple [0.8, 0.8, 0.8].

By setting the `hist` command equal to a variable, as done in line 10, MATLAB returns a vector containing the bin counts for the histogram. This vector is used in the loop that spans lines 11–17 to label each of the six bins with its exact bin count. Lines 12–13 specify the *x*-position, *y*-position, and string arguments required by the `text` command. Basically, a text string with the current bin count is located at the top of the corresponding histogram bar. Lines 14–16 supplement the `text` command and set various text object properties including horizontal and vertical alignment properties to ensure that each text object is centered above its corresponding bin.

As shown in Fig. 4, we can now easily establish that $342 + 360 = 702$ observations are found within one standard deviation of the mean. While it takes a little practice to know exactly what to set, the results are certainly worth the effort. Compared against the original plot in Fig. 1, the new result is clearly better.

Next, let us remake the quadrature sinusoid plot.

```
01   t = (0:0.01:2*pi);
02   x = cos(t); y = sin(t);
03   plot(t,x,'k-',t,y,'k--'); grid;
04   set(gca,'FontSize',24);
05   xlabel('t'); ylabel('Amplitude');
06   set(gca,'GridLineStyle','--',...
07      'YTick',(-1:1),...
08      'XTick',(0:pi/2:2*pi),...
09      'FontName','symbol',...
10      'XTickLabel',[{'0'};{'p/2'};...
11      {'p'};{'3p/2'};{'2p'}],...
12      'XLim',[0,2*pi],'YLim',[-1.1 1.1]);
13   LineHandles = findobj('Type','Line');
14   set(LineHandles,'LineWidth',3);
15   text(pi,0,'sin(t)',...
16      'FontSize',24,...
17      'HorizontalAligment','left',...
18      'VerticalAlignment','bottom');
19   text(pi/2,0,'cos(t)',...
20      'Fontsize',24,...
21      'HorizontalAlignment','right',...
22      'VerticalAlignment','top');
```

Similar to the previous example, the first three lines remain unchanged. Line 4 sets the font size for the axes object to 24 point, a property that is inherited by the line 5 axis labeling commands. Lines 6–12 set various attributes of the axes object. To help visibility, the GridLineStyle is set to dashed rather than dotted. The YTick marks are set rather broadly apart, and the XTick marks are spaced by $\pi/2$ to help emphasize the quadrature relation between the curves. Line 9 changes the axes FontName to symbol; when combined with lines 10–11, the horizontal axis will now show the symbol π rather than decimal numbers since the string character p appears as π with the symbol font set. Line 12 effectively resizes the plot bounding box; making XLim cover one period avoids the wasted space in the previous plot, and expanding YLim to slightly larger than the amplitude extremes helps make the plot seem less crowded.

In line 13, the `findobj` command returns the two line handles corresponding to the $x(t) = \cos(t)$ and $y(t) = \sin(t)$ curves. The set command used in line 14 changes the LineWidth of both curves to 3. This time, it will not strain our eyes to see the curves.

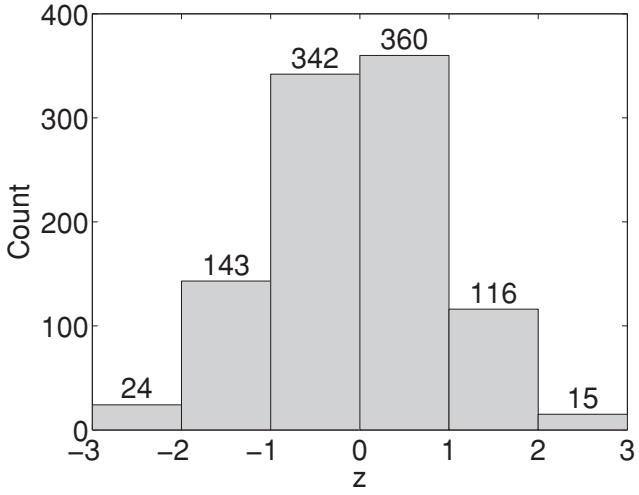

Fig. 4 Custom histogram in the original point size to demonstrate improvement to readability

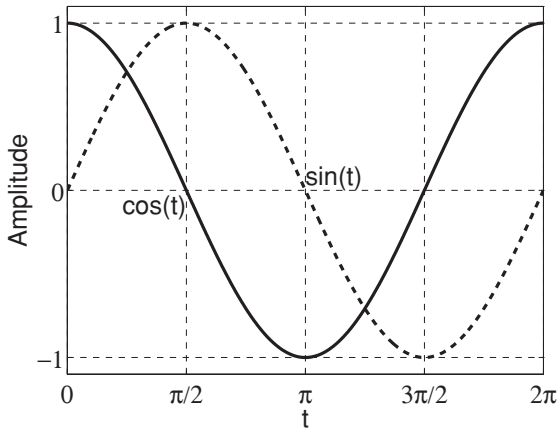

Fig. 5 Custom plot of two quadrature sinusoids in original point size to underscore readability issues

Lines 15–22 effectively replace the `legend` command. Instead of a standard legend, appropriate text objects are placed next to their respective curves. Vertical and horizontal alignment properties are chosen to ensure that the text does not run into the curves. As shown in Fig. 5, the new plot has no glaring faults and is much better than the original.

Lastly, we move on to the following Bessel function plot.

```
01   beta = (0:0.1:15); n = [0:3,10];
02   for i=1:length(n),
03       J(i,:) = besselj(n(i), beta);
04   end
05    set(0,'DefaultAxesColorOrder',[0 0 0],...
06       'DefaultAxesLineStyleOrder',...
07       ('-|--|-.|:|.'),...
08       'DefaultAxesFontSize',24,...
09       'DefaultTextFontSize',16,...
10       'DefaultLineLineWidth',3);
11   plot(beta,J);
```

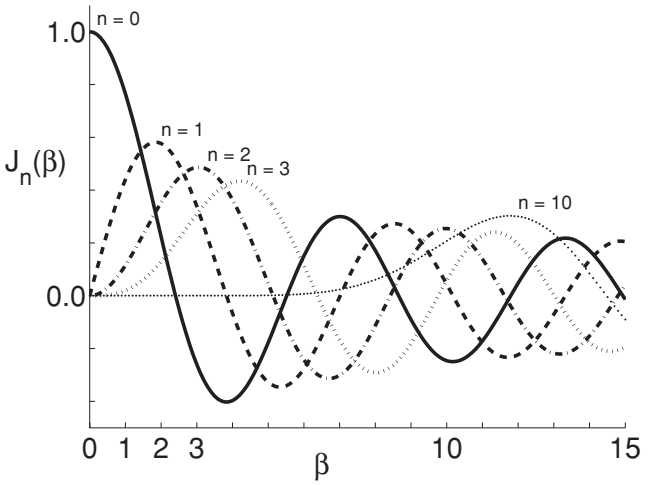

Fig. 6 Custom plot of $J_n(\beta)$, in original point size, for various n is superior to the Fig. 3 plot

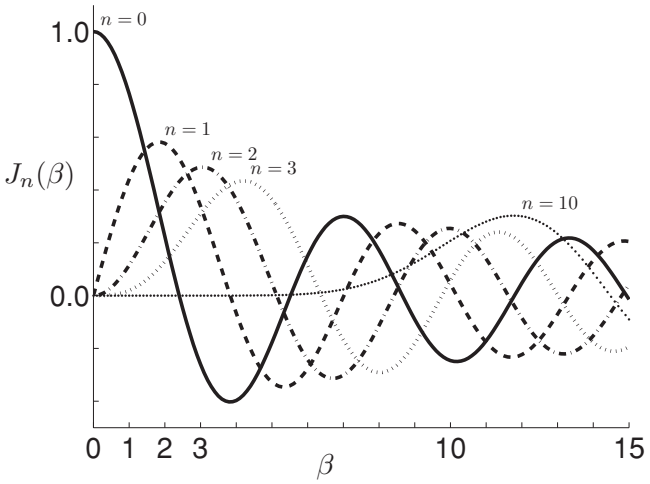

Fig. 7 Custom plot, in original point size, of $J_n(\beta)$ using LaTeX interpreter

```
12    xlabel('\beta','Position',[6.5 -0.6 1.0]);
13    ylabel('J_n(\beta)','Rotation',0,...
14        'Position',[-1.5 0.4 1.0]);
15    set(gca,'box','off','XTick',(0:15),...
16        'XTickLabel',[{'0'}; {'1'};{'2'};...
17        {'3'};{' '};{' '};{' '};{' '};...
18        {' '};{' '};{'10'};{' '};{' '};...
19        {' '};{' '};{'15'}],...
20        'YTick',(-.4:.2:1),...
21        'YTickLabel',[{' '},{' '},{'0.0'},...
22        {' '},{' '},{' '},{' '},{'1.0'}],...
23        'XLim',[0 15],'YLim',[-0.5 1.1]);
24    for i = 1:length(n),
25        [mx,ind] = max(J(i,:));
26        text(beta(ind)+0.2,mx+0.05,...
27             ['n = ',num2str(n(i))]);
28    end
```

The data creation lines 1–4 are identical to the original. Before moving to the `plot` command, however, remember that the main fault of the original plot was that the different Bessel function lines, when printed in black and white, could not be uniquely distinguished. Wouldn't it be nice if MATLAB sequenced through line styles rather than colors when plotting multiple curves? With Handle Graphics, we can tell MATLAB to do just that.

A little explanation is needed to understand lines 5–10. Although these properties do not appear when you type `get(0)`, you can set the default of any graphic object property in the root. You simply concatenate Default with an object name, such as Axes, followed by the desired property, such as ColorOrder. Thus, by typing `set(0,'DefaultAxesColorOrder',[0 0 0])`, every plot will only sequence through a single color, black, which is identified by the RGB triple `[0 0 0]`. In a similar manner, lines 6–7 provide us with our sequencing line styles. Lines 8–10 set future text and axes font sizes as well as line object line widths. Now, the `plot` command of line 11 plots the individual Bessel function curves with sequencing line styles, each black and with a LineWidth of 3.

Lines 12–14 show a method to obtain Greek characters. The axis labels both require β as well as some subscripting. MATLAB will generally produce Greek characters if you simply spell them out and precede them with a \ character (e.g., `\beta` produces β). Each text object has a property called Interpreter that tells MATLAB how to read the string argument. By default, the Interpreter is set to `tex`. TEX founder Donald Knuth's TEX book is an excellent place to start learning TEX syntax.

Notice that the underscore in line 13 causes subscripting. The Position property in lines 12 and 14 allows the axis labels to be positioned to complement axis tick labels. Although this is a two dimensional plot and the z-axis is irrelevant, the position property still requires three arguments to specify the x-, y-, and z-positions, respectively. The rotation of the y-axis label is set to zero so you do not need to crane your neck to read the text.

Lines 15–23 set various axes properties, much like the previous example. As it does not really add anything to the plot, the plot bounding box is turned off with the Box property in line 15. Many of the tick labels are set blank (`' '`) to reduce crowding. Lines 24–28 replace the original `legend` command, placing text objects near the maximum amplitudes of their respective Bessel functions. The result is shown in Fig. 6. Without a doubt, the new plot is superior to the original Fig. 3 plot.

The really astute and detail-oriented reader will note that the symbols in Fig. 6, while good, are not quite right. They are too vertical and do not follow the italic bend like $J_n(\beta)$. It turns out that we can fix this, too, since MATLAB also has a LaTeX interpreter. Users who know LaTeX syntax will find that MATLAB's LaTeX interpreter is rather robust and capable of producing complex expressions. Replacing lines 12–14 and 26–27 with the following produces the nearly perfect result shown in Fig. 7:

```
12    xlabel('$\beta$','Position',[6.5 -0.6 1.0],...
          'Interpreter','latex');
13    ylabel('$J_n(\beta)$','Rotation',0,...
14        'Position',[-1.5 0.4 1.0],...
          'Interpreter','latex');
26    text(beta(ind)+0.2,mx+0.05,...
27        ['$n = ',num2str(n(i)),'$'],...
          'Interpreter','latex');
```

Before concluding, two observations are warranted. First, although many plot annotations and modifications are possible using pull-down menus and mouse clicks, it is preferable to perform such operations using commands, as demonstrated in this article. Only in this way will you be able to reliably, accurately, and quickly produce (and reproduce) your plots, as well as tweak the look of your plots to meet requirements. The more complicated the plot, the more important this advice becomes.

It is also worth pointing out that many computer systems are not "what you see is what you get" (WYSIWYG). In other words, what is shown on your screen may not be identical to what is printed to a file or your printer. Thus, your MATLAB plots, generic or otherwise, may not appear exactly as you expect. You need to look at the end print or use MATLAB's print preview, which is normally WYSIWYG, to ensure that your results are precisely what you want.

Conclusions

This article shows three examples of generic MATLAB plots, discusses techniques to better control MATLAB plots, and then uses these techniques to improve the three original examples. In each case, plot customizations are achieved using MATLAB's Handle Graphics, which provide a convenient and simple manner to obtain and set graphic object properties. Colors and shading, line weights and styles, font sizes and types, axis spacing and labeling, and nearly every other aspect of a MATLAB plot are easily controlled.

This article cannot tell you, however, everything that can go wrong with your plots nor everything you might need to do to fix them. That is your job. Still, the examples and discussion should provide a solid basis for MATLAB users to improve plot quality. With a little creativity and patience, it is possible to produce just about any plot imaginable.

Read more about it

- C. Gomez, S. Steer, and R. Nikoukhah, *Engineering and Scientific Computing with Scilab*. New York: Springer-Verlag, 2006.
- *Getting Started with MATLAB*, Version 7, The MathWorks, Inc., Natick MA, 1984–2006. PDF available as part of MATLAB release R2006b documentation.
- *Graphics*, Version 7, The MathWorks, Inc., Natick MA, 1984–2006. PDF available as part of MATLAB release R2006b documentation.
- P. Marchand and O.T. Holland, *Graphics and GUIs with MATLAB*, 3rd ed. London: Chapman and Hall/CRC, 2003.
- T. Davis and K. Sigmon, *MATLAB Primer*, 7th ed. London: Chapman and Hall/CRC, 2004.
- S. Campbell, J. Chancelier, and R. Nikoukhah, *Modeling and Simulation in Scilab/Scicos*. New York: Springer-Verlag, 2005.

About the author

Roger A. Green (Roger.Green@ndsu.edu) is an associate professor in the Department of Electrical and Computer Engineering at North Dakota State University. He teaches a variety of signal processing courses and uses MATLAB extensively in both his courses and research.