4.6 Nested Functions in Matlab

Normally, we include subfunctions in a primary function as follows.

```
function output=primary(input)
statements---including subfunction calls

% first subfunction
function out=one(in)
statements

% second subfunction
function out=two(in)
statments
```

To write a *nested function*, write one or more functions within the body of another function in an M-file. You must terminate any nested function (as well as the primary function) with an **end** statement.

```
function output=primary(input)
statements---including subfunctionc calls

% first subfunction
    function out=one(in)
        statements
    end %nested function one

% second subfunction
    function out=two(in)
        statments
    end %nested function two

end %function primary
```

M-file functions do not normally require a terminating **end** statement. However, if an M-file contains one or more nested functions, you must terminate *all* functions (including any subfunctions) in the M-file with an **end** statement, whether or not they contain nested functions.

[1] Copyrighted material. See: http://msenux.redwoods.edu/Math4Textbook/

Variable Scope in Nested Functions

So, what's the big deal? Why should we bother to study *nested functions*? What do they offer that we don't already have with the existing functions and subfunctions we've already studied?

The answer lies in *variable scope*. As a short example, open the editor and enter the following lines. Save the function M-file as **varScope1.m**.

```
function varScope1
x=3;
nestfun1

function nestfun1
fprintf('In nestfun1, x equals: %d\n',x)
```

Note that we have a primary function named **varScope1** which contains one subfunction named **nestfun1**. The primary assigns the variable x the scalar 3, then calls the subfunction with the command **nestfun1**. In turn, the subfunction **nestfun1** attempts to print the value of the variable x **fprintf** with a nice format string.

Remember that the primary function **varScope1** and the subfunction **nestfun1** have separate workspaces. The variable x exists in the primary workspace, but the subfunction workspace is independent of the primary workspace and thus has no knowledge of the variable x. Thus, when we run the function, we should not be surprised by the error message.

```
>> varScope1
??? Undefined function or variable 'x'.

Error in ==> varScope1>nestfun1 at 6
fprintf('In nestfun1, x equals: %d\n',x)

Error in ==> varScope1 at 3
nestfun1
```

This is precisely the error message we expected to see. The variable x exists in the primary function workspace, but the subfunction workspace has no knowledge of the variable x. When the subfunction attempts to print the contents of the variable x, an error occurs.

We've seen a couple of ways we can fix this M-file so that the value of x is printed by the subfunction. One way would be to declare the variable x global in both the primary function and the subfunction.

```
function varScope1
global x
x=3;
nestfun1

function nestfun1
global x
fprintf('In nestfun1, x equals: %d\n',x)
```

Running this M-file produces the desired effect. The subfunction now has knowledge of the global variable x and its **fprintf** statement is successful.

```
>> varScope1
In nestfun1, x equals: 3
```

We've also seen that we can pass the value of the variable x to an input argument of the subfunction.

```
function varScope1
x=3;
nestfun1(x)

function nestfun1(x)
fprintf('In nestfun1, x equals: %d\n',x)
```

As readers can see, this also works.

```
>> varScope1
In nestfun1, x equals: 3
```

Nested Functions Redefine the Rules of Variable Scope. If we take our same example, and nest the subfunction, a whole new set of rules are put into place for variable scope, the first of which follows.

The First Rule for Variable Scope in Nested Functions. A variable that has a value assigned to it by the primary function can be read or overwritten by a function nested at any level within the primary.

Enter the following lines exactly as they appear below into the editor and save the function M-file as **varScope2.m**.

```
function varScope2
x=3;
nestfun1

function nestfun1
fprintf('In nestfun1, x equals: %d\n',x)
end  %end nestfun1

end  %end varScope2
```

If you are using the Matlab editor, select all of the above lines in the editor with the key sequence Ctrl+a (Cmd-a on the Mac), then "smart indent" the highlighted selection with Ctrl+i (Cmd-i on the Mac). The resulting indentation is shown in what follows.

```
function varScope2
x=3;
nestfun1

    function nestfun1
        fprintf('In nestfun1, x equals: %d\n',x)
    end  %end nestfun1

end  %end varScope2
```

We've used comments to indicate where the primary and nested functions "end." Note that the combination of the **end** statements, the "smart" indentation in the editor, and our helpful comments all serve to highlight the fact that the function **nestfun1** is "nested" within the primary function **varScope2**. According to the *First Rule for Variable Scope in Nested Functions*, the nested function **nestfun1** should have access to the variable x that is assigned the value of 3 in the primary function **varScope2**.

```
>> varScope2
In nestfun1, x equals: 3
```

Very interesting! No global variables, no passing of values as arguments, but because **nestfun1** is "nested" within the primary function, the nested function **nestfun1** has access to all variables assigned in the primary function.

Graphical User Interfaces

In this section we introduce our readers to Matlab's GUI's *Graphical User Interfaces*). Matlab has a full complement of **uicontrols** (*User Interface Controls*), typical of what you commonly see in most of today's modern sofware: edit boxes, popupmenus, push- and toggle-buttons, sliders, radio buttons, and checkboxes, a few of which we will introduce in this section.

We're going to write our first GUI, keeping it low-key as we introduce the concepts. Our gui design can be described with the following series of tasks:

1. Plot a function.
2. Provide radio buttons to change the color of the graph of the function, offering choices of red, green, or blue.
3. Provide a popup menu with choices of different line styles for the plot: solid, dotted, etc.
4. Provide an edit box that allows the user to change the linewidth.

Let's begin. Open the editor and enter the following lines, then save the function M-file as **plotGUI.m**.

```
function plotGUI

close all
clc

% plot domain
xmin=0;
xmax=4*pi;

% some colors
figure_color=[0.8,0.9,0.8];
panel_color=[1,0.9,0.8];
buttongroup_color=[0.9,0.9,0.8];
```

This function takes no input and export no output. However, it is possible to write GUI functions that accept input and export data back to the caller.

Next we close all open figure windows and clear the command window. This is helpful during the development of the GUI, but once we have a tested and working program, we'll remove these two commands from the GUI.

We set limits on the domain of our plot, which we will also use when we initialize an axes on our figure window. It's helpful to put these here and not have to change them in several places in our program.

Finally, we initialize some colors we will use for our GUI components. Note that each color takes the form of a row vector with three entries, each of which must be a number between 0 and 1. The first entry is the percentage of red, the second the percentage of green, and the third entry is the percentage of blue. Most modern computers have color wheels where you can view a color and the percentages of red, green, and blue (in the RGB system) to make up the color.

Next, we'll add a figure window to the GUI. Add the following lines to the function M-file **plotGUI** and save.

```
hFigure=figure(...
    'Units','Pixels',...
    'Position', [100 100 700 500],...
    'Color',figure_color,...
    'MenuBar','none',...
    'ToolBar','none',...
    'NumberTitle','off',...
    'Name','Plot GUI');
```

Run the program. Some things to note:

1. We set **Units** to **Pixels**. Thereafter, all positioning commands are made in Pixels. Think of a pixel as one dot on your computer screen. Typical resolution on modern displays are of the order of 1024 by 768 pixels, but can be higher (or lower) depending on personal display settings on your computer.
2. Next we set the **Position** of the figure window at **[100 100 700 500]**. The first two measurements (100 and 100) are the location of the lower left-hand corner of the figure window, relative to the lower left-hand corner of your computer screen. The next two measurements (700 and 500) are the width and height of the figure window. All of these measurements are in pixels
3. Finally, we set the color, then turn off the menubar and toolbar as well as the usual number title that is displayed in a typical figure window (Figure 1,

Figure 2, etc.). Then we give our figure window a meaningful name for the project, namely **Plot GUI**.

4. Note that we assign a *handle* to the figure window named **hFigure**. Every Matlab object can be associated with a handle for further reference.

We will now add an axes to the figure window. Note that the axes object will be a "child" of the figure window (equivalently, the figure window is a "parent" of the axes object). Add the following lines to the file **plotGUI** and save.

```
hAxes=axes(...
    'Parent',hFigure,...
    'Units','Pixels',...
    'Position',[50 50 400 400],...
    'Xlim',[xmin,xmax],...
    'XGrid','on',...
    'YGrid','on');
```

Run the program. Things to note:

1. Note that the "Parent" of the axes object is set to **hFigure**, which is the numerical handle assigned to the figure window designated above. This makes the axes object a "child" of the figure window whose numerical handle is **hFigure**.
2. The unit of measurement for the axes object is also set to "Pixels."
3. The position of the axes object is set to **[50 50 400 400]**. The first two numbers (50 and 50) set the lower left-hand corner of the axes object, relative to the lower left-hand corner of its parent, in this case the figure window designated by the handle **hFigure**.
4. The limits on the x-axis are set to **[xmin,xmax]** and the grid in both the x- and y-directions is turned 'on' (equivalent to using the **grid on** command).
5. Finally, note that we've assigned a handle **hAxes** to our axes object for further reference.

Now that we have a figure window and axes object, let's create some data and place a plot of the data on the axes object. Enter the following lines in **plotGUI** and save.

```
% plot data
x=linspace(xmin,xmax);
y=sin(2*x);
hLine=line(x,y);
```

If you've been running the program, note that we've left some room on the right for our controls. First, let's create a **uipanel** (user interface panel) in this free area of the figure window to contain the various controls of the GUI which we will create later. Enter the following lines in **plotGUI** and save.

```
hPanel=uipanel(...
  'Parent',hFigure,...
  'Units','Pixels',...
  'Position',[475,50,200,400],...
  'BackgroundColor',panel_color);
```

Running the program at this stage should produce the result shown in **Figure 4.1**.

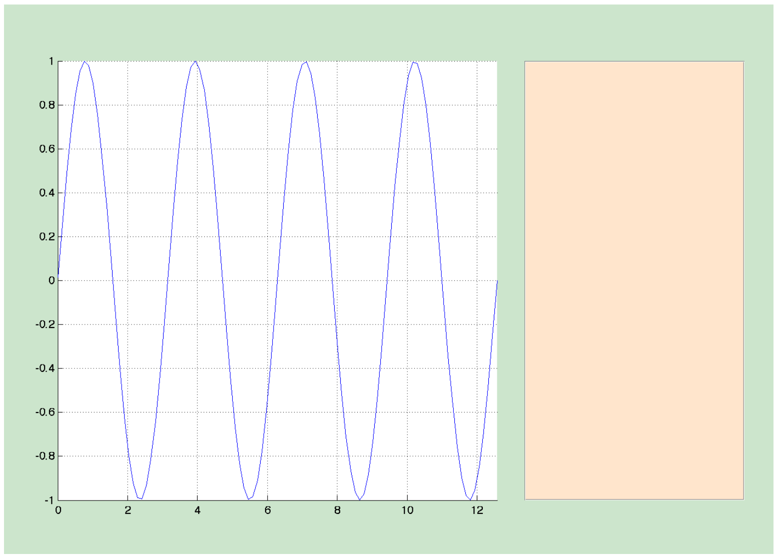

Figure 4.1. Figure acting as parent to child axes and panel objects.

Note that the **uipanel** is a child object of the figure window having handle **hFigure**. Hence, when we set the position with the vector **[475,50,200,400]**, the first two entries (475 and 50) designate the lower left-hand corner of the **uipanel** as measured from the lower left-hand corner of the 'Parent' figure window. The next two entries of the position vector, 200 and 400, designate the width and the height of the **uipanel**.

Button Groups and Radio Buttons

. We're now going to place three radio buttons in the **uipanel**, one for each of the colors red, green, and blue. If the user selects the 'red' radio button, then we'll color the graph red. Similar strategies will apply to the 'green' and 'blue' radio buttons.

We could set the radio buttong directly in the **uipanel**. However, if we first place our three radio buttons in what is known as a **uibuttongroup**, only one of them can be selected at a time. So, enter the following lines in **plotGUI** and save.

```
hButtonGroup=uibuttongroup(...
    'Parent',hPanel,...
    'Units','Pixels',...
    'Position',[10,275,180,100],...
    'BackgroundColor',buttongroup_color}
```

Because the **uibuttongroup** is a child of the **uipanel** with handle **hPanel**, when we position the **uibuttongroup** with the vector **[10,275,180,100]**, the first two entries (10 and 275) are the lower left-hand corner of the **uibuttongroup**, as measured from the lower left-hand corner of the **uipanel** with handle **hPanel**. The second two entries, 180 and 100, designate with width and height of the **uibuttongroup**.

Next, we add a radio button to the **uibuttongroup**. We use a **uicontrol** for this purpose, with **Style** set to **Radio**. It is important to note that the parent of this **uicontrol** is the **uibuttongroup** having handle **hButtonGroup**. Thus, when we set the position to **[10,10,160,20]**, the fist two entries (10 and 10) are the lower left-hand corner of the radion **uicontrol**, as measured from the lower left hand corner of its parent (**uibuttongroup** with handle **hButtonGroup**).

```
r1=uicontrol(...
    'Style','Radio',...
    'Parent',hButtonGroup,...
    'Units','Pixels',...
    'Position', [10,10,160,20],...
    'String','Blue',...
    'BackgroundColor',buttongroup_color);
```

The **BackgroundColor** should be self explanatory, and when we set **String** to **Blue**, this prints the message 'Blue' to the default position (to the immediate right of the radio button.

We add two more radio buttons (one for red and one for green) in a similar manner. First, green.

```
r2=uicontrol(...
    'Style','Radio',...
    'Parent',hButtonGroup,...
    'Units','Pixels',...
    'Position', [10,37,160,20],...
    'String','Green',...
    'BackgroundColor',buttongroup_color);
```

Then, red.

```
r3=uicontrol(...
    'Style','Radio',...
    'Parent',hButtonGroup,...
    'Units','Pixels',...
    'Position', [10,65,160,20],...
    'String','Red',...
    'BackgroundColor',buttongroup_color);
```

Note that each radio button has a unique handle (**r1**, **r2**, and **r3**) for later reference.

Adding the button group and radio buttons produce the GUI shown in **Figure 4.2**.

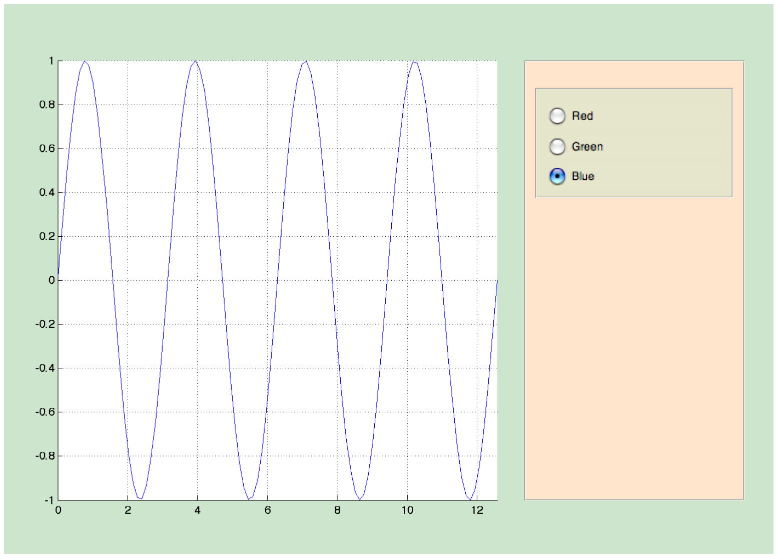

Figure 4.2. Adding a button group and radio buttons.

At this point, you can try clicking each of the radio buttons, red, blue, or green. Although they are not programmed to do anything at this point, note that only one of them can be selected at a time. This behavior (one at a time selection) is due to the fact that we grouped them inside a **uibuttongroup**.

Adding a Function Callback to the UIButtonGroup

Note that the handle **hButtonGroup** can be used to get or set property-value pairs for the **uibuttongroup**. You can obtain information on all possible property-value pairs for a **uibuttongroup** by typing **doc uibuttongroup** at the Matlab prompt. We are particularly interested in the **SelectionChangeFcn** property, to which we must assign a handle to a function. This sort of function is referred to as a *callback function* and will be executed whenever a new radio button is selected with the mouse. (Hence the property name, **SelectionChangeFcn**). It doesn't matter what name you use for the callback function, but writers of Matlab GUI's tend to use a descriptive name that includes the substring 'callback.' Thus, we'll name the callback **colorSelection_callback**.

Add one more property-value pair to the **uibuttongroup** as follows.

```
hButtonGroup=uibuttongroup(...
    'Parent',hPanel,...
    'Units','Pixels',...
    'Position',[10,275,180,100],...
    'BackgroundColor',buttongroup_color,...
    'SelectionChangeFcn',@colorSelection_callback);
```

Note that this is not new code. We simply appended the last property-value pair to the existing code that initialized the **uibuttongroup**.

Now, whenever you click a radio button with your mouse, the callback function named **colorSelection_callback** will be executed. You can give it a try at this point, but the callback function does not yet exist in our GUI code, so you will receive an error message attesting to that fact.

What we need to do next is write the function callback. So, go to the end of the file **plotGUI.m**, just prior to the closing **end** of the primary function **plotGUI**, and add the following lines. We'll fill in the body of this function callback in a moment, but just add these lines at present.

```
    function colorSelection_callback(hObject,eventdata)
    end % end colorSelection_callback
```

Note that this callback function is *nested* inside the primary function **plotGUI**. Note the **end** command, which you must use to terminate any nested function.

Some comments are in order.

- The name of the callback must match the name of the function handle assigned to the property **SelectionChangeFcn** in the **uibuttongroup**.
- Two input arguments are required. You may use any names you wish.
  i. The first argument, which we have named **hObject**, receives the value of the handle of the calling object. In this case, **hObject** receives the handle of the **uibuttongroup**, namely **hButtonGroup**.
  ii. The second argument, which we have named **eventdata**, is passed a structure containing data describing the state of the calling object when triggered by an event (in this case the clicking of a radio button by the mouse). In this case, the structure **eventdata** contains two fields, one named **OldValue**, which contains the handle of the previously selected radio button, and a second named **NewValue**, which contains the handle of the newly selected radio button. Note that the possible handles contained in these fields are three: **r1**, **r2**, or **r3**. You can access the value of these fields by typing the structure name, a period, then the field desired. For example, to get the handle to the newly selected radion button, type **eventdata.NewValue**. To get the handle of the previously selected radio button, type **eventdata.OldValue**.

We'll now code the body of the callback function. First, we get the value of the currently selected radio button and query the handle for the value of 'String,' which could be either 'Red', 'Green', or 'Blue' (see the initialization strings for each radio button described and coded earlier). We set up a switch structure, which will color the plot red, green, or blue, depending on whether **lineColor** equals 'Red', 'Green', or 'Blue'.

```
function colorSelection_callback(hObject,eventdata)
    lineColor=get(eventdata.NewValue, 'String');
    switch lineColor
        case 'Red'
            set(hLine,'Color','r')
        case 'Green'
            set(hLine,'Color','g')
        case 'Blue'
            set(hLine,'Color','b')
    end
end % end colorSelection_callback
```

The thing to note is the fact that the nested callback has access to all variables defined in the outer primary function. In particular, **hLine** is a handle to the sinusoidal plot and it was defined in the primary function. Hence, the nested function **colorSelect_callback** has access to this variable by default. It is not necessary to pass the variable by value or to declare it global. Note that in each **case** of the **switch** structure, the code uses the handle **hLine** to set the 'Color' property of the plot to the appropriate color.

At this point, you should be able to run the function without error. The GUI will initialize, after which you can click the radio buttons and watch the plot change to the color associated with the selected radio button.

Popup Menus

In this section we will add a popup menu containing choices of line styles for our plot. However, before adding the popup **uicontrol**, we will first add some explanatory text by adding a **uicontrol** containing *static text*. Static text is not dynamic[2], as the name implies, and is used specifically to annotate your GUI with help messages and instructions.

So, immediately following the code for your last radio button (the code with handle **r3**), enter the following lines.

```
hLineStyleText=uicontrol(...
    'Style','text',...
    'Parent',hPanel,...
    'Position', [10 240 180 20],...
    'String', 'LineStyle Choices',...
    'HorizontalAlignment','Left',...
    'BackgroundColor', panel_color);
```

Again, comments are in order.

- The 'Style' of the **uicontrol** is 'text'. This declares the control as static text. The text that is actually printed on your GUI is the value of the 'String' property, in this case "LineStyle Choices." Underneath this static text, we will soon place a popup menu with choices for line styles.
- Not that this **uicontrol** is a child of the **uipanel** with handle **hPanel**. Consequently, any positioning is measured from the lower left-hand corner of the **uipanel**.

[2] Although static text cannot be modified by the user, it can be modified by the program code.

Next, we deal with the popup menu. Following the code for our static text, enter these lines to intialize a popup menu of line style choices.

```
hLineStylePopup=uicontrol(...
    'Style','popup',...
    'Parent',hPanel,...
    'Units','Pixels',...
    'Position', [10 225 180 15],...
    'String',{'Solid' 'Dotted' 'DashDot' 'Dashed' 'None'},...
    'BackgroundColor', buttongroup_color,...
    'Callback', @LineStylePopup_callback);
```

Here are a few words of explanation. First, some simple comments.

- The 'Style' of the **uicontrol** is 'popup'. This declares the control to be a popup menu.
- Again, the 'Parent' of this **uicontrol** is the **uipanel** with handle **hPanel**. Thus, the position is again measured from the lower left=hand corner of the **uipanel**.

The 'String' property of the **uicontrol** is a *cell* containing the individual strings that we wish to appear on the popup menu. A cell is a Matlab data structure that allows the user to collect different data types in a set.

One of the simplest ways to build a cell in Matlab is shown in the code that follows. You should enter this code at the Matlab prompt, not in the code of our GUI that is under construction.

```
>> C={'dashed', 1:5, magic(3)}
C =
    'dashed'    [1x5 double]    [3x3 double]
```

Note that we have collected three different data types in a cell, delimiting the entries with curly braces, and assigned the result to the variable C. The first entry is a string, the second a row vector, and the third a 3×3 matrix.

You can access the contents of the individual elements of a cell by using curly braces with the usual indexing. For example, to obtain the contents of the first entry in C, enter the following code (again, at the Matlab prompt, not in the code for the GUI).

```
>> C{1}
ans =
dashed
```

In similar fashion, we can access the contents of the second entry in C with the following command.

```
>> C{2}
ans =
     1     2     3     4     5
```

We will leave it to our readers to access the magic matrix that is the third entry in C.

Our use of cells at this juncture will be at a very elementary level. For a full discussion of cell structure in Matlab, we refer our readers to the Matlab documentation.

To continue with our discussion of the popup **uicontrol**, after setting the background color, note that we set the 'Callback' property of the **uicontrol** to a function handle **LineStylePopup__callback**. At this point, you can run the GUI and note the added presence of our two new **uicontrols** in **Figure 4.3**, the static text and the popup menu. However, selecting an item on the popup menu with the mouse will lead to an error, because the callback function **LineStylePopup__callback** has not yet been defined.

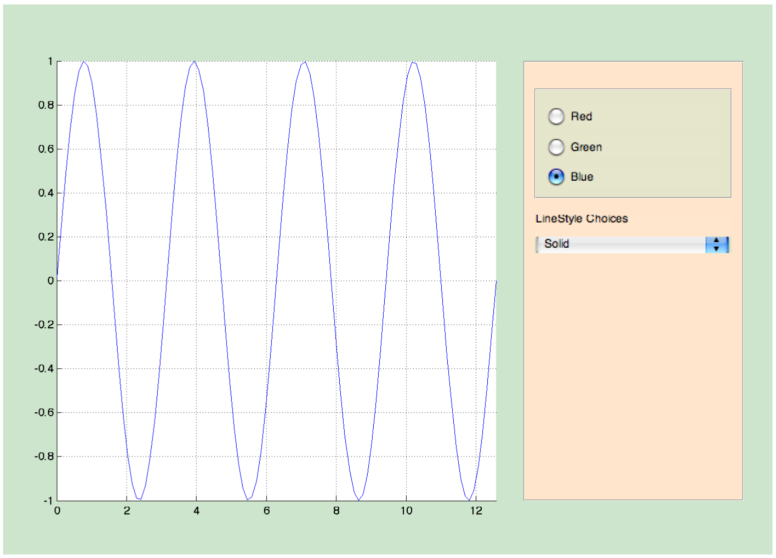

Figure 4.3. Adding a static text and popup menu.

So, let's defined the callback for the popup menu of line styles. Just below the callback function **colorSelection__callback**, add the following lines.

```
    function LineStylePopup_callback(hObject,eventdata)
        lineStyleChoices=get(hObject,'String');
        lineStyleChoice=get(hObject,'Value');
        lineStyle=lineStyleChoices{lineStyleChoice};
        switch lineStyle
            case 'Solid'
                set(hLine,'LineStyle','-')
            case 'Dotted'
                set(hLine,'LineStyle',':')
            case 'DashDot'
                set(hLine,'LineStyle','-.')
            case 'Dashed'
                set(hLine,'LineStyle','--')
            case 'None'
                set(hLine,'LineStyle','none')
        end
    end % end LineStylePopup_callback
```

We will make several comments regarding this callback.

- The first line contains the function keyword and the name of the callback, in this case **LineStylePopup__callback**. The descriptive name of this callback is a reminder of it is designed to provide. Note the oblibatory arguments, **hObject** and **eventdata**. The argument **hObject** will receive the handle of the calling object (in this case, the handle of the popup menu **uicontrol**). In this example, the argument **eventdata** is not used, but it is still required. As more evolutions of Matlab are updated, plans are in the works to make more use of the argument **eventdata** (as we did with our **uibuttongroup**, but for the meantime, it's not used in this callback, but it is still required.
- Because **hObject** is passed the handle to the popup **uicontrol**, we can use it to access two properties of that control, the 'String' and 'Value' properties.
  i. The 'String' property contains a cell of strings, each of which describe a line style.
  ii. The 'Value' property contains a number indicating which entry on the popup menu was selected by the user.

The cell of strings is stored in **lineStyleChoices** and the number indicating the choice is stored in **lineStyleChoice**. We obtain the string describing the choice with cell indexing. Note the use of the curly braces.

- A switch structure is then used to set the line style based on the string describing the choice that is stored in **lineStyle**. Because the callback is a nested function, it can access the handle **hLine** of the plot, then use it to set the line style according to the choice in **lineStyle**.

At this point, you can run the GUI again. Select the 'Red' radio button and the 'Dashed' linestyle from the popup menu. The result is shown in **Figure 4.4**.

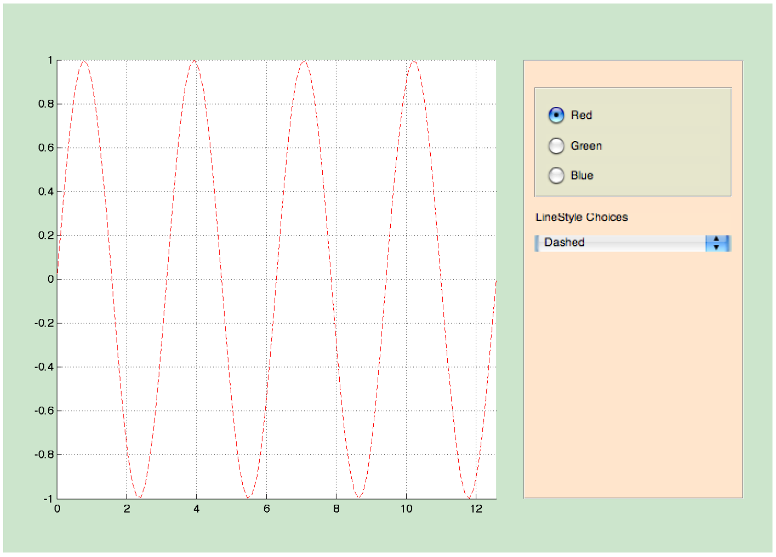

Figure 4.4. Dashed and red.

Edit Boxes

We will add one last **uicontrol** to our GUI, an *edit box*. An edit box provides an area where a user can enter text. Unlike static text, however, an edit box is dynamic. The user can change what is entered there, and on the other hand, your program can also update the contents of an edit box.

However, before we create an edit box on our GUI, we'll first place some static text above the planned location for the edit box. The text will provide the user with information relevant for the edit box. Enter the following lines just below the last **uicontrol**, the popup **uicontrol** for line styles. As static text controls were explained above, and there is nothing new about this current entry other than the 'String' value, we offer the code without comment.

```
hLineWidthText=uicontrol(...
    'Style','text',...
    'Parent',hPanel,...
    'Position', [10 180 180 20],...
    'String', 'Enter Desired LineWidth: (1-10)',...
    'HorizontalAlignment','Left',...
    'BackgroundColor', panel_color);
```

Next, to initialize the edit box for our GUI, enter the following lines just below the last **uicontrol**, the static text control with handle **hLineWidthText**.

```
hLineWidthEditbox=uicontrol(...
    'Style','edit',...
    'Parent',hPanel,...
    'Units','Pixels',...
    'Position', [10 150 180 25],...
    'String','1',...
    'BackgroundColor', buttongroup_color,...
    'Callback', @LineWidthEditbox_callback);
```

Some comments are in order.

- Note that the 'Style' is 'edit', so this **uicontrol** is a dynamic edit box, one where both user and program can modify the text inside.
- The parent is the **uipanel**, so the position of the edit box will be measured from the lower left-hand corner of the panel with handle **hPanel**.
- We intialize 'String' to '1'. This will be the intial value that appears in the edit box. It represents a line width of 1 point (1 pt).
- We assign the function handle **@LineWidthEditbox_callback** to the 'Callback' property.

At this point, if you attempt to modify the text in the dit box, you will receive an error as the callback function **LineWidthEditbox_callback** is not yet written. Let's take care of that immediately and write the callback.

As usual, the obligatory arguments **hObject** and **eventdata** are used.

```
function LineWidthEditbox_callback(hObject,eventdata)
    lineWidth=str2double(get(hObject,'String'));
    if (lineWidth>=1) && (lineWidth<=10)
        set(hLine,'LineWidth',lineWidth);
    else
        set(hObject,'String','Invalid Input---Try again');
    end
end % end LineStyleEditbox_callback
```

Some comments are in order.

- Because **hObject** contains the handle of the calling object, **hLineWidthEditbox** in this case, we can use this handle to 'get' the value of the 'String' property, i.e., the string entered in the edit box. Because this value is a string, we use **str2double** to change it to a number (double precision, the Matlab default), storing the result in **lineWidth**. This is necessary because the 'LineWidth' property of our plot expects a number as its value.
- We use an **if..else..end** construct to determine if the number is greater than or equal to 1 *and* less than or equal to 10. If this test is true, then we use the handle to our plot to set the 'LineWidth' property to the value of **lineWidth**. If not, then we alert the user that the input is invalid and ask them to try again, i.e., enter a linewidth in the edit box between 1 and 10.

Run the program. Choose, red, dashed, and enter 2 for the linewidth to produce the image in **Figure 4.5**.

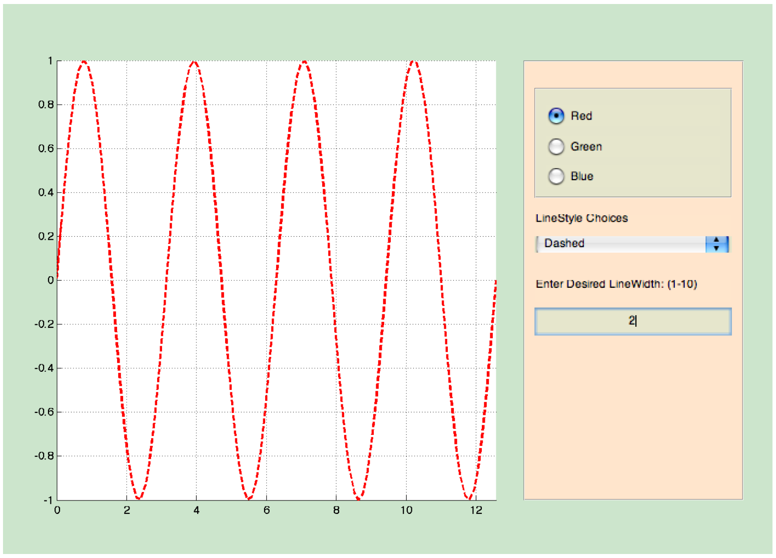

Figure 4.5. Adding an edit box for linewidth.

Appendix

Here we share a complete listing of **plotGUI** for convenience.

```
function plotGUI

close all
clc

% plot domain
xmin=0;
xmax=4*pi;

% some colors
figure_color=[0.8,0.9,0.8];
panel_color=[1,0.9,0.8];
buttongroup_color=[0.9,0.9,0.8];

hFigure=figure(...
    'Units','Pixels',...
    'Position', [100 100 700 500],...
    'Color',figure_color,...
    'MenuBar','none',...
    'ToolBar','none',...
    'NumberTitle','off',...
    'Name','Plot GUI');

hAxes=axes(...
    'Parent',hFigure,...
    'Units','Pixels',...
    'Position',[50 50 400 400],...
    'Xlim',[xmin,xmax],...
    'XGrid','on',...
    'YGrid','on');

hPanel=uipanel(...
    'Parent',hFigure,...
    'Units','Pixels',...
    'Position',[475,50,200,400],...
    'BackgroundColor',panel_color);
```

```
hButtonGroup=uibuttongroup(...
    'Parent',hPanel,...
    'Units','Pixels',...
    'Position',[10,275,180,100],...
    'BackgroundColor',buttongroup_color,...
    'SelectionChangeFcn',@colorSelection_callback);

r1=uicontrol(...
    'Style','Radio',...
    'Parent',hButtonGroup,...
    'Units','Pixels',...
    'Position', [10,10,160,20],...
    'String','Blue',...
    'BackgroundColor',buttongroup_color);

r2=uicontrol(...
    'Style','Radio',...
    'Parent',hButtonGroup,...
    'Units','Pixels',...
    'Position', [10,37,160,20],...
    'String','Green',...
    'BackgroundColor',buttongroup_color);

r3=uicontrol(...
    'Style','Radio',...
    'Parent',hButtonGroup,...
    'Units','Pixels',...
    'Position', [10,65,160,20],...
    'String','Red',...
    'BackgroundColor',buttongroup_color);

hLineStyleText=uicontrol(...
    'Style','text',...
    'Parent',hPanel,...
    'Position', [10 240 180 20],...
    'String', 'LineStyle Choices',...
    'HorizontalAlignment','Left',...
    'BackgroundColor', panel_color);
```

```
hLineStylePopup=uicontrol(...
    'Style','popup',...
    'Parent',hPanel,...
    'Units','Pixels',...
    'Position', [10 225 180 15],...
    'String',{'Solid' 'Dotted' 'DashDot' 'Dashed' 'None'},...
    'BackgroundColor', buttongroup_color,...
    'Callback', @LineStylePopup_callback);

hLineWidthText=uicontrol(...
    'Style','text',...
    'Parent',hPanel,...
    'Position', [10 180 180 20],...
    'String', 'Enter Desired LineWidth: (1-10)',...
    'HorizontalAlignment','Left',...
    'BackgroundColor', panel_color);

hLineWidthEditbox=uicontrol(...
    'Style','edit',...
    'Parent',hPanel,...
    'Units','Pixels',...
    'Position', [10 150 180 25],...
    'String','1',...
    'BackgroundColor', buttongroup_color,...
    'Callback', @LineWidthEditbox_callback);

% plot data
x=linspace(xmin,xmax);
y=sin(2*x);
hLine=line(x,y);

    function colorSelection_callback(hObject,eventdata)
        lineColor=get(eventdata.NewValue, 'String');
        switch lineColor
            case 'Red'
                set(hLine,'Color','r')
            case 'Green'
                set(hLine,'Color','g')
            case 'Blue'
                set(hLine,'Color','b')
        end
    end % end colorSelection_callback
    function LineStylePopup_callback(hObject,eventdata)
        lineStyleChoices=get(hObject,'String');
        lineStyleChoice=get(hObject,'Value');
        lineStyle=lineStyleChoices{lineStyleChoice};
        switch lineStyle
            case 'Solid'
                set(hLine,'LineStyle','-')
            case 'Dotted'
                set(hLine,'LineStyle',':')
            case 'DashDot'
                set(hLine,'LineStyle','-.')
            case 'Dashed'
                set(hLine,'LineStyle','--')
            case 'None'
                set(hLine,'LineStyle','none')
        end
    end % end LineStylePopup_callback

    function LineWidthEditbox_callback(hObject,eventdata)
        lineWidth=str2double(get(hObject,'String'));
        if (lineWidth>=1) && (lineWidth<=10)
            set(hLine,'LineWidth',lineWidth);
        else
            set(hObject,'String','Invalid Input---Try again');
        end
    end % end LineStyleEditbox_callback

end  % end plotGUI
```

4.6 Exercises

1. Write a Matlab GUI that contains one axes object. Set each of the following properties of the axes object to the indicated value.

i. Set both ‘XLim’ and ‘YLim’ properties to $[-10, 10]$.
ii. Set both ‘XTick’ and ‘YTick’ properties to **-10:2:10**.
iii. Set both ‘XGrid’ and ‘YGrid’ properties to ‘on’.
iv. Set the ‘ButtonDownFcn’ property of the axes object to the callback function handle **@changeBackgroundColor_Callback**.

Use a cell to contain of list of strings representing colors.

```
color_list={'r','g','b','m','c','w'};
```

Write the callback function **changeBackgroundColor_Callback** that will change the ‘Color’ property of the axes object each time the mouse is clicked on the axes object. The colors should cycle through the list of colors in the order presented in **color_list**, returning to the first color of the list when the last color is used.

2. Write a Matlab GUI that contains one axes object. Set each of the following properties of the axes object to the indicated value.

i. Set both ‘XLim’ and ‘YLim’ properties to $[-10, 10]$.
ii. Set both ‘XTick’ and ‘YTick’ properties to **-10:2:10**.
iii. Set both ‘XGrid’ and ‘YGrid’ properties to ‘on’.
iv. Set the ‘ButtonDownFcn’ property of the axes object to the callback function handle **@plotCurrentPoint_Callback**.

Write the callback function **plotCurrentPoint_Callback** so that it uses the **line** command to plot a point at the position of the mouse click. Set the ‘LineStyle’ to ’none’, ’Marker’ to ’o’, ’MarkerSize’ to 12, ’MarkerFaceColor’ to ’r’, and ’MarkerEdgeColor’, to ’k’. *Hint: There’s a bit of trickiness here as ‘CurrentPoint’ returns a* 2×3 *array. Try the following experiment. At the command prompt, type:*

```
>> figure
>> axes
>> grid
>> x=get(gca,'CurrentPoint')
```

The response might be something like the following:

```
x =
    0.3744    0.5658    1.0000
    0.3744    0.5658    0.0000
```

Note that **x(1,1)** *and* **x(1,2)** *contains the coordinates you need to use in your GUI.*

3. Write a Matlab GUI that contains one axes object. Set each of the following properties of the axes object to the indicated value.

i. Set both 'XLim' and 'YLim' properties to $[-10, 10]$.
ii. Set both 'XTick' and 'YTick' properties to **-10:2:10**.
iii. Set both 'XGrid' and 'YGrid' properties to 'on'.
iv. Set the 'ButtonDownFcn' property of the axes object to the callback function handle **@cursorCoordinates_Callback**.

Use Matlab's **text** command to print the coordinates of the origin on the screen.

```
hText=text(0,0,'(0,0)')
set(hText,'HorizontalAlignment','center')
```

Write the callback function **cursorCoordinates_Callback** that gets the 'CurrentPoint' of a mouse click in the axes object. Set the 'Position' property of the text object to the coordinates provided by 'CurrentPoint'. Concatenate the following string.

```
textStr=['(',num2str(x(1,1)),',',num2str(x(1,2)),')'];
```

Assign **textStr** as the value of the 'String' property of the text object.

4. Write a Matlab GUI that contains one axes object. Set each of the following properties of the axes object to the indicated value.

i. Set both 'XLim' and 'YLim' properties to $[-10, 10]$.
ii. Set both 'XTick' and 'YTick' properties to **-10:2:10**.

iii. Set both 'XGrid' and 'YGrid' properties to 'on'.
iv. Set the 'ButtonDownFcn' property of the axes object to the callback function handle **@cursorUpdateText__Callback**.

Create a panel as a child of the figure object with Matlab's **uipanel** command. Inside the panel, create a static text **uicontrol** as a child of the panel. Set the 'String' property of this control to '(0,0)' to start with. Write the callback function **cursorUpdateText__Callback** that gets the 'CurrentPoint' of a mouse click in the axes object and stores it in the variable **x**. Concatenate the following string.

```
textStr=['(',num2str(x(1,1)),',',num2str(x(1,2)),')'];
```

Assign **textStr** as the value of the 'String' property of the static text **uicontrol**. Thus, clicking the mouse in the axes should update the text object with the coordinates of the mouse cursor in the axes object.

5. Write a Matlab GUI that contains one axes object. Set each of the following properties of the axes object to the indicated value.

i. Set both 'XLim' and 'YLim' properties to $[-10, 10]$.
ii. Set both 'XTick' and 'YTick' properties to **-10:2:10**.
iii. Set both 'XGrid' and 'YGrid' properties to 'on'.
iv. Set the 'ButtonDownFcn' property of the axes object to the callback function handle **@updateLine__Callback**.

Initialize a line object with the command **hLine=line(0,0)**. Write a callback function **updateLine__Callback** that gets the 'CurrentPoint' of the mouse click in the axes object. Append the x- and y-coordinates of this new point to the value of the 'XData' and 'YData' properties of the line object having handle **hLine**. For example, to update the x-data, you will need these lines in the body of the callback.

```
newPoint=get(hObject,'CurrentPoint');
x=get(hLine,'XData');
set(hLine,'XData',[x,newPoint(1,1)])
```

Similar code is required to update the 'YData'. Thus, each time you click your mouse in the axes, a new line segmented is added, connecting the last point plotted to the current point clicked with the mouse.

6. Write a Matlab GUI that opens a figure with handle **hFig**. Set the property **WindowButtonMotionFcn** of the figure object to the function callback handle **@mouseCursor_Callback**. Create a **uipanel** object as a child of the figure object. Initialize a static text **uicontrol** that is a child of the **uipanel**. Initialize the 'String' property of this static text control to the empty string.

Write the callback with the form:

```
function mouseCursor_Callback(hObject,eventdata)
```

Get the current point of the mouse position in the figure window with:

```
x=get(hObject,'CurrentPoint');
```

Concatenate a text string as follows:

```
coordsStr=['(',num2str(x(1,1)),',',num2str(x(1,2)),')'];
```

Finally, set the 'String' property of the text **uicontrol** to the value **coordStr**. Moving the mouse in the figure window should show the pixel coordinates of the mouse in the static text control.

7. Write a Matlab GUI that opens a figure window with handle **hFig**. Set the **WindowButtonMotionFcn** property of the figure window to the callback **@mouseCursor_Calback**. Create an axes object with handle **hAx**. Set each of the following properties of the axes object to the indicated value.

i. Set both 'XLim' and 'YLim' properties to $[-10, 10]$.
ii. Set both 'XTick' and 'YTick' properties to **-10:2:10**.
iii. Set both 'XGrid' and 'YGrid' properties to 'on'.

Use the axes 'Position' property to locate and size the axes object within the figure window. Create a **uipanel** object as a child of the figure object. Initialize a static text **uicontrol** that is a child of the **uipanel**. Initialize the 'String' property of this static text control to the empty string. Write the callback with the form:

```
function mouseCursor_Callback(hObject,eventdata)
```

In the callback, get the current point of the mouse position in the figure window with:

```
newPoint=get(hObject,'CurrentPoint');
```

Next, get the position of the axes in the figure window with:

```
pos=get(hAx,'Position');
```

Test if the cursor is "in" the axes object with:

```
inAxes=(newPoint(1,1)>=pos(1)) && ...
            (newPoint(1,1)<pos(1)+pos(3)) && ...
            (newPoint(1,2)>pos(2)) && ...
            (newPoint(1,2)<pos(2)+pos(4));
```

Now, if the previous logical **inAxes** is **true**, get the 'CurrentPoint' in the axes (which differs from the 'CurrentPoint' in the figure window — it's given in axes coordinates), then use this value to set the 'String' property of the static text control. That is:

```
if inAxes
       x=get(hAx,'CurrentPoint');
       coordsStr=['(',num2str(x(1,1)),',',num2str(x(1,2)),')'];
       set(hText,'String',coordsStr);
end
```

Moving the mouse in the figure window should show mouse cursor coordinates in the static text control only when the mouse passes over the axes object. The coordinates will also be in axes coordinates, not figure pixel coordinates.

8. Adjust the GUI in **Exercise 7** so that the cursor will draw a path as it is dragged over the axes object. This is best accomplished by declaring and intializing a line object in the primary function.

```
hLine=line;
set(hLine,'XData',[],'YData',[]);
```

Now, if the mouse cursor is **inAxes**, then get ‘CurrentPoint’ from the axes object and use it to update the ’XData’ and ’YData’ properties of the line object, much as you did in **Exercise 5**.

9. The Matlab command **eval** is used to execute a string as a Matlab command. For example, try the following:

```
>> str='x=-3:3; y=x.^2';
>> eval(str)
y =
     9     4     1     0     1     4     9
```

Note that Matlab’s **eval** command simply executes the string as a command. You can also store the output of the **eval** command as follows:

```
>> x=-3:3;
>> y=eval('x.^2')
y =
     9     4     1     0     1     4     9
```

This gives us a way that we can interact with the user who wants to plot an equation. We set up an edit box, the user types in the equation, then we use the **eval** command to evaluate the equation on a domain, then plot the result.

Write a Matlab GUI that contains one axes object. Set each of the following properties of the axes object to the indicated value.

i. Set both ‘XLim’ and ‘YLim’ properties to $[-10, 10]$.
ii. Set both ‘XTick’ and ‘YTick’ properties to **-10:2:10**.
iii. Set both ‘XGrid’ and ‘YGrid’ properties to ‘on’.

Set up a **uicontrol** edit box in the figure window. If you wish, you can make things look nicer by first adding a panel, then placing your edit box inside the panel. Make the edit box large enough so that the user has room to enter his equation. You might set up a static text **uicontrol** in front of the edit box with its ‘String’ property set to ’y = ’ so that the user knows that he should only enter the right-hand side of the equation $y = f(x)$.

Set the 'Callback' property of the edit box to the handle **@plotFunction_Callback**. In writing the callback **plotFunction_Callback**, you will have to vectorize the equation string before evaluating it with **eval**, setting all the `*`, `/`, and `^` to their "dot" equivalents. Matlab provides the **vectorize** command for this purpose. For example, try the following at the command prompt.

```
>> str='x^2*sin(x)/exp(x+1)';
>> vectorize(str)
ans =
x.^2.*sin(x)./exp(x+1)
```

In the callback, you will want retrieve the string from the edit box, vectorize it, then evaluate the string on the domain $[-10, 10]$ and store the result. You can then plot the resulting data.

```
x=linspace(-10,10);
eqnStr=get(hEditEquation,'String');
eqnStr=vectorize(eqnStr);
y=eval(eqnStr);
plot(x,y)
```

A more sophisticated approach would be to create an empty line object in the primary function just prior to the callback.

```
hLine=line;
set(hLine,'XData',[],'YData',[]);
```

Then, instead of using the **plot** command, you can update the data in the line object.

```
set(hLine,'XData',x,'YData',y);
```

10. Following the hints given in **Exercise 9**, write a Matlab GUI that emulates the behavior of a graphing calculator. First, set up an axes object with the properties set as follows.

i. Set both ‘XLim’ and ‘YLim’ properties to $[-10, 10]$.
ii. Set both ‘XTick’ and ‘YTick’ properties to **-10:2:10**.
iii. Set both ‘XGrid’ and ‘YGrid’ properties to ‘on’.

Create three panels. One panel will hold an edit box where the user enters his equation, a second will hold edit boxes for the window parameters xmin, xmax, xscl, ymin, ymax, and yscl. You might also want to create an edit box in this second panel that contains the number of points to be plotted. Should a graph have a “jagged” appearance, increasing the number in this edit box would smoothen the graph. A third panel should contain a **uicontrol** that is a pushbutton. This pushbutton should have its ‘String’ property set to ‘Graph’ and ‘Callback’ property set to the handle **@plotFunction_Callback**.

The design should allow the user to enter his equation, window parameters, and number of points. When the user clicks the pushbutton with his mouse, the callback function **plotFunction_Callback** will grab the window parameter data and adjust ‘Xlim’, ‘YLim’, ‘XTick’, and ‘YTick’ of the axes object accordingly. Then the user’s equation should be drawn over the domain [Xmin,Xmax].

This problem could potentially be expanded into a semester ending project, adding ‘Zoom’ menu or a ‘Calc’ menu for finding zeros and extrema. This project is very open-ended with lots of potential for further implementation of ideas and utilities.

4.6 Answers

2. **linePlot.m**

4. **cursorText.m**

8. **mouseDraw.m**