MATLAB®

App Building

MATLAB®

MathWorks®

How to Contact MathWorks

Latest news: www.mathworks.com

Sales and services: www.mathworks.com/sales_and_services

User community: www.mathworks.com/matlabcentral

Technical support: www.mathworks.com/support/contact_us

Phone: 508-647-7000

The MathWorks, Inc.
3 Apple Hill Drive
Natick, MA 01760-2098

Contents

Introduction to Creating UIs

Create UIs with GUIDE

<div align="right">

Lay Out a UI Using GUIDE

</div>

6|

<div align="right">

Programming a GUIDE UI

</div>

7

Create UIs Programmatically

Lay Out a Programmatic UI

9

Code a Programmatic UI

10

Manage Application-Defined Data

11

12

App Designer

13

App Programming

16

Keyboard Shortcuts

17

App Packaging

Packaging GUIs as Apps

18

Introduction to Creating UIs

About Apps in MATLAB Software

Ways to Build Apps

There are different ways to build MATLAB® apps:

- "Use GUIDE" on page 1-2
- "Use MATLAB Functions to Create Apps Programmatically" on page 1-3
- "Use App Designer" on page 1-6

Each of these approaches offers a different workflow and a slightly different set of functionality. The best choice for you depends on your project requirements and how you prefer to work.

Use GUIDE

GUIDE is a drag-and-drop environment for laying out user interfaces (UIs). You code the interactive behavior of your app separately, in the MATLAB editor. Apps you create using GUIDE can display any type of MATLAB plot. GUIDE also provides various interactive components, including menus, tool bars, and tables. Use this approach to create simple apps that can display any type of plot.
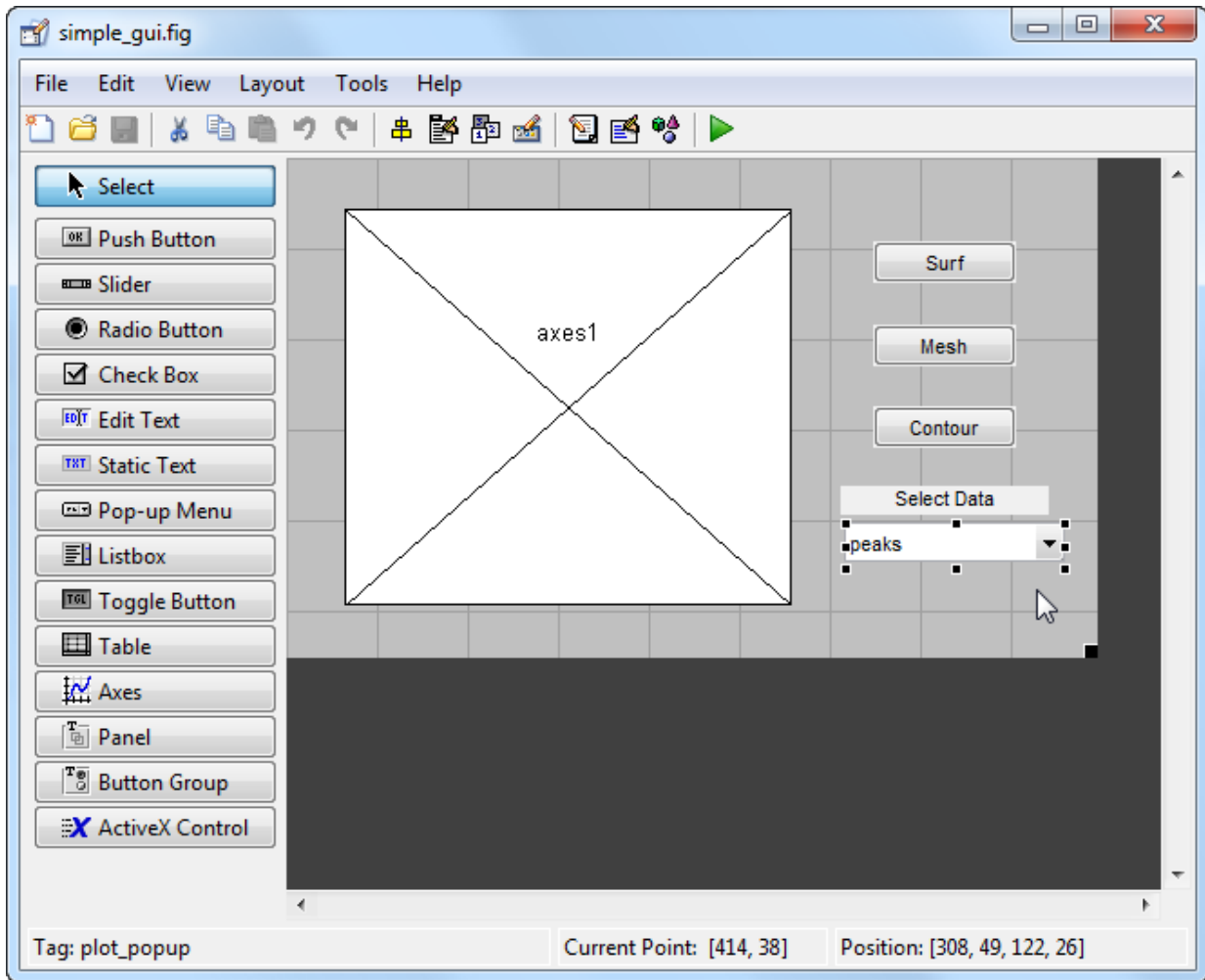
Use MATLAB Functions to Create Apps Programmatically

Instead of using GUIDE, you can code the layout and behavior of your app entirely using MATLAB functions. In this approach, you create a traditional figure and place interactive components in that figure programmatically. These apps support the same types of graphics and interactive components that GUIDE supports, as well as tabbed

panels. Use this approach to build complex apps with many interdependent components that can display any type of plot.
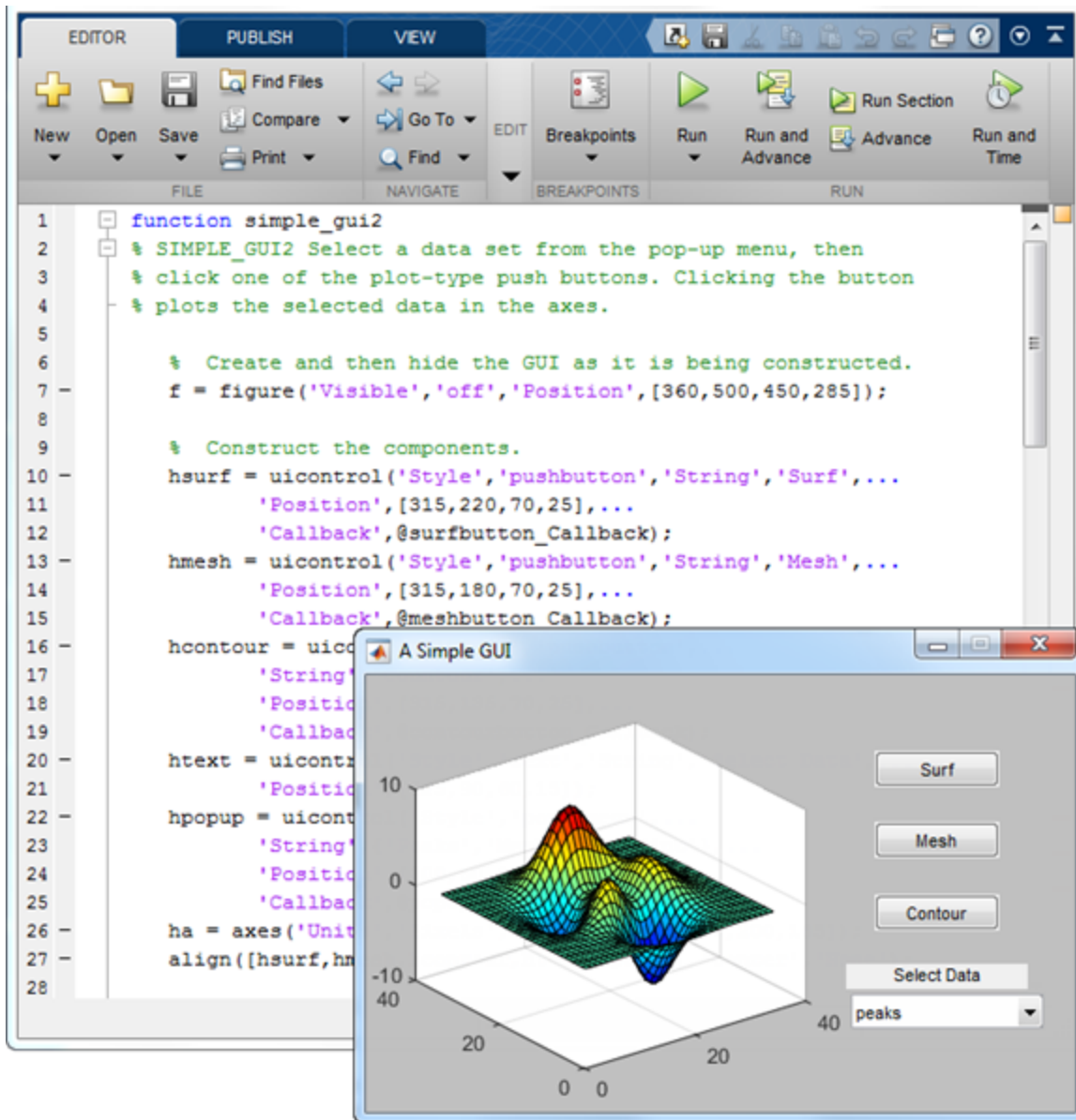
Use App Designer

App Designer is a rich drag-and-drop environment introduced in R2016a. It includes a fully integrated version of the MATLAB editor. The layout and code views are tightly linked so that changes you make in one view immediately affect the other. A larger set of interactive controls are available, including gauges, lamps, knobs, and switches. Simple scatter and line plots are also supported, but other graphics features are not available. Menus, tool bars, and tables are also not available. Use this approach for apps that require some of the new interactive components but do not require more than simple line or scatter plots.
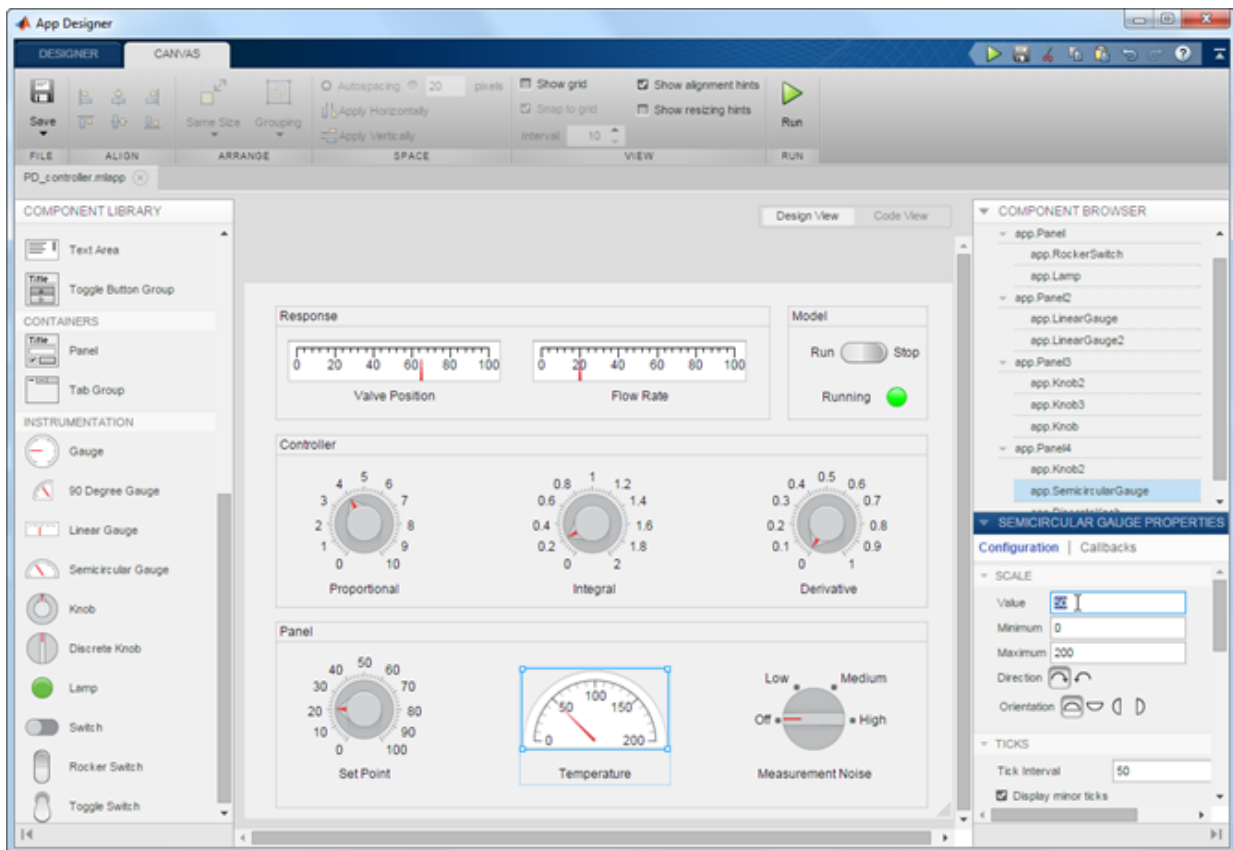
Related Examples

- "Create a Simple App Using GUIDE" on page 2-2
- "Create a Simple App Programmatically" on page 3-2
- "Create Simple App Using App Designer" on page 13-2

More About

- "Differences Between App Designer and GUIDE" on page 13-6
- "Graphics Support in App Designer" on page 13-11

How to Create a App with GUIDE

Create a Simple App Using GUIDE

Note: This topic applies to apps you create using GUIDE. For alternative ways to build apps, see "Ways to Build Apps" on page 1-2.

This example shows how to use GUIDE to create an app that has a simple user interface (UI), such as the one shown here.
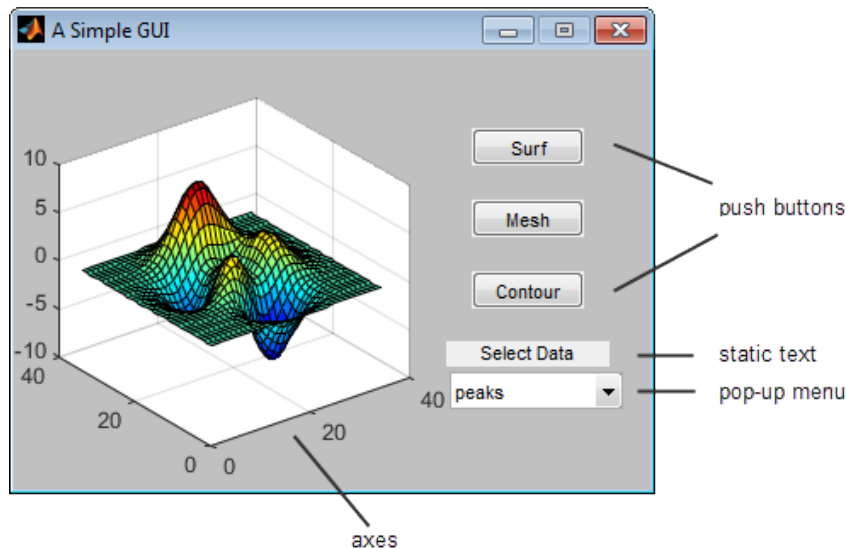
Subsequent sections guide you through the process of creating this app.

If you only want to view and run the code that created this app, set your current folder to one to which you have write access. Copy the example code and open it in the Editor by issuing the following MATLAB commands:

```
copyfile(fullfile(docroot, 'techdoc','creating_guis',...
   'examples','simple_gui*.*')),fileattrib('simple_gui*.*', '+w');
guide simple_gui.fig;
edit simple_gui.m
```

Click the **Run** ▶ button to run the app.

Open a New UI in the GUIDE Layout Editor

1 Start GUIDE by typing `guide` at the MATLAB prompt.
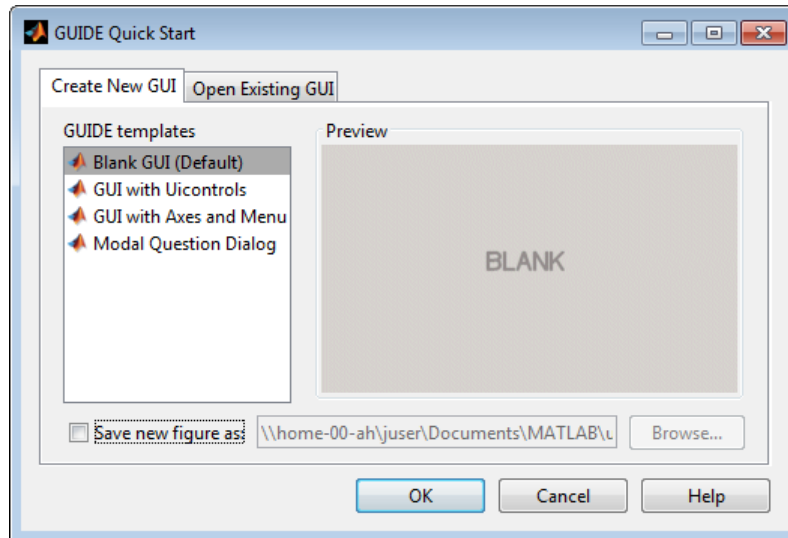
2 In the GUIDE Quick Start dialog box, select the **Blank (Default)** template, and then click **OK**.
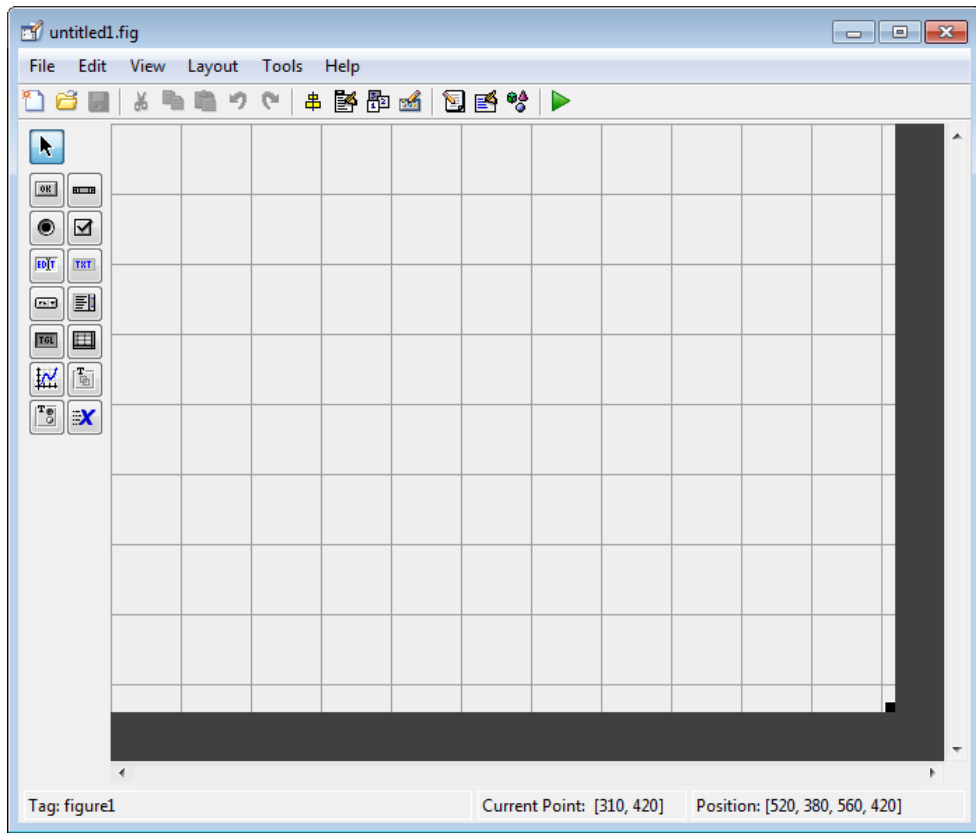
3 Display the names of the components in the component palette:

a Select **File** > **Preferences** > **GUIDE**.

b Select **Show names in component palette**.

c Click **OK**.
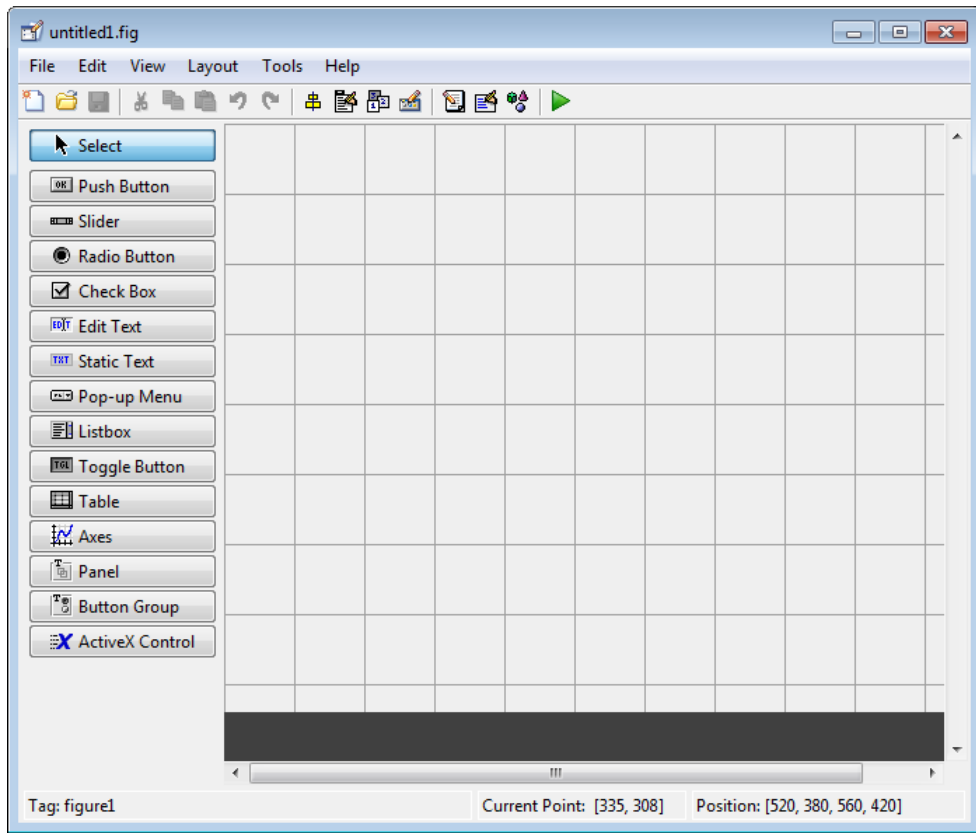
Set the Window Size in GUIDE

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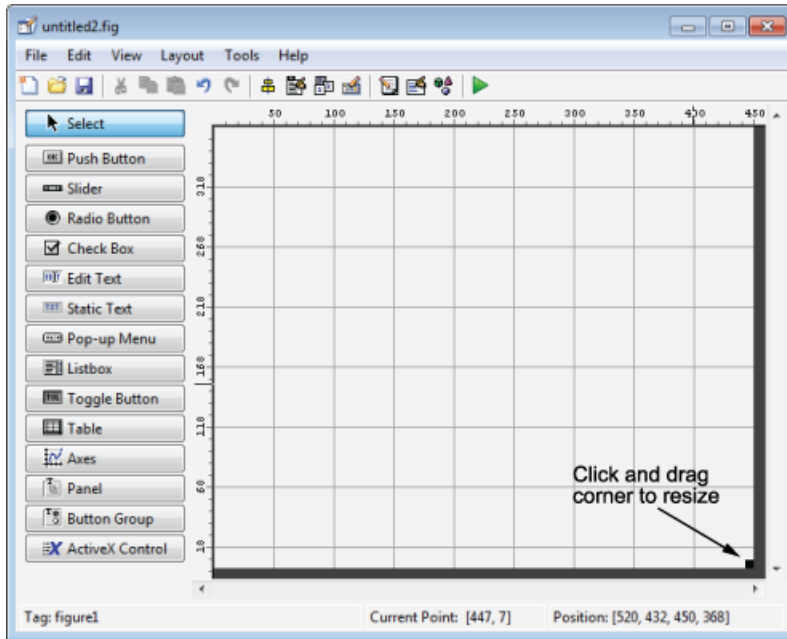
Layout the UI

Add, align, and label the components in the UI.

1 Add the three push buttons to the UI. Select the push button tool from the component palette at the left side of the Layout Editor and drag it into the layout area. Create three buttons, positioning them approximately as shown in the following figure.
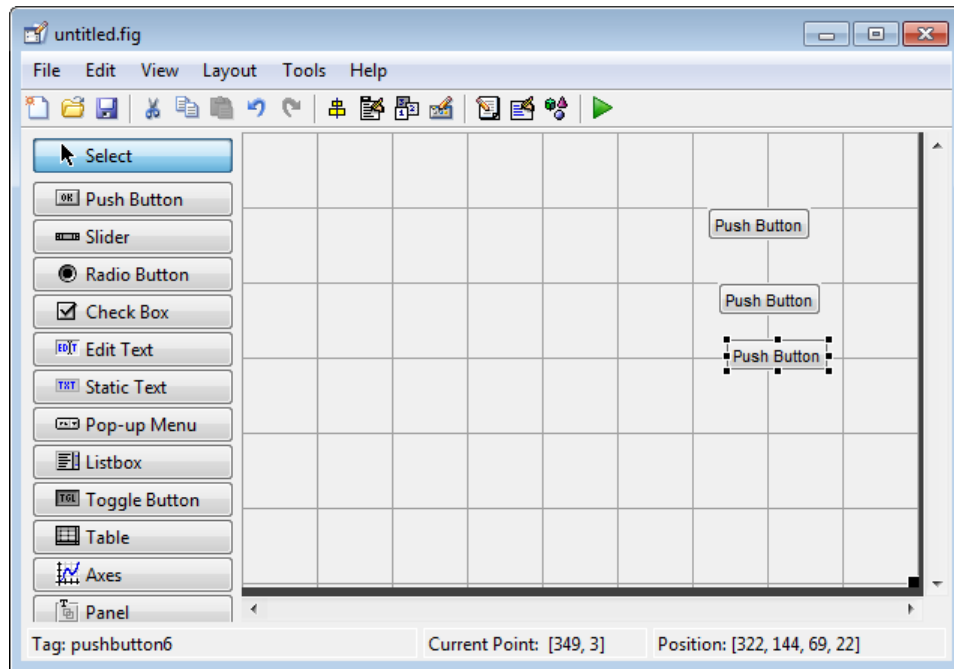
2 Add the remaining components to the UI.

- A static text area
- A pop-up menu
- An axes

Arrange the components as shown in the following figure. Resize the axes component to approximately 2-by-2 inches.
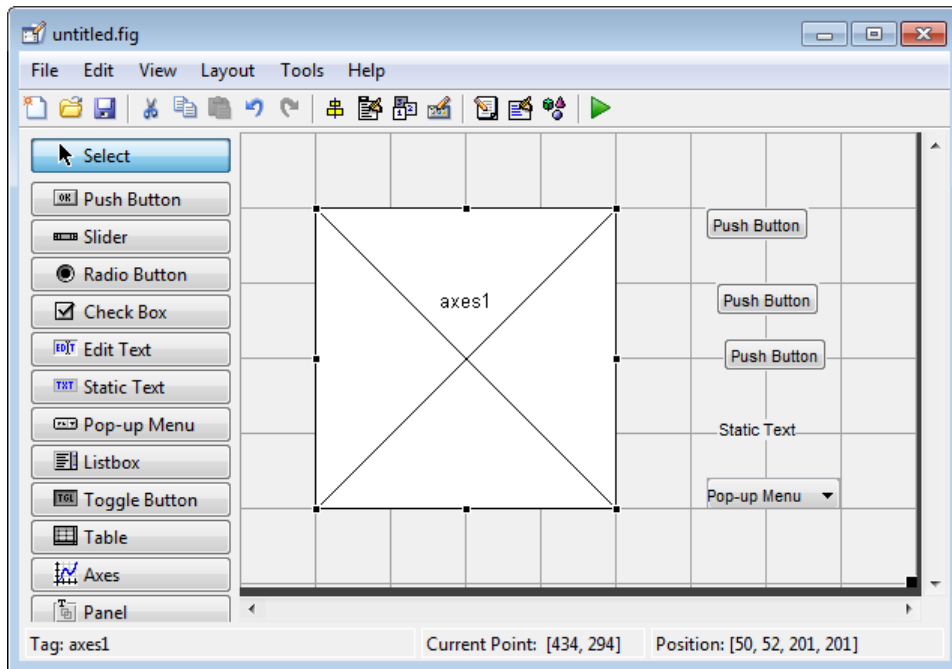
Align the Components

If several components have the same parent, you can use the Alignment Tool to align them to one another. To align the three push buttons:

1 Select all three push buttons by pressing **Ctrl** and clicking them.

2 Select **Tools** > **Align Objects**.

3 Make these settings in the Alignment Tool:

   •   Left-aligned in the horizontal direction.

   •   20 pixels spacing between push buttons in the vertical direction.
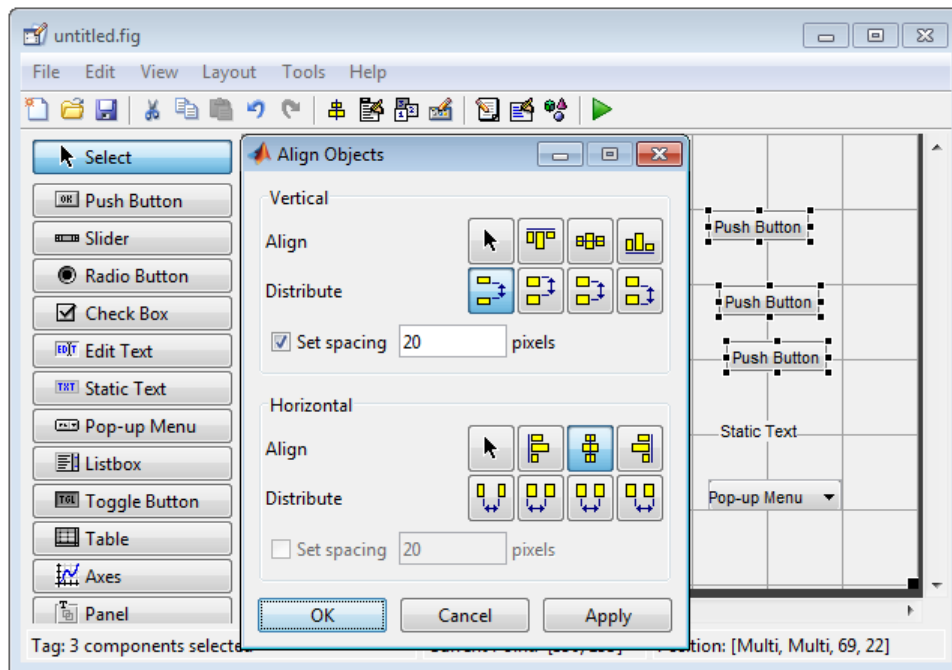
4 Click **OK**.
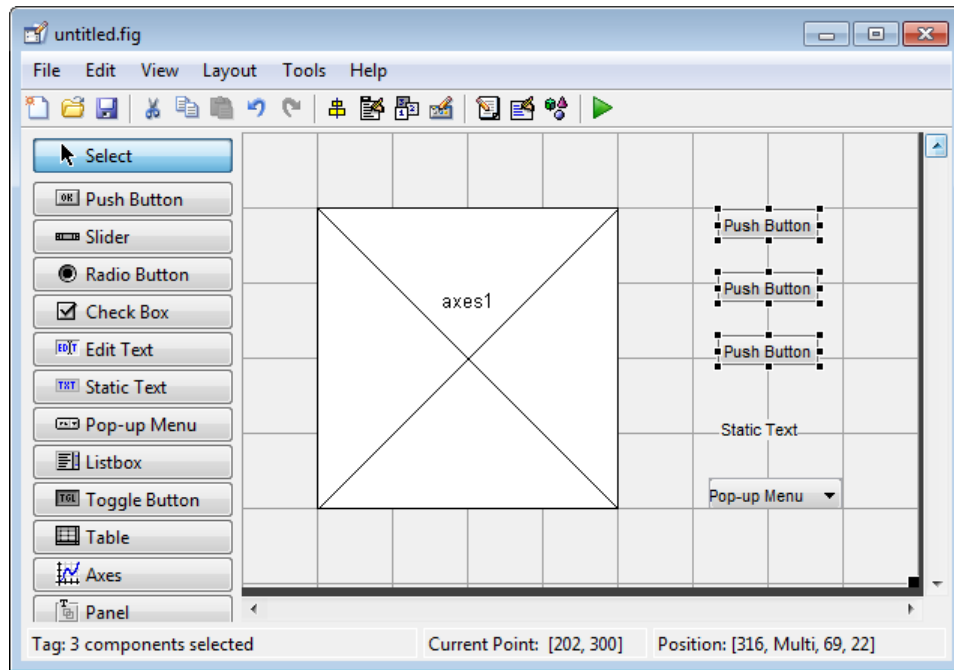
Label the Push Buttons

Each of the three push buttons specifies a plot type: surf, mesh, and contour. This section shows you how to label the buttons with those options.

1   Select **View** > **Property Inspector**.
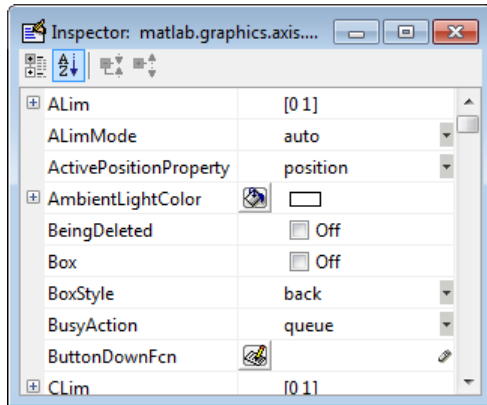
2 In the layout area, click the top push button.
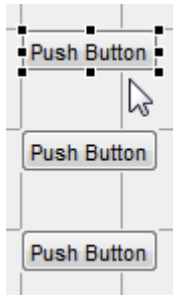
3 In the Property Inspector, select the `String` property, and then replace the existing value with the word `Surf`.

4 Click outside the `String` field. The push button label changes to **Surf**.

5 Click each of the remaining push buttons in turn and repeat steps 3 and 4. Label the middle push button **Mesh**, and the bottom button **Contour**.

List Pop-Up Menu Items

The pop-up menu provides a choice of three data sets: peaks, membrane, and sinc. These data sets correspond to MATLAB functions of the same name. This section shows you how to list those data sets as choices in the pop-menu.

1 In the layout area, click the pop-up menu.

2 In the Property Inspector, click the button next to `String`. The String dialog box displays.
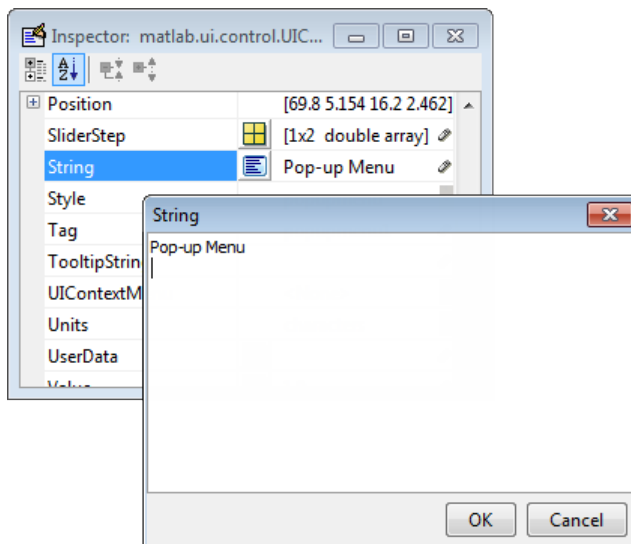
3 Replace the existing text with the names of the three data sets: peaks, membrane, and sinc. Press **Enter** to move to the next line.
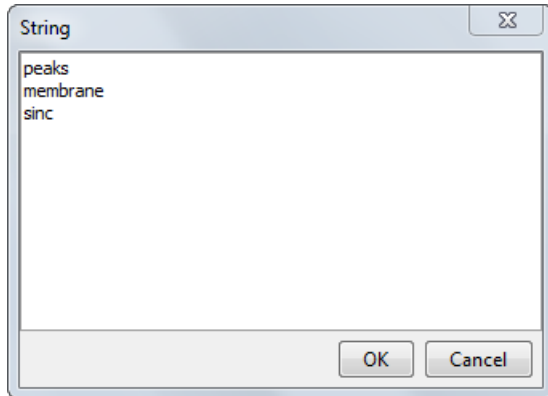
4 When you finish editing the items, click **OK**.

The first item in your list, `peaks`, appears in the pop-up menu in the layout area.
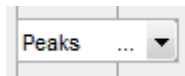
Modify the Static Text

In this UI, the static text serves as a label for the pop-up menu. This section shows you how to change the static text to read `Select Data`.

1 In the layout area, click the static text.

2 In the Property Inspector, click the button next to `String`. In the String dialog box that displays, replace the existing text with the phrase `Select Data`.
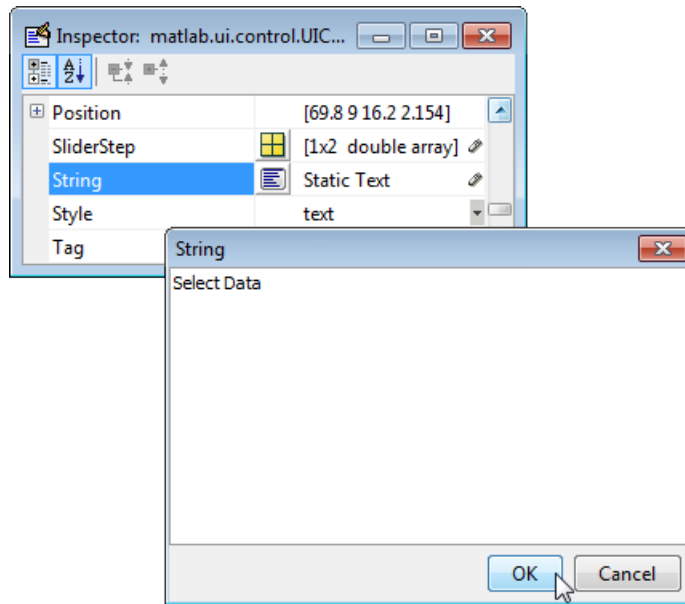
3 Click **OK**.

The phrase `Select Data` appears in the static text component above the pop-up menu.
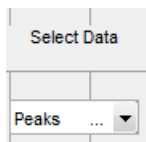
Save the Layout

When you save a layout, GUIDE creates two files, a FIG-file and a code file. The FIG-file, with extension `.fig`, is a binary file that contains a description of the layout. The code file, with extension `.m`, contains MATLAB functions that control the app's behavior.

1 Save and run your program by selecting **Tools** > **Run**.

2 GUIDE displays a dialog box displaying: "Activating will save changes to your figure file and MATLAB code. Do you wish to continue?

Click **Yes**.

3 GUIDE opens a **Save As** dialog box in your current folder and prompts you for a FIG-file name.

4 Browse to any folder for which you have write privileges, and then enter the file name `simple_gui` for the FIG-file. GUIDE saves both the FIG-file and the code file using this name.

5 If the folder in which you save the files is not on the MATLAB path, GUIDE opens a dialog box that allows you to change the current folder.

6 GUIDE saves the files `simple_gui.fig` and `simple_gui.m`, and then runs the program. It also opens the code file in your default editor.

The app opens in a new window. Notice that the window lacks the standard menu bar and toolbar that MATLAB figure windows display. You can add your own menus and toolbar buttons with GUIDE, but by default a GUIDE app includes none of these components.

When you run `simple_gui`, you can select a data set in the pop-up menu and click the push buttons, but nothing happens. This is because the code file contains no statements to service the pop-up menu and the buttons.
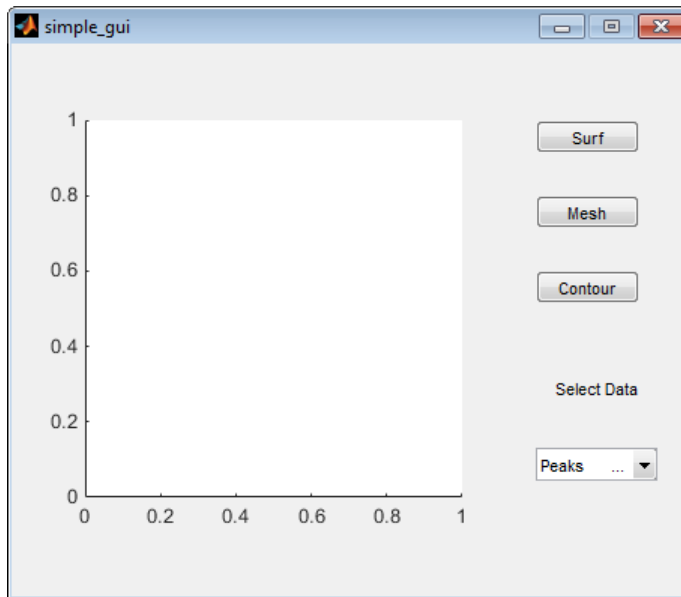
To run an app created with GUIDE without opening GUIDE, execute its code file by typing its name.

```
 simple_gui
```
You can also use the run command with the code file, for example,

```
run simple_gui
```

Note: Do not attempt to run your app by opening its FIG-file outside of GUIDE. If you do so, the figure opens and appears ready to use, but the UI does not initialize and the callbacks do not function.

Code the Behavior of the App

When you saved your layout in the previous section, "Save the Layout" on page 2-14, GUIDE created two files: a FIG-file, simple_gui.fig, and a program file, simple_gui.m. However, the app is not responsive because simple_gui.m does not contain any statements that perform actions. This section shows you how to add code to the file to make the app functional.

Generate Data to Plot

This section shows you how to generate the data to be plotted when the user clicks a button. The *opening function* generates this data by calling MATLAB functions. The opening function initializes the UI when it opens, and it is the first callback in every GUIDE-generated code file.

In this example, you add code that creates three data sets to the opening function. The code uses the MATLAB functions `peaks`, `membrane`, and `sinc`.

1   Display the opening function in the MATLAB Editor.

    If the file `simple_gui.m` is not already open in the editor, open from the Layout Editor by selecting **View** > **Editor**.

2   On the **EDITOR** tab, in the **NAVIGATE** section, click **Go To**, and then select `simple_gui_OpeningFcn`.

    The cursor moves to the opening function, which contains this code:

```
% --- Executes just before simple_gis made visible.
function simple_gui_OpeningFcn(hObject, eventdata, handles, varargin)
% This function has no output args, see OutputFcn.
% hObject    handle to figure
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)
% varargin   command line arguments to simple_g(see VARARGIN)

% Choose default command line output for simple_gui
handles.output = hObject;

% Update handles structure
guidata(hObject, handles);

% UIWAIT makes simple_gwait for user response (see UIRESUME)
% uiwait(handles.figure1);
```

3   Create data to plot by adding the following code to the opening function immediately after the comment that begins `% varargin...`

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

2-17

```
handles.current_data = handles.peaks;
surf(handles.current_data)
```

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

The last two lines create a current data value and set it to peaks, and then display the surf plot for peaks. The following figure shows how the looks when it first displays.
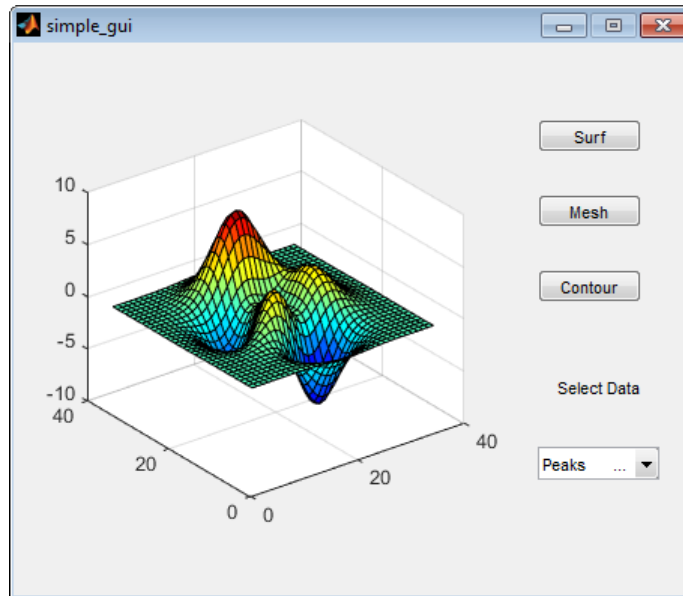
Code Pop-Up Menu Behavior

The pop-up menu presents options for plotting the data. When the user selects one of the three plots, MATLAB software sets the pop-up menu `Value` property to the index of the selected string. The pop-up menu callback reads the pop-up menu `Value` property to determine the item that the menu currently displays, and sets `handles.current_data` accordingly.

1   Display the pop-up menu callback in the MATLAB Editor. In the GUIDE Layout Editor, right-click the pop-up menu component, and then select **View Callbacks** > **Callback**.
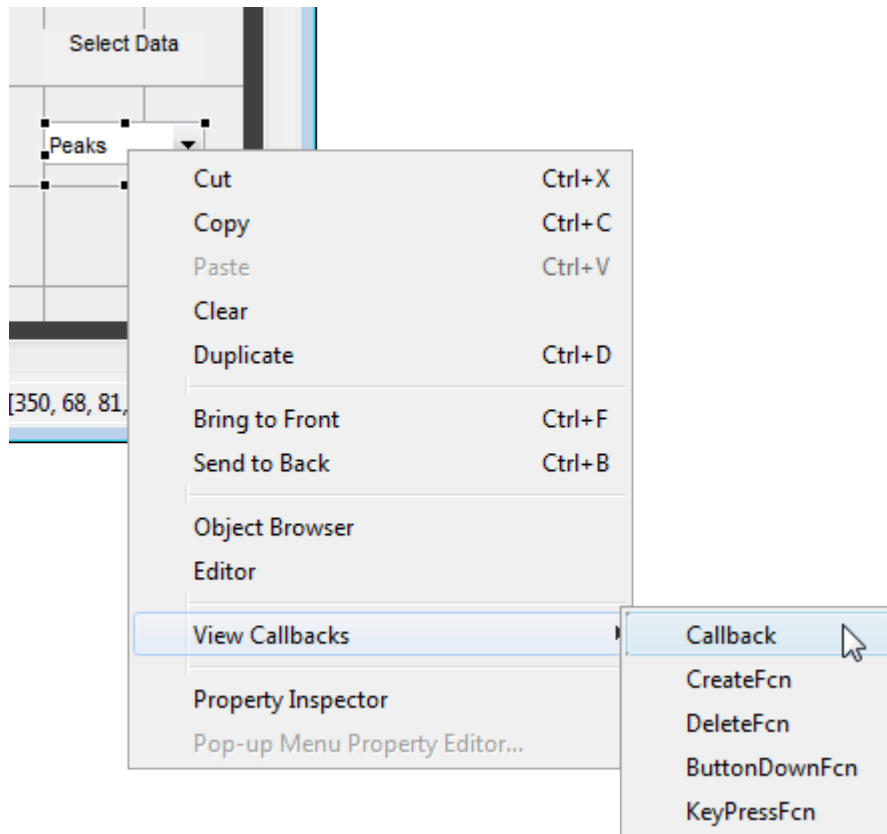
GUIDE displays the code file in the Editor, and moves the cursor to the pop-menu callback, which contains this code:

```
% --- Executes on selection change in popupmenu1.
function popupmenu1_Callback(hObject, eventdata, handles)
% hObject    handle to popupmenu1 (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)
```

2   Add the following code to the popupmenu1_Callback after the comment that begins % handles...

This code first retrieves two pop-up menu properties:

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

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

```
% Determine the selected data set.
str = get(hObject, 'String');
val = get(hObject,'Value');
% Set current data to the selected data set.
switch str{val};
case 'peaks' % User selects peaks.
   handles.current_data = handles.peaks;
case 'membrane' % User selects membrane.
   handles.current_data = handles.membrane;
case 'sinc' % User selects sinc.
   handles.current_data = handles.sinc;
end
% Save the handles structure.
guidata(hObject,handles)
```

Code Push Button Behavior

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

1 Display the **Surf** push button callback in the MATLAB Editor. In the Layout Editor, right-click the **Surf** push button, and then select **View Callbacks** > **Callback**.
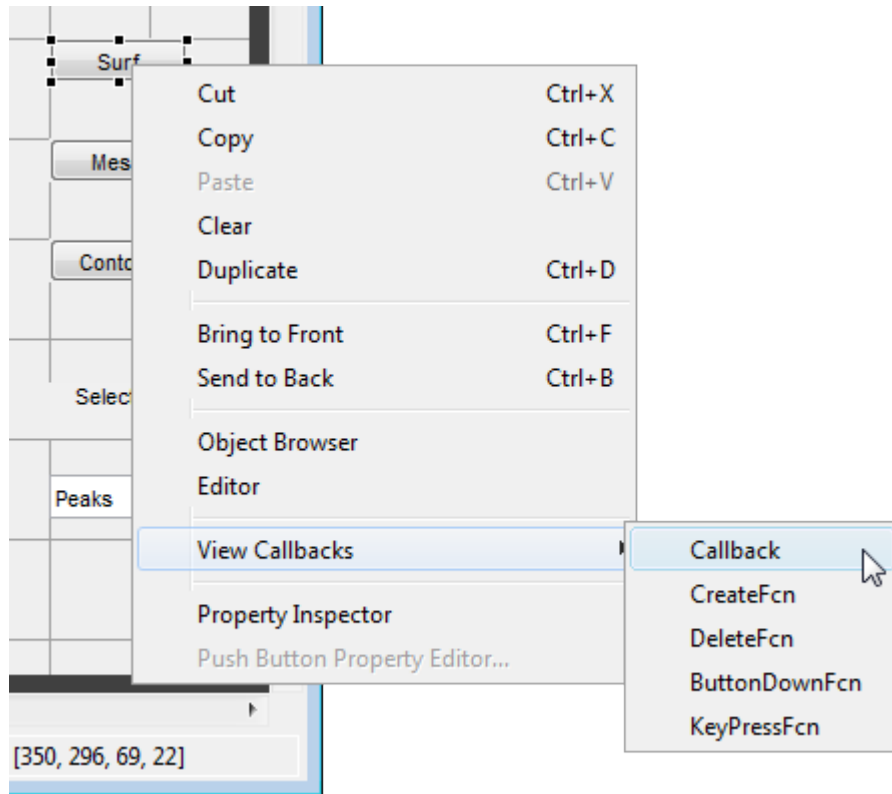
In the Editor, the cursor moves to the **Surf** push button callback in the code file, which contains this code:

```
% --- Executes on button press in pushbutton1.
function pushbutton1_Callback(hObject, eventdata, handles)
% hObject    handle to pushbutton1 (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)
```

2 Add the following code to the callback immediately after the comment that begins `%` `handles...`

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

3 Repeat steps 1 and 2 to add similar code to the **Mesh** and **Contour** push button callbacks.

- Add this code to the **Mesh** push button callback, `pushbutton2_Callback`:

  ```
  % Display mesh plot of the currently selected data.
  mesh(handles.current_data);
  ```
- Add this code to the **Contour** push button callback, `pushbutton3_Callback`:

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

4   Save your code by selecting **File** > **Save**.

Run the App

In "Code the Behavior of the App" on page 2-16, you programmed the pop-up menu and the push buttons. You also created data for them to use and initialized the display. Now you can run your program to see how it works.

1   Run your program from the Layout Editor by selecting **Tools** > **Run**.
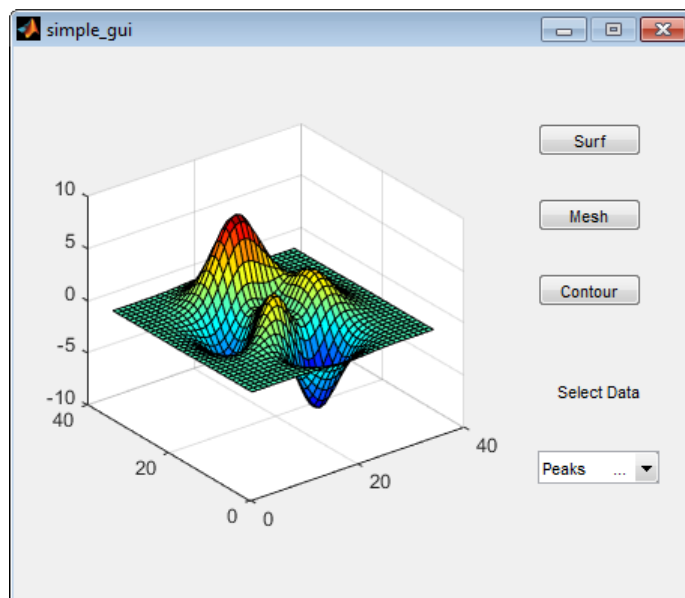
2   In the pop-up menu, select **Membrane**, and then click the **Mesh** button. The displays a mesh plot of the MathWorks® L-shaped Membrane logo.

3 Try other combinations before closing the window.

Related Examples

- "Create a Simple App Programmatically" on page 3-2
- "Create Simple App Using App Designer" on page 13-2

More About

- "Ways to Build Apps" on page 1-2

Files Generated by GUIDE

In this section...
"Code Files and FIG-Files" on page 2-24
"Code File Structure" on page 2-24
"Adding Callback Templates to an Existing Code File" on page 2-25
"About GUIDE-Generated Callbacks" on page 2-26

Code Files and FIG-Files

By default, the first time you save or run your program, GUIDE save two files:

- A FIG-file, with extension `.fig`, that contains a complete description of the layout and each component, such as push buttons, axes, panels, menus, and so on. The FIG-file is a binary file and you cannot modify it except by changing the layout in GUIDE. FIG-files are specializations of MAT-files. See "Custom Applications to Access MAT-Files" for more information.

- A code file, with extension `.m`, that initially contains initialization code and templates for some callbacks that control behavior. You generally add callbacks you write for your components to this file. As the callbacks are functions, the code file can never be a MATLAB script.

  When you save your the first time, GUIDE automatically opens the code file in your default editor.

The FIG-file and the code file must have the same name. These two files usually reside in the same folder, and correspond to the tasks of laying out and programming the UI. When you lay out the in the Layout Editor, your components and layout is stored in the FIG-file. When you program the UI, your code is stored in the corresponding code file.

If your includes ActiveX® components, GUIDE also generates a file for each ActiveX component.

Code File Structure

The code file that GUIDE generates is a function file. The name of the main function is the same as the name of the code file. For example, if the name of the code file is

`mygui.m`, then the name of the main function is `mygui`. Each callback in the file is a local function of that main function.

When GUIDE generates a code file, it automatically includes templates for the most commonly used callbacks for each component. The code file also contains initialization code, as well as an opening function callback and an output function callback. It is your job to add code to the component callbacks for your program to work as you want. You can also add code to the opening function callback and the output function callback. The code file orders functions as shown in the following table.

Section	Description
Comments	Displayed at the command line in response to the `help` command.
Initialization	GUIDE initialization tasks. *Do not edit this code.*
Opening function	Performs your initialization tasks before the user has access to the UI.
Output function	Returns outputs to the MATLAB command line after the opening function returns control and before control returns to the command line.
Component and figure callbacks	Control the behavior of the window and of individual components. MATLAB software calls a callback in response to a particular event for a component or for the figure itself.
Utility/helper functions	Perform miscellaneous functions not directly associated with an event for the figure or a component.

Adding Callback Templates to an Existing Code File

When you save the UI, GUIDE automatically adds templates for some callbacks to the code file. If you want to add other callbacks to the file, you can easily do so.

Within GUIDE, you can add a local callback function template to the code in any of the following ways. Select the component for which you want to add the callback, and then:

- Right-click the mouse button, and from the **View callbacks** submenu, select the desired callback.
- From **View** > **View Callbacks**, select the desired callback.

- Double-click a component to show its properties in the Property Inspector. In the

  Property Inspector, click the pencil-and-paper icon ![pencil icon] next to the name of the callback you want to install in the code file.

- For toolbar buttons, in the Toolbar Editor, click the **View** button next to **Clicked Callback** (for Push Tool buttons) or **On Callback**, or **Off Callback** (for Toggle Tools).

When you perform any of these actions, GUIDE adds the callback template to the code file, saves it, and opens it for editing at the callback you just added. If you select a callback that currently exists in the code file, GUIDE adds no callback, but saves the file and opens it for editing at the callback you select.

For more information, see "GUIDE-Generated Callback Functions and Property Values" on page 7-4.

About GUIDE-Generated Callbacks

Callbacks created by GUIDE for components are similar to callbacks created programmatically, with certain differences.

- GUIDE generates callbacks as function templates within the code file.

  GUIDE names callbacks based on the callback type and the component `Tag` property. For example, `togglebutton1_Callback` is such a default callback name. If you change a component `Tag`, GUIDE renames all its callbacks in the code file to contain the new tag. You can change the name of a callback, replace it with another function, or remove it entirely using the Property Inspector.

- GUIDE provides three arguments to callbacks, always named the same.

- You can append arguments to GUIDE-generated callbacks, but never alter or remove the ones that GUIDE places there.

- You can rename a GUIDE-generated callback by editing its name or by changing the component `Tag`.

- You can delete a callback from a component by clearing it from the Property Inspector; this action does not remove anything from the code file.

- You can specify the same callback function for multiple components to enable them to share code.

After you delete a component in GUIDE, all callbacks it had remain in the code file. If you are sure that no other component uses the callbacks, you can then remove the

callback code manually. For details, see "Renaming and Removing GUIDE-Generated Callbacks" on page 7-6.

Related Examples

A Simple Programmatic App

Create a Simple App Programmatically

Note: This topic applies to apps you create programmatically using the figure function. For alternative ways to build apps, see "Ways to Build Apps" on page 1-2.

This example shows how to create a simple app programmatically, such as the one shown here.
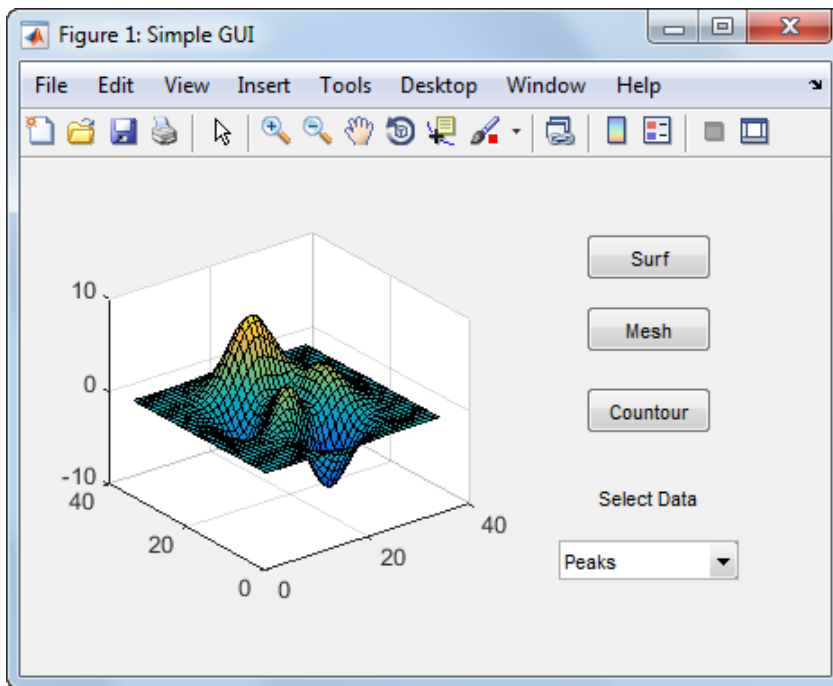
Subsequent sections guide you through the process of creating this app.

If you prefer to view and run the code that created this app without creating it, set your current folder to one to which you have write access. Copy the example code and open it in the Editor by issuing the following MATLAB commands:

```
copyfile(fullfile(docroot, 'techdoc','creating_guis',...
  'examples','simple_gui2*.*')), fileattrib('simple_gui2*.*', '+w');
```

```
edit simple_gui2.m
```

Note: This code uses dot notation to set graphics object properties. Dot notation runs in R2014b and later. If you are using an earlier release, use the `set` function instead. For example, change `f.Visible = 'on'` to `set(f,'Visible','on')`.

To run the code, go to the **Run** section in the **Editor** tab. Then click **Run** ▶.

Create a Code File

Create a function file (as opposed to a *script file*, which contains a sequence of MATLAB commands but does not define functions).

1 At the MATLAB prompt, type `edit`.

2 Type the following statement in the first line of the Editor.

```
function simple_gui2
```

3 Following the function statement, type these comments, ending with a blank line. (The comments display at the command line in response to the `help` command.)

```
% SIMPLE_GUI2 Select a data set from the pop-up menu, then
% click one of the plot-type push buttons. Clicking the button
% plots the selected data in the axes.
```
(Leave a blank line here)

4 At the end of the file, after the blank line, add an `end` statement. This `end` statement is required because the example uses nested functions. To learn more, see "Nested Functions".

5 Save the file in your current folder or at a location that is on your MATLAB path.

Create the Figure Window

To create a container for your app's user interface (UI), add the following code before the `end` statement in your file:

```
% Create and then hide the UI as it is being constructed.
f = figure('Visible','off','Position',[360,500,450,285]);
```

The call to the `figure` function creates a traditional figure and sets the following properties:

- The `Visible` property makes the window invisible so that the user cannot see the components being added or initialized.

  The window becomes visible when the UI has all its components and is initialized.
- The `Position` property is a four-element vector that specifies the location of the UI on the screen and its size: [distance from left, distance from bottom, width, height]. Default units are pixels.

Add Components to the UI

Create the push buttons, static text, pop-up menu, and axes components to the UI.

1 Following the call to `figure`, add these statements to your code file to create three push button components.

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

Each statement uses a series of `uicontrol` property/value pairs to define a push button:

- The `Style` property specifies that the uicontrol is a push button.
- The `String` property specifies the label on each push button: `Surf`, `Mesh`, and `Contour`.
- The `Position` property specifies the location and size of each push button: [distance from left, distance from bottom, width, height]. Default units for push buttons are pixels.

Each `uicontrol` call returns the handle of the push button created.

2 Add the pop-up menu and its static text label by adding these statements to the code file following the push button definitions. The first statement creates a popup menu. The second statement creates a text component that serves as a label for the popup menu.

```
htext  = uicontrol('Style','text','String','Select Data',...
                'Position',[325,90,60,15]);
```

```
hpopup = uicontrol('Style','popupmenu',...
           'String',{'Peaks','Membrane','Sinc'},...
           'Position',[300,50,100,25]);
```

The pop-up menu component `String` property uses a cell array to specify the three items in the pop-up menu: `Peaks`, `Membrane`, and `Sinc`.

The text component, the `String` property specifies instructions for the user.

For both components, the `Position` property specifies the size and location of each component: [distance from left, distance from bottom, width, height]. Default units for these components are pixels.

3 Add the axes by adding this statement to the code file.

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

The `Units` property specifies pixels so that the axes has the same units as the other components.

4 Following all the component definitions, add this line to the code file to align all components, except the axes, along their centers.

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

5 Add this command following the `align` command.

Note: This code uses dot notation to set object properties. Dot notation runs in R2014b and later. If you are using an earlier release, use the `set` function instead. For example, change `f.Visible = 'on'` to `set(f,'Visible','on')`.

```
f.Visible = 'on';
```

Your code file should look like this:

```
function simple_gui2
% SIMPLE_GUI2 Select a data set from the pop-up menu, then
% click one of the plot-type push buttons. Clicking the button
% plots the selected data in the axes.

    %  Create and then hide the UI as it is being constructed.
    f = figure('Visible','off','Position',[360,500,450,285]);
```

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

%Make the UI visible.
f.Visible = 'on';

end
```

6   Run your code by typing `simple_gui2` at the command line. You can select a data
    set in the pop-up menu and click the push buttons, but nothing happens. This
    is because there is no callback code in the file to service the pop-up menu or the
    buttons.
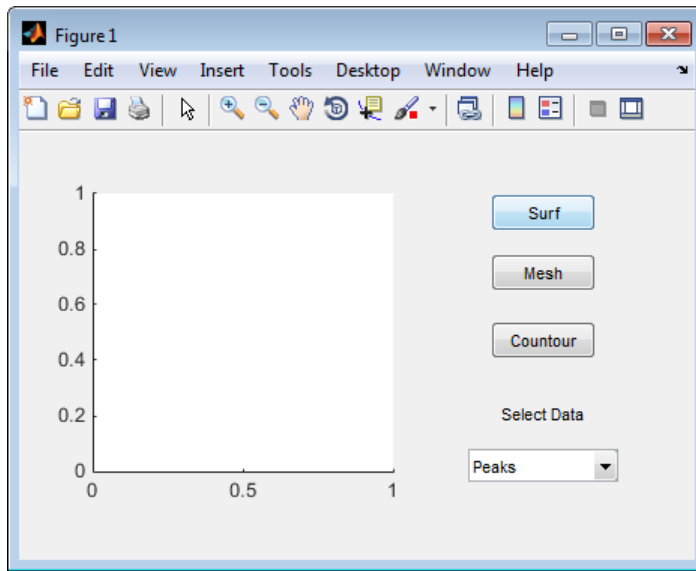
Code the App's Behavior

Program the Pop-Up Menu

The pop-up menu enables users to select the data to plot. When a user selects one of the three data sets in the pop-up menu, MATLAB software sets the pop-up menu `Value` property to the index of the selected string. The pop-up menu callback reads the pop-up menu `Value` property to determine which item is currently displayed and sets `current_data` accordingly.

Add the following callback to your file following the initialization code and before the final `end` statement.

Note: This code uses dot notation to get object properties. Dot notation runs in R2014b and later. If you are using an earlier release, use the `get` function instead. For example, change `str = source.String` to `str = get(source,'String')`.

```
%  Pop-up menu callback. Read the pop-up menu Value property to
```

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

Program the Push Buttons

Each of the three push buttons creates a different type of plot using the data specified by the current selection in the pop-up menu. The push button callbacks plot the data in current_data. They automatically have access to current_data because they are nested at a lower level.

Add the following callbacks to your file following the pop-up menu callback and before the final end statement.

```matlab
% Push button callbacks. Each callback plots current_data in the
% specified plot type.

function surfbutton_Callback(source,eventdata)
% Display surf plot of the currently selected data.
     surf(current_data);
end

function meshbutton_Callback(source,eventdata)
% Display mesh plot of the currently selected data.
     mesh(current_data);
end

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

Program the Callbacks

When the user selects a data set from the pop-up menu or clicks one of the push buttons, MATLAB software executes the callback associated with that particular event. Use each component's `Callback` property to specify the name of the callback with which each event is associated.

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

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

so that the statement looks like this:

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

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

2 To the `uicontrol` statement that defines the **Mesh** push button, add the property/ value pair

```
'Callback',@meshbutton_Callback
```

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

```
'Callback',@contourbutton_Callback
```

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

```
'Callback',@popup_menu_Callback
```

For more information, see "Write Callbacks for Apps Created Programmatically" on page 10-5.

Initialize the UI

Initialize the UI, so it is ready when the window becomes visible. Make the UI behave properly when it is resized by changing the component and figure units to `normalized`. This causes the components to resize when the UI is resized. Normalized units map the lower-left corner of the figure window to `(0,0)` and the upper-right corner to `(1.0, 1.0)`.

Note: This code uses dot notation to set object properties. Dot notation runs in R2014b and later. If you are using an earlier release, use the `set` function instead. For example, change `f.Units = 'normalized'` to `set(f,'Units','normalized')`.

Replace this code in editor:

```matlab
% Make the UI visible.
f.Visible = 'on';
```

with this code:

```matlab
% Initialize the UI.
% Change units to normalized so components resize automatically.
f.Units = 'normalized';
ha.Units = 'normalized';
hsurf.Units = 'normalized';
hmesh.Units = 'normalized';
hcontour.Units = 'normalized';
htext.Units = 'normalized';
hpopup.Units = 'normalized';

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

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

% Assign a name to appear in the window title.
f.Name = 'Simple GUI';

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

% Make the UI visible.
f.Visible = 'on';
```

Verify Code and Run the App

Make sure your code appears as it should, and then run it.

Note: This code uses dot notation to get and set object properties. Dot notation runs in R2014b and later. If you are using an earlier release, use the `get` and `set` functions instead. For example, change `f.Units = 'normalized'` to `set(f,'Units','normalized')`.

---

1 Verify that your code file looks like this:

```
function simple_gui2
% SIMPLE_GUI2 Select a data set from the pop-up menu, then
% click one of the plot-type push buttons. Clicking the button
% plots the selected data in the axes.

%  Create and then hide the UI as it is being constructed.
f = figure('Visible','off','Position',[360,500,450,285]);

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

% Initialize the UI.
% Change units to normalized so components resize automatically.
f.Units = 'normalized';
```

```matlab
      ha.Units = 'normalized';
      hsurf.Units = 'normalized';
      hmesh.Units = 'normalized';
      hcontour.Units = 'normalized';
      htext.Units = 'normalized';
      hpopup.Units = 'normalized';

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

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

      % Assign the a name to appear in the window title.
      f.Name = 'Simple GUI';

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

      % Make the window visible.
      f.Visible = 'on';

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

```
 end

% Push button callbacks. Each callback plots current_data in the
% specified plot type.

function surfbutton_Callback(source,eventdata)
% Display surf plot of the currently selected data.
     surf(current_data);
end

function meshbutton_Callback(source,eventdata)
% Display mesh plot of the currently selected data.
     mesh(current_data);
end

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

2   Run your app by typing `simple_gui2` at the command line. The initialization code
causes it to display the default `peaks` data with the `surf` function, making the UI
look like this.
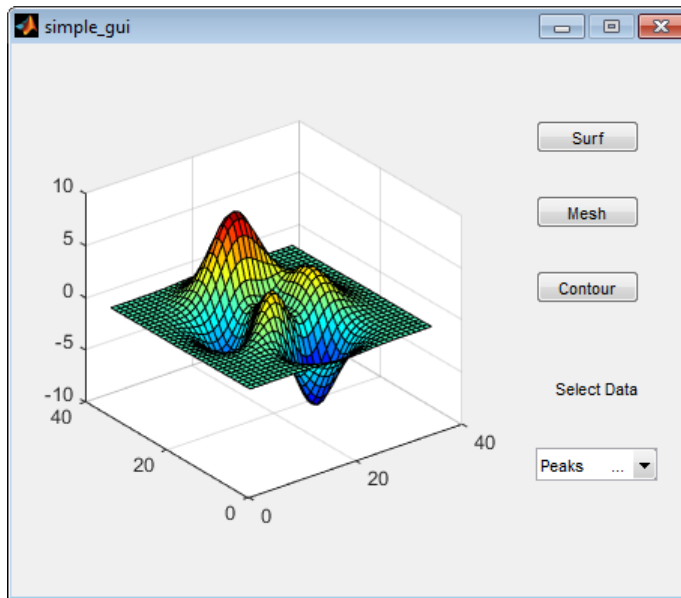
3. In the pop-up menu, select **Membrane**, and then click the **Mesh** button. The UI displays a mesh plot of the MathWorks L-shaped Membrane logo.

4. Try other combinations before closing the UI.

5. Type help simple_gui2 at the command line. MATLAB software displays the help text.

```
help simple_gui2
  SIMPLE_GUI2 Select a data set from the pop-up menu, then
  click one of the plot-type push buttons. Clicking the button
  plots the selected data in the axes.
```

Related Examples

- "Create a Simple App Using GUIDE" on page 2-2
- "Create Simple App Using App Designer" on page 13-2

More About

- "Ways to Build Apps" on page 1-2

Create UIs with GUIDE

What Is GUIDE?

GUIDE: Getting Started

In this section...
"UI Layout" on page 4-2
"UI Programming" on page 4-2

UI Layout

GUIDE is a development environment that provides a set of tools for creating user interfaces (UIs). These tools simplify the process of laying out and programming UIs.

Using the GUIDE Layout Editor, you can populate a UI by clicking and dragging UI components—such as axes, panels, buttons, text fields, sliders, and so on—into the layout area. You also can create menus and context menus for the UI. From the Layout Editor, you can size the UI, modify component look and feel, align components, set tab order, view a hierarchical list of the component objects, and set UI options.

UI Programming

GUIDE automatically generates a program file containing MATLAB functions that controls how the UI behaves. This code file provides code to initialize the UI, and it contains a framework for the UI callbacks. Callbacks are functions that execute when the user interacts with a UI component. Use the MATLAB Editor to add code to these callbacks.

Note MATLAB software provides a selection of standard dialog boxes that you can create with a single function call. For an example, see the documentation for msgbox, which also provides links to functions that create specialized predefined dialog boxes.

GUIDE Tools Summary

The GUIDE tools are available from the Layout Editor shown in the figure below. The tools are called out in the figure and described briefly below. Subsequent sections show you how to use them.
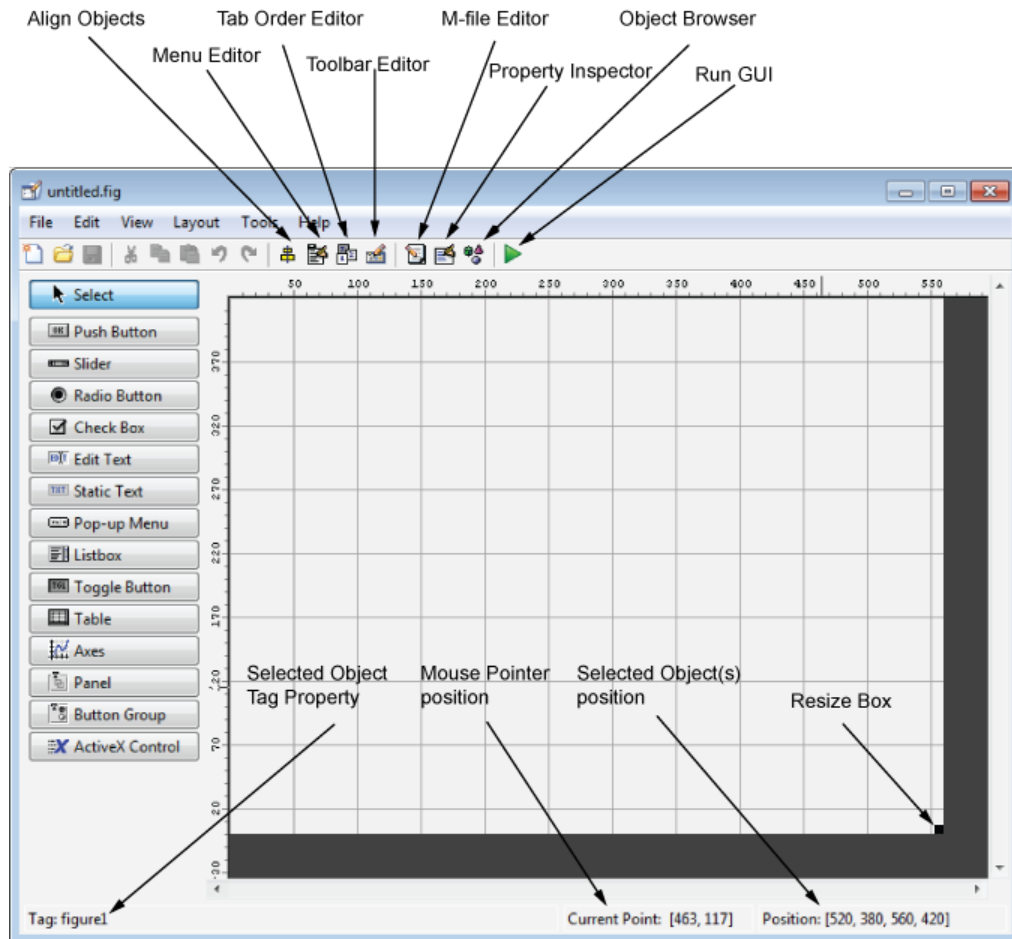
Use This Tool...	To...
Layout Editor	Select components from the component palette, at the left side of the Layout Editor, and arrange them in the layout area. See "Add

Use This Tool...	To...
	Components to the GUIDE Layout Area" on page 6-13 for more information.
Figure Resize Tab	Set the size at which the UI is initially displayed when you run it. See "Set the UI Window Size in GUIDE" on page 6-11 for more information.
Menu Editor	Create menus and context, i.e., pop-up, menus. See "Create Menus for GUIDE UIs" on page 6-77 for more information.
Align Objects	Align and distribute groups of components. Grids and rulers also enable you to align components on a grid with an optional snap-to-grid capability. See "Align GUIDE UI Components" on page 6-65 for more information.
Tab Order Editor	Set the tab and stacking order of the components in your layout. See "Customize Tabbing Behavior in a GUIDE UI" on page 6-74 for more information.
Toolbar Editor	Create Toolbars containing predefined and custom push buttons and toggle buttons. See "Create Toolbars for GUIDE UIs" on page 6-94 for more information.
Icon Editor	Create and modify icons for tools in a toolbar. See "Create Toolbars for GUIDE UIs" on page 6-94 for more information.
Property Inspector	Set the properties of the components in your layout. It provides a list of all the properties you can set and displays their current values.
Object Browser	Display a hierarchical list of the objects in the UI.
Run	Save the UI and run the program.
Editor	Display, in your default editor, the code file associated with the UI. See "Files Generated by GUIDE" on page 2-24 for more information.
Position Readouts	Continuously display the mouse cursor position and the positions of selected objects

Related Examples
- "Create a Simple App Using GUIDE" on page 2-2

More About
- "Ways to Build Apps" on page 1-2

GUIDE Preferences and Options

GUIDE Preferences

In this section...
"Set Preferences" on page 5-2
"Confirmation Preferences" on page 5-2
"Backward Compatibility Preference" on page 5-4
"All Other Preferences" on page 5-4

Set Preferences

You can set preferences for GUIDE. From the MATLAB **Home** tab, in the **Environment** section, click **Preferences**. These preferences apply to GUIDE and to all UIs you create.

The preferences are in different locations within the Preferences dialog box:

Confirmation Preferences

GUIDE provides two confirmation preferences. You can choose whether you want to display a confirmation dialog box when you

- Activate a UI from GUIDE.
- Export a UI from GUIDE.
- Change a callback signature generated by GUIDE.

In the Preferences dialog box, click **MATLAB** > **General** > **Confirmation Dialogs** to access the GUIDE confirmation preferences. Look for the word GUIDE in the **Tool** column.
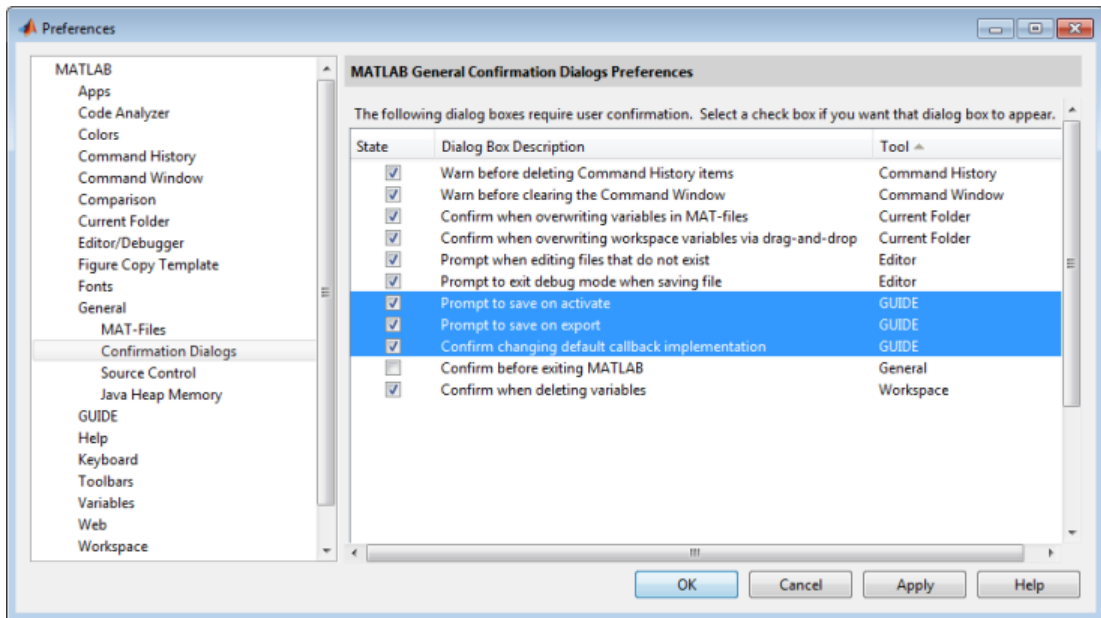
Prompt to Save on Activate

When you activate a UI from the Layout Editor by clicking the **Run** button ▶, a dialog box informs you of the impending save and lets you choose whether or not you want to continue.
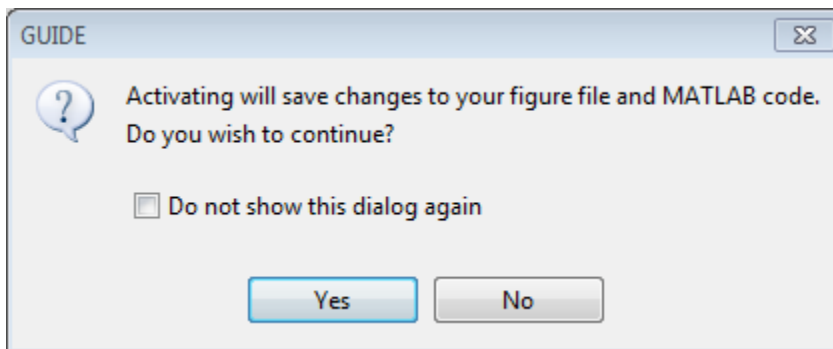
Prompt to Save on Export

From the Layout Editor, when you select **File** > **Export**, a dialog box informs you of the impending save and lets you choose whether or not you want to continue.
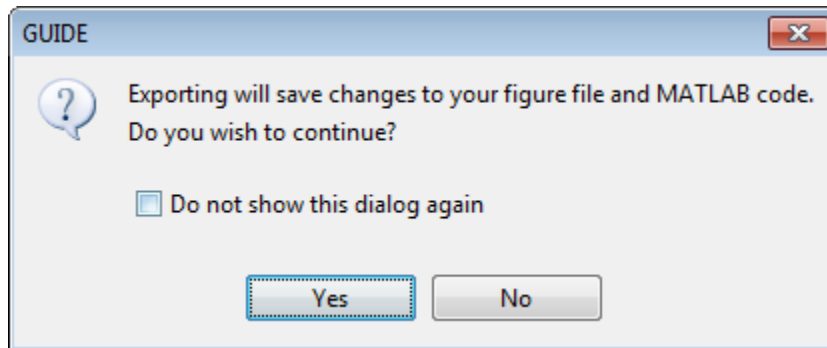
Backward Compatibility Preference

MATLAB Version 5 or Later Compatibility

UI FIG-files created or modified with MATLAB 7.0 or a later version are not automatically compatible with Version 6.5 and earlier versions. GUIDE automatically generates FIG-files, which are binary files that contain the UI layout information.

To make a FIG-file backward compatible, from the Layout Editor, select **File** > **Preferences** > **General** > **MAT-Files**, and then select **MATLAB Version 5 or later (save -v6)**.

Note: The **-v6** option discussed in this section is obsolete and will be removed in a future version of MATLAB.

All Other Preferences

GUIDE provides other preferences, for the Layout Editor interface and for inserting code comments. In the Preferences dialog box, click **GUIDE** to access these preferences.
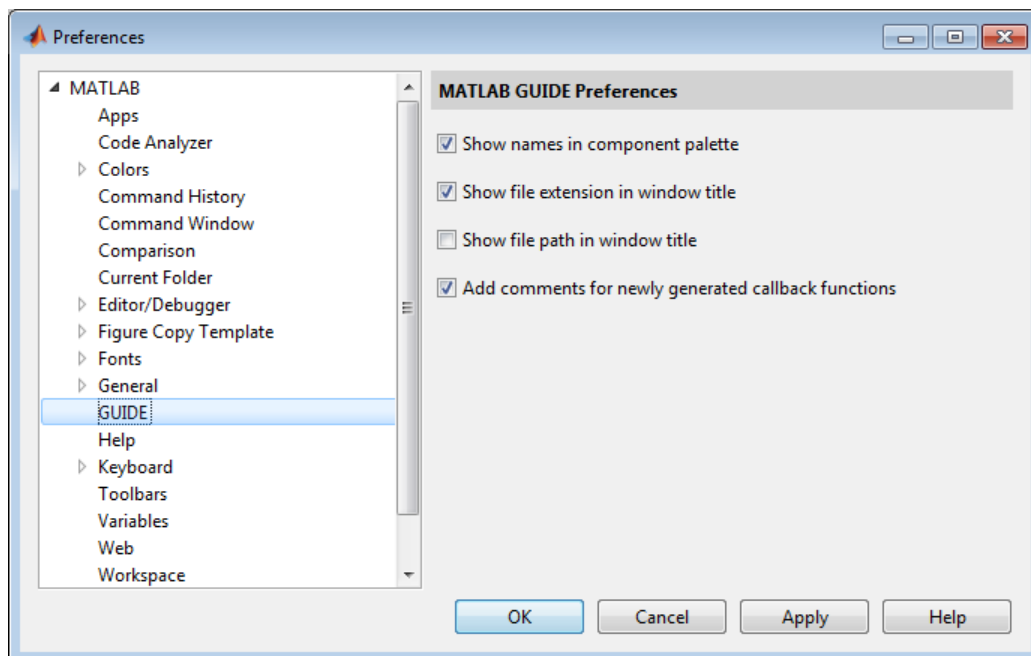
The following topics describe the preferences in this dialog:

- "Show Names in Component Palette" on page 5-5
- "Show File Extension in Window Title" on page 5-6
- "Show File Path in Window Title" on page 5-6
- "Add Comments for Newly Generated Callback Functions" on page 5-6

Show Names in Component Palette

Displays both icons and names in the component palette, as shown below. When unchecked, the icons alone are displayed in two columns, with tooltips.
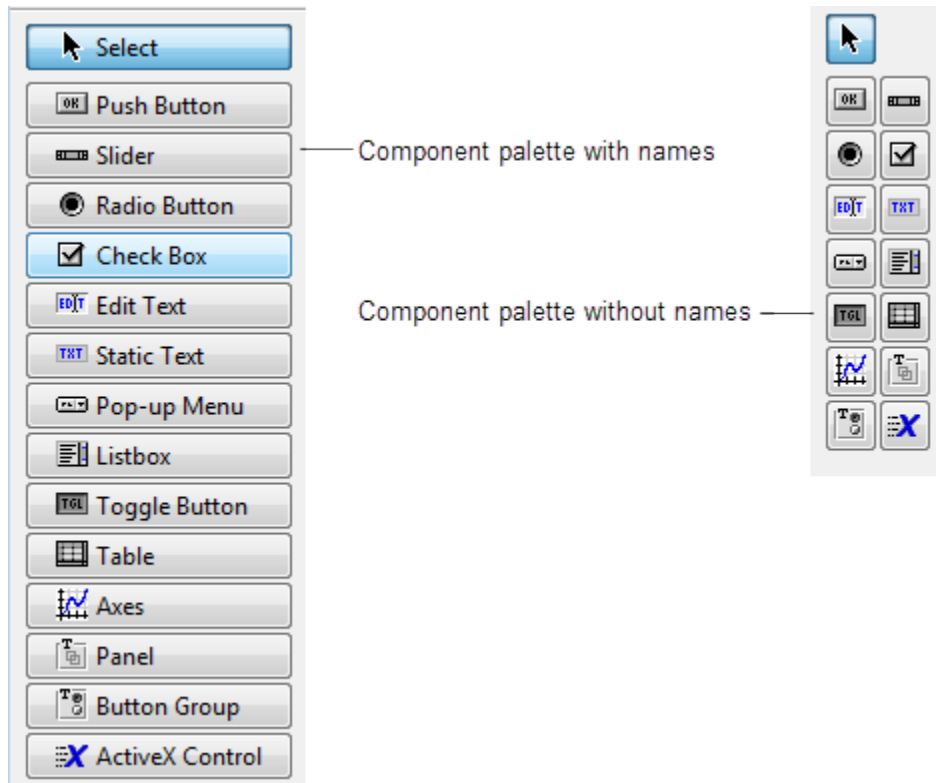
Show File Extension in Window Title

Displays the FIG-file file name with its file extension, `.fig`, in the Layout Editor window title. If unchecked, only the file name is displayed.

Show File Path in Window Title

Displays the full file path in the Layout Editor window title. If unchecked, the file path is not displayed.

Add Comments for Newly Generated Callback Functions

Callbacks are blocks of code that execute in response to actions by the user, such as clicking buttons or manipulating sliders. By default, GUIDE sets up templates that declare callbacks as functions and adds comments at the beginning of each one. Most of the comments are similar to the following.

```
% --- Executes during object deletion, before destroying properties.
function figure1_DeleteFcn(hObject, eventdata, handles)
% hObject    handle to figure1 (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)
```

Some callbacks are added automatically because their associated components are part of the original GUIDE template that you chose. Other commonly used callbacks are added automatically when you add components. You can also add callbacks explicitly by selecting them from **View** > **View Callbacks** menu or on the component's context menu.

If you deselect this preference, GUIDE includes comments only for callbacks that are automatically included to support the original GUIDE template. GUIDE does not include comments for callbacks subsequently added to the code.

See "Write Callbacks in GUIDE" on page 7-2 for more information about callbacks and about the arguments described in the preceding comments.

More About

- "GUIDE Options" on page 5-8

GUIDE Options

The GUI Options Dialog Box

Access the dialog box from the GUIDE Layout Editor by selecting **Tools** > **GUI Options**. The options you select take effect the next time you save your UI.
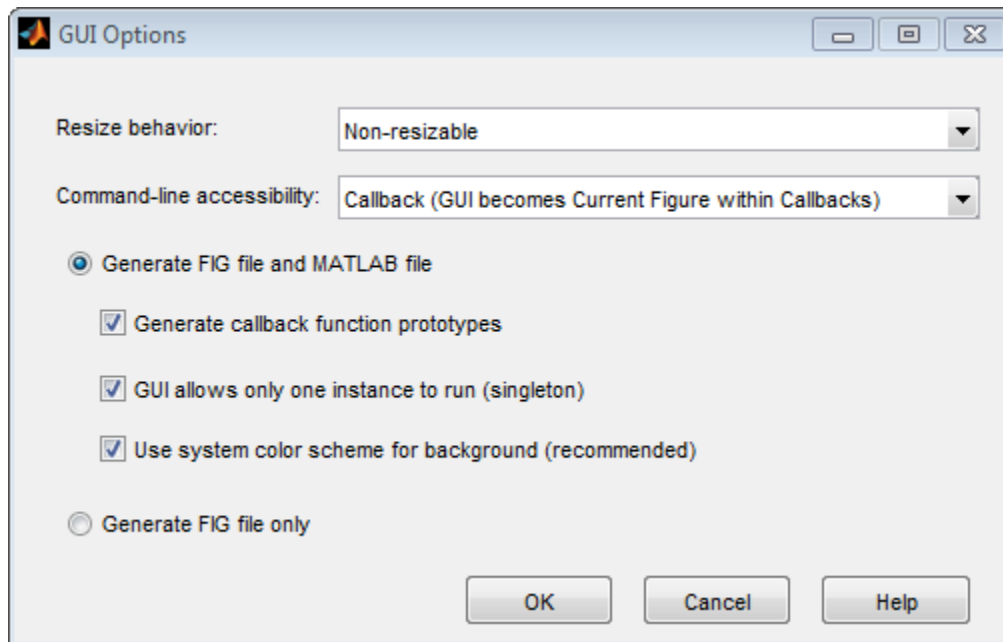
Resize Behavior

You can control whether users can resize the window and how MATLAB handles resizing. GUIDE provides three options:

- **Non-resizable** — Users cannot change the window size (default).
- **Proportional** — The software automatically scales the components in the UI in proportion to the new figure window size.
- **Other (Use SizeChangedFcn)** — Program the UI to behave in a certain way when users resize the figure window.

The first two options set figure and component properties appropriately and require no other action. **Other (Use SizeChangedFcn)** requires you to write a callback routine that recalculates sizes and positions of the components based on the new figure size.

Command-Line Accessibility

You can restrict access to a figure window from the command line or from a code file with the GUIDE **Command-line accessibility** options.

Unless you explicitly specify a figure handle, many commands, such as `plot`, alter the current figure (the figure specified by the root `CurrentFigure` property and returned by the `gcf` command). The current figure is usually the figure that is most recently created, drawn into, or mouse-clicked. You can programmatically designate a figure `h` (where `h` is its handle) as the current figure in four ways:

1 `set(groot,'CurrentFigure',h)` — Makes figure `h` current, but does not change its visibility or stacking with respect to other figures

2 `figure(h)` — Makes figure `h` current, visible, and displayed on top of other figures

3 `axes(h)` — Makes existing axes `h` the current axes and displays the figure containing it on top of other figures

4 `plot(h,...)`, or any plotting function that takes an axes as its first argument, also makes existing axes `h` the current axes and displays the figure containing it on top of other figures

The `gcf` function returns the handle of the current figure.

```
h = gcf
```

For a UI created in GUIDE, set the **Command-line accessibility** option to prevent users from inadvertently changing the appearance or content of a UI by executing commands at the command line or from a script or function, such as `plot`. The following table briefly describes the four options for **Command-line accessibility**.

Option	Description
Callback (GUI becomes Current Figure within Callbacks)	The UI can be accessed only from within a callback. The UI cannot be accessed from the command line or from a script. This is the default.
Off (GUI never becomes Current Figure)	The UI can not be accessed from a callback, the command line, or a script, without the handle.
On (GUI may become Current Figure from Command Line)	The UI can be accessed from a callback, from the command line, and from a script.
Other (Use settings from Property Inspector)	You control accessibility by setting the `HandleVisibility` and `IntegerHandle` properties from the Property Inspector.

Generate FIG-File and MATLAB File

Select **Generate FIG-file and MATLAB file** in the **GUI Options** dialog box if you want GUIDE to create both the FIG-file and the UI code file (this is the default). Once you have selected this option, you can select any of the following items in the frame to configure UI code:

- "Generate Callback Function Prototypes" on page 5-10
- "GUI Allows Only One Instance to Run (Singleton)" on page 5-11
- "Use System Color Scheme for Background" on page 5-11

See "Files Generated by GUIDE" on page 2-24 for information about these files.

Generate Callback Function Prototypes

If you select **Generate callback function prototypes** in the **GUI Options** dialog, GUIDE adds templates for the most commonly used callbacks to the code file for most components. You must then insert code into these templates.

GUIDE also adds a callback whenever you edit a callback routine from the Layout Editor's right-click context menu and when you add menus to the UI using the Menu Editor.

See "Write Callbacks in GUIDE" on page 7-2 for general information about callbacks.

Note This option is available only if you first select the **Generate FIG-file and MATLAB file** option.

GUI Allows Only One Instance to Run (Singleton)

This option allows you to select between two behaviors for the figure window:

- Allow MATLAB software to display only one instance of the UI at a time.
- Allow MATLAB software to display multiple instances of the UI.

If you allow only one instance, the software reuses the existing figure whenever the command to run your program is executed. If a UI window already exists, the software brings it to the foreground rather than creating a new figure.

If you clear this option, the software creates a new figure whenever you issue the command to run the program.

Even if you allow only one instance of a UI to exist, initialization can take place each time you invoke it from the command line. For example, the code in an `OpeningFcn` will run each time a GUIDE program runs unless you take steps to prevent it from doing so. Adding a flag to the `handles` structure is one way to control such behavior. You can do this in the `OpeningFcn`, which can run initialization code if this flag doesn't yet exist and skip that code if it does.

Note This option is available only if you first select the **Generate FIG-file and MATLAB file** option.

Use System Color Scheme for Background

The default color used for UI components is system dependent. This option enables you to make the figure background color the same as the default component background color.

To ensure that the figure background matches the color of the components, select **Use system color scheme for background** in the **GUI Options** dialog.

Note This option is available only if you first select the **Generate FIG-file and MATLAB file** option.

Generate FIG-File Only

The **Generate FIG-file only** option enables you to open figures and UIs to perform limited editing. These can be any figures and need not be UIs. UIs need not have been generated using GUIDE. This mode provides limited editing capability and may be useful for UIs generated in MATLAB Versions 5.3 and earlier. See the `guide` function for more information.

GUIDE selects **Generate FIG-file only** as the default if you do one of the following:

- Start GUIDE from the command line by providing one or more figure objects as arguments.

  ```
  guide(f)
  ```

  In this case, GUIDE selects **Generate FIG-file only**, even when a code file with a corresponding name exists in the same folder.

- Start GUIDE from the command line and provide the name of a FIG-file for which no code file with the same name exists in the same folder.

  ```
  guide('myfig.fig')
  ```

- Use the GUIDE **Open Existing GUI** tab to open a FIG-file for which no code file with the same name exists in the same folder.

When you save the figure or UI with **Generate FIG-file only** selected, GUIDE saves only the FIG-file. You must update any corresponding code files yourself, as appropriate.

If you want GUIDE to manage the UI code file for you, change the selection to **Generate FIG-file and MATLAB file** before saving the UI. If there is no corresponding code file in the same location, GUIDE creates one. If a code file with the same name as the original figure or UI exists in the same folder, GUIDE overwrites it. To prevent overwriting an existing file, save the UI using **Save As** from the **File** menu. Select another file name for the two files. GUIDE updates variable names in the new code file as appropriate.

Callbacks for UIs without Code

Even when there is no code file associated with a UI FIG-file, you can still provide callbacks for UI components to make them perform actions when used. In the Property Inspector, you can type callbacks in the form of strings, built-in functions, or MATLAB code file names; when your program runs, it will execute them if possible. If the callback is a file name, it can include arguments to that function. For example, setting the `Callback` property of a push button to `sqrt(2)` causes the result of the expression to display in the Command Window:

```
ans =
    1.4142
```
Any file that a callback executes must be in the current folder or on the MATLAB path. For more information on how callbacks work, see "Write Callbacks in GUIDE" on page 7-2

More About

· "GUIDE Preferences" on page 5-2

6

Lay Out a UI Using GUIDE

GUIDE Templates

In this section...
"Access the Templates" on page 6-2
"Template Descriptions" on page 6-3

Access the Templates

GUIDE provides several templates that you can modify to create your own UIs. The templates are fully functional programs.

You can access the templates in two ways:

- From the MATLAB toolstrip, on the **HOME** tab, in the **FILE** section, select**New > Graphical User Interface**
- If the Layout Editor is already open, select **File > New**.

In either case, GUIDE displays the **GUIDE Quick Start** dialog box with the **Create New GUI** tab selected as shown in the following figure. This tab contains a list of the available templates.
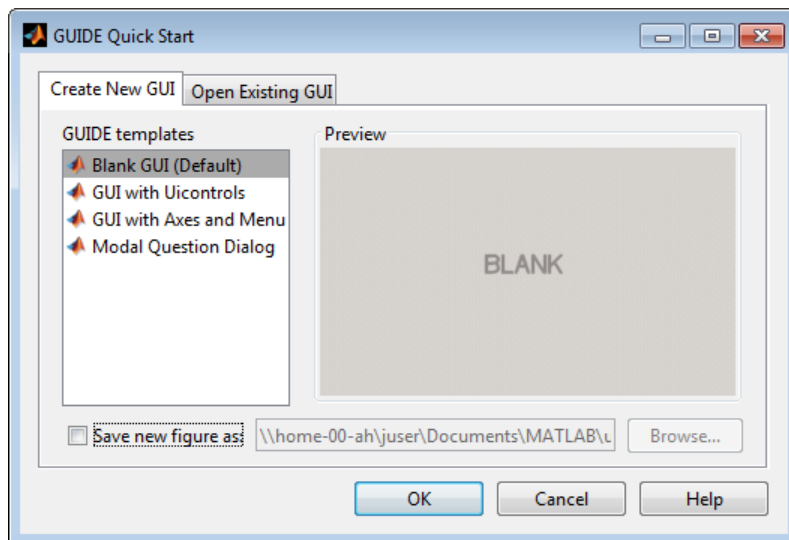
To use a template:

1   Select a template in the left pane. A preview displays in the right pane.

2   Optionally, name your UI now by selecting **Save new figure as** and typing the name in the field to the right. GUIDE saves the UI before opening it in the Layout Editor. If you choose not to name the UI at this point, GUIDE prompts you to save it and give it a name the first time you run your program.

3   Click **OK** to open the UI template in the Layout Editor.

Template Descriptions

GUIDE provides four fully functional templates. They are described in the following sections:

- "Blank GUI" on page 6-3
- "GUI with Uicontrols" on page 6-4
- "GUI with Axes and Menu" on page 6-6
- "Modal Question Dialog" on page 6-8

"Out of the box," none of the UI templates include a menu bar or a toolbar. Neither can they dock in the MATLAB desktop. You can, however, override these GUIDE defaults to provide and customize these controls. See the sections "Create Menus for GUIDE UIs" on page 6-77 and "Create Toolbars for GUIDE UIs" on page 6-94 for details.

Note To see how the templates work, you can view their code and look at their callbacks. You can also modify the callbacks for your own purposes. To view the code file for any of these templates, open the template in the Layout Editor and click the **Editor** button on the toolbar.

Blank GUI

The following figure shows an example of this template.
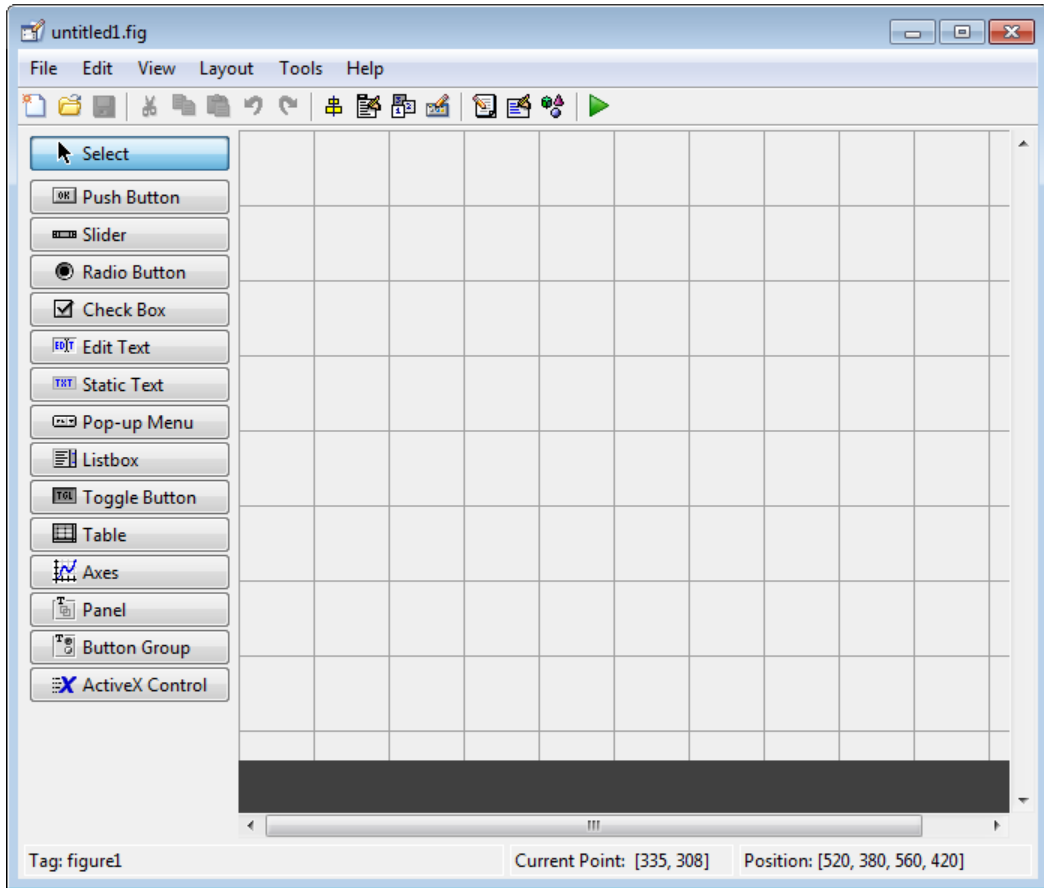
Select this template when the other templates are not suitable for the UI you want to create.

GUI with Uicontrols

The following figure shows an example of this template. The user interface controls shown in this template are the push buttons, radio buttons, edit text, and static text.
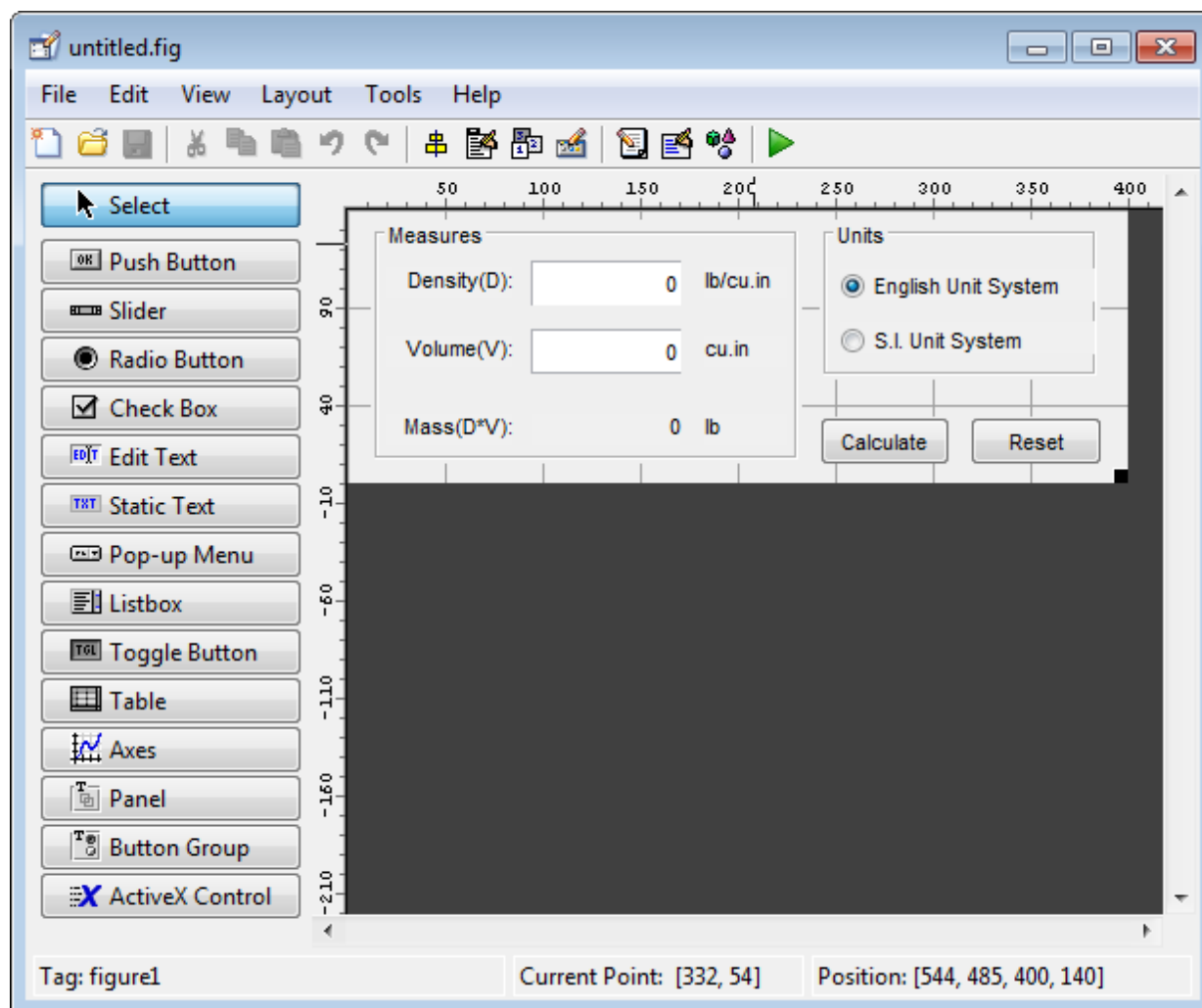
When you click the **Run** button ▶, the UI appears as shown in the following figure.
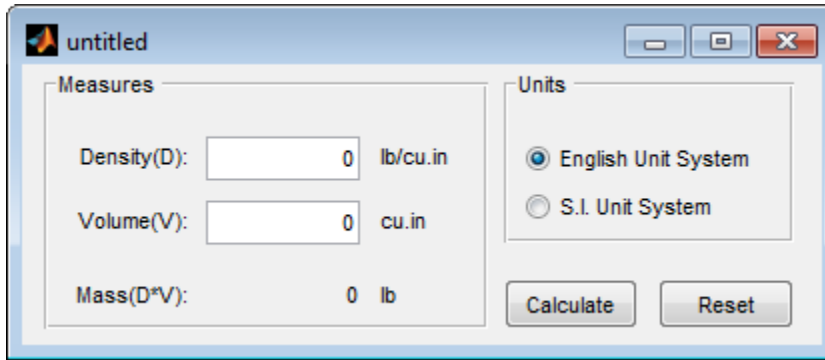
When you enter values for the density and volume of an object, and click the **Calculate** button, the program calculates the mass of the object and displays the result next to **Mass(D*V)**.

To view the code for these user interface controls, open the template in the Layout Editor and click the **Editor** button on the toolbar.

GUI with Axes and Menu

The following figure shows an example of this template.
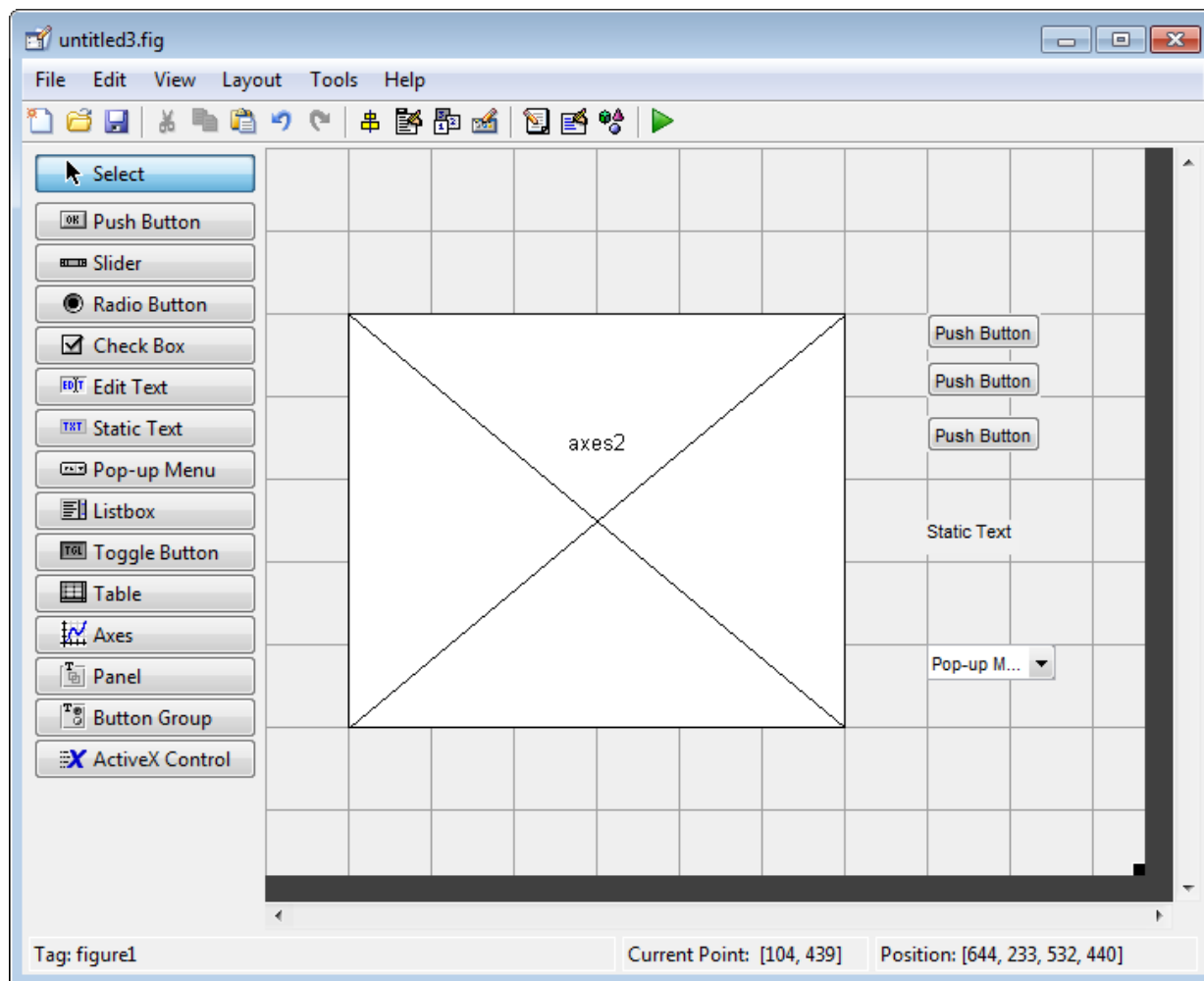
When you click the **Run** button ► on the toolbar, the UI displays a plot of five lines, each of which is generated from random numbers using the MATLAB `rand(5)` command. The following figure shows an example.
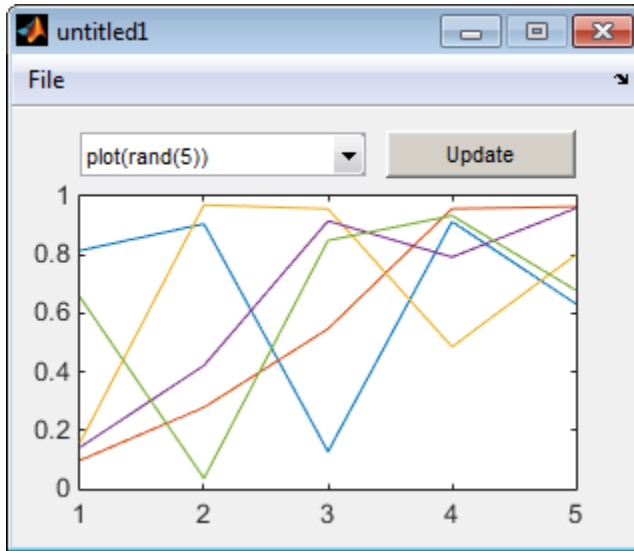
You can select other plots in the pop-up menu. Clicking the **Update** button displays the currently selected plot on the axes.

The UI also has a **File** menu with three items:

- **Open** displays a dialog box from which you can open files on your computer.
- **Print** opens the Print dialog box. Clicking **OK** in the Print dialog box prints the figure.
- **Close** closes the UI.

To view the code for these menu choices, open the template in the Layout Editor and click the **Editor** button on the toolbar.

Modal Question Dialog

The following figure shows an example of this template.
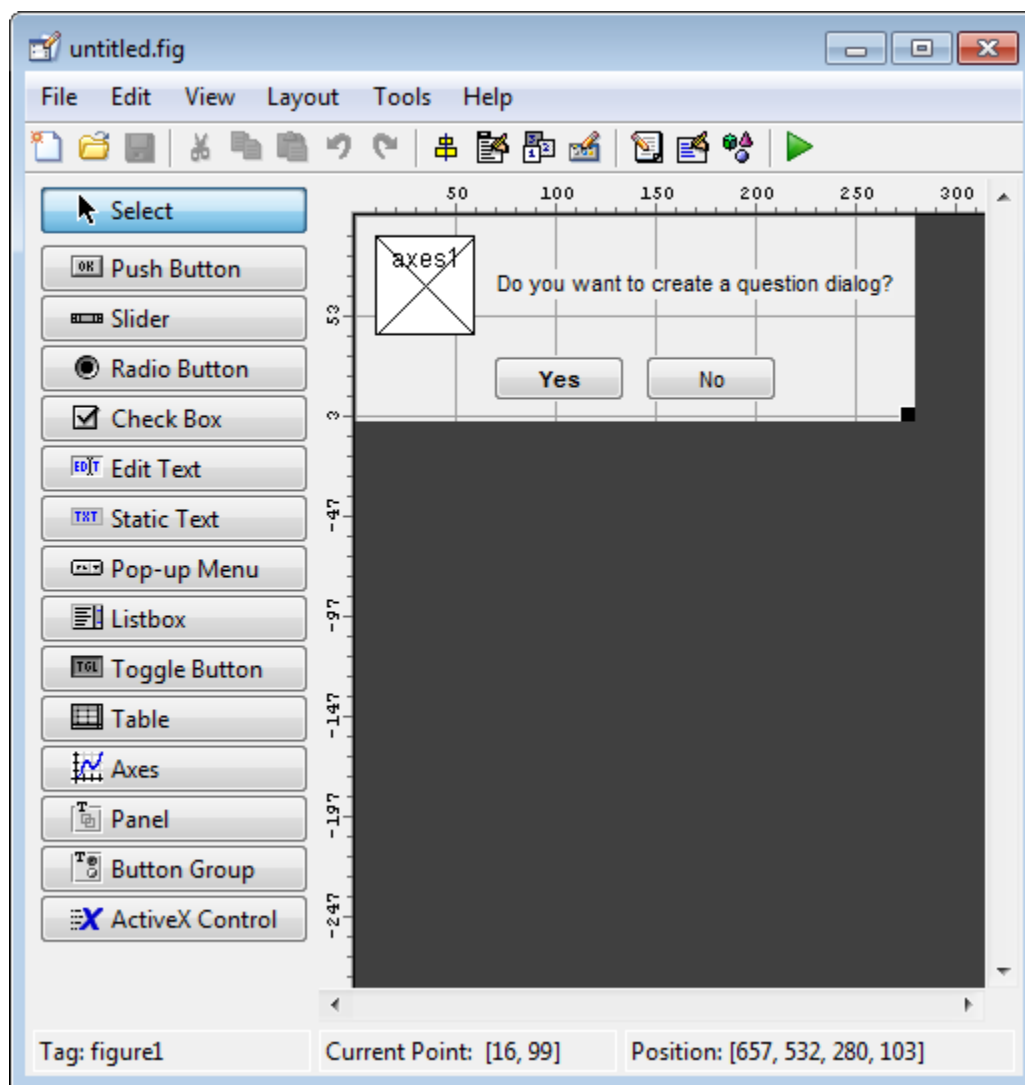
When you click the **Run** button, the following dialog displays.
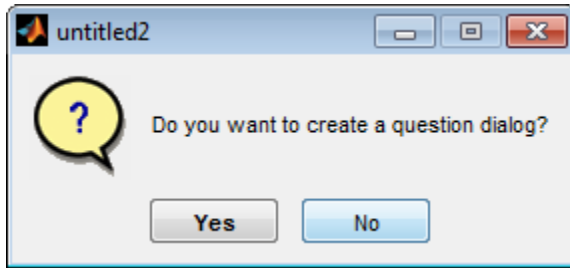
The dialog returns the text string, Yes or No, depending on which button you click.

Select this template if you want your UI to return a string or to be *modal*.

Modal dialogs are *blocking*, which means that the code stops executing while dialog exists. This means that the user cannot interact with other MATLAB windows until they click one of the dialog buttons.

To view the code for this dialog, open the template in the Layout Editor and click the **Editor** button 🖼 on the toolbar.

Related Examples

- "Create a Simple App Using GUIDE" on page 2-2
- "Add Components to the GUIDE Layout Area" on page 6-13

More About

- "Ways to Build Apps" on page 1-2

Set the UI Window Size in GUIDE

Set the size of the UI window by resizing the grid area in the Layout Editor. Click the lower-right corner of the layout area and drag it until the UI is the desired size. If necessary, make the window larger.
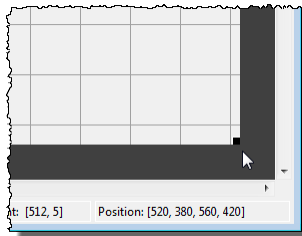
As you drag the corner handle, the readout in the lower right corner shows the current position of the UI in pixels.

Note Setting the `Units` property to `characters` (nonresizable UIs) or `normalized` (resizable UIs) gives the UI a more consistent appearance across platforms. See "Cross-Platform Compatible Units" on page 6-107 for more information.

Prevent Existing Objects from Resizing with the Window

Existing objects within the UI resize with the window if their `Units` are set to `'normalized'`. To prevent them from resizing with the window, perform these steps:

1   Set each object's `Units` property to an absolute value, such as inches or pixels before enlarging the UI.

    To change the `Units` property for all the objects in your UI simultaneously, drag a selection box around all the objects, and then click the Property Inspector button ![icon] and set the `Units`.

2   When you finish enlarging the UI, set each object's `Units` property back to `normalized`.

6-11

Set the Window Position or Size to an Exact Value

1 In the Layout Editor, open the Property Inspector for the figure by clicking the ![icon] button (with no components selected).

2 In the Property Inspector, scroll to the `Units` property and note whether the current setting is `characters` or `normalized`.

3 Click the down arrow at the far right in the `Units` row, and select `inches`.

4 In the Property Inspector, display the `Position` property elements by clicking the `+` sign to the left of `Position`.

5 Change the `x` and `y` coordinates to the point where you want the lower-left corner of the window to appear, and its width and height.

6 Reset the `Units` property to its previous setting, as noted in step 2.

Maximize the Layout Area

You can make maximum use of space within the Layout Editor by hiding the GUIDE toolbar and status bar, and showing only tool icons, as follows:

1 From the **View** menu, deselect **Show Toolbar**.

2 From the **View** menu, deselect **Show Status Bar**.

3 Select **File** > **Preferences**, and then clear **Show names in component palette**

Related Examples

- "Create a Simple App Using GUIDE" on page 2-2

More About

- "Ways to Build Apps" on page 1-2
- "GUIDE Options" on page 5-8

Add Components to the GUIDE Layout Area

In this section...
"Place Components" on page 6-13
"User Interface Controls" on page 6-19
"Panels and Button Groups" on page 6-40
"Axes" on page 6-45
"Table" on page 6-49
"ActiveX Component" on page 6-60
"Resize GUIDE UI Components" on page 6-62

Place Components

The component palette at the left side of the Layout Editor contains the components that you can add to your UI.

Note See "Create Menus for GUIDE UIs" on page 6-77 for information about adding menus to a UI. See "Create Toolbars for GUIDE UIs" on page 6-94 for information about working with the toolbar.

To place components in the GUIDE layout area and give each component a unique identifier, follow these steps:

1 Display component names on the palette.

   a On the MATLAB **Home** tab, in the **Environment** section, click **Preferences**.

   b In the Preferences dialog box, click **GUIDE**.

   c Select **Show Names in Component Palette**, and then click **OK** .

2 Place components in the layout area according to your design.

   • Drag a component from the palette and drop it in the layout area.

   • Click a component in the palette and move the cursor over the layout area. The cursor changes to a cross. Click again to add the component in its default size, or click and drag to size the component as you add it.

Once you have defined a UI component in the layout area, selecting it automatically shows it in the Property Inspector. If the Property Inspector is not open or is not visible, double-clicking a component raises the inspector and focuses it on that component.

The components listed in the following table have additional considerations; read more about them in the sections described there.

If You Are Adding...	Then...
Panels or button groups	See "Add a Component to a Panel or Button Group" on page 6-16.
Menus	See "Create Menus for GUIDE UIs" on page 6-77
Toolbars	See "Create Toolbars for GUIDE UIs" on page 6-94
ActiveX controls	See "ActiveX Component" on page 6-60.

See "Grid and Rulers" on page 6-71 for information about using the grid.

3 Assign a unique identifier to each component. Do this by setting the value of the component `Tag` properties. See"Assign an Identifier to Each Component" on page 6-19 for more information.

4 Specify the look and feel of each component by setting the appropriate properties. The following topics contain specific information.

- "User Interface Controls" on page 6-19
- "Panels and Button Groups" on page 6-40
- "Axes" on page 6-45
- "Table" on page 6-49
- "ActiveX Component" on page 6-60

This is an example of a UI in the Layout Editor. Components in the Layout Editor are not active.
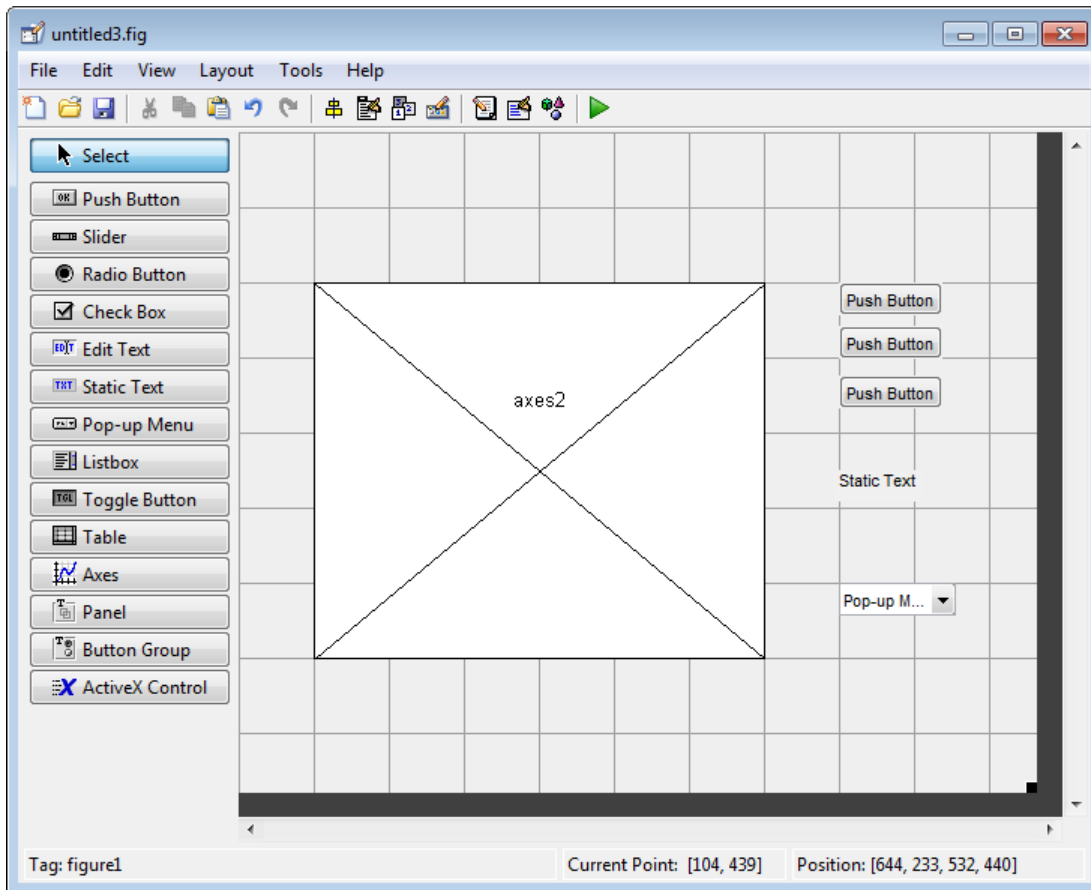
Use Coordinates to Place Components

The status bar at the bottom of the GUIDE Layout Editor displays:

- **Current Point** — The current location of the mouse relative to the lower left corner of the grid area in the Layout Editor.
- **Position** — The `Position` property of the selected component is a vector: [distance from left, distance from bottom, width, height], where distances are relative to the parent figure, panel, or button group.

Here is how to interpret the coordinates in the status bar and rulers:

- The **Position** values updates as you move and resize components. The first two elements in the vector change as you move the component. The last two elements of the vector change as the height and width of the component change.

- When no components are selected, the **Position** value displays the location and size of the figure.

Add a Component to a Panel or Button Group

To add a component to a panel or button group, select the component in the component palette then move the cursor over the desired panel or button group. The position of the cursor determines the component's parent.

GUIDE highlights the potential parent as shown in the following figure. The highlight indicates that if you drop the component or click the cursor, the component will be a child of the highlighted panel, button group, or figure.
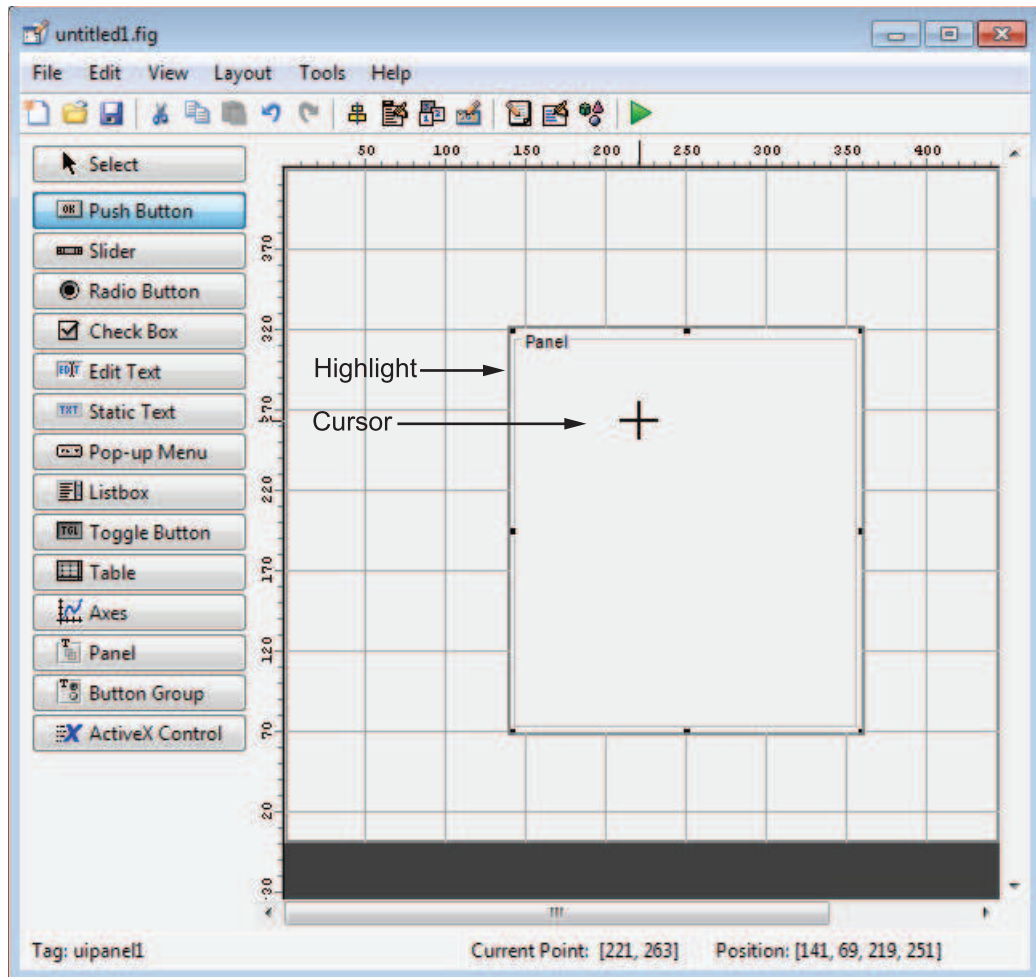
Note Assign a unique identifier to each component in your panel or button group by setting the value of its Tag property. See "Assign an Identifier to Each Component" on page 6-19 for more information.

Include Existing Components in Panels and Button Groups

When you add a new component or drag an existing component to a panel or button group, it will become a member, or child, of the panel or button group automatically, whether fully or partially enclosed by it. However, if the component is not entirely contained in the panel or button group, it appears to be clipped in the Layout Editor. When you run the program, the entire component is displayed and straddles the panel or button group border. The component is nevertheless a child of the panel and behaves accordingly. You can use the Object Browser to determine the child objects of a panel or button group.

You can add a new panel or button group to a UI in order to group any of its existing controls. In order to include such controls in a new panel or button group, do the following. The instructions refer to panels, but you do the same for components inside button groups.

1   Select the New Panel or New Button Group tool and drag out a rectangle to have the size and position you want.

    The panel will not obscure any controls within its boundary unless they are axes, tables, or other panels or button groups. Only overlap panels you want to nest, and then make sure the overlap is complete.

2   You can use **Send Backward** or **Send to Back** on the **Layout** menu to layer the new panel behind components you do not want it to obscure, if your layout has this problem. As you add components to it or drag components into it, the panel will automatically layer itself behind them.

    Now is a good time to set the panel's `Tag` and `String` properties to whatever you want them to be, using the Property Inspector.

3   Open the Object Browser from the **View** menu and find the panel you just added. Use this tool to verify that it contains all the controls you intend it to group together. If any are missing, perform the following steps.

4   Drag controls that you want to include but don't fit within the panel inside it to positions you want them to have. Also, slightly move controls that are already in their correct positions to group them with the panel.

    The panel highlights when you move a control, indicating it now contains the control. The Object Browser updates to confirm the relationship. If you now move the panel, its child controls move with it.

> **Tip** You need to move controls with the mouse to register them with the surrounding panel or button group, even if only by a pixel or two. Selecting them and using arrow keys to move them does not accomplish this. Use the Object Browser to verify that controls are properly nested.

See "Panels and Button Groups" on page 6-40 for more information on how to incorporate panels and button groups into a UI.

Assign an Identifier to Each Component

Use the `Tag` property to assign each component a unique meaningful string identifier.

When you place a component in the layout area, GUIDE assigns a default value to the `Tag` property. Before saving the UI, replace this value with a string that reflects the role of the component in the UI.

The string value you assign `Tag` is used by code to identify the component and must be unique in the UI. To set `Tag`:

1   Select **View** > **Property Inspector** or click the **Property Inspector** button.

2   In the layout area, select the component for which you want to set `Tag`.

3   In the Property Inspector, select `Tag` and then replace the value with the string you want to use as the identifier. In the following figure, `Tag` is set to `mybutton`.
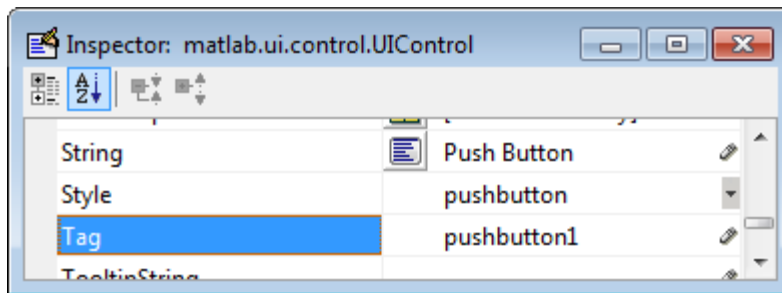
User Interface Controls

User interface controls include push buttons, toggle buttons, sliders, radio buttons, edit text controls, static text controls, pop-up menus, check boxes, and list boxes.

To define user interface controls, you must set certain properties. To do this:

1 Use the Property Inspector to modify the appropriate properties. Open the Property Inspector by selecting **View** > **Property Inspector** or by clicking the Property Inspector button .

2 In the layout area, select the component you are defining.

Subsequent topics describe commonly used properties of user interface controls and offer a simple example for each kind of control:

Commonly Used Properties

The most commonly used properties needed to describe a user interface control are shown in the following table. Instructions for a particular control may also list properties that are specific to that control.

Property	Value	Description
`Enable`	`on`, `inactive`, `off`. Default is `on`.	Determines whether the control is available to the user
`Max`	Scalar. Default is 1.	Maximum value. Interpretation depends on the type of component.
`Min`	Scalar. Default is 0.	Minimum value. Interpretation depends on the type of component.

Property	Value	Description
Position	4-element vector: [distance from left, distance from bottom, width, height].	Size of the component and its location relative to its parent.
String	String. Can also be a cell array or character array of strings.	Component label. For list boxes and pop-up menus it is a list of the items.
Units	characters, centimeters, inches, normalized, pixels, points. Default is characters.	Units of measurement used to interpret the Position property vector
Value	Scalar or vector	Value of the component. Interpretation depends on the type of component.

For a complete list of properties and for more information about the properties listed in the table, see Uicontrol Properties.

Push Button

To create a push button with label **Button 1**, as shown in this figure:
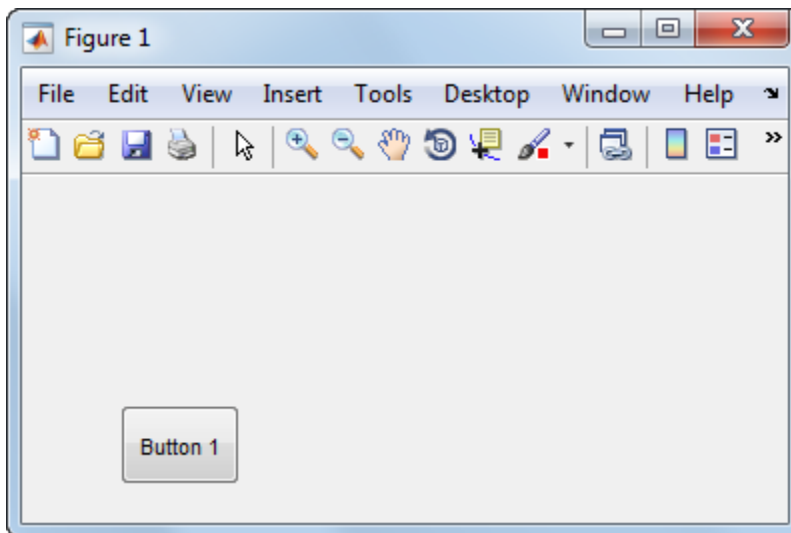
- Specify the push button label by setting the `String` property to the desired label, in this case, `Button 1`.
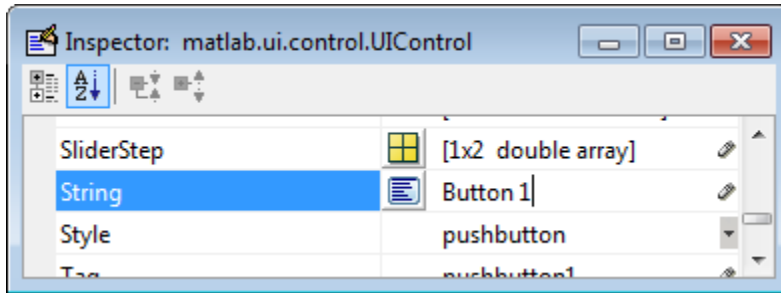
To display the & character in a label, use two & characters in the string. The words `remove`, `default`, and `factory` (case sensitive) are reserved. To use one of these as a label, prepend a backslash (\) to the string. For example, `\remove` yields **remove.**

The push button accommodates only a single line of text. If you specify more than one line, only the first line is shown. If you create a push button that is too narrow to accommodate the specified `String`, MATLAB software truncates the string with an ellipsis.

- If you want to set the position or size of the component to an exact value, then modify its `Position` property.
- To add an image to a push button, assign the button's CData property as an m-by-n-by-3 array of RGB values that defines a truecolor image. You must do this programmatically in the opening function of the UI code file. For example, the array `img` defines a 16-by-64-by-3 truecolor image using random values between 0 and 1 (generated by `rand`).

```
img = rand(16,64,3);
set(handles.pushbutton1,'CData',img);
```

where `pushbutton1` is the push button's `Tag` property.
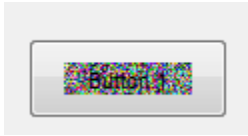
> **Note** See `ind2rgb` for information on converting a matrix `X` and corresponding colormap, i.e., an `(X, MAP)` image, to RGB (truecolor) format.

Slider

To create a slider as shown in this figure:
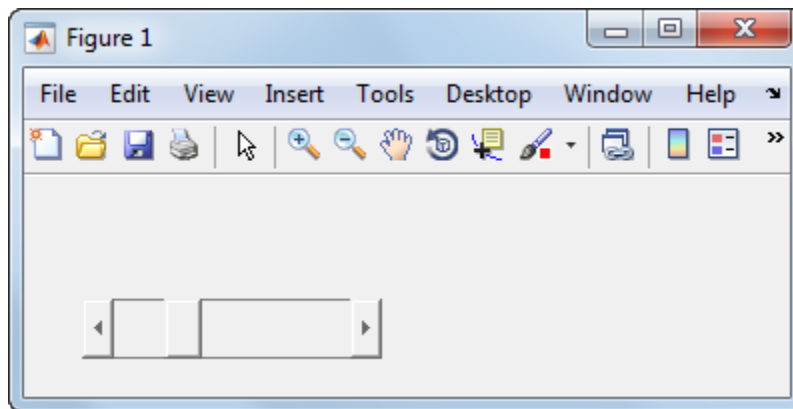
- Specify the range of the slider by setting its `Min` property to the minimum value of the slider and its `Max` property to the maximum value. The `Min` property must be less than `Max`.

- Specify the value indicated by the slider when it is created by setting the `Value` property to the appropriate number. This number must be less than or equal to `Max` and greater than or equal to `Min`. If you specify `Value` outside the specified range, the slider is not displayed.

- The slider `Value` changes by a small amount when a user clicks the arrow button, and changes by a larger amount when the user clicks the trough (also called the channel). Control how the slider responds to these actions by setting the

SliderStep property. Specify SliderStep as a two-element vector, [minor_step major_step], where minor_step is less than or equal to major_step. Because specifying very small values can cause unpredictable slider behavior, make both minor_step and major_step greater than 1e-6. Set major_step to the proportion of the range that clicking the trough moves the slider thumb. Setting it to 1 or higher causes the thumb to move to Max or Min when the trough is clicked.

As major_step increases, the thumb grows longer. When major_step is 1, the thumb is half as long as the trough. When major_step is greater than 1, the thumb continues to grow, slowly approaching the full length of the trough. When a slider serves as a scroll bar, you can uses this behavior to indicate how much of the document is currently visible by changing the value of major_step.
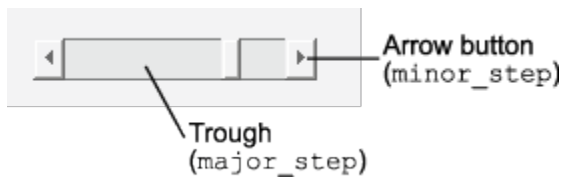
- If you want to set the location or size of the component to an exact value, then modify its Position property.

  The slider component provides no text description or data entry capability. Use a "Static Text" on page 6-30 component to label the slider. Use an "Edit Text" on page 6-28 component to enable a user to input a value to apply to the slider.

Note: On Mac platforms, the height of a horizontal slider is constrained. If the height you set in the position vector exceeds this constraint, the displayed height of the slider is the maximum allowed. The height element of the position vector is not changed.

Radio Button

To create a radio button with label **Indent nested functions**, as shown in this figure:
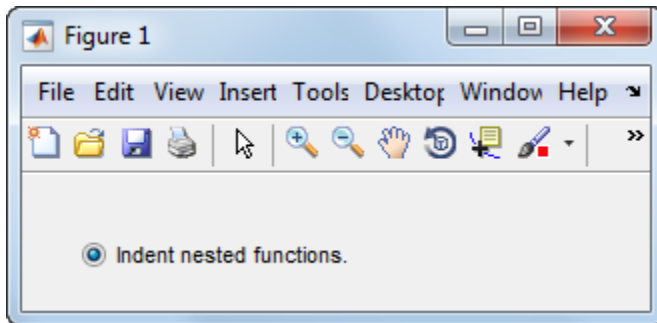
- Specify the radio button label by setting the `String` property to the desired label, in this case, `Indent nested functions`.
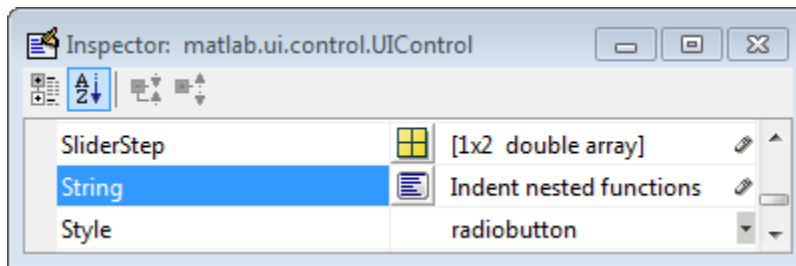
To display the & character in a label, use two & characters in the string. The words `remove`, `default`, and `factory` (case sensitive) are reserved. To use one of these as a label, prepend a backslash (\\) to the string. For example, `\remove` yields **remove.**

The radio button accommodates only a single line of text. If you specify more than one line, only the first line is shown. If you create a radio button that is too narrow to accommodate the specified `String`, MATLAB software truncates the string with an ellipsis.
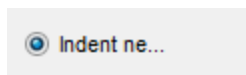
- Create the radio button with the button selected by setting its `Value` property to the value of its `Max` property (default is `1`). Set `Value` to `Min` (default is `0`) to leave the radio button unselected. Correspondingly, when the user selects the radio button, the software sets `Value` to `Max`, and to `Min` when the user deselects it.

6-25

- If you want to set the position or size of the component to an exact value, then modify its `Position` property.

- To add an image to a radio button, assign the button's CData property an m-by-n-by-3 array of RGB values that defines a truecolor image. You must do this programmatically in the opening function of the UI code file. For example, the array `img` defines a 16-by-24-by-3 truecolor image using random values between 0 and 1 (generated by `rand`).

```
img = rand(16,24,3);
set(handles.radiobutton1,'CData',img);
```

> **Note** To manage exclusive selection of radio buttons and toggle buttons, put them in a button group. See "Button Group" on page 6-43 for more information.

Check Box

To create a check box with label **Display file extension** that is initially checked, as shown in this figure:
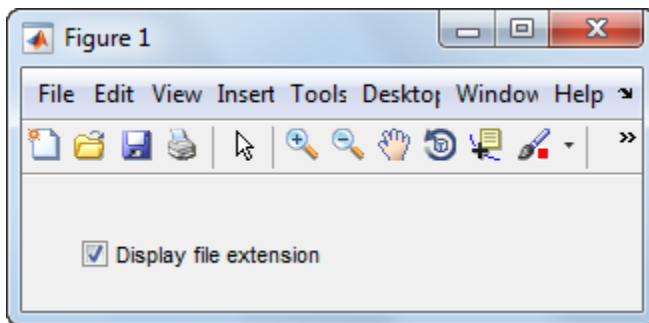
- Specify the check box label by setting the `String` property to the desired label, in this case, `Display file extension`.
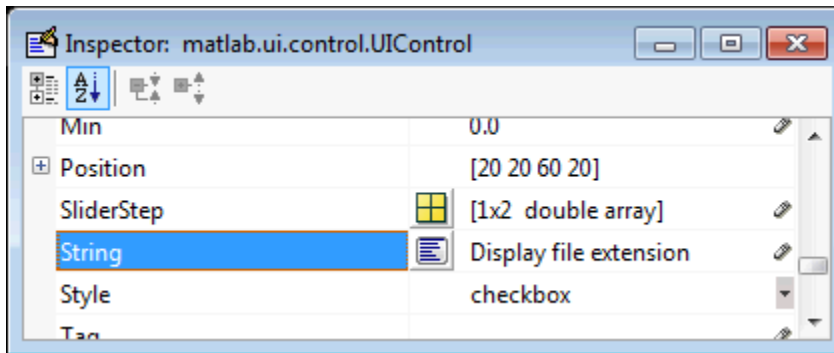
To display the & character in a label, use two & characters in the string. The words `remove`, `default`, and `factory` (case sensitive) are reserved. To use one of these as a label, prepend a backslash (\) to the string. For example, `\remove` yields **remove.**

The check box accommodates only a single line of text. If you specify a component width that is too small to accommodate the specified `String`, MATLAB software truncates the string with an ellipsis.
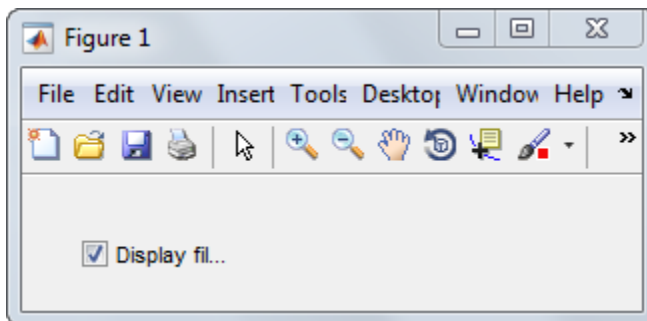
- Create the check box with the box checked by setting the `Value` property to the value of the `Max` property (default is `1`). Set `Value` to `Min` (default is `0`) to leave the box unchecked. Correspondingly, when the user clicks the check box, the software sets `Value` to `Max` when the user checks the box and to `Min` when the user clears it.
- If you want to set the position or size of the component to an exact value, then modify its `Position` property.

Edit Text

To create an edit text component that displays the initial text **Enter your name here**, as shown in this figure:
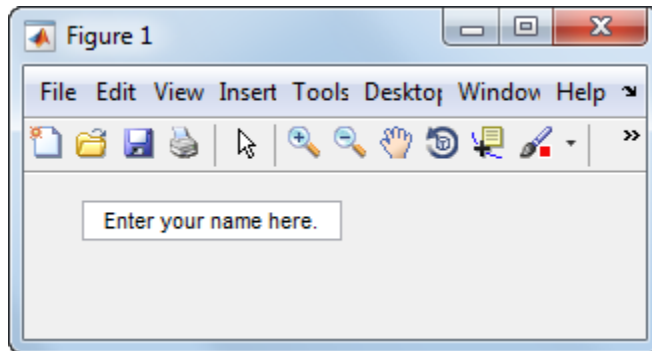
- Specify the text to be displayed when the edit text component is created by setting the `String` property to the desired string, in this case, `Enter your name here`.
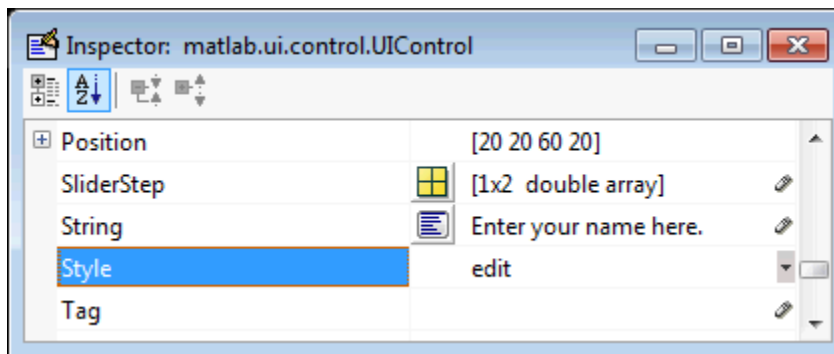
To display the & character in a label, use two & characters in the string. The words `remove`, `default`, and `factory` (case sensitive) are reserved. To use one of these as a label, prepend a backslash (\) to the string. For example, \remove yields **remove.**

- To enable multiple-line input, specify the `Max` and `Min` properties so that their difference is greater than 1. For example, `Max = 2`, `Min = 0`. `Max` default is `1`, `Min` default is `0`. MATLAB software wraps the string and adds a scroll bar if necessary. On all platforms, when the user enters a multiline text box via the **Tab** key, the editing cursor is placed at its previous location and no text highlights.
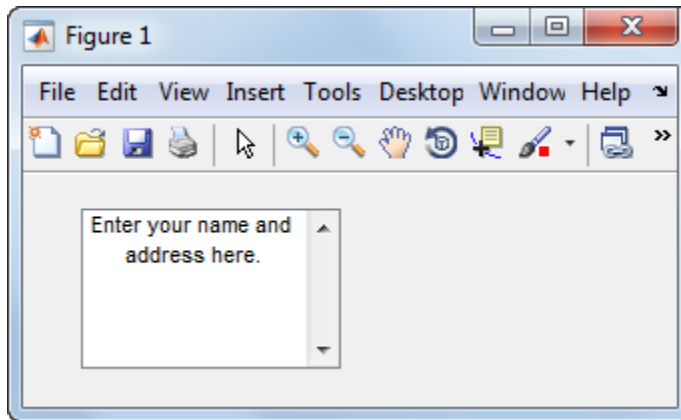
If `Max-Min` is less than or equal to 1, the edit text component admits only a single line of input. If you specify a component width that is too small to accommodate the specified string, MATLAB software displays only part of the string. The user can use the arrow keys to move the cursor through the entire string. On all platforms, when the user enters a single-line text box via the **Tab** key, the entire contents is highlighted and the editing cursor is at the end (right side) of the string.
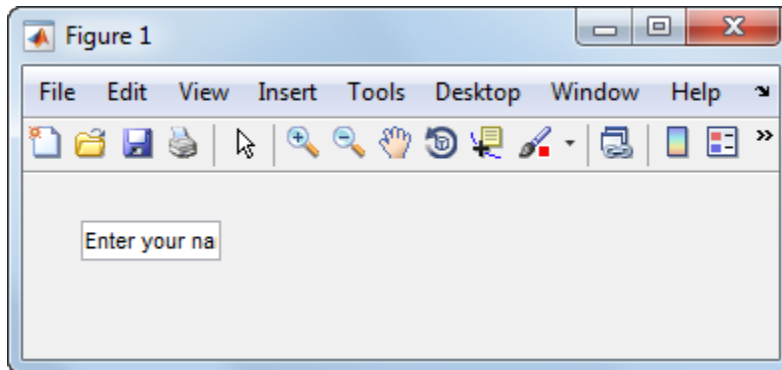
- If you want to set the position or size of the component to an exact value, then modify its `Position` property.

- You specify the text font to display in the edit box by typing the name of a font residing on your system into the `FontName` entry in the Property Inspector. On

Microsoft® Windows® platforms, the default is MS Sans Serif; on Macintosh and
UNIX® platforms, the default is Helvetica.

Tip To find out what fonts are available, type uisetfont at the MATLAB prompt; a
dialog displays containing a list box from which you can select and preview available
fonts. When you select a font, its name and other characteristics are returned in
a structure, from which you can copy the FontName string and paste it into the
Property Inspector. Not all fonts listed may be available to users of your UI on their
systems.

Static Text

To create a static text component with text **Select a data set**, as shown in this figure:
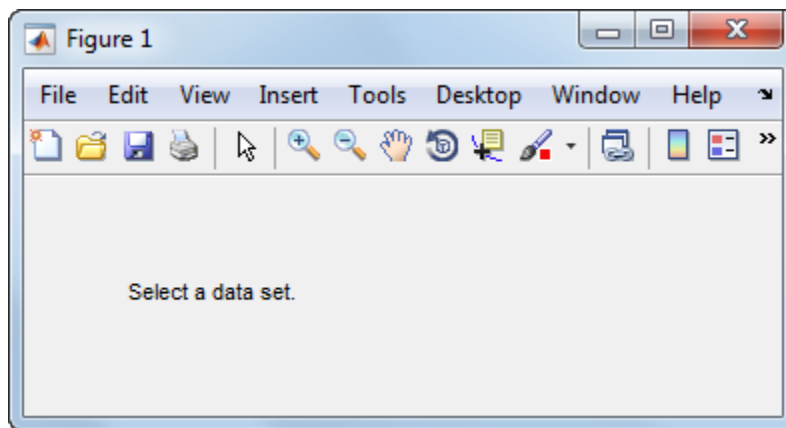
- Specify the text that appears in the component by setting the component String
  property to the desired text, in this case Select a data set.
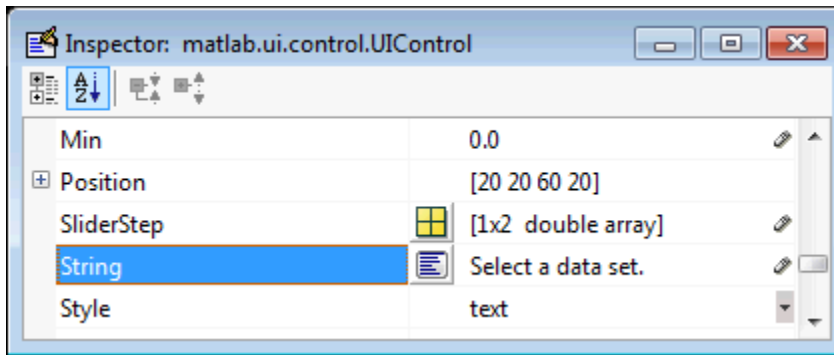
To display the & character in a list item, use two **&** characters in the string. The words `remove`, `default`, and `factory` (case sensitive) are reserved. To use one of these as a label, prepend a backslash (\) to the string. For example, `\remove` yields **remove.**

If your component is not wide enough to accommodate the specified `String`, MATLAB software wraps the string.
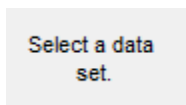
- If you want to set the position or size of the component to an exact value, then modify its `Position` property.
- You can specify a text font, including its `FontName`, `FontWeight`, `FontAngle`, `FontSize`, and `FontUnits` properties. For details, see the previous topic, "Edit Text" on page 6-28, and for a programmatic approach, the section "How to Set Font Characteristics" on page 9-25.

Pop-Up Menu

To create a pop-up menu (also known as a drop-down menu or combo box) with items **one**, **two**, **three**, and **four**, as shown in this figure:
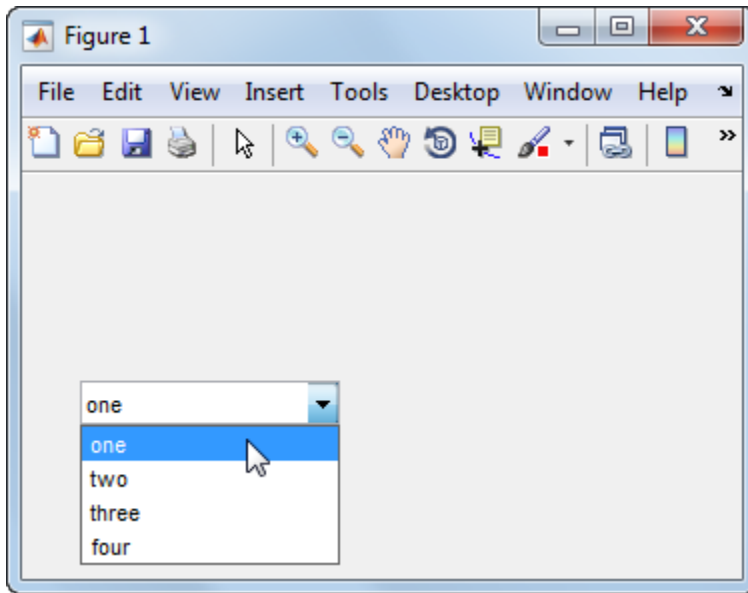
- Specify the pop-up menu items to be displayed by setting the `String` property to the desired items. Click the

  button to the right of the property name to open the Property Inspector editor.
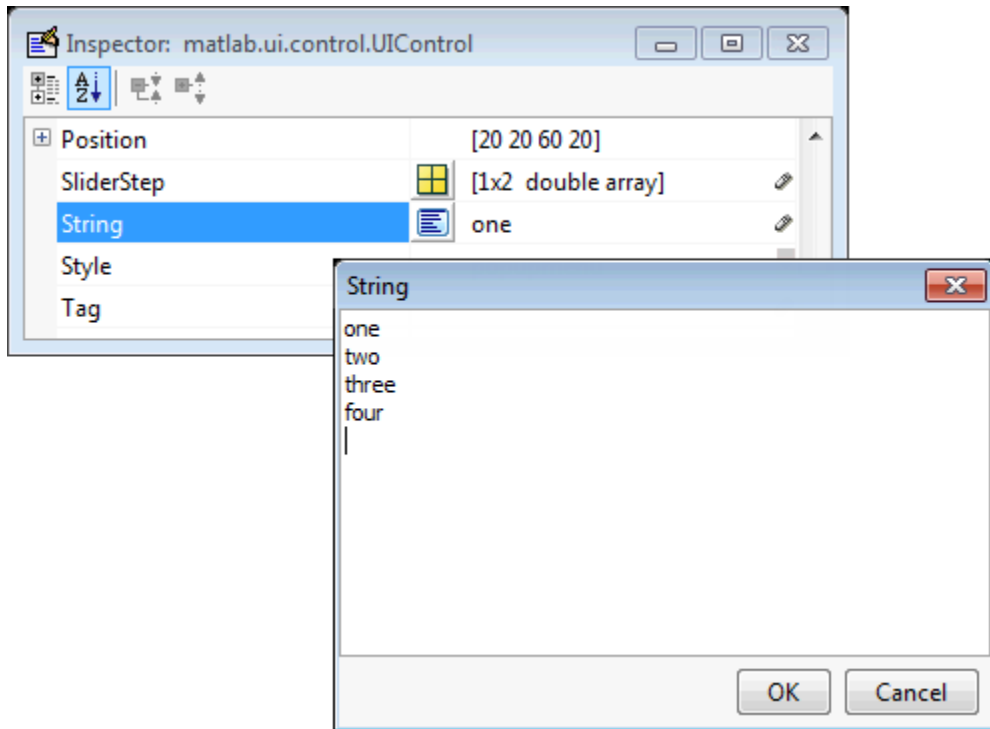
To display the & character in a menu item, use two & characters in the string. The words `remove`, `default`, and `factory` (case sensitive) are reserved. To use one of these as a label, prepend a backslash (\) to the string. For example, `\remove` yields **remove.**

If the width of the component is too small to accommodate one or more of the specified strings, MATLAB software truncates those strings with an ellipsis.

- To select an item when the component is created, set `Value` to a scalar that indicates the index of the selected list item, where 1 corresponds to the first item in the list. If you set `Value` to `2`, the menu looks like this when it is created:
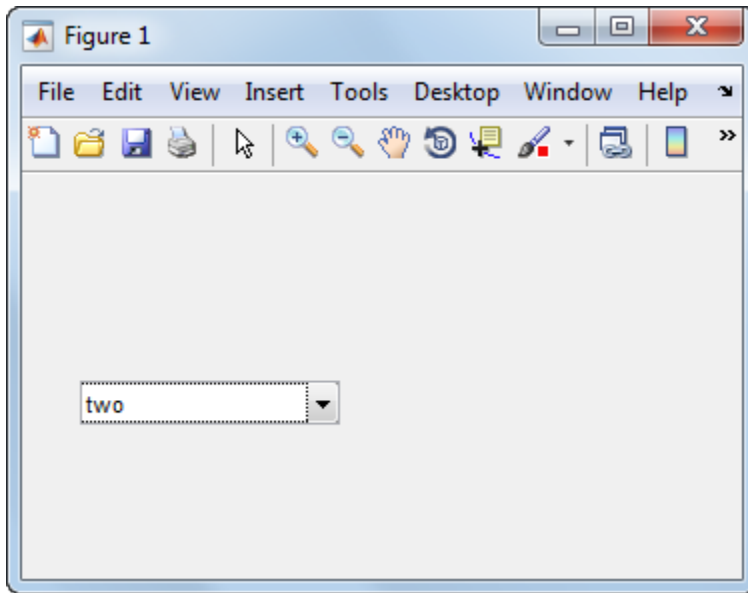
- If you want to set the position and size of the component to exact values, then modify its `Position` property. The height of a pop-up menu is determined by the font size. The height you set in the position vector is ignored.

Note The pop-up menu does not provide for a label. Use a "Static Text" on page 6-30 component to label the pop-up menu.

List Box

To create a list box with items **one**, **two**, **three**, and **four**, as shown in this figure:
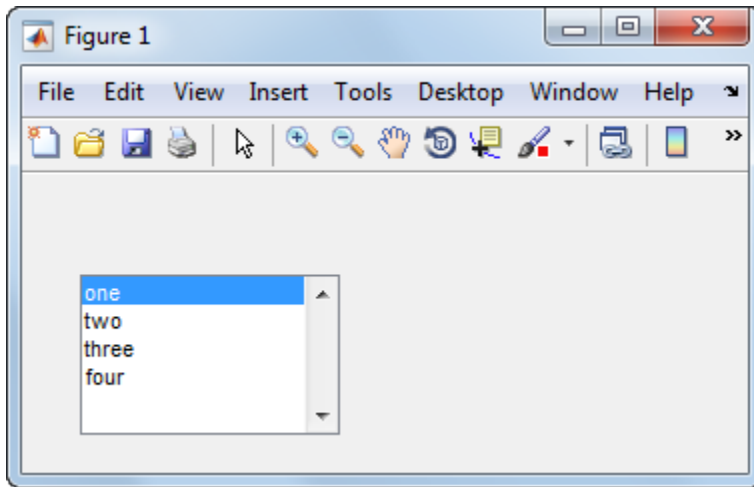
- Specify the list of items to be displayed by setting the `String` property to the desired list. Use the Property Inspector editor to enter the list. You can open the editor by

  clicking the button to the right of the property name.
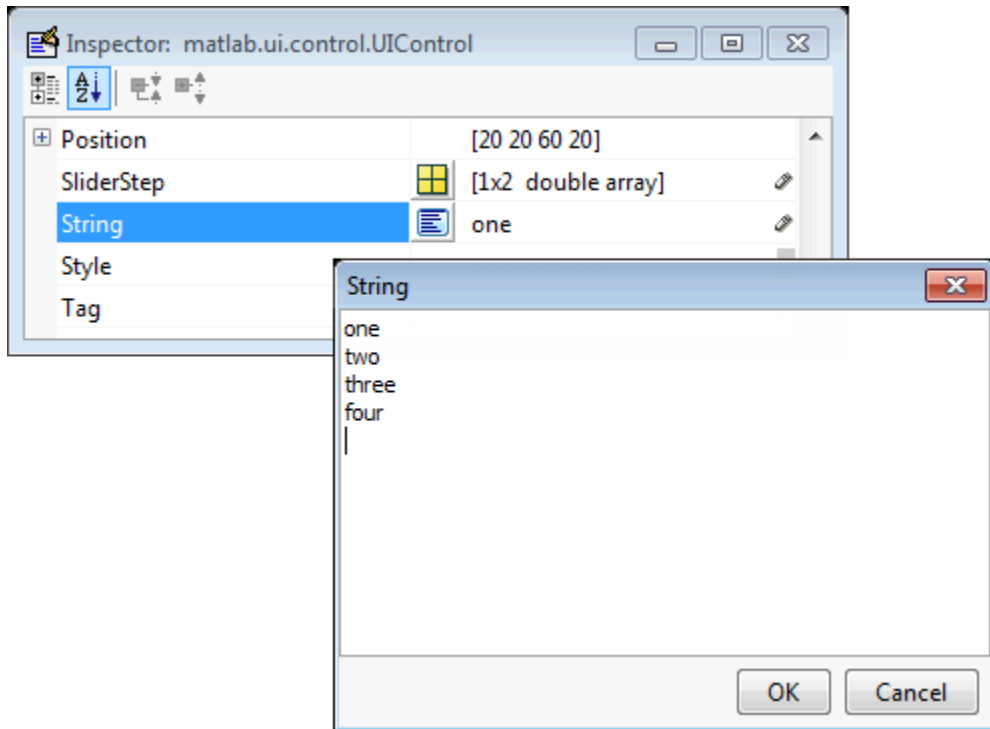
To display the & character in a label, use two & characters in the string. The words `remove`, `default`, and `factory` (case sensitive) are reserved. To use one of these as a label, prepend a backslash (\) to the string. For example, `\remove` yields **remove.**

If the width of the component is too small to accommodate one or more of the specified strings, MATLAB software truncates those strings with an ellipsis.

- Specify selection by using the `Value` property together with the `Max` and `Min` properties.

  - To select a single item when the component is created, set `Value` to a scalar that indicates the index of the selected list item, where 1 corresponds to the first item in the list.

  - To select more than one item when the component is created, set `Value` to a vector of indices of the selected items. `Value = [1,3]` results in the following selection.
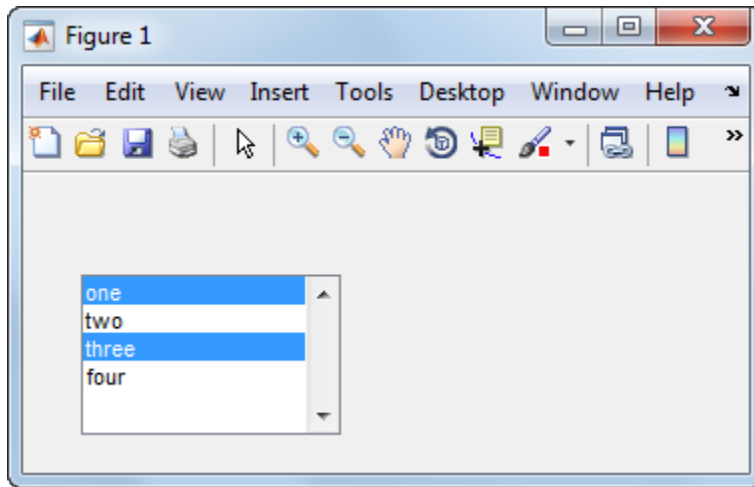
To enable selection of more than one item, you must specify the Max and Min properties so that their difference is greater than 1. For example, Max = 2, Min = 0. Max default is 1, Min default is 0.

- If you want no initial selection, set the Max and Min properties to enable multiple selection, i.e., Max - Min > 1, and then set the Value property to an empty matrix [].

- If the list box is not large enough to display all list entries, you can set the ListBoxTop property to the index of the item you want to appear at the top when the component is created.

- If you want to set the position or size of the component to an exact value, then modify its Position property.

Note The list box does not provide for a label. Use a "Static Text" on page 6-30 component to label the list box.

Toggle Button

To create a toggle button with label **Left/Right Tile**, as shown in this figure:
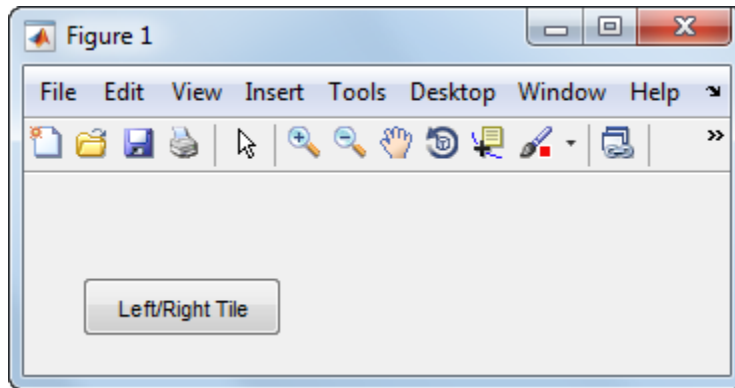
- Specify the toggle button label by setting its `String` property to the desired label, in this case, `Left/Right Tile`.
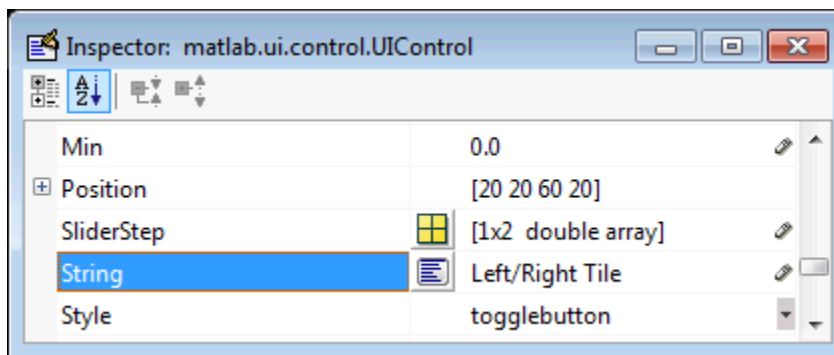
To display the & character in a label, use two & characters in the string. The words `remove`, `default`, and `factory` (case sensitive) are reserved. To use one of these as a label, prepend a backslash (\) to the string. For example, `\remove` yields **remove.**

The toggle button accommodates only a single line of text. If you specify more than one line, only the first line is shown. If you create a toggle button that is too narrow to accommodate the specified `String`, MATLAB software truncates the string with an ellipsis.
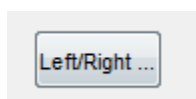
- Create the toggle button with the button selected (depressed) by setting its `Value` property to the value of its `Max` property (default is `1`). Set `Value` to `Min` (default is `0`) to leave the toggle button unselected (raised). Correspondingly, when the user selects the toggle button, MATLAB software sets `Value` to `Max`, and to `Min` when the user deselects it. The following figure shows the toggle button in the depressed position.
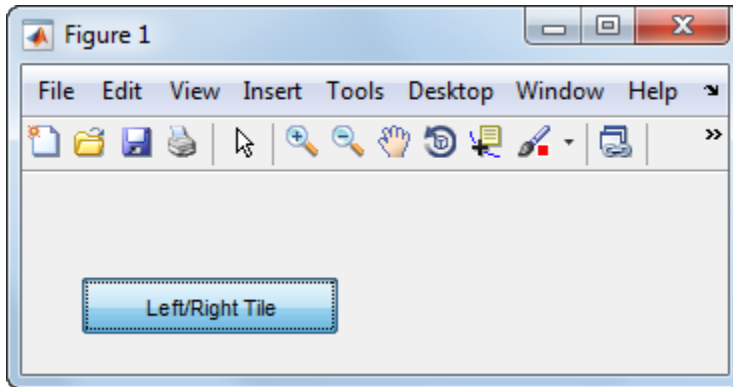
- If you want to set the position or size of the component to an exact value, then modify its `Position` property.

- To add an image to a toggle button, assign the button's CData property an m-by-n-by-3 array of RGB values that defines a truecolor image. You must do this programmatically in the opening function of the UI code file. For example, the array `img` defines a 16-by-64-by-3 truecolor image using random values between 0 and 1 (generated by `rand`).

```
img = rand(16,64,3);
set(handles.togglebutton1,'CData',img);
```

where `togglebutton1` is the toggle button's `Tag` property.
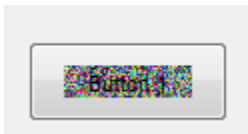
Note To manage exclusive selection of radio buttons and toggle buttons, put them in a button group. See "Button Group" on page 6-43 for more information.

6-39

Panels and Button Groups

Panels and button groups are containers that arrange UI components into groups. If you move the panel or button group, its children move with it and maintain their positions relative to the panel or button group.

To define panels and button groups, you must set certain properties. To do this:

1 Use the Property Inspector to modify the appropriate properties. Open the Property Inspector by selecting **View** > **Property Inspector** or by clicking the Property Inspector button .

2 In the layout area, select the component you are defining.

Subsequent topics describe commonly used properties of panels and button groups and offer a simple example for each component.

- "Commonly Used Properties" on page 6-40
- "Panel" on page 6-41
- "Button Group" on page 6-43

Commonly Used Properties

The most commonly used properties needed to describe a panel or button group are shown in the following table:

Property	Values	Description
`Position`	4-element vector: [distance from left, distance from bottom, width, height].	Size of the component and its location relative to its parent.
`Title`	String	Component label.
`TitlePosition`	`lefttop`, `centertop`, `righttop`, `leftbottom`, `centerbottom`, `rightbottom`. Default is `lefttop`.	Location of title string in relation to the panel or button group.
`Units`	`characters`, `centimeters`, `inches`, `normalized`, `pixels`,	Units of measurement used to interpret the `Position` property vector

Property	Values	Description
	`points`. Default is `characters`.	

For a complete list of properties and for more information about the properties listed in the table, see the Uipanel Properties and Uibuttongroup Properties.

Panel

To create a panel with title **My Panel** as shown in the following figure:
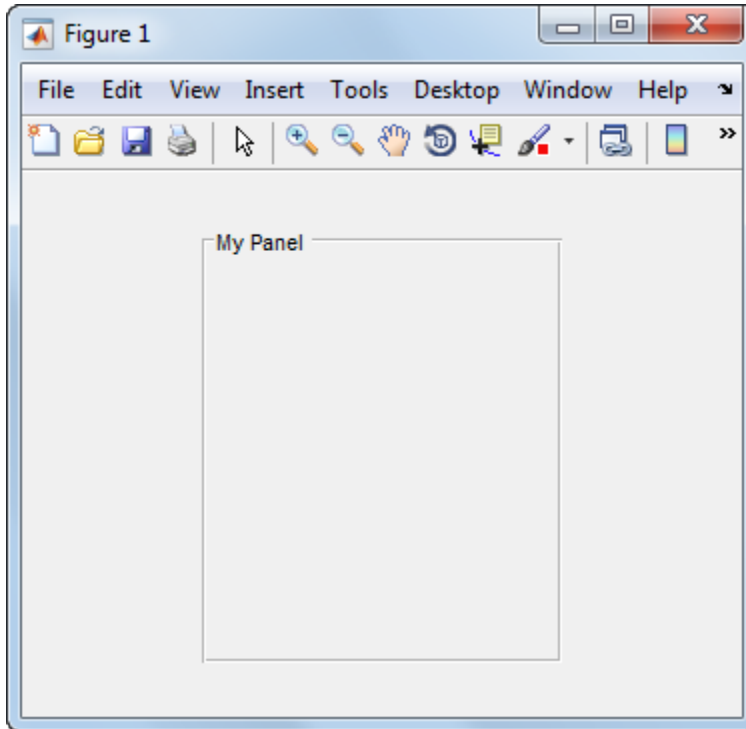
- Specify the panel title by setting the `Title` property to the desired string, in this case `My Panel`.
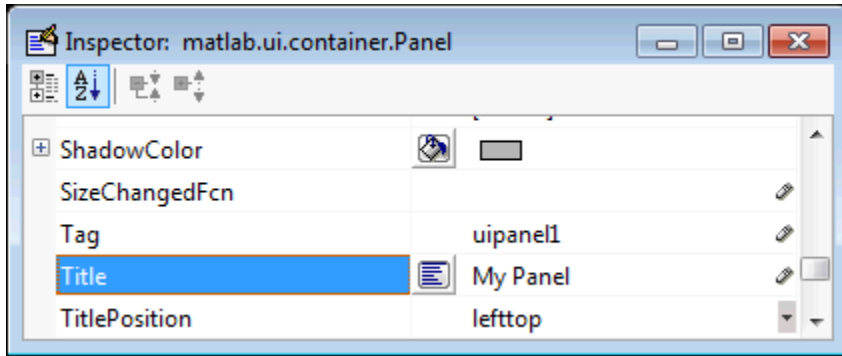
6-41

To display the & character in the title, use two & characters in the string. The words remove, default, and factory (case sensitive) are reserved. To use one of these as a label, prepend a backslash (\) to the string. For example, \remove yields **remove**.

- Specify the location of the panel title by selecting one of the available TitlePosition property values from the pop-up menu, in this case lefttop. You can position the title at the left, middle, or right of the top or bottom of the panel.
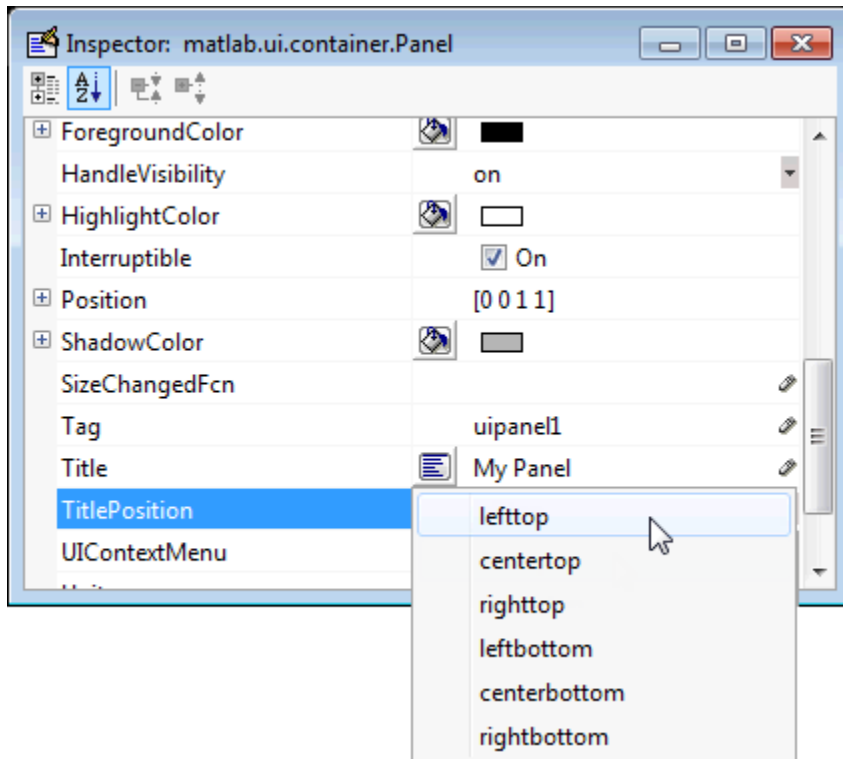
- If you want to set the position or size of the panel to an exact value, then modify its Position property.

Button Group

To create a button group with title **My Button Group** as shown in the following figure:
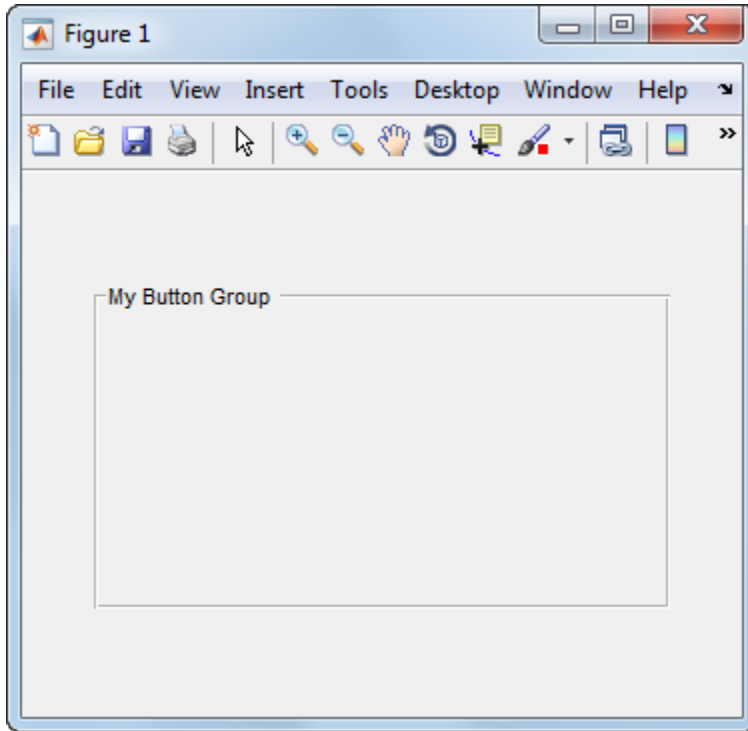
- Specify the button group title by setting the `Title` property to the desired string, in this case `My Button Group`.
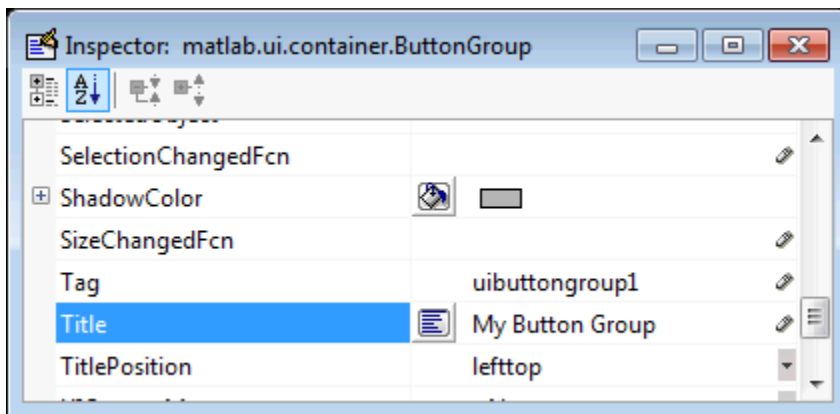
To display the & character in the title, use two & characters in the string. The words remove, default, and factory (case sensitive) are reserved. To use one of these as a label, prepend a backslash (\) to the string. For example, \remove yields **remove**.

- Specify the location of the button group title by selecting one of the available `TitlePosition` property values from the pop-up menu, in this case `lefttop`. You can position the title at the left, middle, or right of the top or bottom of the button group.
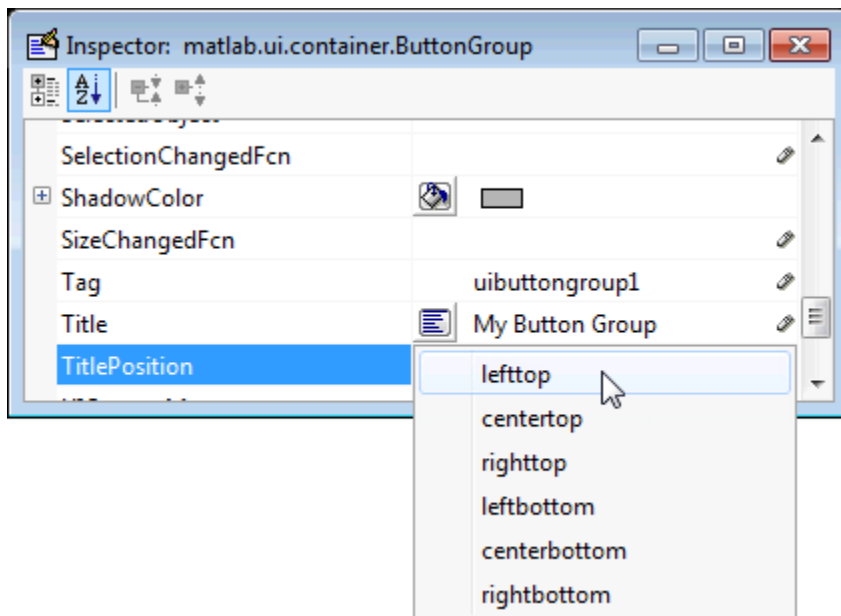
- If you want to set the position or size of the button group to an exact value, then modify its `Position` property.

Axes

Axes enable your UI to display graphics such as graphs and images using commands such as: `plot`, `surf`, `line`, `bar`, `polar`, `pie`, `contour`, and `mesh`.

To define an axes, you must set certain properties. To do this:

1 Use the Property Inspector to modify the appropriate properties. Open the Property Inspector by selecting **View** > **Property Inspector** or by clicking the Property Inspector button📝.

2 In the layout area, select the component you are defining.

Subsequent topics describe commonly used properties of axes and offer a simple example.

- "Commonly Used Properties" on page 6-46
- "Create Axes" on page 6-47

Commonly Used Properties

The most commonly used properties needed to describe an axes are shown in the following table:

Property	Values	Description
NextPlot	add, replace, replacechildren. Default is replace	Specifies whether plotting adds graphics, replaces graphics and resets axes properties to default, or replaces graphics only.
Position	4-element vector: [distance from left, distance from bottom, width, height].	Size of the component and its location relative to its parent.
Units	normalized, centimeters, characters, inches, pixels, points. Default is normalized.	Units of measurement used to interpret position vector

For a complete list of properties and for more information about the properties listed in the table, see Axes Properties.

See commands such as the following for more information on axes objects: `plot`, `surf`, `line`, `bar`, `polar`, `pie`, `contour`, `imagesc`, and `mesh`.

Many of these graphing functions reset axes properties by default, according to the setting of its `NextPlot` property, which can cause unwanted behavior in a UI, such as resetting axis limits and removing axes context menus and callbacks. See "Create Axes"

on page 6-47 and "Axes" on page 9-23 for information about setting the `NextPlot` property.

Create Axes

Here is an axes in a GUIDE UI:
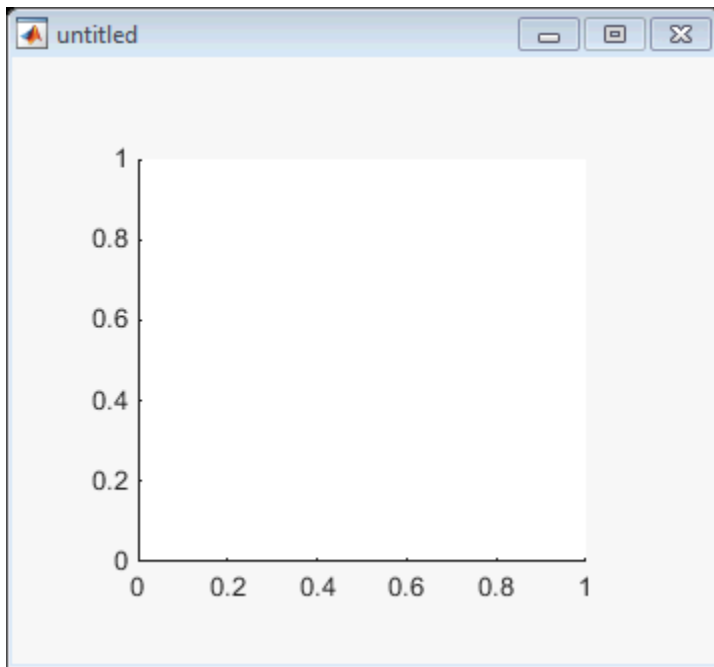
Use these guidelines when you create axes objects in GUIDE:

- Allow for tick marks to be placed outside the box that appears in the Layout Editor. The axes above looks like this in the layout editor; placement allows space at the left and bottom of the axes for tick marks. Functions that draw in the axes update the tick marks appropriately.
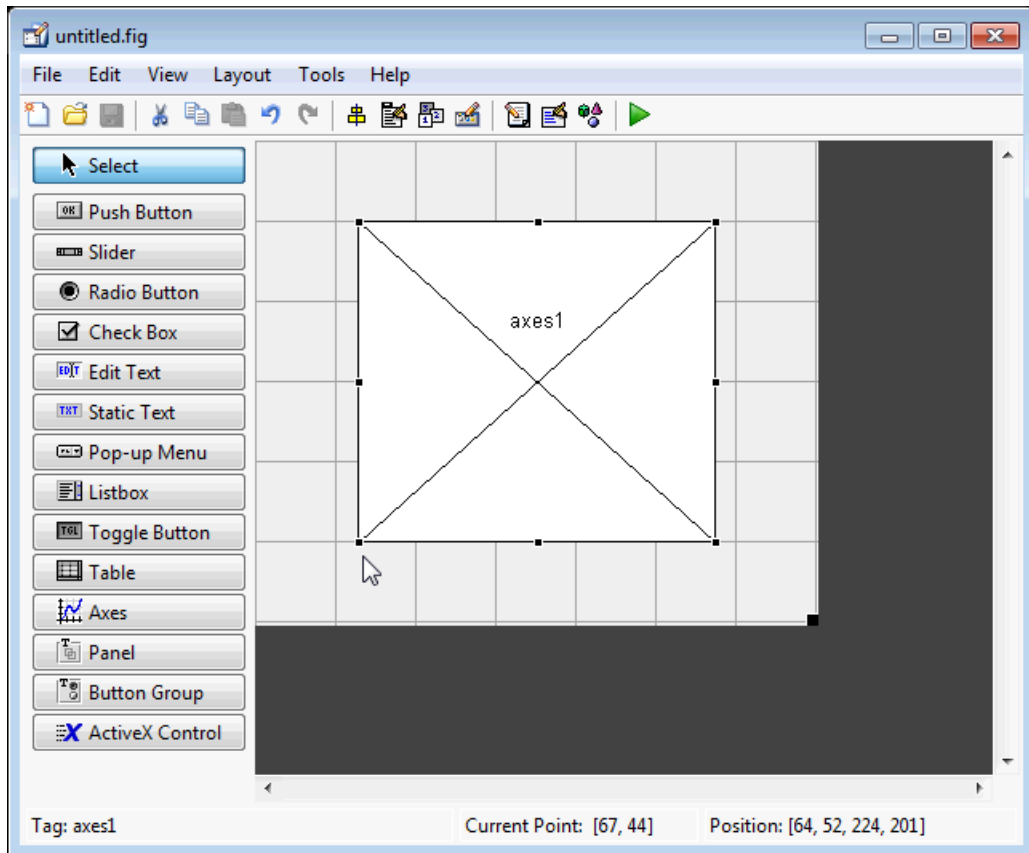
- Use the `title`, `xlabel`, `ylabel`, `zlabel`, and `text` functions in the UI code file to label an axes component. For example,

```
xlh = (axes_handle,'Years')
```

labels the X-axis as `Years`. The handle of the X-axis label is `xlh`.

The words `remove`, `default`, and `factory` (case sensitive) are reserved. To use one of these in component text, prepend a backslash (\) to the string. For example, `\remove` yields **remove.**

- If you want to set the position or size of the axes to an exact value, then modify its `Position` property.

- If you customize axes properties, some of them (or example, callbacks, font characteristics, and axis limits and ticks) may get reset to default every time you draw a graph into the axes when the `NextPlot` property has its default value of `'replace'`. To keep customized properties as you want them, set `NextPlot` to `'replacechildren'` in the Property Inspector, as shown here.
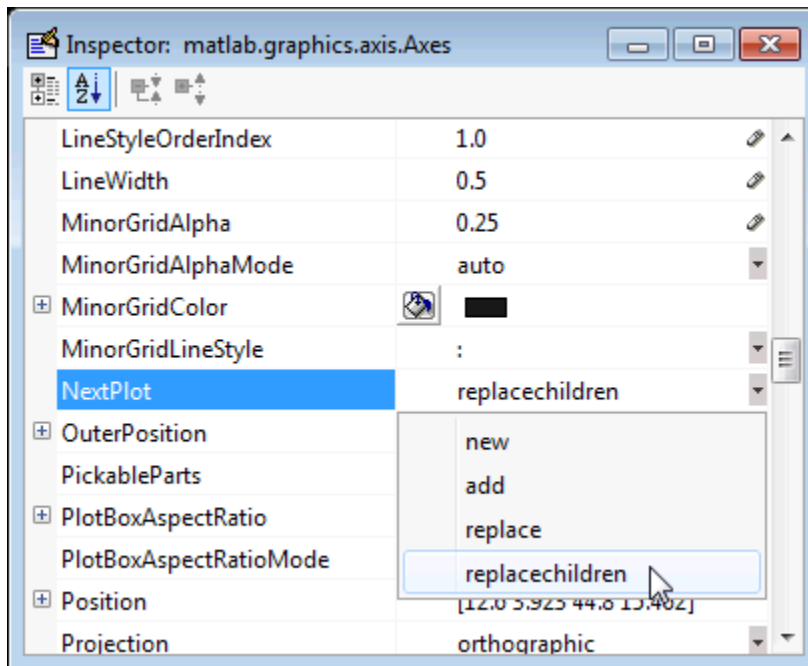
Table

Tables enable you to display data in a two dimensional table. You can use the Property Inspector to get and set the object property values.
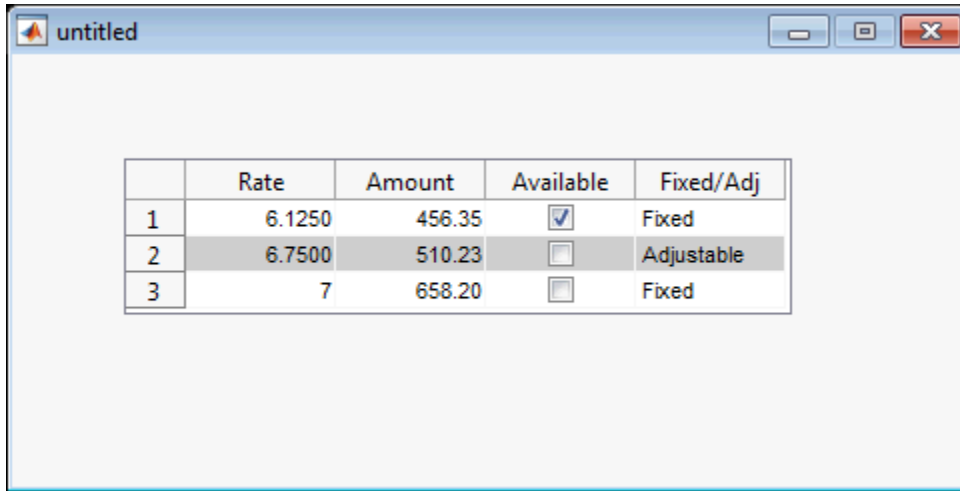
Commonly Used Properties

The most commonly used properties of a table component are listed in the table below. These are grouped in the order they appear in the Table Property Editor. Please refer to `uitable` documentation for detail of all the table properties:

Group	Property	Values	Description
Column	ColumnName	1-by-*n* cell array of strings | {'numbered'} | empty matrix ([])	The header label of the column.
	ColumnFormat	Cell array of strings	Determines display and editability of columns
	ColumnWidth	1-by-*n* cell array or 'auto'	Width of each column in pixels; individual column widths can also be set to 'auto'
	ColumnEditable	logical 1-by-*n* matrix | scalar logical value | empty matrix ([])	Determines data in a column as editable
Row	RowName	1-by-*n* cell array of strings	Row header label names
Color	BackgroundColor	*n*-by-3 matrix of RGB triples	Background color of cells
	RowStriping	{on} | off	Color striping of table rows
Data	Data	Matrix or cell array of numeric, logical, or character data	Table data.

Create a Table

To create a UI with a table in GUIDE as shown, do the following:

Drag the table icon on to the Layout Editor and right click in the table. From the table's context menu, select **Table Property Editor**. You can also select **Table Property Editor** from the **Tools** menu when you select a table by itself.
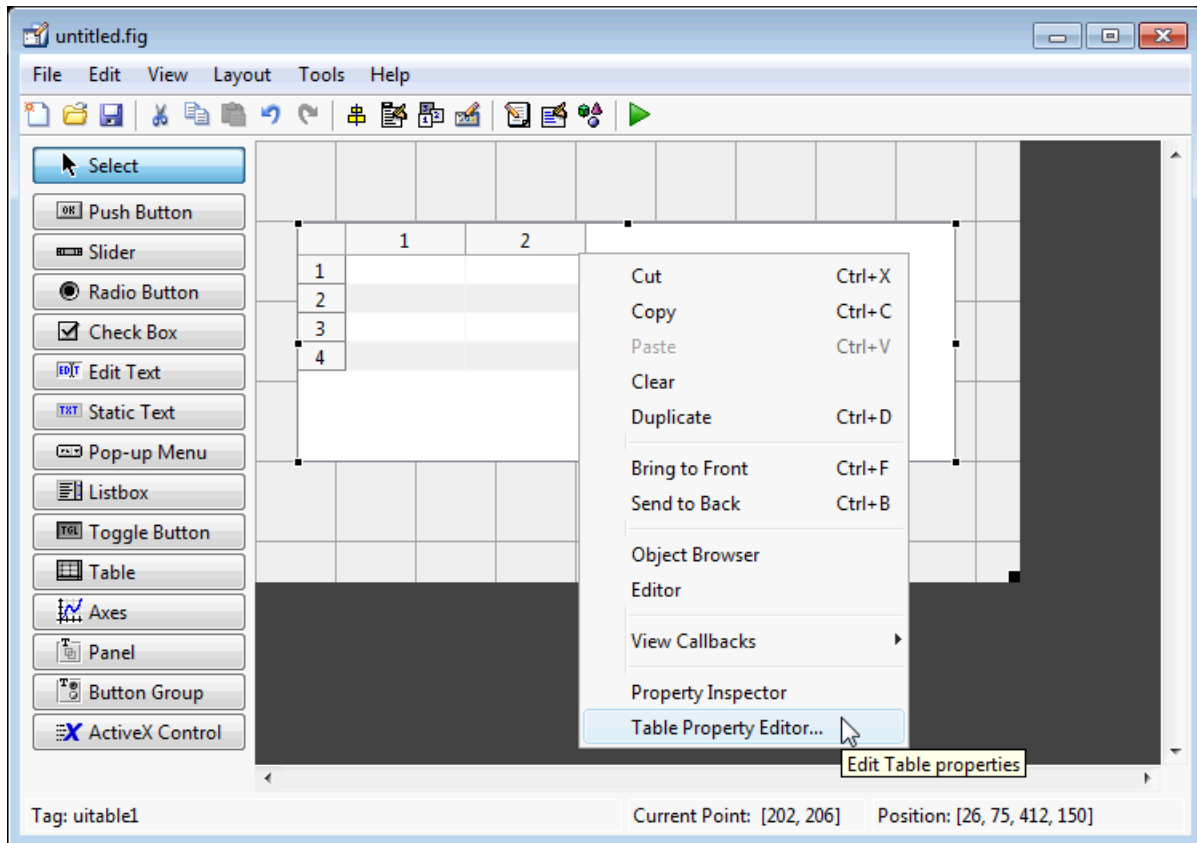
Use the Table Property Editor

When you open it this way, the Table Property Editor displays the **Column** pane. You can also open it from the Property Inspector by clicking one of its Table Property Editor icons , in which case the Table Property Editor opens to display the pane appropriate for the property you clicked.

Clicking items in the list on the left hand side of the Table Property Editor changes the contents of the pane to the right . Use the items to activate controls for specifying the table's **Columns**, **Rows**, **Data**, and **Color** options.

The **Columns** and **Rows** panes each have a data entry area where you can type names and set properties. on a per-column or per-row basis. You can edit only one row or column

definition at a time. These panes contain a vertical group of five buttons for editing and navigating:

Button	Purpose	Accelerator Keys	
		Windows	Macintosh
Insert	Inserts a new column or row definition entry below the current one	**Insert**	**Insert**
Delete	Deletes the current column or row definition entry (no undo)	**Ctrl+D**	**Cmd+D**
Copy	Inserts a Copy of the selected entry in a new row below it	**Ctrl+P**	**Cmd+P**
Up	Moves selected entry up one row	**Ctrl+ uparrow**	**Cmd+ uparrow**
Down	Moves selected entry down one row	**Ctrl+ downarrow**	**Cmd+ downarrow**

Keyboard equivalents only operate when the cursor is in the data entry area. In addition to those listed above, typing **Ctrl+T** or **Cmd+T** selects the entire field containing the cursor for editing (if the field contains text).

To save changes to the table you make in the Table Property Editor, click **OK**, or click **Apply** commit changes and keep on using the Table Property Editor.

Set Column Properties

Click **Insert** to add two more columns.
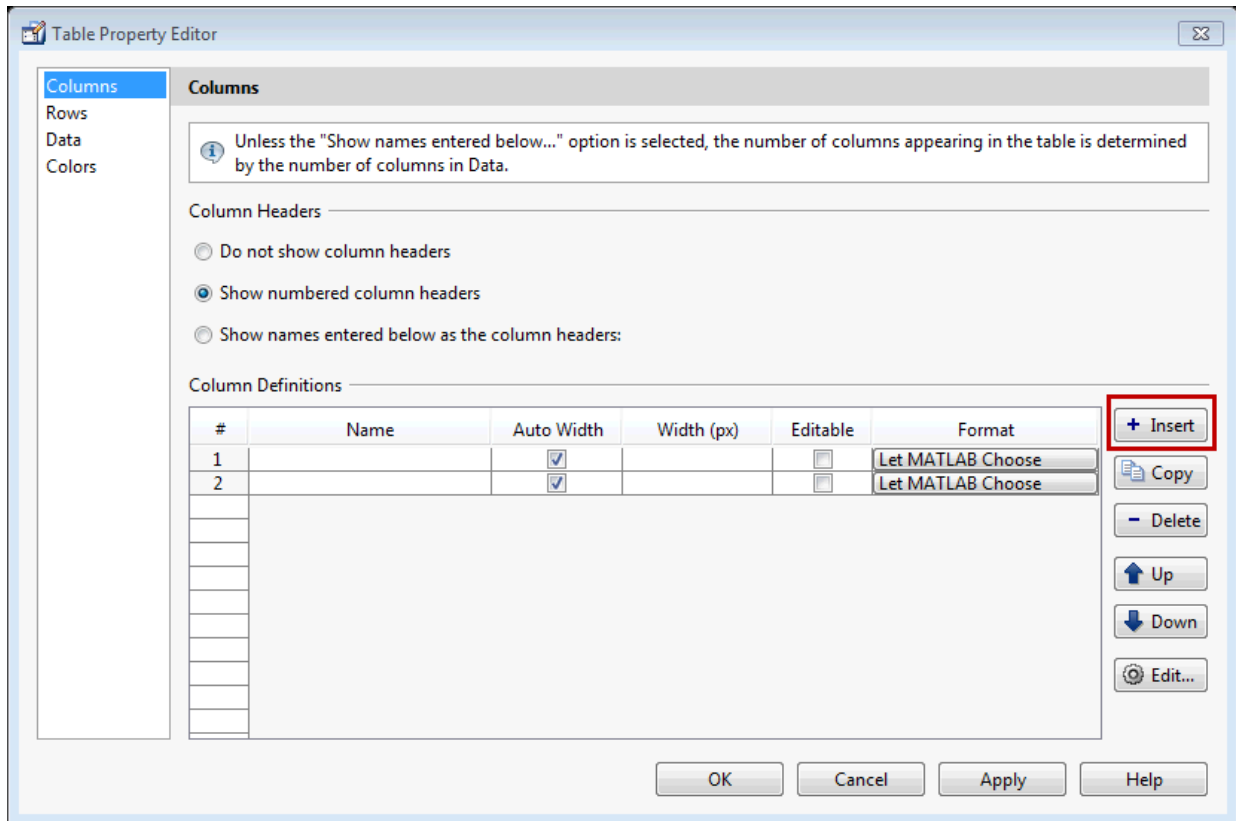
Select **Show names entered below as the column headers** and set the ColumnName by entering Rate, Amount, Available, and Fixed/Adj in **Name** group. for the Available and Fixed/Adj columns set the ColumnEditable property to on. Lastly set the ColumnFormat for the four columns
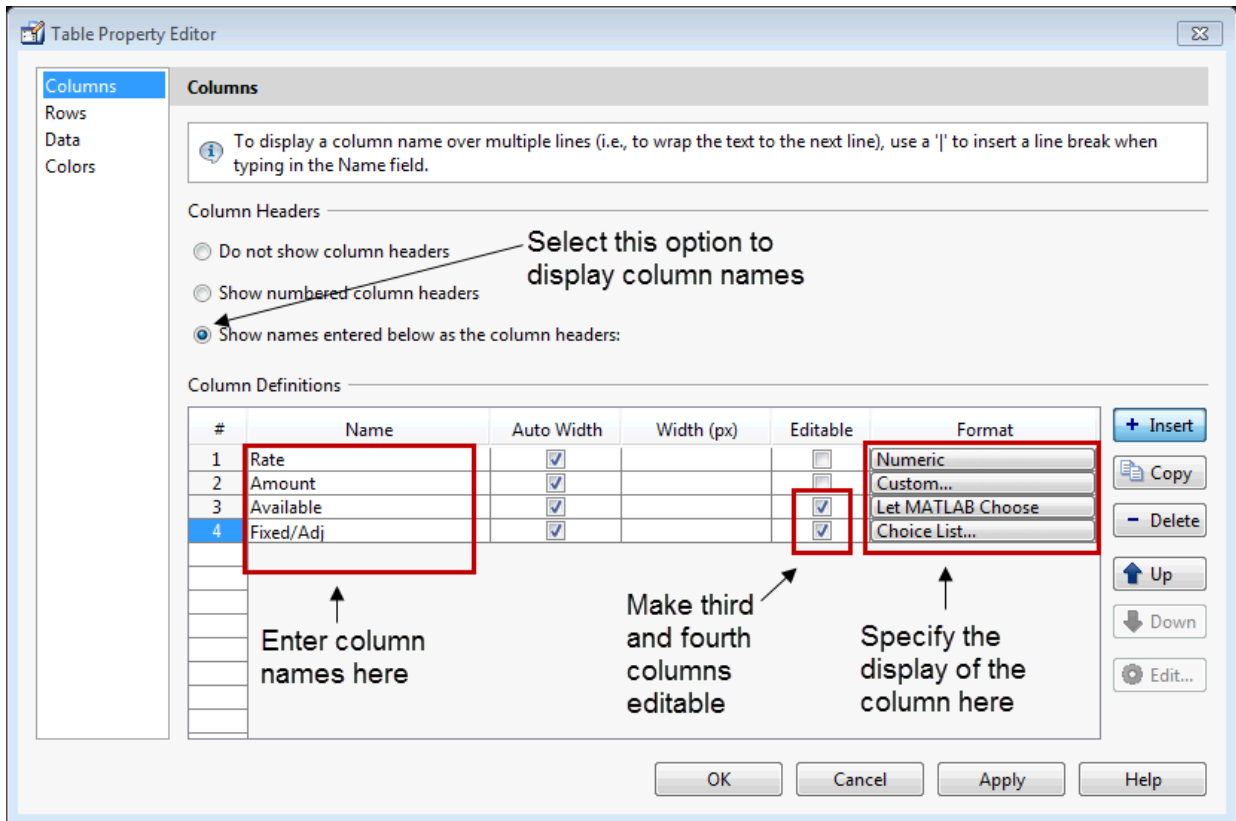
For the Rate column, select **Numeric**. For the Amount Column select **Custom** and in the Custom Format Editor, choose **Bank**.

Custom Format Editor

Select an output format for numeric values:

short	3.1416	Scaled fixed point with 4 digits precision
long	3.14159265358979	Scaled fixed point with 7/14 digits precision
short e	3.1416e+000	Floating point with 4 digits precision
long e	3.141592653589793e+000	Floating point with 7/14 digits precision
short g	3.1416	Best of fixed or floating point with 4 digits precision
long g	3.14159265358979	Best of fixed or floating point with 7/14 digits precision
short eng	3.1416e+000	Engineering format with 4 digits precision
long eng	3.14159265358979e+000	Engineering format with 7/14 digits precision
bank	3.14	Fixed dollars and cents
rat	355/113	Ratio of small integers
+	+	+ for positive, - for negative, blank for zero

Example: PI displays as 3.14

OK   Cancel

Leave the Available column at the default value. This allows MATLAB to chose based on the value of the Data property of the table. For the Fixed/Adj column select `Choice List` to create a pop-up menu. In the Choice List Editor, click **Insert** to add a second choice and type Fixed and Adjustable as the 2 choices.
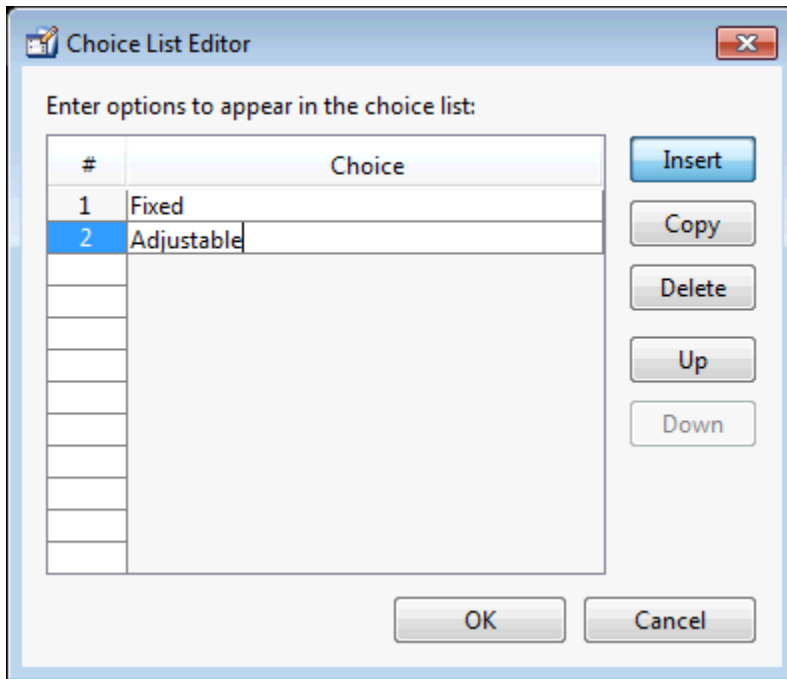
Note: For a user to select items from a choice list, the ColumnEditable property of the column that the list occupies must be set to `'true'`. The pop-up control only appears when the column is editable.

Set Row Properties

In the Row tab, leave the default RowName, **Show numbered row headers**.
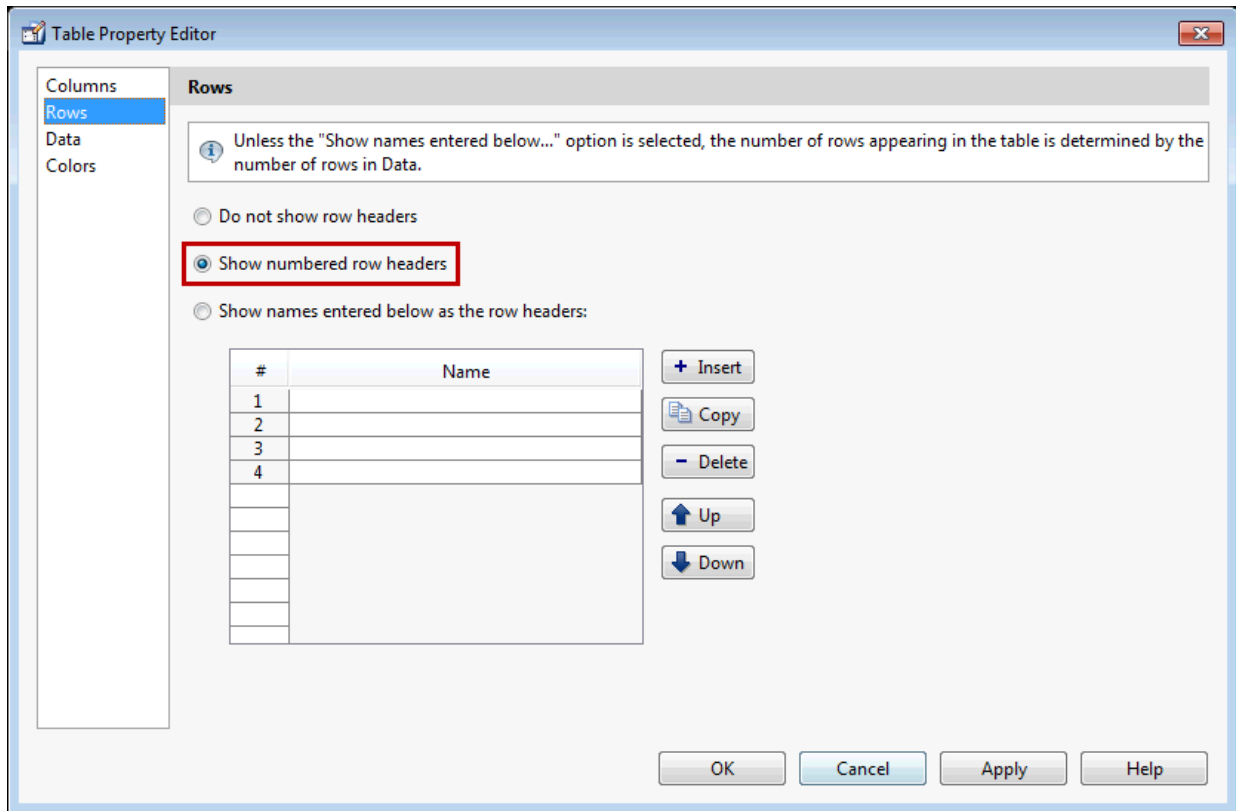
Set Data Properties

Use the Data property to specify the data in the table. Create the data in the command window before you specify it in GUIDE. For this example, type:

```
dat = {6.125, 456.3457, true,  'Fixed';...
6.75,  510.2342, false, 'Adjustable';...
7,     658.2,    false, 'Fixed';};
```

In the Table Property Editor, select the data that you defined and select **Change data value to the selected workspace variable below**.
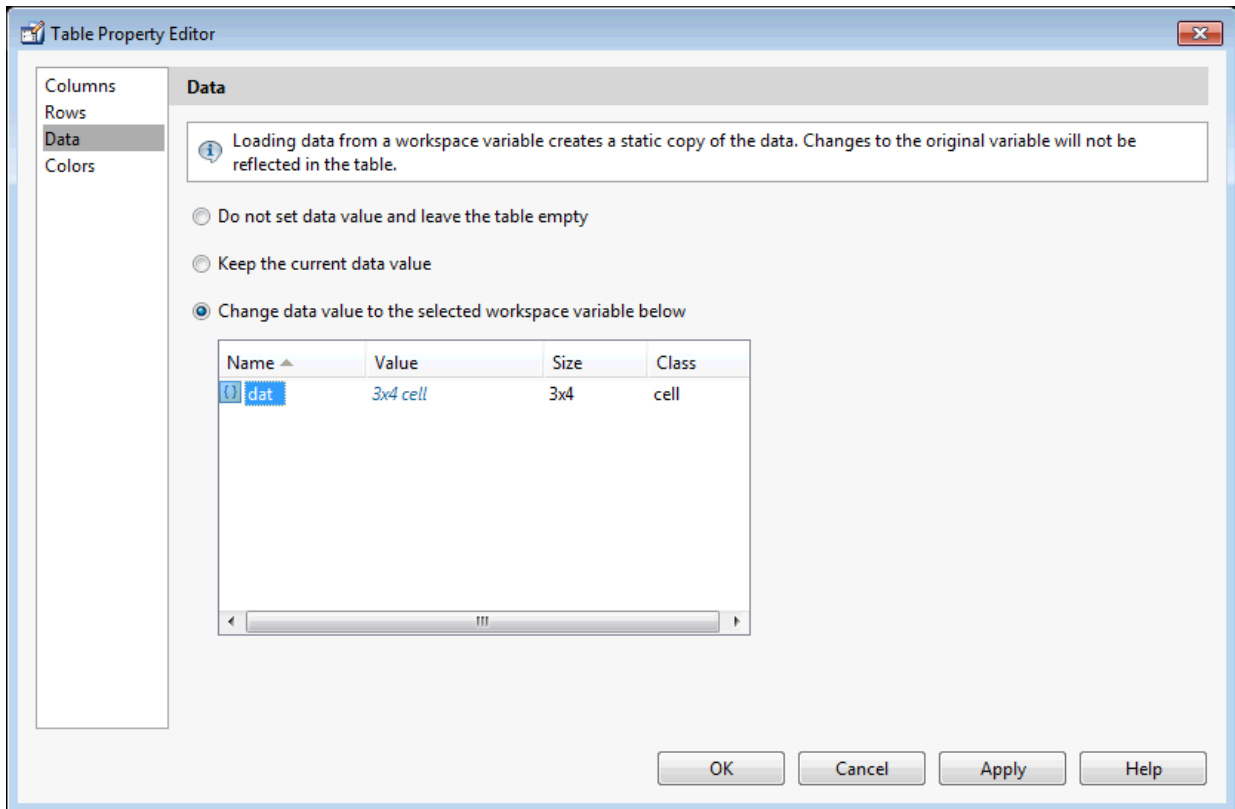
Set Color Properties

Specify the BackgroundColor and RowStriping for your table in the Color tab.
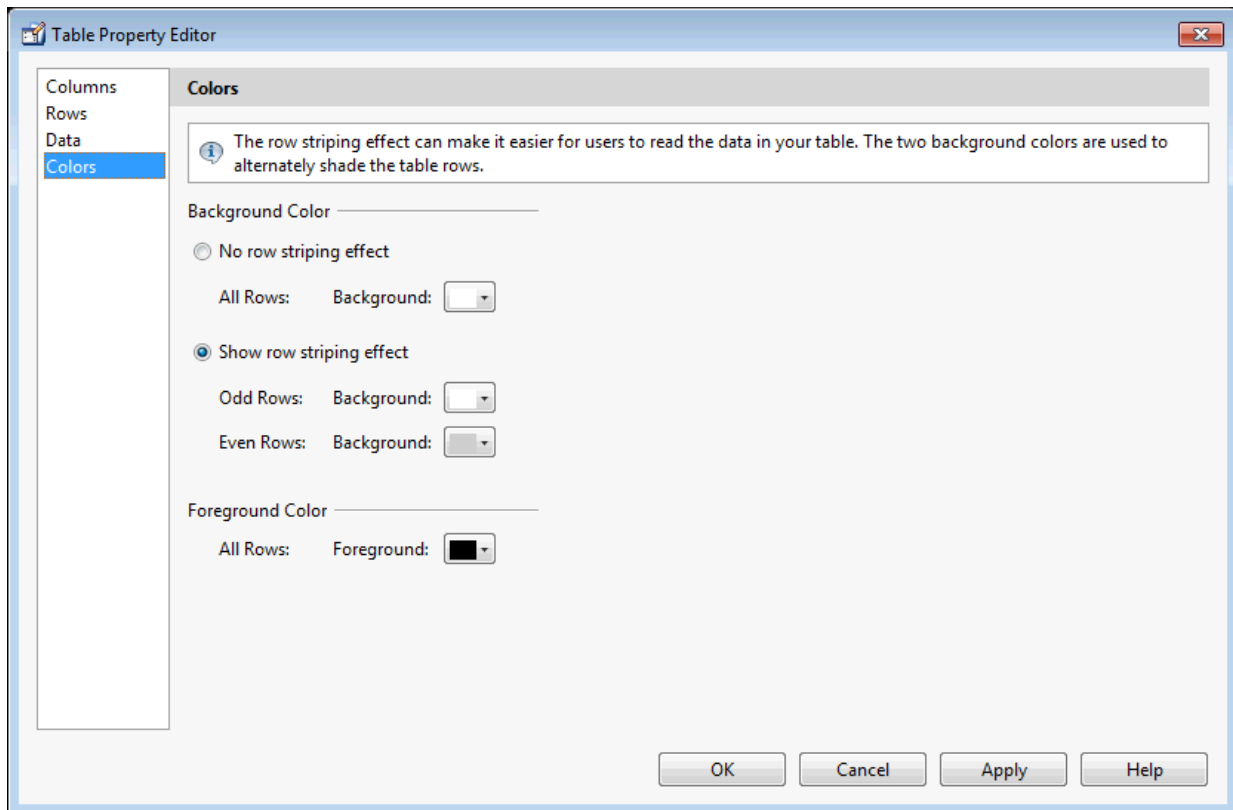
You can change other `uitable` properties to the table via the Property Inspector.

ActiveX Component

When you drag an ActiveX component from the component palette into the layout area, GUIDE opens a dialog box, similar to the following, that lists the registered ActiveX controls on your system.

Note If MATLAB software is not installed locally on your computer — for example, if you are running the software over a network — you might not find the ActiveX control described in this example. To register the control, see "Registering Controls and Servers".
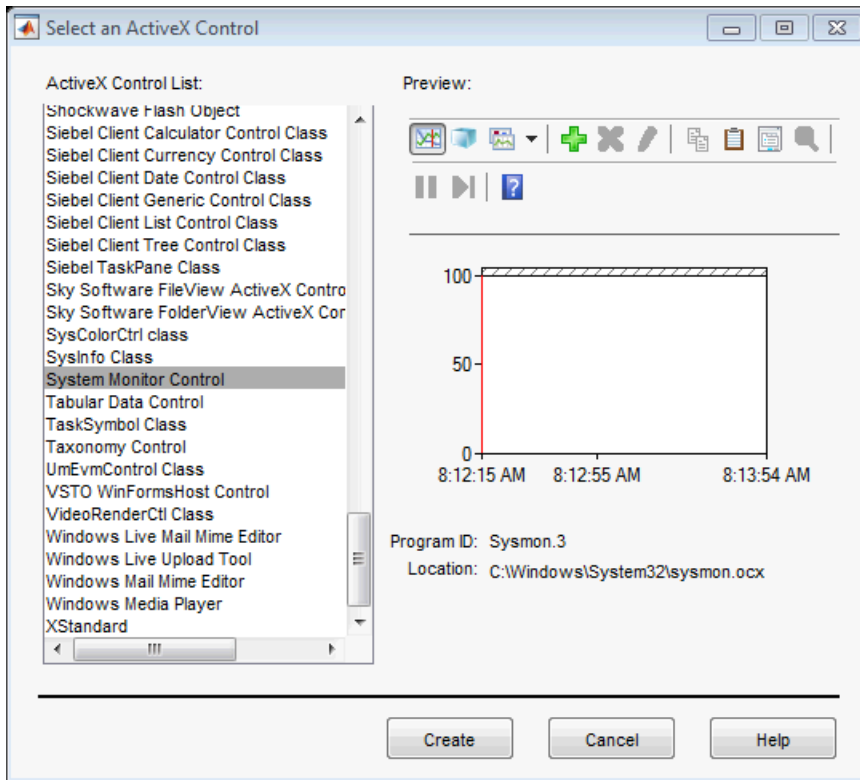
1   Select the desired ActiveX control. The right panel shows a preview of the selected control.

2   Click **Create**. The control appears as a small box in the Layout Editor.

3   Resize the control to approximately the size of the square shown in the preview pane. You can do this by clicking and dragging a corner of the control, as shown in the following figure.
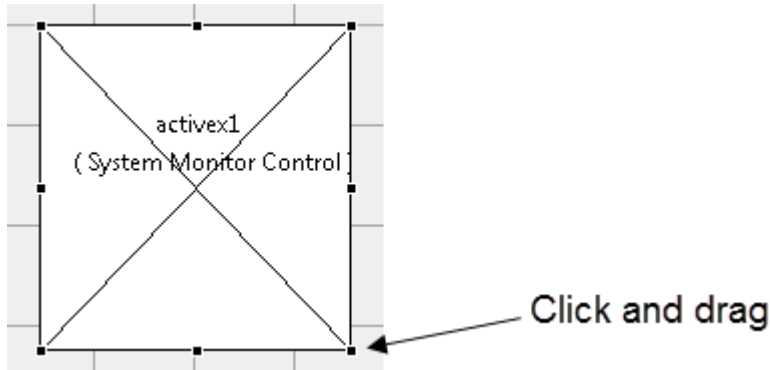
When you select an ActiveX control, you can open the ActiveX Property Editor by right-clicking and selecting **ActiveX Property Editor** from the context menu or clicking the **Tools** menu and selecting it from there.

Note: What an **ActiveX Property Editor** contains and looks like is dependent on what user controls that the authors of the particular ActiveX object have created and stored in the UI for the object. In some cases, a UI without controls or no UI at all appears when you select this menu item.

Resize GUIDE UI Components

You can resize components in one of the following ways:

- "Drag a Corner of the Component" on page 6-62
- "Set the Component's Position Property" on page 6-63

Drag a Corner of the Component

Select the component you want to resize. Click one of the corner handles and drag it until the component is the desired size.

Set the Component's Position Property

Select one or more components that you want to resize. Then select **View** > **Property Inspector**or click the Property Inspector button ![icon].

1   In the Property Inspector, scroll to the `Units` property and note whether the current setting is `characters` or `normalized`. Click the button next to `Units` and then change the setting to `inches` from the pop-up menu.
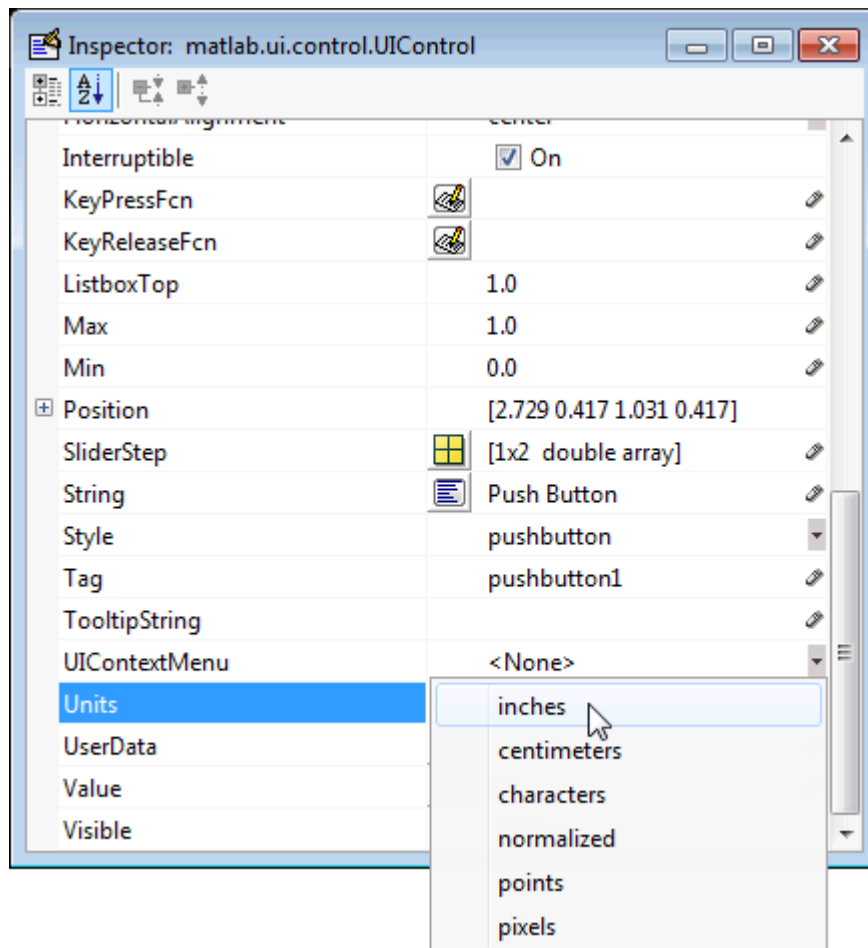
2 Click the **+** sign next to `Position`. The Property Inspector displays the elements of the `Position` property.
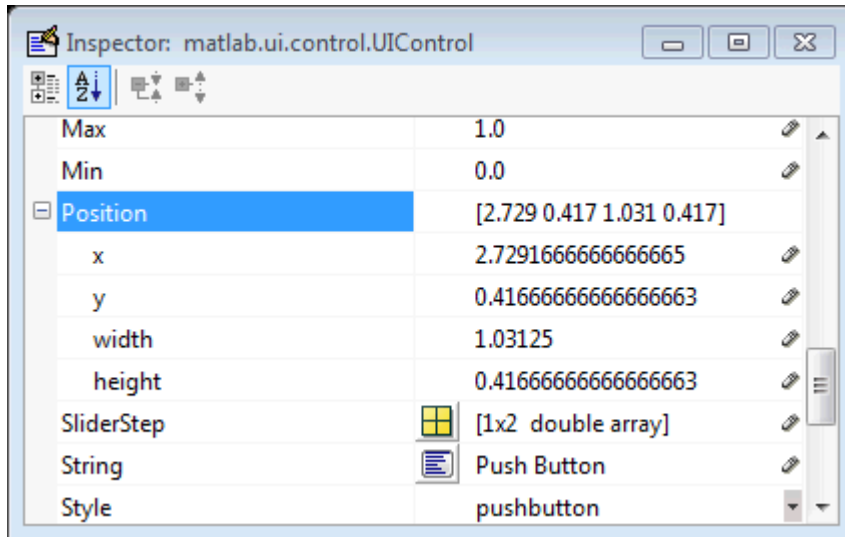
3 Type the `width` and `height` you want the components to be.

4 Reset the `Units` property to its previous setting, either `characters` or `normalized`.

Note To select multiple components, they must have the same parent. That is, they must be contained in the same figure, panel, or button group. Setting the `Units` property to `characters` (nonresizable UIs) or `normalized` (resizable UIs) gives the UI a more consistent appearance across platforms.

Related Examples

- "Create a Simple App Using GUIDE" on page 2-2
- "Callbacks for Specific Components" on page 7-12

More About

- "Ways to Build Apps" on page 1-2
- "Write Callbacks in GUIDE" on page 7-2

Align GUIDE UI Components

In this section...

Align Objects Tool

The Align Objects tool enables you to position objects with respect to each other and to adjust the spacing between selected objects. The specified alignment operations apply to all components that are selected when you press the **Apply** button. To open the Align Objects tool in the GUIDE Layout Editor, select **Tools** > **Align Objects**.

Note To select multiple components, they must have the same parent. That is, they must be contained in the same figure, panel, or button group.
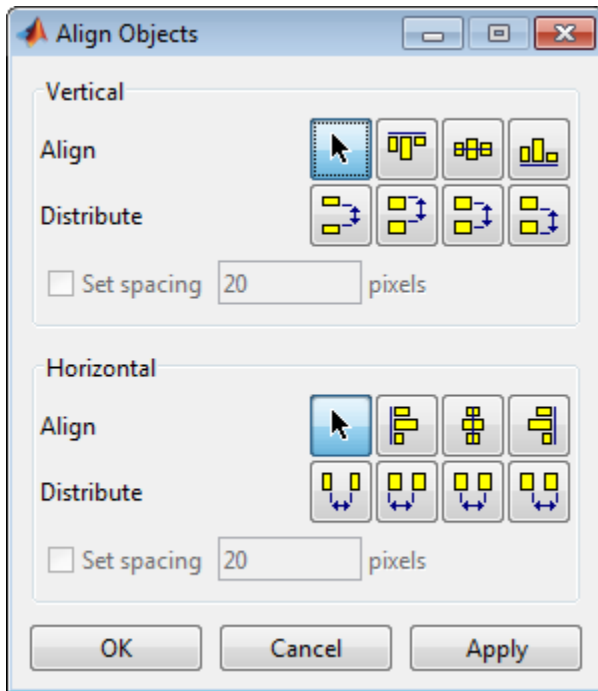
The Align Objects tool provides two types of alignment operations:

- **Align** — Align all selected components to a single reference line.
- **Distribute** — Space all selected components uniformly with respect to each other.

Both types of alignment can be applied in the vertical and horizontal directions. In many cases, it is better to apply alignments independently to the vertical and horizontal using two separate steps.

Align Options

There are both vertical and horizontal align options. Each option aligns selected components to a reference line, which is determined by the bounding box that encloses the selected objects. For example, the following picture of the layout area shows the bounding box (indicated by the dashed line) formed by three selected push buttons.
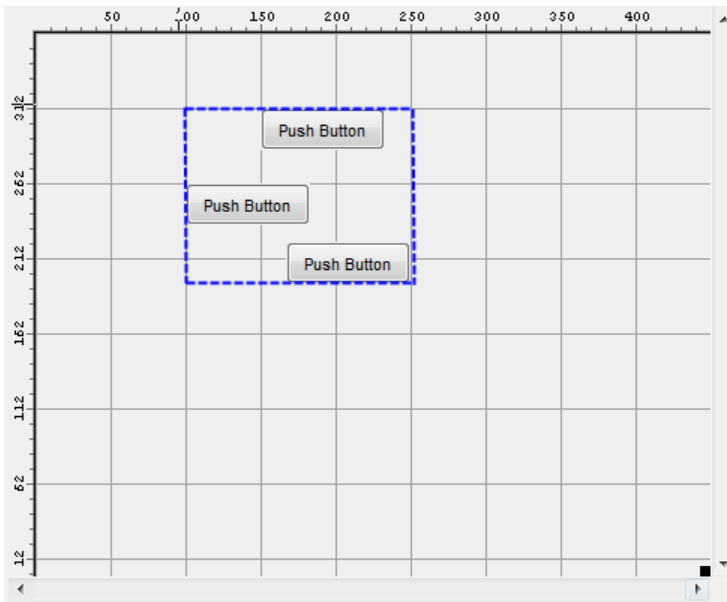
All of the align options (vertical top, center, bottom and horizontal left, center, right) place the selected components with respect to the corresponding edge (or center) of this bounding box.

Distribute Options

Distributing components adds equal space between all components in the selected group. The distribute options operate in two different modes:

- Default behavior — MATLAB distributes space equally among components within the bounding box.
- Select the **Set spacing** check box — You specify the number of pixels between each component.

Both modes enable you to specify how the spacing is measured, as indicated by the button labels on the alignment tool. These options include spacing measured with respect to the following edges:

- Vertical — inner, top, center, and bottom
- Horizontal — inner, left, center, and right

Property Inspector

About the Property Inspector

In GUIDE, as in MATLAB generally, you can see and set most components' properties using the Property Inspector. To open it from the GUIDE Layout Editor, do any of the following:

· Select the component you want to inspect, or double-click it to open the Property Inspector and bring it to the foreground

· Select **View** > **Property Inspector**.

· Click the **Property Inspector** button

The Property Inspector window opens, displaying the properties of the selected component. For example, here is a view of a push button's properties.
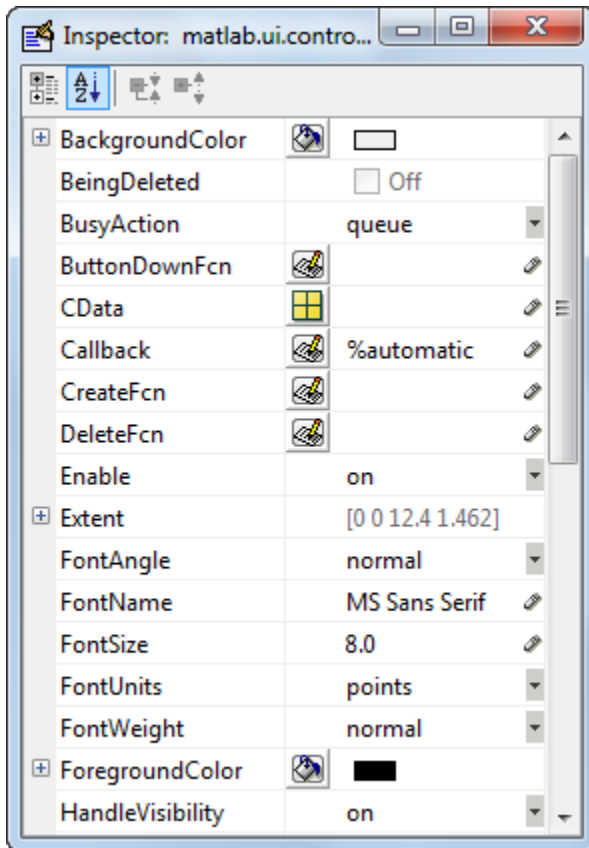
Scroll down to see additional properties. Click any property value or icon to set its value.

The Property Inspector provides context-sensitive help for individual properties. To see a definition of any property, right-click the name or value in the Property Inspector and click the **What's This?** menu item that appears. A context-sensitive help window opens displaying the definition of the property.
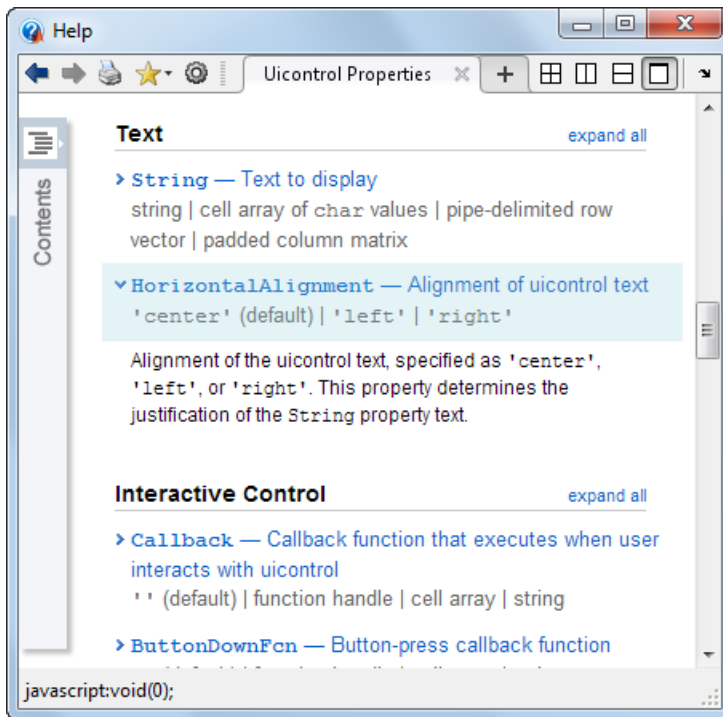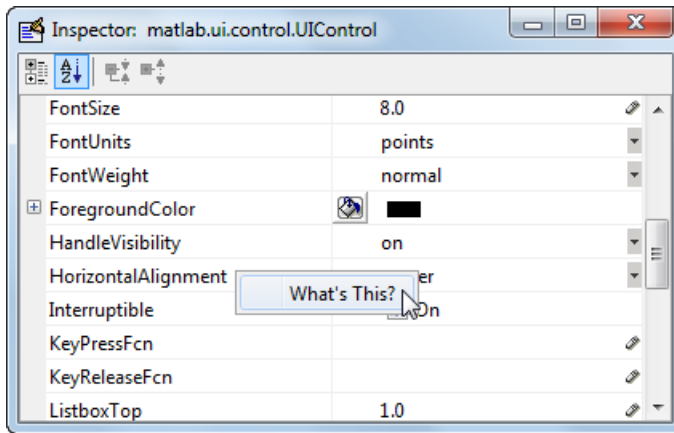
Use the Property Inspector to Align Components

The Property Inspector enables you to align components by setting their `Position` properties. A component's `Position` property is a four-element vector that specifies the size and location of the component: [distance from left, distance from bottom, width, height]. The values are given in the units specified by the `Units` property of the component.

1 Select the components you want to align.

2 Select **View** > **Property Inspector** or click the **Property Inspector** button 📝.

3 In the Property Inspector, scroll to the `Units` property and note its current setting, then change the setting to `inches`.

4 Scroll to the `Position` property. This figure shows the `Position` property for multiple components of the same size.
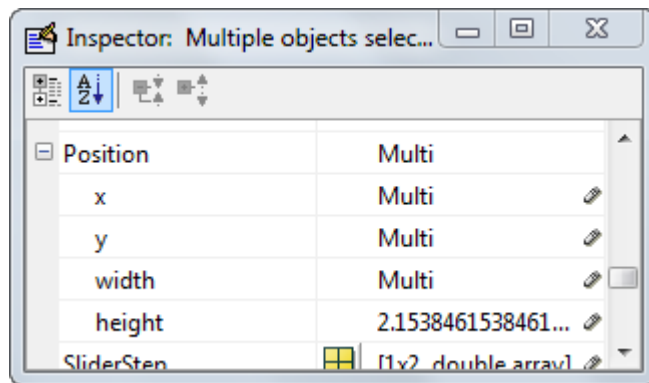
5 Change the value of `x` to align their left sides. Change the value of `y` to align their bottom edges. For example, setting `x` to 2.0 aligns the left sides of the components 2 inches from the left side of the window.

6 When the components are aligned, change the `Units` property back to its original setting.

Grid and Rulers

The layout area displays a grid and rulers to facilitate component layout. Grid lines are spaced at 50-pixel intervals by default. The size of each pixel is 1/96th of an inch on Windows systems and 1/72nd of an inch on Macintosh systems. On Linux® systems, the size of a pixel is determined by your system resolution.

You can optionally enable *snap-to-grid*, which causes any object that is moved close to a grid line to jump to that line. Snap-to-grid works with or without a visible grid.
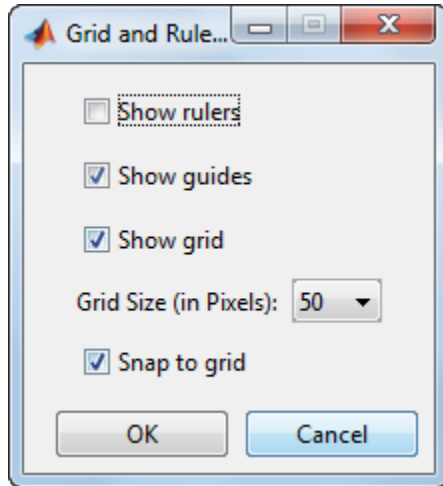
Use the Grid and Rulers dialog (select **Tools > Grid and Rulers**) to:

- Control visibility of rulers, grid, and guide lines
- Set the grid spacing
- Enable or disable snap-to-grid

Guide Lines

The Layout Editor has both vertical and horizontal snap-to guide lines. Components snap to the line when you move them close to the line.

Guide lines are useful when you want to establish a reference for component alignment at an arbitrary location in the Layout Editor.

Creating Guide Lines

To create a guide line, click the top or left ruler and drag the line into the layout area.

Guide line for
horizontal alignment

Guide line for
vertical alignment
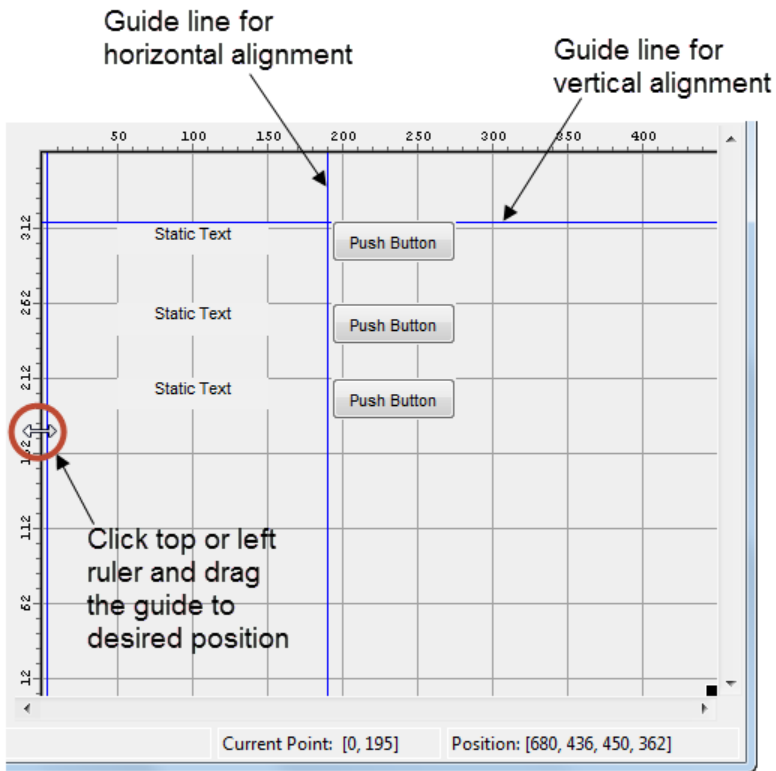
More About

- "GUIDE Options" on page 5-8

Customize Tabbing Behavior in a GUIDE UI

A UI's tab order is the order in which components of the UI acquire focus when a user presses the **Tab** key on the keyboard. Focus is generally denoted by a border or a dotted border.

You can set, independently, the tab order of components that have the same parent. The figure window, each panel, and each button group has its own tab order. For example, you can set the tab order of components that have the figure as a parent. You can also set the tab order of components that have a panel or button group as a parent.

If, in tabbing through the components at the figure level, a user tabs to a panel or button group, then subsequent tabs sequence through the components of the panel or button group before returning to the level from which the panel or button group was reached.

Note Axes cannot be tabbed. From GUIDE, you cannot include ActiveX components in the tab order.

When you create a UI, GUIDE sets the tab order at each level to be the order in which you add components to that level in the Layout Editor. This may not be the best order for the user.

Note Tab order also affects the stacking order of components. If components overlap, those that appear lower in the tabbing order, are drawn on top of those that appear higher in the order.

The figure in the following UI contains an axes component, a slider, a panel, static text, and a pop-up menu. Of these, only the slider, the panel, and the pop-up menu at the figure level can be tabbed. The panel contains three push buttons, which can all be tabbed.
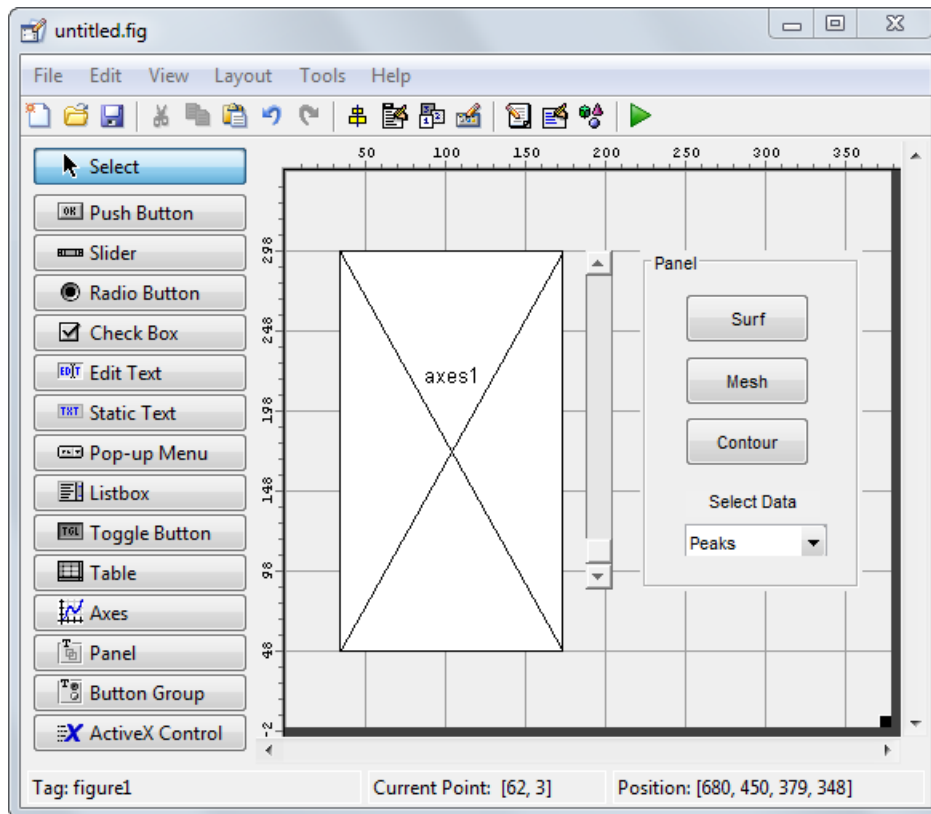
To examine and change the tab order of the panel components, click the panel background to select it, then select **Tools** > **Tab Order Editor** in the Layout Editor.
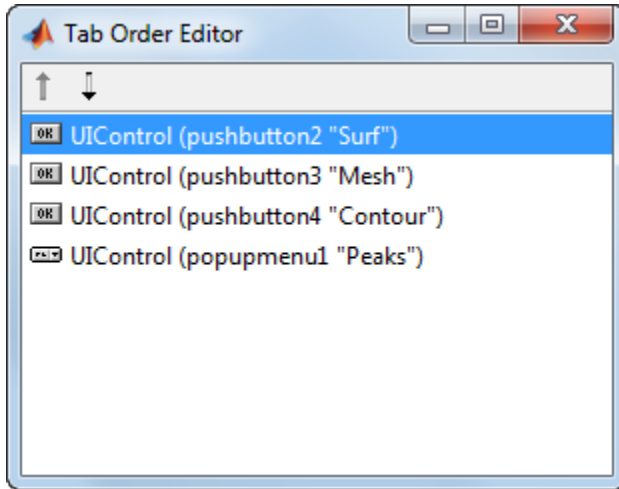
The Tab Order Editor displays the panel's components in their current tab order. To change the tab order, select a component and press the up or down arrow to move the component up or down in the list. If you set the tab order for the first three components in the example to be

1   **Surf** push button

2   **Contour** push button

3   **Mesh** push button

the user first tabs to the **Surf** push button, then to the **Contour** push button, and then to the **Mesh** push button. Subsequent tabs sequence through the remaining components at the figure level.

Create Menus for GUIDE UIs

In this section...
"Menus for the Menu Bar" on page 6-77
"Context Menus" on page 6-87

You can use GUIDE to give UIs menu bars with pull-down menus as well as context menus that you attach to components. You can create both types of menus using the Menu Editor. Access the Menu Editor from the **Tools** menu or click the **Menu Editor** button .
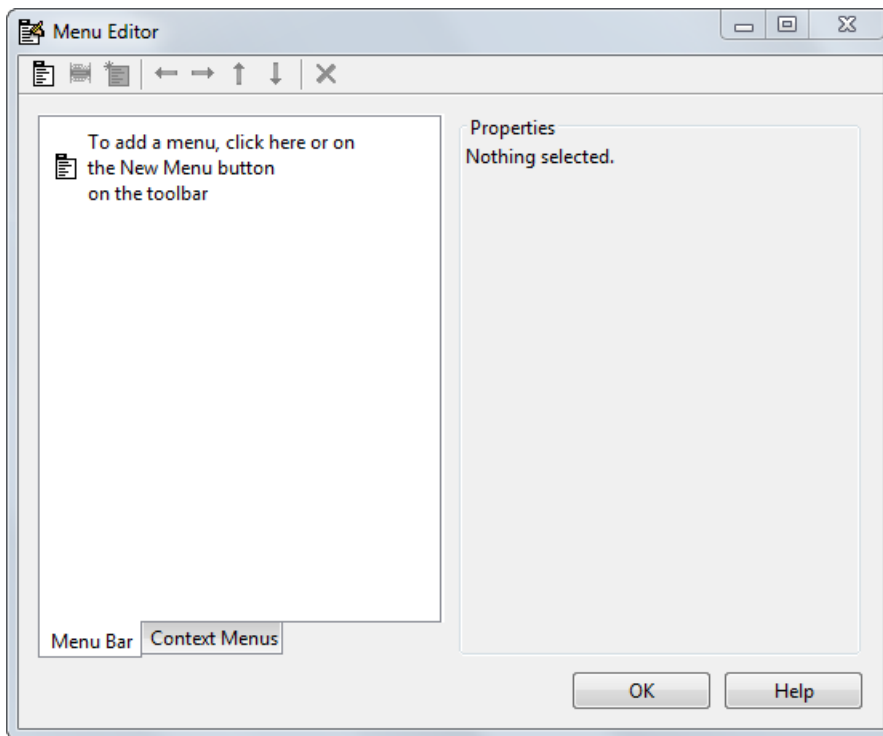
Menus for the Menu Bar

- "How Menus Affect Figure Docking" on page 6-78

- "Add Standard Menus to the Menu Bar" on page 6-79
- "Create a Menu" on page 6-79
- "Add Items to a Menu" on page 6-82
- "Additional Drop-Down Menus" on page 6-84
- "Cascading Menus" on page 6-84

When you create a drop-down menu, GUIDE adds its title to the menu bar. You then can create menu items for that menu. Each menu item can have a cascading menu, also known as a submenu, and these items can have cascading menus, and so on.

How Menus Affect Figure Docking

By default, when you create a UI with GUIDE, it does not create a menu bar for that UI. You might not need menus for your UI, but if you want the user to be able to dock or undock the UI window, it must contain a menu bar or a toolbar. This is because docking is controlled by the docking icon, a small curved arrow near the upper-right corner of the menu bar or the toolbar, as the following illustration shows.
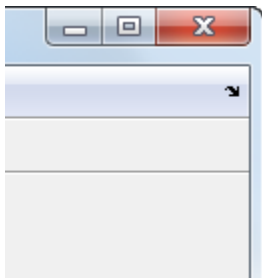
Figure windows with a standard menu bar also have a **Desktop** menu from which the user can dock and undock them.

To display the docking arrow and the **Desktop** > **Dock Figure** menu item, use the Property Inspector to set the figure property DockControls to 'on'. You must also set the MenuBar and/or ToolBar figure properties to 'figure' to display docking controls.

The WindowStyle figure property also affects docking behavior. The default is 'normal', but if you change it to 'docked', then the following applies:

- The UI window opens docked in the desktop when you run it.

- The `DockControls` property is set to `'on'` and cannot be turned off until `WindowStyle` is no longer set to `'docked'`.

- If you undock a UI window created with `WindowStyle` `'docked'`, it will have not have a docking arrow unless the figure displays a menu bar or a toolbar (either standard or customized). When it has no docking arrow, users can undock it from the desktop, but will be unable to redock it there.

However, when you provide your own menu bar or toolbar using GUIDE, it can display the docking arrow if you want the UI window to be dockable. See the following sections and "Create Toolbars for GUIDE UIs" on page 6-94 for details.

Note: UIs that are modal dialogs (figures with `WindowStyle` set to `'modal'`) cannot have menu bars, toolbars, or docking controls.

For more information, see the `DockControls`, `MenuBar`, `ToolBar`, and `WindowStyle` property descriptions in Figure Properties.

Add Standard Menus to the Menu Bar

The figure `MenuBar` property controls whether your UI displays the MATLAB standard menus on the menu bar. GUIDE initially sets the value of `MenuBar` to `none`. If you want your UI to display the MATLAB standard menus, use the Property Inspector to set `MenuBar` to `figure`.

- If the value of `MenuBar` is `none`, GUIDE automatically adds a menu bar that displays only the menus you create.

- If the value of `MenuBar` is `figure`, the UI displays the MATLAB standard menus and GUIDE adds the menus you create to the right side of the menu bar.

In either case, you can enable the user to dock and undock the window by setting the figure's `DockControls` property to `'on'`.

Create a Menu

1 Start a new menu by clicking the New Menu button in the toolbar. A menu title, `Untitled 1`, appears in the left pane of the dialog box.
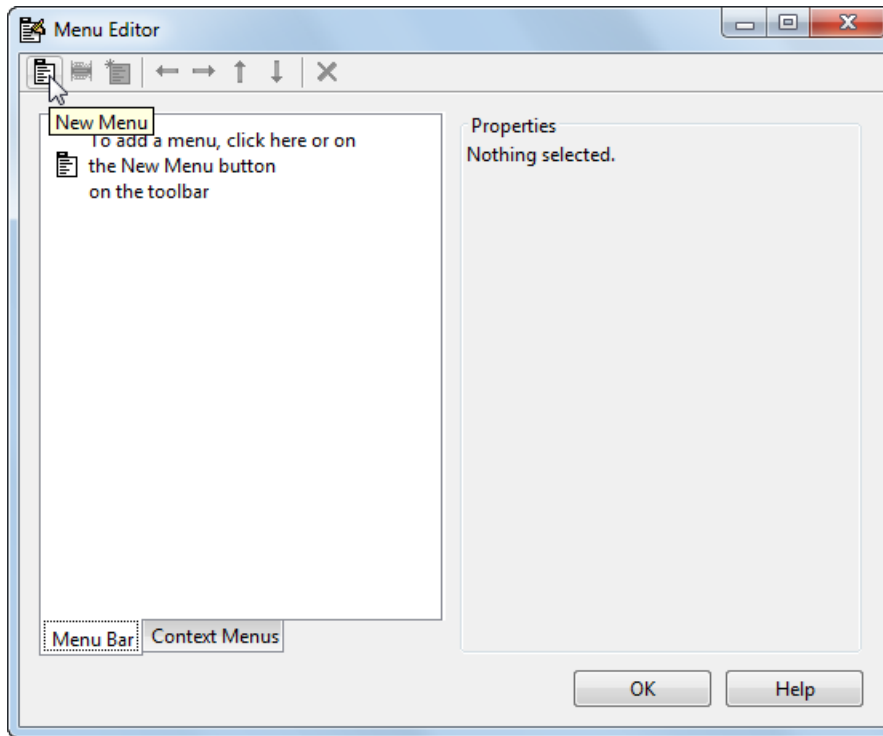
Note By default, GUIDE selects the **Menu Bar** tab when you open the Menu Editor.

2 Click the menu title to display a selection of menu properties in the right pane.
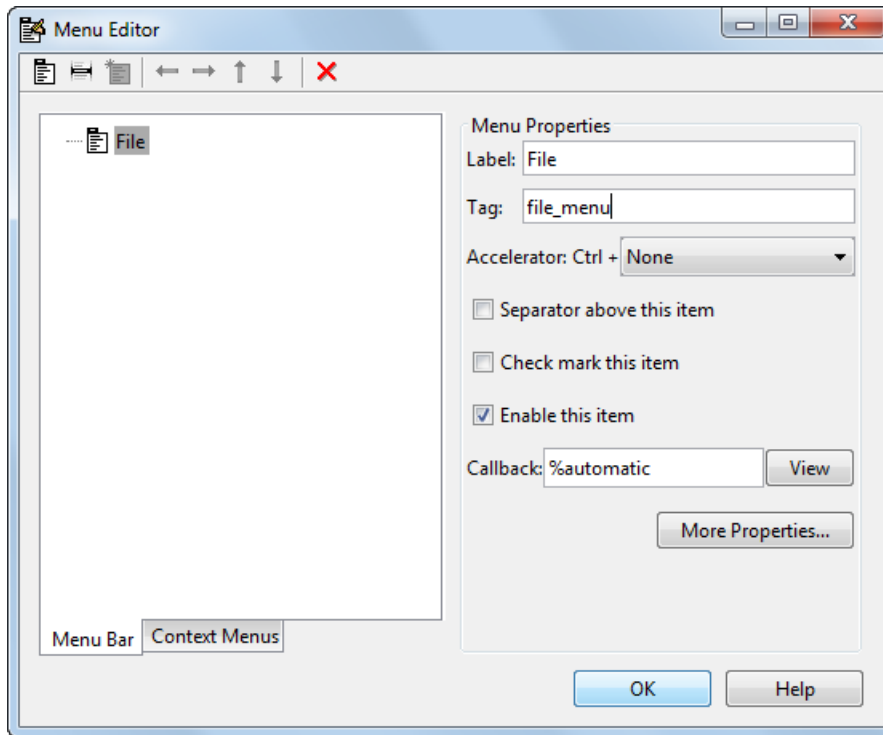
3   Fill in the **Label** and **Tag** fields for the menu. For example, set **Label** to `File` and set **Tag** to `file_menu`. Click outside the field for the change to take effect.

Label is a string that specifies the text label for the menu item. To display the & character in a label, use two & characters in the string. The words `remove`, `default`, and `factory` (case sensitive) are reserved. To use one of these as labels, prepend a backslash (\) to the string. For example, `\remove` yields **remove.**

Tag is a string that is an identifier for the menu object. It is used in the code to identify the menu item and must be unique in your UI code file.

Add Items to a Menu

Use the **New Menu Item** tool to create menu items that are displayed in the drop-down menu.

1 Add an **Open** menu item under `File`, by selecting `File` then clicking the **New Menu Item** button in the toolbar. A temporary numbered menu item label, `Untitled`, appears.
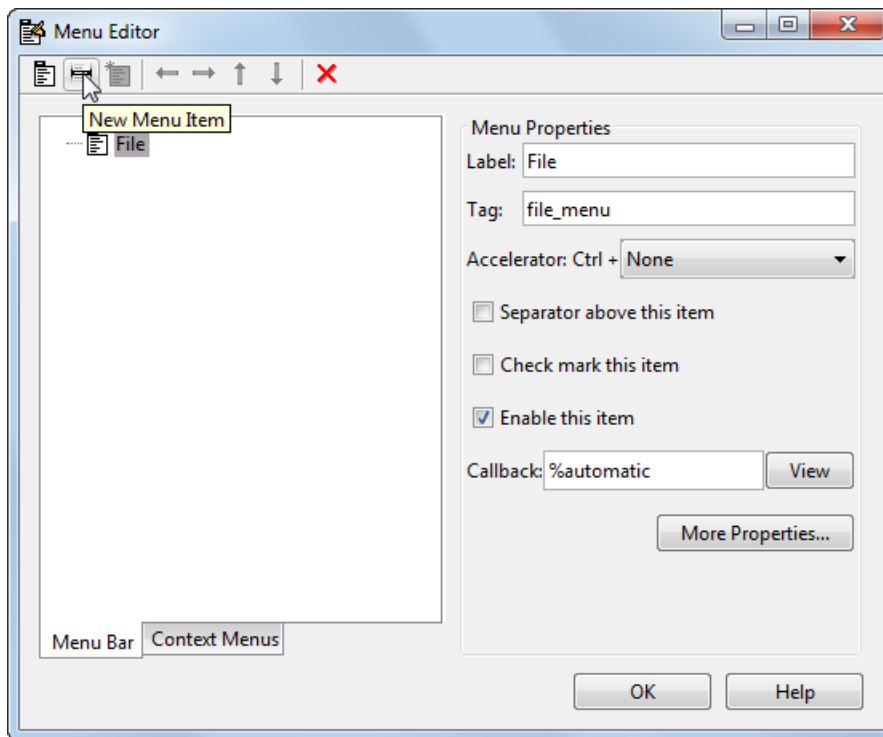
2 Fill in the **Label** and **Tag** fields for the new menu item. For example, set **Label** to `Open` and set **Tag** to `menu_file_open`. Click outside the field for the change to take effect.
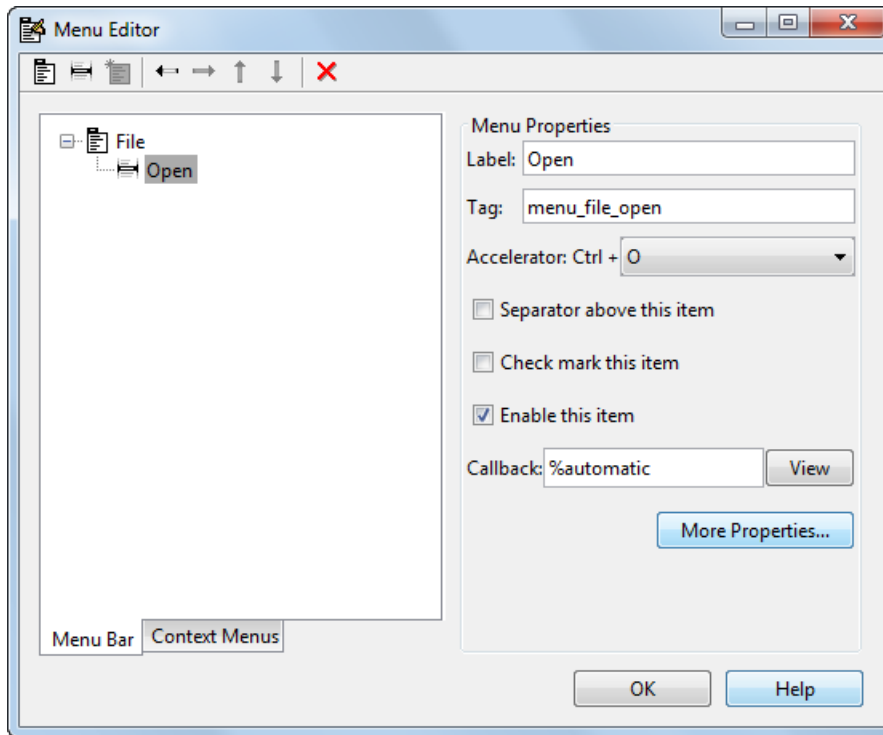
You can also

- Choose an alphabetic keyboard accelerator for the menu item with the **Accelerator** pop-up menu. In combination with **Ctrl**, this is the keyboard equivalent for a menu item that does not have a child menu. Note that some accelerators may be used for other purposes on your system and that other actions may result.

- Display a separator above the menu item by checking **Separator above this item**.

- Display a check next to the menu item when the menu is first opened by checking **Check mark this item**. A check indicates the current state of the menu item. See the example in "Add Items to the Context Menu" on page 6-89.

- Enable this item when the menu is first opened by checking **Enable this item**. This allows the user to select this item when the menu is first opened. If you clear this option, the menu item appears dimmed when the menu is first opened, and the user cannot select it.

- Specify a string for the routine, i.e., the **Callback**, that performs the action associated with the menu item. If you have not yet saved the UI, the default value is `%automatic`. When you save the UI, and if you have not changed this field, GUIDE automatically sets the value using a combination of the **Tag** field and the UI file name. See "Menu Item" on page 7-23 for more information about specifying this field and for programming menu items.

  The **View** button displays the callback, if there is one, in an editor. If you have not yet saved the UI, GUIDE prompts you to save it.

- Open the Property Inspector, where you can change all menu properties, by clicking the **More Properties** button. For detailed information about the properties, see Uimenu Properties.

Note See "Menu Item" on page 7-23 and "How to Update a Menu Item Check" on page 7-25 for programming information and basic examples.

Additional Drop-Down Menus

To create additional drop-down menus, use the New Menu button in the same way you did to create the `File` menu. For example, the following figure also shows an `Edit` drop-down menu.

Cascading Menus

To create a cascading menu, select the menu item that will be the title for the cascading menu, then click the **New Menu Item** button. In the example below, `Copy` is a cascading menu.
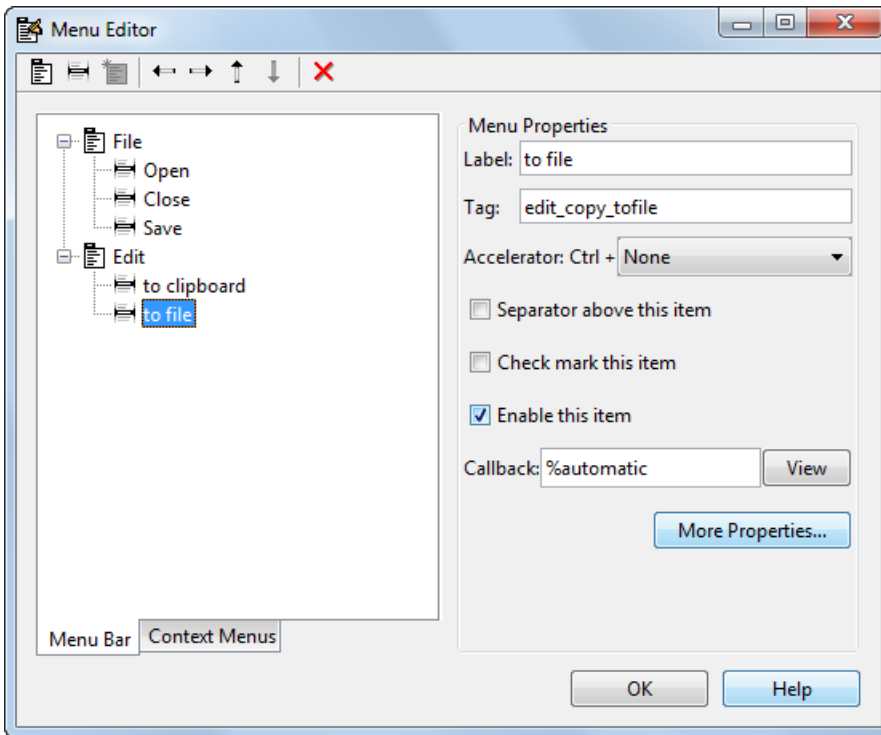
Note See "Menu Item" on page 7-23 for information about programming menu items.

The following Menu Editor illustration shows three menus defined for the figure menu bar.
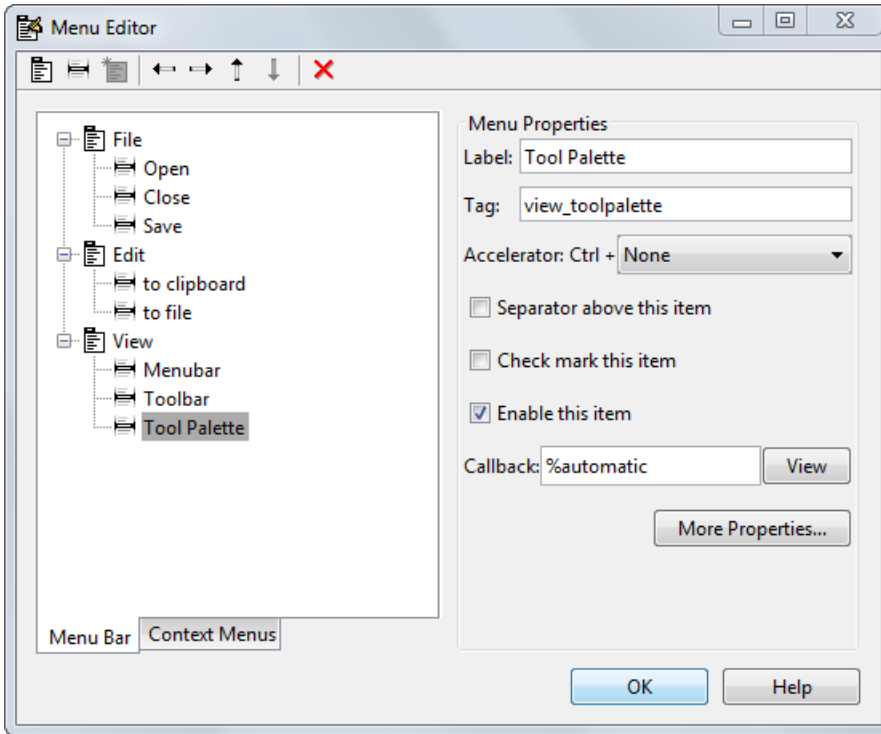
When you run your program, the menu titles appear in the menu bar.
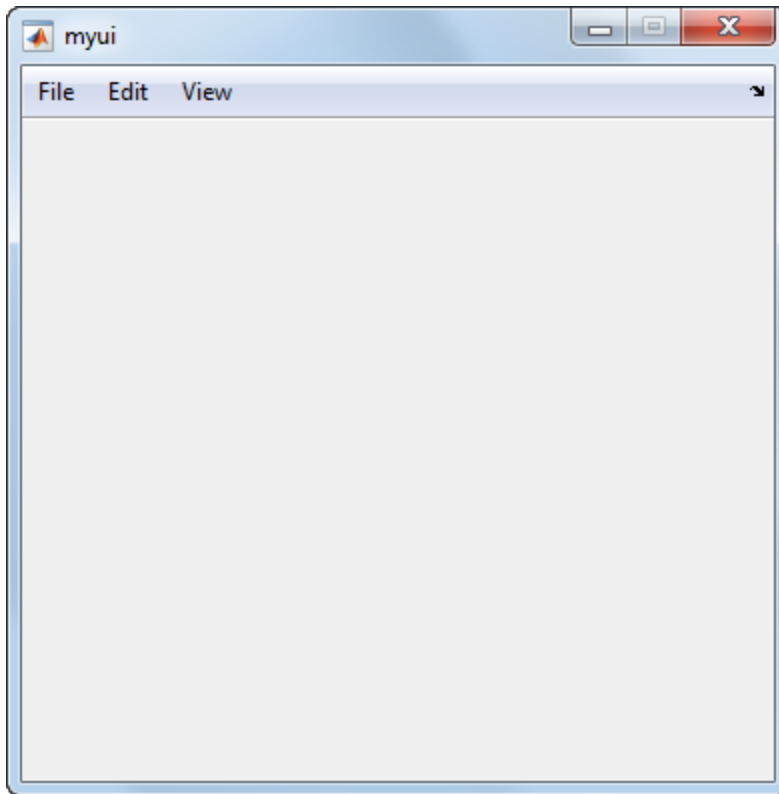
Context Menus

A context menu is displayed when a user right-clicks the object for which the menu is defined. The Menu Editor enables you to define context menus and associate them with objects in the layout. The process has three steps:

1 "Create the Parent Menu" on page 6-88
2 "Add Items to the Context Menu" on page 6-89
3 "Associate the Context Menu with an Object" on page 6-92

Note See "Menus for the Menu Bar" on page 6-77 for information about defining menus in general. See "Menu Item" on page 7-23 for information about defining local callback functions for your menus.

Create the Parent Menu

All items in a context menu are children of a menu that is not displayed on the figure menu bar. To define the parent menu:

1   Select the Menu Editor's **Context Menus** tab and select the New Context Menu button from the toolbar.
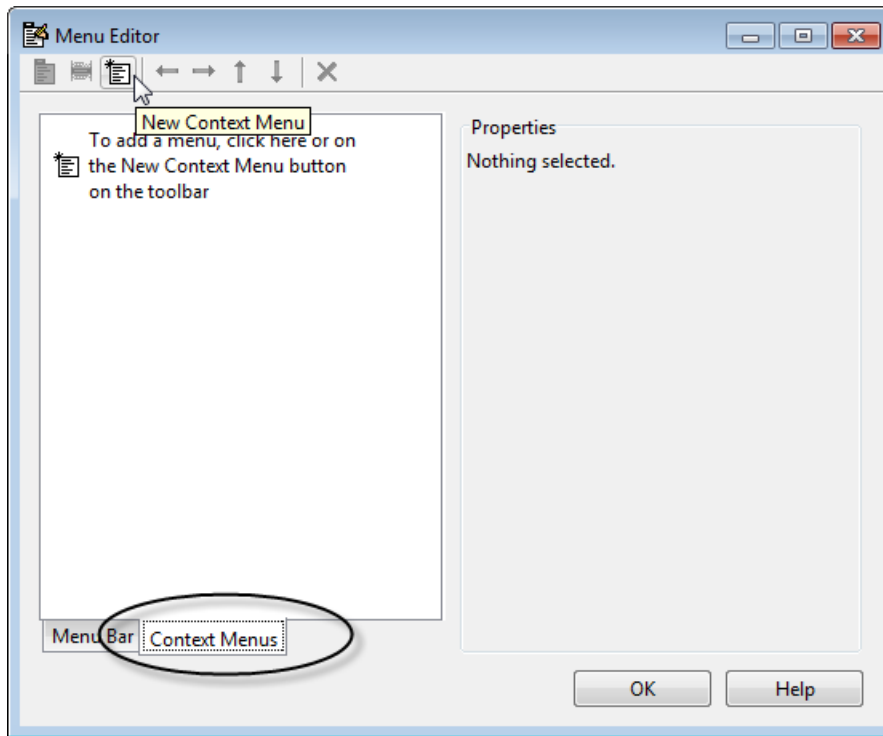
2   Select the menu, and in the **Tag** field type the context menu tag (`axes_context_menu` in this example).
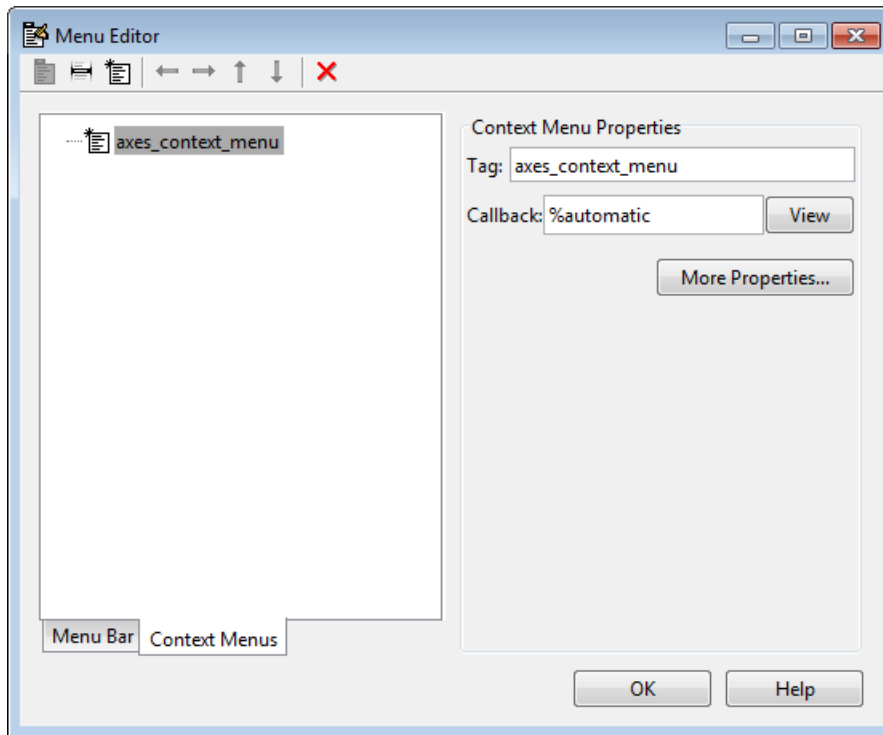
Add Items to the Context Menu

Use the New Menu Item button to create menu items that are displayed in the context menu.

1   Add a **Blue background color** menu item to the menu by selecting
    axes_context_menu and clicking the **New Menu Item** tool. A temporary
    numbered menu item label, Untitled, appears.
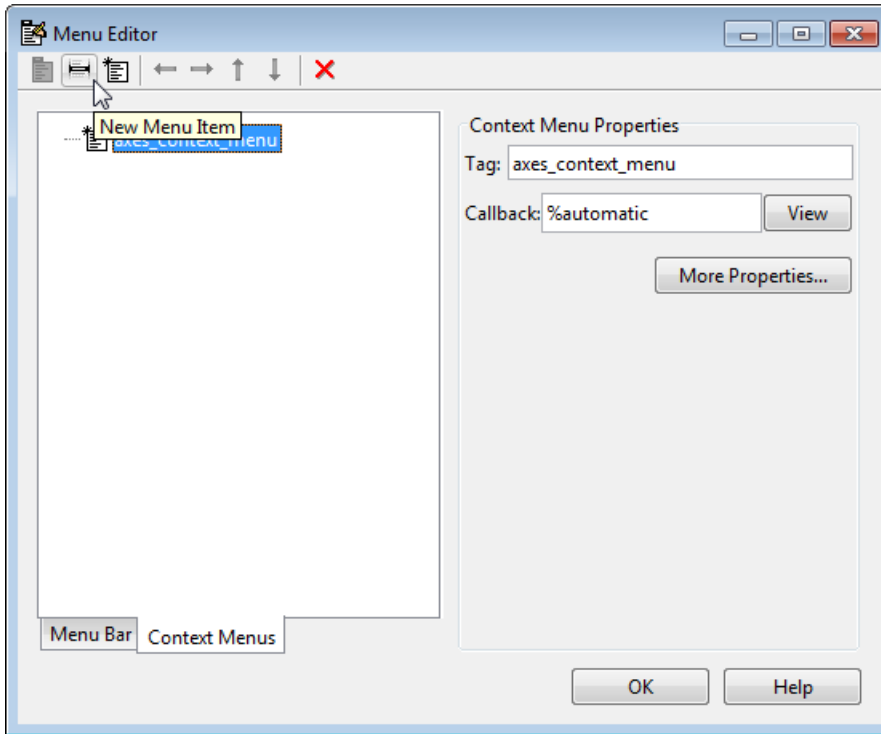
2 Fill in the **Label** and **Tag** fields for the new menu item. For example, set **Label** to `Blue background color` and set **Tag** to `blue_background`. Click outside the field for the change to take effect.
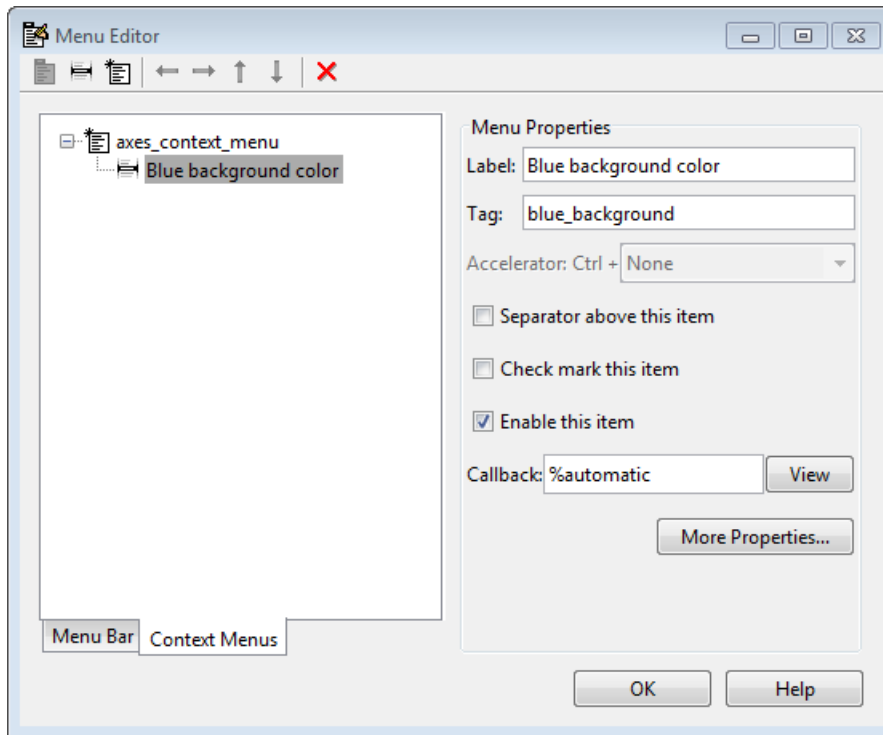
You can also modify menu items in these ways:

- Display a separator above the menu item by checking **Separator above this item**.

- Display a check next to the menu item when the menu is first opened by checking **Check mark this item**. A check indicates the current state of the menu item. See the example in "Add Items to the Context Menu" on page 6-89. See "How to Update a Menu Item Check" on page 7-25 for a code example.

- Enable this item when the menu is first opened by checking **Enable this item**. This allows the user to select this item when the menu is first opened. If you clear this option, the menu item appears dimmed when the menu is first opened, and the user cannot select it.

- Specify a **Callback** for the menu that performs the action associated with the menu item. If you have not yet saved the UI, the default value is %automatic. When you save the UI, and if you have not changed this field, GUIDE automatically creates a

callback in the code file using a combination of the **Tag** field and the UI file name. The callback's name does not display in the **Callback** field of the Menu Editor, but selecting the menu item does trigger it.

You can also type an unquoted string into the **Callback** field to serve as a callback. It can be any valid MATLAB expression or command. For example, the string

```
set(gca, 'Color', 'y')
```
sets the current axes background color to yellow. However, the preferred approach to performing this operation is to place the callback in the UI code file. This avoids the use of `gca`, which is not always reliable when several figures or axes exist. Here is a version of this callback coded as a function in the UI code file:

```
function axesyellow_Callback(hObject, eventdata, handles)
% hObject    handle to axesyellow (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)
set(handles.axes1,'Color','y')
```
This code sets the background color of the axes with Tag `axes1` no matter to what object the context menu is attached to.

If you enter a callback string in the Menu Editor, it overrides the callback for the item in the code file, if any has been saved. If you delete a string you have entered in the **Callback** field, the callback for the item in the UI code file is executed when the user selects that item in the UI.

See "Menu Item" on page 7-23 for more information about specifying this field and for programming menu items. For another example of programming context menus in GUIDE, see "Synchronized Data Presentations in a GUIDE UI" on page 8-17.

The **View** button displays the callback, if there is one, in an editor. If you have not yet saved the UI, GUIDE prompts you to save it.

- Open the Property Inspector, where you can change all menu properties except callbacks, by clicking the **More Properties** button. For detailed information about these properties, see Uicontextmenu Properties.

Associate the Context Menu with an Object

1 In the Layout Editor, select the object for which you are defining the context menu.

2 Use the Property Inspector to set this object's `UIContextMenu` property to the name of the desired context menu.

The following figure shows the `UIContextMenu` property for the `axes` object with `Tag` property `axes1`.
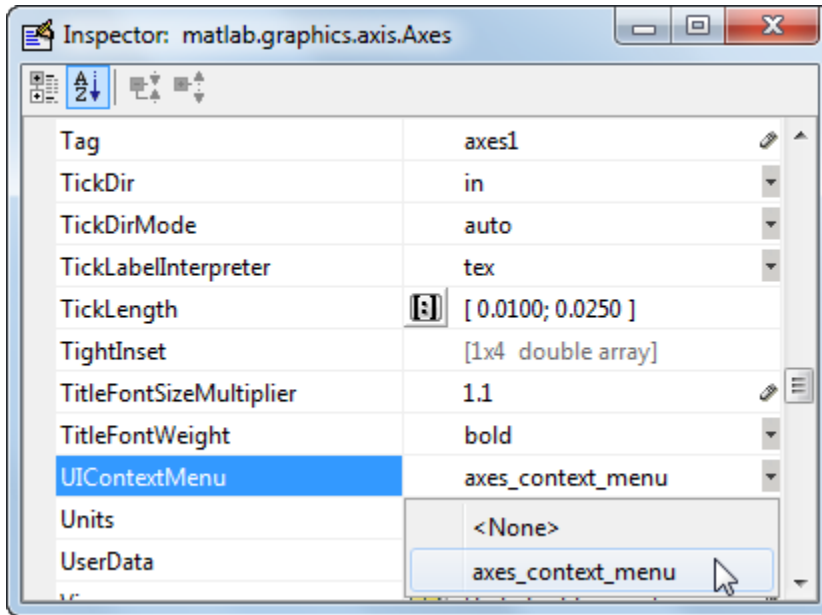
In the UI code file, complete the local callback function for each item in the context menu. Each callback executes when a user selects the associated context menu item. See "Menu Item" on page 7-23 for information on defining the syntax.

Note See "Menu Item" on page 7-23 and "How to Update a Menu Item Check" on page 7-25 for programming information and basic examples.

Related Examples

- "Callbacks for Specific Components" on page 7-12
- "Create Toolbars for GUIDE UIs" on page 6-94

More About

- "Write Callbacks in GUIDE" on page 7-2

Create Toolbars for GUIDE UIs

In this section...
"Toolbar and Tools" on page 6-94
"Editing Tool Icons" on page 6-102

Toolbar and Tools

To add a toolbar to a UI, select the Toolbar Editor.
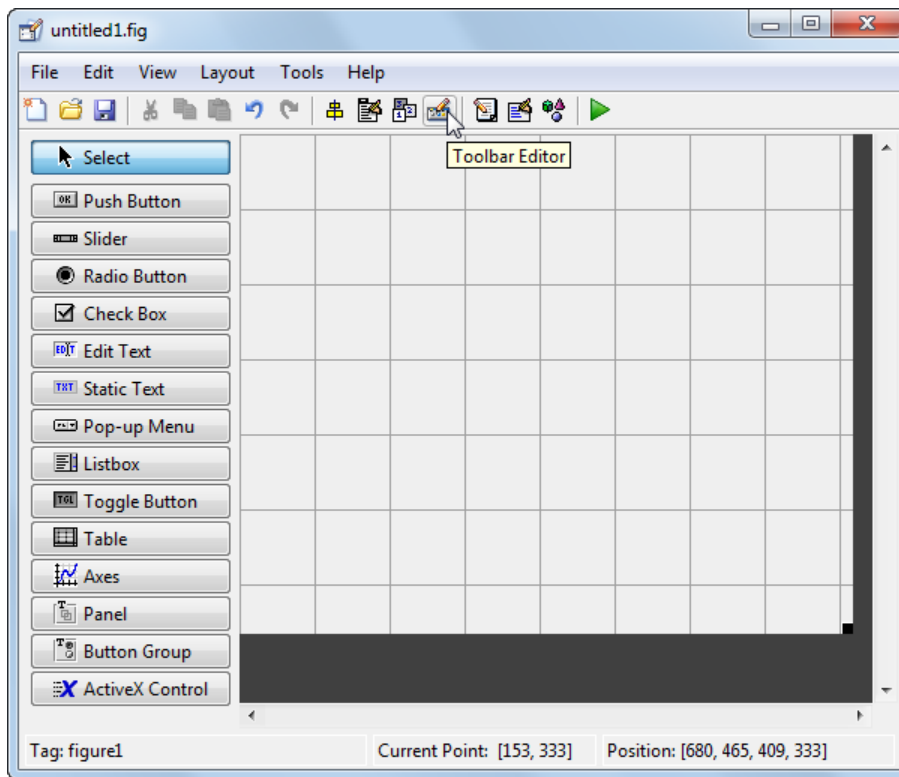
You can also open the Toolbar Editor from the **Tools** menu.
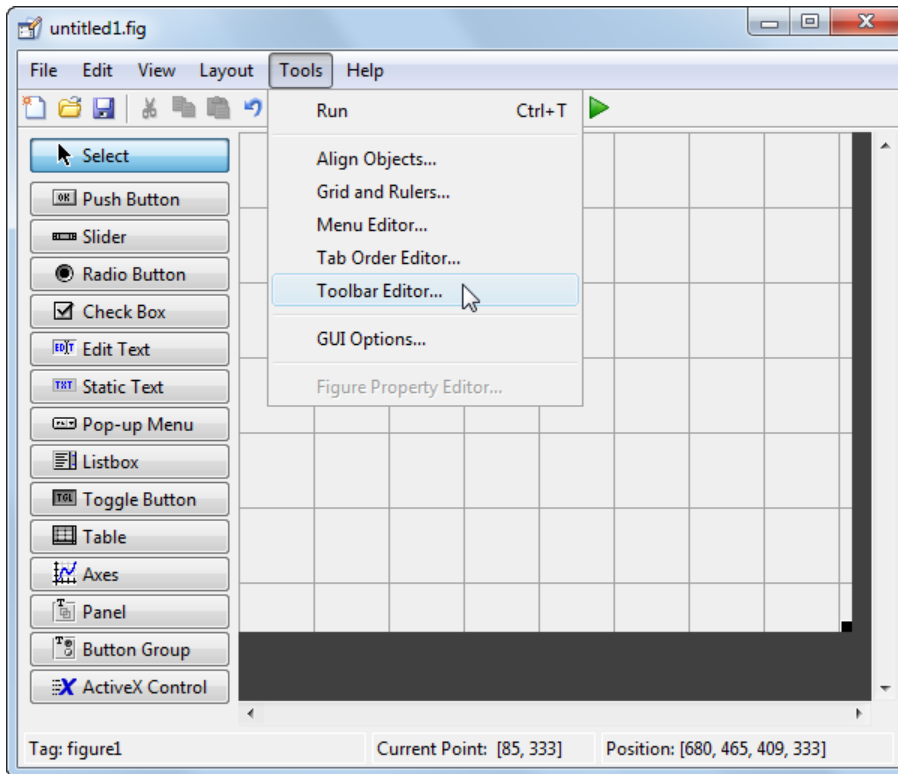
The Toolbar Editor gives you interactive access to all the features of the `uitoolbar`, `uipushtool`, and `uitoggletool` functions. It only operates in the context of GUIDE; you cannot use it to modify any of the built-in MATLAB toolbars. However, you can use the Toolbar Editor to add, modify, and delete a toolbar from any UI in GUIDE.

Currently, you can add one toolbar to your UI in GUIDE. However, your UI can also include the standard MATLAB figure toolbar. If you need to, you can create a toolbar that looks like a normal figure toolbar, but customize its callbacks to make tools (such as pan, zoom, and open) behave in specific ways.

Note: You do not need to use the Toolbar Editor if you simply want your UI to have a standard figure toolbar. You can do this by setting the figure's `ToolBar` property to `'figure'`, as follows:

1   Open the UI in GUIDE.

2   From the **View** menu, open **Property Inspector**.

3   Set the `ToolBar` property to `'figure'` using the drop-down menu.

4   Save the figure

If you later want to remove the figure toolbar, set the `ToolBar` property to `'auto'` and resave the UI. Doing this will not remove or hide your custom toolbar. See "Create Toolbars for Programmatic UIs" on page 9-54 for more information about making toolbars manually.

If you want users to be able to dock and undock a UI window on the MATLAB desktop, it must have a toolbar or a menu bar, which can either be the standard ones or ones you create in GUIDE. In addition, the figure property `DockControls` must be turned on. For details, see "How Menus Affect Figure Docking" on page 6-78.

Use the Toolbar Editor

The Toolbar Editor contains three main parts:

· The **Toolbar Layout** preview area on the top

· The **Tool Palette** on the left

· Two tabbed property panes on the right
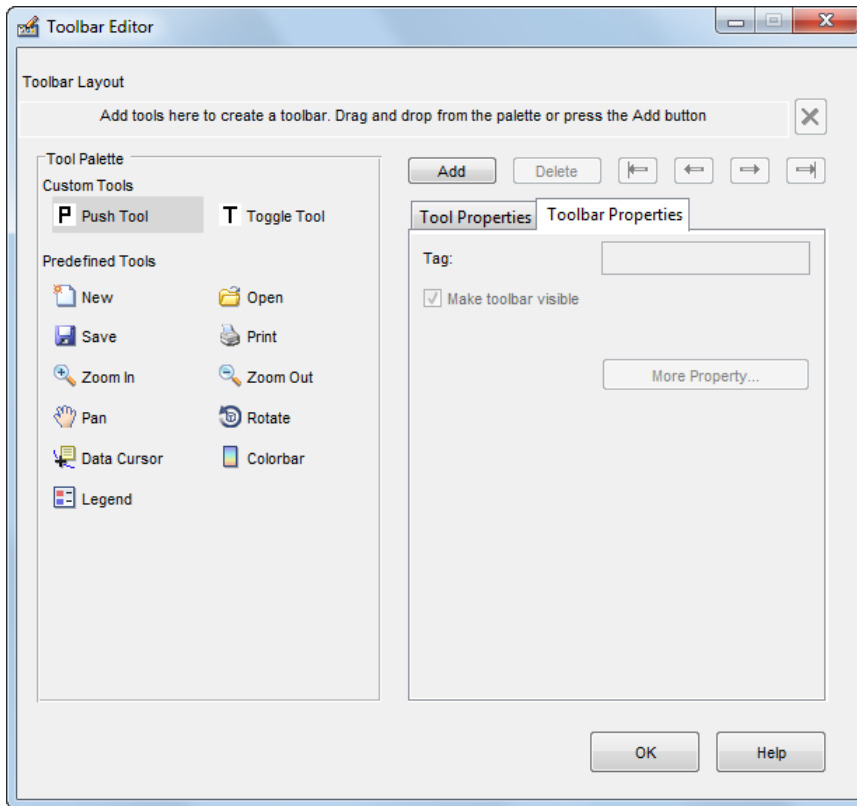
To add a tool, drag an icon from the **Tool Palette** into the **Toolbar Layout** (which initially contains the text prompt shown above), and edit the tool's properties in the **Tool Properties** pane.

When you first create a UI, no toolbar exists on it. When you open the Toolbar Editor and place the first tool, a toolbar is created and a preview of the tool you just added appears in the top part of the window. If you later open a UI that has a toolbar, the Toolbar Editor shows the existing toolbar, although the Layout Editor does not.

Add Tools

You can add a tool to a toolbar in three ways:

•   Drag and drop tools from the **Tool Palette**.

- Select a tool in the palette and click the **Add** button.
- Double-click a tool in the palette.

Dragging allows you to place a tool in any order on the toolbar. The other two methods place the tool to the right of the right-most tool on the **Toolbar Layout**. The new tool is selected (indicated by a dashed box around it) and its properties are shown in the **Tool Properties** pane. You can select only one tool at a time. You can cycle through the **Tool Palette** using the tab key or arrow keys on your computer keyboard. You must have placed at least one tool on the toolbar.

After you place tools from the **Tool Palette** into the **Toolbar Layout** area, the Toolbar Editor shows the properties of the currently selected tool, as the following illustration shows.
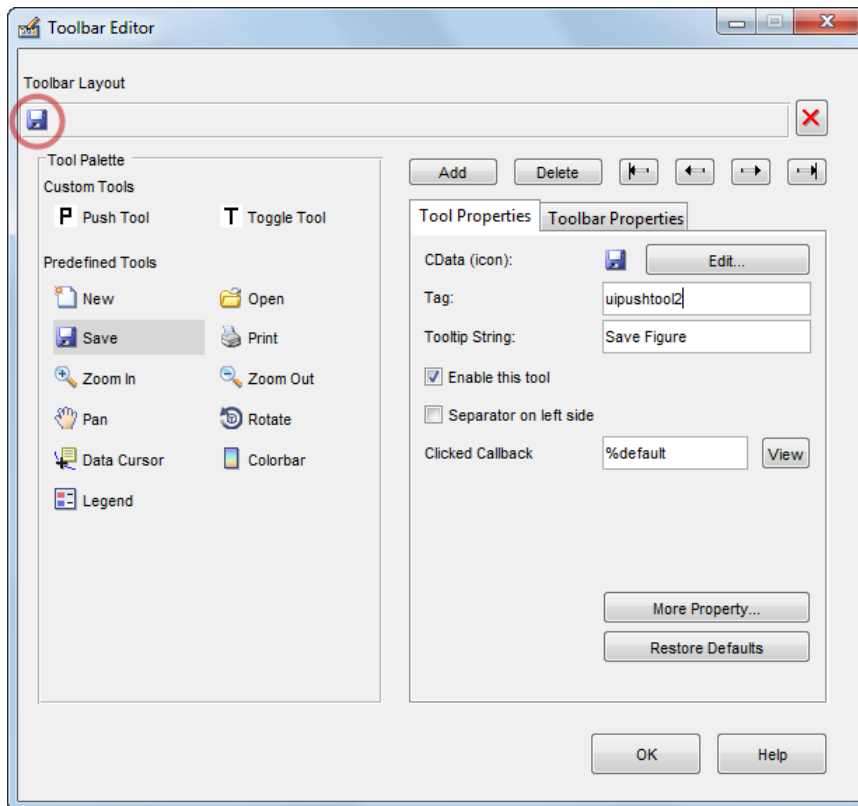
Predefined and Custom Tools

The Toolbar Editor provides two types of tools:

- Predefined tools, having standard icons and behaviors
- Custom tools, having generic icons and no behaviors

Predefined Tools

The set of icons on the bottom of the **Tool Palette** represent standard MATLAB figure tools. Their behavior is built in. Predefined tools that require an axes (such as pan and zoom) do not exhibit any behavior in UIs lacking axes. The callback(s) defining the behavior of the predefined tool are shown as %default, which calls the same function that the tool calls in standard figure toolbars and menus (to open files, save figures, change modes, etc.). You can change %default to some other callback to customize the tool; GUIDE warns you that you will modify the behavior of the tool when you change a callback field or click the **View** button next to it, and asks if you want to proceed or not.

Custom Tools

The two icons at the top of the Tool Palette create pushtools and toggletools. These have no built-in behavior except for managing their appearance when clicked on and off. Consequently, you need to provide your own callback(s) when you add one to your toolbar. In order for custom tools to respond to clicks, you need to edit their callbacks to create the behaviors you desire. Do this by clicking the **View** button next to the callback in the **Tool Properties** pane, and then editing the callback in the Editor window.

Add and Remove Separators

Separators are vertical bars that set off tools, enabling you to group them visually. You can add or remove a separator in any of three ways:

- Right-click on a tool's preview and select **Show Separator**, which toggles its separator on and off.
- Check or clear the check box **Separator** to the left in the tool's property pane.
- Change the Separator property of the tool from the Property Inspector

After adding a separator, that separator appears in the **Toolbar Layout** to the left of the tool. The separator is not a distinct object or icon; it is a property of the tool.

Move Tools

You can reorder tools on the toolbar in two ways:

- Drag a tool to a new position.
- Select a tool in the toolbar and click one of the arrow buttons below the right side of the toolbar.

If a tool has a separator to its left, the separator moves with the tool.

Remove Tools

You can remove tools from the toolbar in three ways:

- Select a tool and press the **Delete** key.
- Select a tool and click the **Delete** button on the UI.
- Right-click a tool and select **Delete** from the context menu.

You cannot undo any of these actions.

Edit a Tool's Properties

You edit the appearance and behavior of the currently selected tool using the **Tool Properties** pane, which includes controls for setting the most commonly used tool properties:

- CData — The tool's icon
- Tag — The internal name for the tool
- Enable — Whether users can click the tool
- Separator — A bar to the left of the icon for setting off and grouping tools
- Clicked Callback — The function called when users click the tool
- Off Callback (uitoggletool only) — The function called when the tool is put in the *off* state
- On Callback (uitoggletool only) — The function called when the tool is put in the *on* state

See "Write Callbacks in GUIDE" on page 7-2 for details on programming the tool callbacks. You can also access these and other properties of the selected tool with the Property Inspector. To open the Property Inspector, click the **More Properties** button on the **Tool Properties** pane.

Edit Tool Icons

To edit a selected toolbar icon, click the **Edit** button in the **Tool Properties** pane, next to **CData (icon)** or right-click the **Toolbar Layout** and select **Edit Icon** from the

context menu. The Icon Editor opens with the tool's `CData` loaded into it. For information about editing icons, see "Use the Icon Editor" on page 6-102.

Edit Toolbar Properties

If you click an empty part of the toolbar or click the **Toolbar Properties** tab, you can edit two of its properties:

- `Tag` — The internal name for the toolbar
- `Visible` — Whether the toolbar is displayed in your UI

The `Tag` property is initially set to `uitoolbar1`. The `Visible` property is set to `on`. When on, the `Visible` property causes the toolbar to be displayed on the UI regardless of the setting of the figure's `Toolbar` property. If you want to toggle a custom toolbar as you can built-in ones (from the **View** menu), you can create a menu item, a check box, or other control to control its Visible property.

To access nearly all the properties for the toolbar in the Property Inspector, click **More Properties**.

Test Your Toolbar

To try out your toolbar, click the **Run** button in the Layout Editor. The software asks if you want to save changes to its `.fig` file first.

Remove a Toolbar

You can remove a toolbar completely—destroying it—from the Toolbar Editor, leaving your UI without a toolbar (other than the figure toolbar, which is not visible by default). The are two ways to remove a toolbar:

- Click the **Remove** button ❌ on the right end of the toolbar.
- Right-click a blank area on the toolbar and select **Remove Toolbar** from the context menu.

If you remove all the individual tools in the ways shown in "Remove Tools" on page 6-100 without removing the toolbar itself, your UI will contain an empty toolbar.

Close the Toolbar Editor

You can close the Toolbar Editor window in two ways:

- Press the **OK** button.

• Click the Close box in the title bar.

When you close the Toolbar Editor, the current state of your toolbar is saved with the UI you are editing. You do not see the toolbar in the Layout Editor, but you can run your program to see it.

Editing Tool Icons

GUIDE includes its own Icon Editor, a dialog for creating and modifying icons such as icons on toolbars. You can access this editor only from the Toolbar Editor. This figure shows the Icon Editor loaded with a standard Save icon.
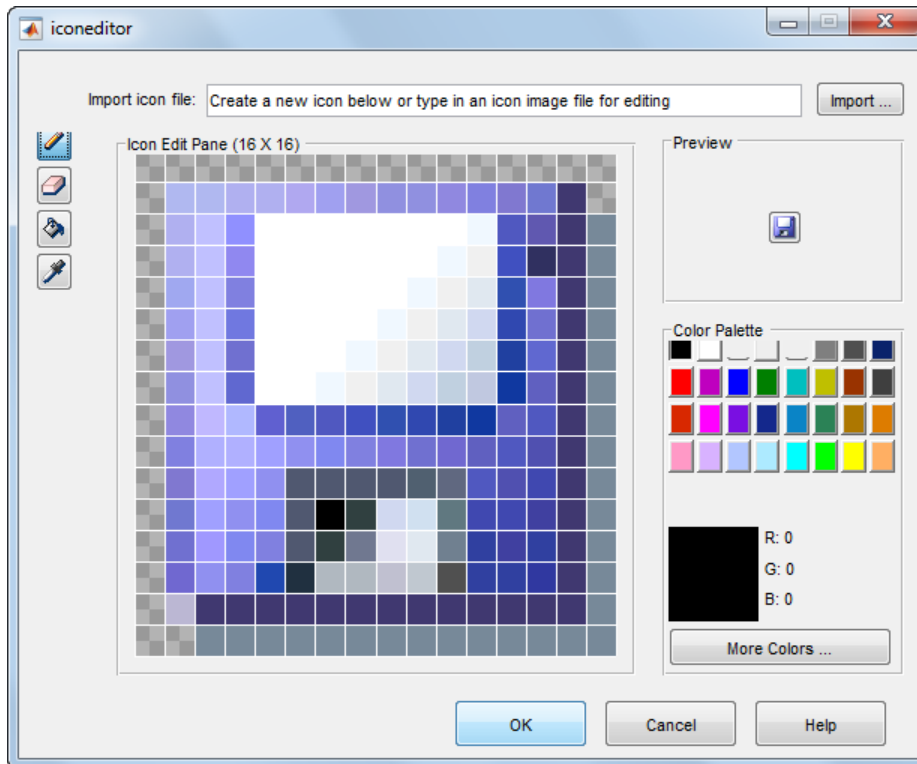
Use the Icon Editor

The Icon Editor dialog includes the following components:

- **Icon file name** — The icon image file to be loaded for editing
- **Import** button — Opens a file dialog to select an existing icon file for editing
- Drawing tools — A group of four tools on the left side for editing icons

  - Pencil tool — Color icon pixels by clicking or dragging
  - Eraser tool — Erase pixels to be transparent by clicking or dragging
  - Paint bucket tool — Flood regions of same-color pixels with the current color
  - Pick color tool — Click a pixel or color palette swatch to define the current color
- **Icon Edit** pane — A n-by-m grid where you color an icon
- **Preview** pane — A button with a preview of current state of the icon
- **Color Palette** — Swatches of color that the pencil and paint tools can use
- **More Colors** button — Opens the Colors dialog box for choosing and defining colors
- **OK** button — Dismisses the dialog and returns the icon in its current state
- **Cancel** button — Closes the dialog without returning the icon

To work with the Icon Editor,

1 Open the Icon Editor for a selected tool's icon.

2 Using the Pencil tool, color the squares in the grid:

  - Click a color cell in the palette.
  - That color appears in the **Color Palette** preview swatch.
  - Click in specific squares of the grid to transfer the selected color to those squares.
  - Hold down the left mouse button and drag the mouse over the grid to transfer the selected color to the squares that you touch.
  - Change a color by writing over it with another color.

3 Using the Eraser tool, erase the color in some squares

  - Click the Eraser button on the palette.
  - Click in specific squares to erase those squares.
  - Click and drag the mouse to erase the squares that you touch.
  - Click a another drawing tool to disable the Eraser.

4 Click **OK** to close the dialog and return the icon you created or click **Cancel** to close the dialog without modifying the selected tool's icon.

The Toolbar Editor and Icon Editor are shown together below.
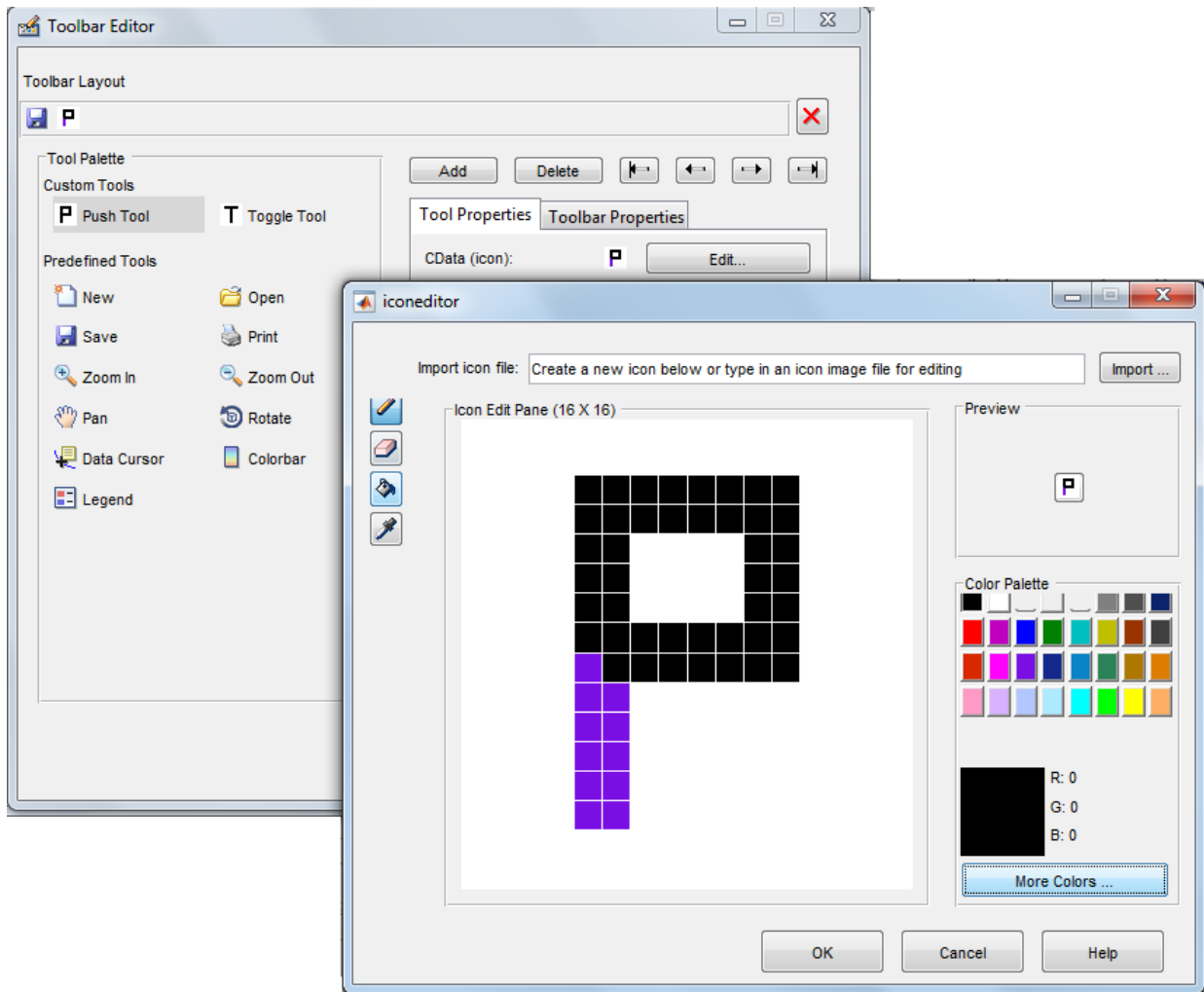
Related Examples

- "Callbacks for Specific Components" on page 7-12
- "Create Menus for GUIDE UIs" on page 6-77

More About

- "Write Callbacks in GUIDE" on page 7-2

Design Cross-Platform UIs in GUIDE

Default System Font

By default, user interface controls (uicontrols) use the default font for the platform on which they are running. For example, when displaying your UI on PCs, uicontrols use MS San Serif. When your program runs on a different platform, it uses that computer's default font. This provides a consistent look with respect to your UI and other applications.

If you have set the FontName property to a named font and want to return to the default value, you can set the property to the string `default`. This ensures that the software uses the system default at run-time.

You can use the Property Inspector to set this property:
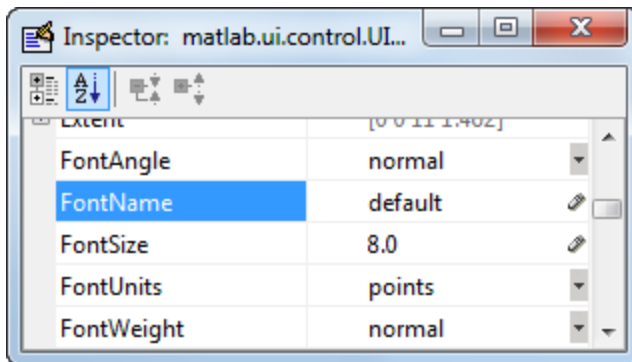
As an alternative, use the `set` command to set the property in the UI code file. For example, if there is a push button in your UI and its handle is stored in the `pushbutton1` field of the `handles` structure, then the statement

```
set(handles.pushbutton1,'FontName','default')
```

sets the `FontName` property to use the system default.

Specify a Fixed-Width Font

If you want to use a fixed-width font for a user interface control, set its `FontName` property to the string `fixedwidth`. This special identifier ensures that your UI uses the standard fixed-width font for the target platform.

You can find the name of the fixed-width font that is used on a given platform by querying the root `FixedWidthFontName` property.

```
get(groot,'FixedWidthFontName')
```

Use a Specific Font Name

You can specify an actual font name (such as Times or Courier) for the `FontName` property. However, doing so may cause your UI to not look as you intended when run on a different computer. If the target computer does not have the specified font, it will substitute another font that may not look good in your UI or may not be the standard font used for UIs on that system. Also, different versions of the same named font may have different size requirements for a given set of characters.

Standard Background Color

The default component background color is the standard system background color on which the UI is displaying. This color varies on different computer systems, e.g., the standard shade of gray on the PC differs from that on UNIX system, and may not match the default UI background color.

If you use the default component background color, you can use that same color as the background color for your UI. This provides a consistent look with respect to your UI and other applications. To do this in GUIDE, check **Options** > **Use system color scheme for background** on the Layout Editor **Tools** menu.

Note This option is available only if you first select the **Generate FIG-file and MATLAB File** option.

Cross-Platform Compatible Units

Cross-platform compatible UIs should look correct on computers having different screen sizes and resolutions. Since the size of a pixel can vary on different computer displays,

using the default figure `Units` of `pixels` does not produce a UI that looks the same on all platforms.

For this reason, GUIDE defaults the `Units` property for the figure to `characters`.

System-Dependent Units

Character units are defined by characters from the default system font. The width of a character unit equals the width of the letter `x` in the system font. The height of a character unit is the distance between the baselines of two lines of text. Note that character units are not square.

Units and Resize Behavior

If you set the resize behavior from the **GUI Options** dialog box, GUIDE automatically sets the units for the UI components in a way that maintains the intended look and feel across platforms. To specify the resize behavior option, select **Tools > GUI Options**, and select an item from the **Resize behavior** pop-up menu:

- If you choose **Non-resizable**, GUIDE defaults the component units to `characters`.
- If you choose **Proportional**, GUIDE defaults the component units to `normalized`.
- If you choose **Other (Use SizeChangedFcn)**, GUIDE defaults the component units to `characters`. However, you must provide a `SizeChangedFcn` callback to customize the resize behavior.

The **Non-resizable** and **Proportional** options enable your UI to automatically adjust the size and relative spacing of components when the UI displays on different computers.

Note GUIDE does not automatically adjust component units if you modify the figure's `Resize` property programmatically or in the Property Inspector.

At times, it might be convenient to use a more familiar unit of measure, e.g., inches or centimeters, when you are laying out the UI. However, to preserve the look of your UI on different computers, remember to change the figure `Units` property back to `characters`, and the components' `Units` properties to `characters` (nonresizable UIs) or `normalized` (resizable UIs) before you save the UI.

Programming a GUIDE UI

Write Callbacks in GUIDE

In this section...
"Callbacks for Different User Actions" on page 7-2
"GUIDE-Generated Callback Functions and Property Values" on page 7-4
"GUIDE Callback Syntax" on page 7-5
"Renaming and Removing GUIDE-Generated Callbacks" on page 7-6

Callbacks for Different User Actions

UI and graphics components have certain properties that you can associate with specific callback functions. Each of these properties corresponds to a specific user action. For example, a uicontrol has a property called `Callback`. You can set the value of this property to be a handle to a callback function, an anonymous function, or a string containing MATLAB commands. Setting this property makes your UI respond when the user interacts with the uicontrol. If the `Callback` property has no specified value, then nothing happens when the end user interacts with the uicontrol.

This table lists the callback properties that are available, the user actions that trigger the callback function, and the most common UI and graphics components that use them.

Callback Property	User Action	Components That Use This Property
`ButtonDownFcn`	End user presses a mouse button while the pointer is on the component or figure.	`axes`, `figure`, `uibuttongroup`, `uicontrol`, `uipanel`, `uitable`,
`Callback`	End user triggers the component. For example: selecting a menu item, moving a slider, or pressing a push button.	`uicontextmenu`, `uicontrol`, `uimenu`
`CellEditCallba`	End user edits a value in a table whose cells are editable.	`uitable`
`CellSelection`	End user selects cells in a table.	`uitable`
`ClickedCallba`	End user clicks the push tool or toggle tool with the left mouse button.	`uitoggletool`, `uipushtool`
`CloseRequestF`	The figure closes.	`figure`

Callback Property	User Action	Components That Use This Property
`CreateFcn`	Callback executes when MATLAB creates the object, but before it is displayed.	`axes`, `figure`, `uibuttongroup`, `uicontextmenu`, `uicontrol`, `uimenu`, `uipushtool`, `uipanel`, `uitable`, `uitoggletool`, `uitoolbar`
`DeleteFcn`	Callback executes just before MATLAB deletes the figure.	`axes`, `figure`, `uibuttongroup`, `uicontextmenu`, `uicontrol`, `uimenu`, `uipushtool`, `uipanel`, `uitable`, `uitoggletool`, `uitoolbar`
`KeyPressFcn`	End user presses a keyboard key while the pointer is on the object.	`figure`, `uicontrol`, `uipanel`, `uipushtool`, `uitable`, `uitoolbar`
`KeyReleaseFcn`	End user releases a keyboard key while the pointer is on the object.	`figure`, `uicontrol`, `uitable`
`OffCallback`	Executes when the `State` of a toggle tool changes to `'off'`.	`uitoggletool`
`OnCallback`	Executes when the `State` of a toggle tool changes to `'on'`.	`uitoggletool`
`SizeChangedFcn`	End user resizes a button group, figure, or panel whose `Resize` property is `'on'`.	`figure`, `uipanel`, `uibuttongroup`
`SelectionChan`	End user selects a different radio button or toggle button within a button group.	`uibuttongroup`
`WindowButtonD`	End user presses a mouse button while the pointer is in the figure window.	`figure`
`WindowButtonM`	End user moves the pointer within the figure window.	`figure`
`WindowButtonU`	End user releases a mouse button.	`figure`
`WindowKeyPres`	End user presses a key while the pointer is on the figure or any of its child objects.	`figure`

Callback Property	User Action	Components That Use This Property
`WindowKeyRele`	End user releases a key while the pointer is on the figure or any of its child objects.	`figure`
`WindowScrollW`	End user turns the mouse wheel while the pointer is on the figure.	`figure`

GUIDE-Generated Callback Functions and Property Values

How GUIDE Manages Callback Functions and Properties

After you add a `uicontrol`, `uimenu`, or `uicontextmenu` component to your UI, but before you save it, GUIDE populates the `Callback` property with the value, `%automatic`. This value indicates that GUIDE will generate a name for the callback function.

When you save your UI, GUIDE adds an empty callback function definition to your UI code file, and it sets the control's `Callback` property to be an anonymous function. This function definition is an example of a GUIDE-generated callback function for a push button.

```
function pushbutton1_Callback(hObject,eventdata,handles)
% hObject    handle to pushbutton1 (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)

end
```

If you save this UI with the name, `myui`, then GUIDE sets the push button's `Callback` property to the following value:

```
@(hObject,eventdata)myui('pushbutton1_Callback',hObject,eventdata,guidata(hObject))
```

This is an anonymous function that serves as a reference to the function, `pushbutton1_Callback`. This anonymous function has four input arguments. The first argument is a string containing the name of the callback function. The last three arguments are provided by Handle Graphics®, and are discussed in the section, "GUIDE Callback Syntax" on page 7-5.

Note: GUIDE does not automatically generate callback functions for other UI components, such as tables, panels, or button groups. If you want any of these components to execute a callback function, then you must create the callback by right-

---

clicking on the component in the layout, and selecting an item under **View Callbacks** in the context menu.

GUIDE Callback Syntax

All callbacks must accept at least three input arguments:

- `hObject` — Handle to the UI component that triggered the callback.
- `eventdata` — A variable that contains detailed information about specific mouse or keyboard actions.
- `handles` — A `struct` that contains handles to all the objects in the UI. GUIDE uses the `guidata` function to store and maintain this structure.

For the callback function to accept additional arguments, you must put the additional arguments at the end of the argument list in the function definition.

The eventdata Argument

The `eventdata` argument provides detailed information to certain callback functions. For example, if the end user triggers the `KeyPressFcn`, then MATLAB provides information regarding the specific key (or combination of keys) that the end user pressed. If `eventdata` is not available to the callback function, then MATLAB passes it as an empty array. The following table lists the callbacks and components that use `eventdata`.

Callback Property Name	Component

`WindowKeyPressFcn`
`WindowKeyReleaseFcn`
`WindowScrollWheel`
	`figure`

`KeyPressFcn`	`figure`, `uicontrol`, `uitable`
`KeyReleaseFcn`	`figure`, `uicontrol`, `uitable`
`SelectionChangedFcn`	`uibuttongroup`
`CellEditCallback` `CellSelectionCallback`	`uitable`

Renaming and Removing GUIDE-Generated Callbacks

Renaming Callbacks

GUIDE creates the name of a callback function by combining the component's `Tag` property and the callback property name. If you change the component's `Tag` value, then GUIDE changes the callback's name the next time you save the UI.

If you decide to change the `Tag` value after saving the UI, then GUIDE updates the following items (assuming that all components have unique `Tag` values).

- Component's callback function definition
- Component's callback property value
- References in the code file to the corresponding field in the `handles` structure

To rename a callback function without changing the component's `Tag` property:

1 Change the name in the callback function definition.

2 Update the component's callback property by changing the first argument passed to the anonymous function. For example, the original callback property for a push button might look like this:

```
@(hObject,eventdata)myui('pushbutton1_Callback',...
                        hObject,eventdata,guidata(hObject))
```

In this example, you must change the string, `'pushbutton1_Callback'` to the new function name.

3 Change all other references to the old function name to the new function name in the UI code file.

Deleting Callbacks

You can delete a callback function when you want to remove or change the function that executes when the end user performs a specific action. To delete a callback function:

1 Search and replace all instances that refer to the callback function in your code.

2 Open the UI in GUIDE and replace all instances that refer to the callback function in the Property Inspector.

3 Delete the callback function.

Related Examples

- "Callbacks for Specific Components" on page 7-12

More About

- "Anonymous Functions"
- "Share Data Among Callbacks" on page 11-2

Initialize UIs Created Using GUIDE

In this section...

Opening Function

The opening function is the first callback in every UI code file. It is executed just before the UI is made visible to the user, but after all the components have been created, i.e., after the components' `CreateFcn` callbacks, if any, have been run.

You can use the opening function to perform your initialization tasks before the user has access to the UI. For example, you can use it to create data or to read data from an external source. MATLAB passes any command-line arguments to the opening function.

Function Naming and Template

GUIDE names the opening function by appending `_OpeningFcn` to the name of the UI. This is an example of an opening function template as it might appear in the `myui` code file.

```
% --- Executes just before myui is made visible.
function myui_OpeningFcn(hObject, eventdata, handles, varargin)
% This function has no output args, see OutputFcn.
% hObject    handle to figure
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)
% varargin   command line arguments to myui (see VARARGIN)

% Choose default command line output for myui
handles.output = hObject;

% Update handles structure
guidata(hObject, handles);

% UIWAIT makes myui wait for user response (see UIRESUME)
% uiwait(handles.myui);
```

Input Arguments

The opening function has four input arguments `hObject`, `eventdata`, `handles`, and `varargin`. The first three are the same as described in "GUIDE Callback Syntax" on page 7-5. the last argument, `varargin`, enables you to pass arguments from the

command line to the opening function. The opening function can take actions with them (for example, setting property values) and also make the arguments available to callbacks by adding them to the `handles` structure.

For more information about using `varargin`, see the `varargin` reference page and "Support Variable Number of Inputs".

Passing Object Properties to an Opening Function

You can pass property name-value pairs as two successive command line arguments when you run your program. If you pass a name-value pair that corresponds to a figure property, MATLAB sets the property automatically. For example, `my_gui('Color', 'Blue')` sets the background color of the UI window to blue.

If you want your program to accept an input argument that is not a valid figure property, then your code must recognize and handle that argument. Otherwise, the argument is ignored. The following example is from the opening function for the Modal Question Dialog template, available from the GUIDE Quick Start dialog box. The added code opens the modal dialog with a message, specified from the command line or by another program that calls this one. For example, this command displays the text, `'Do you want to exit?'` on the window.

```
myui('String','Do you want to exit?')
```

To accept this name-value pair, you must customize the opening function because `'String'` is not a valid figure property. The Modal Question Dialog template file contains code to performs these tasks:

- Uses the `nargin` function to determine the number of user-specified arguments (which do not include `hObject`, `eventdata`, and `handles`)
- Parses `varargin` to obtain property name/value pairs, converting each name string to lower case
- Handles the case where the argument `'title'` is used as an alias for the figure `Name` property
- Handles the case `'string'`, assigning the following value as a `String` property to the appropriate static text object

```
function modalgui_OpeningFcn(hObject, eventdata, handles, varargin)
.
.
.
% Insert custom Title and Text if specified by the user
% Hint: when choosing keywords, be sure they are not easily confused
```

```
% with existing figure properties.  See the output of set(figure) for
% a list of figure properties.
if(nargin > 3)
    for index = 1:2:(nargin-3),
        if nargin-3==index, break, end
        switch lower(varargin{index})
         case 'title'
          set(hObject, 'Name', varargin{index+1});
         case 'string'
          set(handles.text1, 'String', varargin{index+1});
        end
    end
end
.
.
.
```

The if block loops through the odd elements of varargin checking for property names
or aliases, and the case blocks assign the following (even) varargin element as a value
to the appropriate property of the figure or one of its components. You can add more
cases to handle additional property assignments that you want the opening function to
perform.

Initial Template Code

Initially, the input function template contains these lines of code:

- handles.output = hObject adds a new element, output, to the handles
  structure and assigns it the value of the input argument hObject, which is the figure
  object.

- guidata(hObject,handles) saves the handles structure. You must use the
  guidata function to save any changes that you make to the handles structure. It is
  not sufficient just to set the value of a handles field.

- uiwait(handles.myui), initially commented out, blocks program execution until
  uiresume is called or the window is closed. Note that uiwait allows the user access
  to other MATLAB windows. Remove the comment symbol for this statement if you
  want the UI to be blocking when it opens.

Output Function

The output function returns, to the command line, outputs that are generated during its
execution. It is executed when the opening function returns control and before control
returns to the command line. This means that you must generate the outputs in the
opening function, or call uiwait in the opening function to pause its execution while
other callbacks generate outputs.

Function Naming and Template

GUIDE names the output function by appending `_OutputFcn` to the name of the UI. This is an example of an output function template as it might appear in the `myui` code file.

```
% --- Outputs from this function are returned to the command line.
function varargout = myui_OutputFcn(hObject, eventdata,...
                        handles)
% varargout  cell array for returning output args (see VARARGOUT);
% hObject    handle to figure
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)

% Get default command line output from handles structure
varargout{1} = handles.output;
```

Input Arguments

The output function has three input arguments: `hObject`, `eventdata`, and `handles`. They are the same as described in "GUIDE Callback Syntax" on page 7-5.

Output Arguments

The output function has one output argument, `varargout`, which it returns to the command line. By default, the output function assigns `handles.output` to `varargout`.

You can change the output by taking one of these actions:

- Change the value of `handles.output`. It can be any valid MATLAB value including a structure or cell array.
- Add output arguments to `varargout`. The `varargout` argument is a cell array. It can contain any number of output arguments. By default, GUIDE creates just one output argument, `handles.output`. To create an additional output argument, create a new field in the `handles` structure and add it to `varargout` using a command similar to

  ```
  varargout{2} = handles.second_output;
  ```

Related Examples

- "Create a Simple App Using GUIDE" on page 2-2

Callbacks for Specific Components

Coding the behavior of a UI component involves specific tasks that are unique to the type of component you are working with. This topic contains simple examples of callbacks for each type of component. The examples are written for GUIDE unless otherwise stated. For general information about coding callbacks, see "Write Callbacks in GUIDE" on page 7-2 or "Write Callbacks for Apps Created Programmatically" on page 10-5.

How to Use the Example Code

If you are working in GUIDE, then right-click on the component in your layout and select the appropriate callback property from the **View Callbacks** menu. Doing so creates an empty callback function that is automatically associated with the component. The specific function name that GUIDE creates is based on the component's Tag property, so your function name might be slightly different than the function name in the example code. Do not change the function name that GUIDE creates in your code. To use the example code in your UI, copy the code from the example's function body into your function's body.

If you are creating a UI programmatically, (without GUIDE), then you can adapt the example code into your code. To adapt an example into your code, omit the third input argument, `handles`, from the function definition. Also, replace any references to the `handles` array with the appropriate object handle. To associate the callback function with the component, set the component's callback property to be a handle to the callback function. For example, this command creates a push button component and sets the `Callback` property to be a handle to the function, `pushbutton1_callback`.

```
pb =
uicontrol('Style','pushbutton','Callback',@pushbutton1_Callback);
```

Push Button

This code is an example of a push button callback function in GUIDE. Associate this function with the push button `Callback` property to make it execute when the end user clicks on the push button.

```
function pushbutton1_Callback(hObject, eventdata, handles)
% hObject    handle to pushbutton1 (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)
display('Goodbye');
close(gcf);
```

The first line of code, `display('Goodbye')`, displays the string, `'Goodbye'`, in the Command Window. The next line gets a handle to the UI window using `gcf` and then closes it.

Toggle Button

This code is an example of an example of a toggle button callback function in GUIDE. Associate this function with the toggle button `Callback` property to make it execute when the end user clicks on the toggle button.

```
function togglebutton1_Callback(hObject,eventdata,handles)
% hObject    handle to togglebutton1 (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)

% Hint: get(hObject,'Value') returns toggle state of togglebutton1
button_state = get(hObject,'Value');
```

```
if button_state == get(hObject,'Max')
 display('down');
elseif button_state == get(hObject,'Min')
 display('up');
end
```

The toggle button's `Value` property matches the `Min` property when the toggle button is up. The `Value` changes to the `Max` value when the toggle button is depressed. This callback function gets the toggle button's `Value` property and then compares it with the `Max` and `Min` properties. If the button is depressed, then the function displays `'down'` in the Command Window. If the button is up, then the function displays `'up'`.

Radio Button

This code is an example of a radio button callback function in GUIDE. Associate this function with the radio button `Callback` property to make it execute when the end user clicks on the radio button.

```
function radiobutton1_Callback(hObject, eventdata, handles)
% hObject    handle to radiobutton1 (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)

% Hint: get(hObject,'Value') returns toggle state of radiobutton1

if (get(hObject,'Value') == get(hObject,'Max'))
 display('Selected');
else
 display('Not selected');
end
```

The radio button's `Value` property matches the `Min` property when the radio button is not selected. The `Value` changes to the `Max` value when the radio button is selected. This callback function gets the radio button's `Value` property and then compares it with the `Max` and `Min` properties. If the button is selected, then the function displays `'Selected'` in the Command Window. If the button is not selected, then the function displays `'Not selected'`.

Note Use a button group to manage exclusive selection behavior for radio buttons. See "Button Group" on page 7-22 for more information.

Check Box

This code is an example of a check box callback function in GUIDE. Associate this function with the check box `Callback` property to make it execute when the end user clicks on the check box.

```matlab
function checkbox1_Callback(hObject, eventdata, handles)
% hObject    handle to checkbox1 (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)

% Hint: get(hObject,'Value') returns toggle state of checkbox1

if (get(hObject,'Value') == get(hObject,'Max'))
 display('Selected');
else
 display('Not selected');
end
```

The check box's `Value` property matches the `Min` property when the check box is not selected. The `Value` changes to the `Max` value when the check box is selected. This callback function gets the check box's `Value` property and then compares it with the `Max` and `Min` properties. If the check box is selected, the function displays `'Selected'` in the Command Window. If the check box is not selected, it displays `'Not selected'`.

Edit Text Field

This code is an example of a callback for an edit text field in GUIDE. Associate this function with the uicontrol's `Callback` property to make it execute when the end user types inside the text field.

```matlab
function edit1_Callback(hObject, eventdata, handles)
% hObject    handle to edit1 (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)

% Hints: get(hObject,'String') returns contents of edit1 as text
%        str2double(get(hObject,'String')) returns contents as double
input = get(hObject,'String');
display(input);
```

When the end user types characters inside the text field and presses the **Enter** key, the callback function retrieves the string value and displays it in the Command Window.

To enable users to enter multiple lines of text, set the Max and Min properties to numeric values that satisfy Max - Min > 1. For example, set Max to 2, and Min to 0 to satisfy the inequality. In this case, the callback function triggers when the end user clicks on an area in the UI that is outside of the text field.

Retrieve Numeric Values

If you want to interpret the contents of an edit text field as numeric values, then convert the characters to numbers using the str2double function. The str2double function returns NaN for nonnumeric input.

This code is an example of an edit text field callback function that interprets the user's input as numeric values.

```
function edit1_Callback(hObject, eventdata, handles)
% hObject    handle to edit1 (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)

% Hints: get(hObject,'String') returns contents of edit1 as text
% str2double(get(hObject,'String')) returns contents as a double
input = str2double(get(hObject,'string'));
if isnan(input)
  errordlg('You must enter a numeric value','Invalid Input','modal')
  uicontrol(hObject)
  return
else
  display(input);
end
```

When the end user enters values into the edit text field and presses the **Enter** key, the callback function gets the value of the String property and converts it to a numeric value. Then, it checks to see if the value is NaN (nonnumeric). If the input is NaN, then the callback presents an error dialog box.

Slider

This code is an example of a slider callback function in GUIDE. Associate this function with the slider Callback property to make it execute when the end user moves the slider.

```
function slider1_Callback(hObject, eventdata, handles)
```

```
% hObject    handle to slider1 (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)

% Hints: get(hObject,'Value') returns position of slider
%        get(hObject,'Min') and get(hObject,'Max') to determine...
slider_value = get(hObject,'Value');
display(slider_value);
```

When the end user moves the slider, the callback function gets the current value of the slider and displays it in the Command Window. By default, the slider's range is [0, 1]. To modify the range, set the slider's `Max` and `Min` properties to the maximum and minimum values, respectively.

List Box

Populate Items in the List Box

If you are developing a UI using GUIDE, use the list box `CreateFcn` callback to add items to the list box.

This code is an example of a list box `CreateFcn` callback that populates the list box with the items, `Red`, `Green`, and `Blue`.

```
function listbox1_CreateFcn(hObject, eventdata, handles)
% hObject    handle to listbox1 (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    empty - handles not created until after all CreateFcns

% Hint: listbox controls usually have a white background on Windows.
if ispc && isequal(get(hObject,'BackgroundColor'), ...
    get(0,'defaultUicontrolBackgroundColor'))
    set(hObject,'BackgroundColor','white');
end
set(hObject,'String',{'Red';'Green';'Blue'});
```
The last line, `set(hObject,'String',{'Red';'Green';'Blue'})`, populates the contents of the list box.

If you are developing a UI programmatically (without GUIDE), then populate the list box when you create it. For example:

```
function myui()
```

```
figure
uicontrol('Style','Listbox',...
'String',{'Red';'Green';'Blue'},...
 'Position',[40 70 80 50]);
end
```

Change the Selected Item

When the end user selects a list box item, the list box's `Value` property changes to a number that corresponds to the item's position in the list. For example, a value of `1` corresponds to the first item in the list. If you want to change the selection in your UI code, then change the `Value` property to another number between `1` and the number of items in the list.

For example, you can use the `handles` structure in GUIDE to access the list box and change the `Value` property:

```
set(handles.listbox1,'Value',2)
```

The first argument, `handles.listbox1`, might be different in your code, depending on the value of the list box `Tag` property.

Write the Callback Function

This code is an example of a list box callback function in GUIDE. Associate this function with the list box `Callback` property to make it execute when a selects an item in the list box.

```
function listbox1_Callback(hObject, eventdata, handles)
% hObject    handle to listbox1 (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)
% Hints: contents = cellstr(get(hObject,'String')) returns contents
% contents{get(hObject,'Value')} returns selected item from listbox1
items = get(hObject,'String');
index_selected = get(hObject,'Value');
item_selected = items{index_selected};
display(item_selected);
```

When the end user selects an item in the list box, the callback function performs the following tasks:

- Gets all the items in the list box and stores them in the variable, `items`.

- Gets the numeric index of the selected item and stores it in the variable, `index_selected`.
- Gets the string value of the selected item and stores it in the variable, `item_selected`.
- Displays the selected item in the MATLAB Command Window.

The example, "Interactive List Box in a GUIDE UI" on page 8-33 shows how to populate a list box with directory names.

Pop-Up Menu

Populate Items in the Pop-Up Menu

If you are developing a UI using GUIDE, use the pop-up menu `CreateFcn` callback to add items to the pop-up menu.

This code is an example of a pop-up menu `CreateFcn` callback that populates the menu with the items, `Red`, `Green`, and `Blue`.

```matlab
function popupmenu1_CreateFcn(hObject, eventdata, handles)
% hObject    handle to popupmenu1 (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    empty - handles not created until after all CreateFcns

% Hint: popupmenu controls usually have a white background on Windows.
if ispc && isequal(get(hObject,'BackgroundColor'),...
    get(0,'defaultUicontrolBackgroundColor'))
    set(hObject,'BackgroundColor','white');
end
set(hObject,'String',{'Red';'Green';'Blue'});
```

The last line, `set(hObject,'String',{'Red';'Green';'Blue'})`, populates the contents of the pop-up menu.

If you are developing a UI programmatically (without GUIDE), then populate the pop-up menu when you create it. For example:

```matlab
function myui()
 figure
 uicontrol('Style','popupmenu',...
 'String',{'Red';'Green';'Blue'},...
 'Position',[40 70 80 20]);
end
```

Change the Selected Item

When the end user selects an item, the pop-up menu's `Value` property changes to a number that corresponds to the item's position in the menu. For example, a value of `1` corresponds to the first item in the list. If you want to change the selection in your UI code, then change the `Value` property to another number between `1` and the number of items in the menu.

For example, you can use the `handles` structure in GUIDE to access the pop-up menu and change the `Value` property:

```
set(handles.popupmenu1,'Value',2)
```

The first argument, `handles.popupmenu1`, might be different in your code, depending on the value of the pop-up menu `Tag` property.

Write the Callback Function

This code is an example of a pop-up menu callback function in GUIDE. Associate this function with the pop-up menu `Callback` property to make it execute when the end user selects an item from the menu.

```
function popupmenu1_Callback(hObject, eventdata, handles)
% hObject    handle to popupmenu1 (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)

% Hints: contents = cellstr(get(hObject,'String')) returns contents...
%        contents{get(hObject,'Value')} returns selected item...
items = get(hObject,'String');
index_selected = get(hObject,'Value');
item_selected = items{index_selected};
display(item_selected);
```

When the end user selects an item in the pop-up menu, the callback function performs the following tasks:

- Gets all the items in the pop-up menu and stores them in the variable, `items`.
- Gets the numeric index of the selected item and stores it in the variable, `index_selected`.
- Gets the string value of the selected item and stores it in the variable, `item_selected`.
- Displays the selected item in the MATLAB Command Window.

Panel

Make the Panel Respond to Button Clicks

You can create a callback function that executes when the end user right-clicks or left-clicks on the panel. If you are working in GUIDE, then right-click the panel in the layout and select **View Callbacks** > **ButtonDownFcn** to create the callback function.

This code is an example of a `ButtonDownFcn` callback in GUIDE.

```
function uipanel1_ButtonDownFcn(hObject, eventdata, handles)
% hObject    handle to uipanel1 (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)
display('Mouse button was pressed');
```

When the end user clicks on the panel, this function displays the text, `'Mouse button was pressed'`, in the Command Window.

Resize the Window and Panel

By default, GUIDE UIs cannot be resized, but you can override this behavior by selecting **Tools** > **GUI Options** and setting **Resize behavior** to **Proportional**.

Programmatic UIs can be resized by default, and you can change this behavior by setting the `Resize` property of the figure on or off.

When your UI is resizable, the position of components in the window adjust as the user resizes it. If you have a panel in your UI, then the panel's size will change with the window's size. Use the panel's `SizeChangedFcn` callback to make your UI perform specific tasks when the panel resizes.

This code is an example of a panel's `SizeChangedFcn` callback in a GUIDE UI. When the end user resizes the window, this function modifies the font size of static text inside the panel.

```
function uipanel1_SizeChangedFcn(hObject, eventdata, handles)
% hObject    handle to uipanel1 (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)
set(hObject,'Units','Points')
panelSizePts = get(hObject,'Position');
panelHeight = panelSizePts(4);
```

```
set(hObject,'Units','normalized');
newFontSize = 10 * panelHeight / 115;
texth = findobj('Tag','text1');
set(texth,'FontSize',newFontSize);
```

If your UI has nested panels, then they will resize from the inside-out (in child-to-parent order).

Note: To make the text inside a panel resize automatically, set the `fontUnits` property to `'normalized'`.

Button Group

Button groups are similar to panels, but they also manage exclusive selection of radio buttons and toggle buttons. When a button group contains multiple radio buttons or toggle buttons, the button group allows the end user to select only one of them.

Do not code callbacks for the individual buttons that are inside a button group. Instead, use the button group's `SelectionChangedFcn` callback to respond when the end user selects a button.

This code is an example of a button group `SelectionChangedFcn` callback that manages two radio buttons and two toggle buttons.

```
function uibuttongroup1_SelectionChangedFcn(hObject, eventdata, handles)
% hObject     handle to the selected object in uibuttongroup1
% eventdata   structure with the following fields
% EventName: string 'SelectionChanged' (read only)
% OldValue: handle of the previously selected object or empty
% NewValue: handle of the currently selected object
% handles     structure with handles and user data (see GUIDATA)
switch get(eventdata.NewValue,'Tag') % Get Tag of selected object.
    case 'radiobutton1'
        display('Radio button 1');
    case 'radiobutton2'
        display('Radio button 2');
    case 'togglebutton1'
        display('Toggle button 1');
    case 'togglebutton2'
        display('Toggle button 2');
end
```

When the end user selects a radio button or toggle button in the button group, this function determines which button the user selected based on the button's `Tag` property. Then, it executes the code inside the appropriate `case`.

Note: The button group's `SelectedObject` property contains a handle to the button that user selected. You can use this property elsewhere in your UI code to determine which button the user selected.

Menu Item

The code in this section contains example callback functions that respond when the end user selects **Edit** > **Copy** > **To File** in this menu.
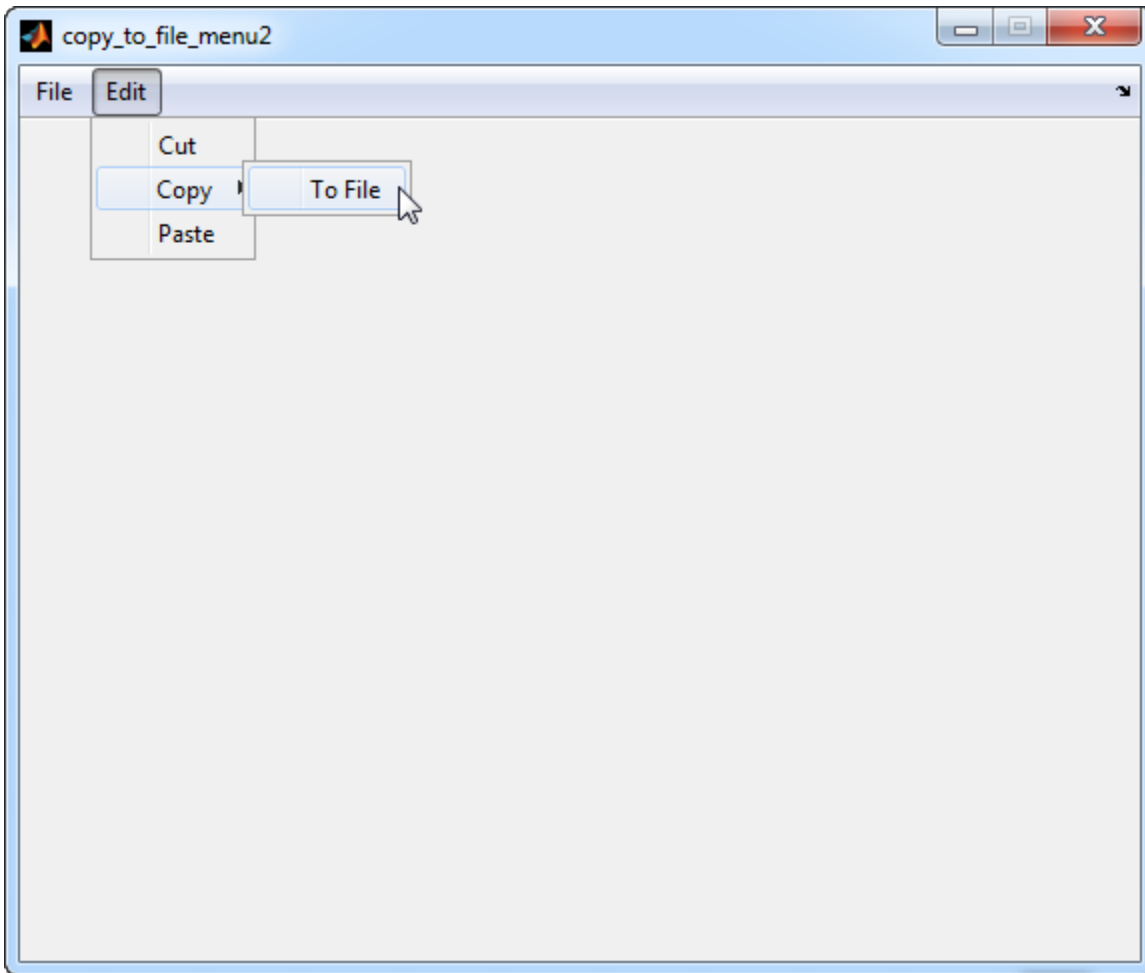
```matlab
% --------------------------------------------------------------------
function edit_menu_Callback(hObject, eventdata, handles)
% hObject    handle to edit_menu (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)
display('Edit menu selected');

% --------------------------------------------------------------------
function copy_menu_item_Callback(hObject, eventdata, handles)
```

```
% hObject    handle to copy_menu_item (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)
display('Copy menu item selected');

% --------------------------------------------------------------------
function tofile_menu_item_Callback(hObject, eventdata, handles)
% hObject    handle to tofile_menu_item (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)
[filename,path] = uiputfile('myfile.m','Save file name');
```

The function names might be different in your UI, depending on the tag names you specify in the GUIDE Menu Editor.

The callback functions trigger in response to these actions:

- When the end user selects the **Edit** menu, the `edit_menu_Callback` function displays the text, `'Edit menu selected'`, in the MATLAB Command Window.
- When the end user hovers the mouse over the **Copy** menu item, the `copy_menu_item_Callback` function displays the text, `'Copy menu item selected'`, in the MATLAB Command Window.
- When the end user clicks and releases the mouse button on the **To File** menu item, the `tofile_menu_item_Callback` function displays a dialog box that prompts the end user to select a destination folder and file name.

The `tofile_menu_item_Callback` function calls the `uiputfile` function to prompt the end user to supply a destination file and folder. If you want to create a menu item that prompts the user for an existing file, for example, if your UI has an **Open File** menu item, then use the `uigetfile` function.

When you create a cascading menu like this one, the intermediate menu items trigger when the mouse hovers over them. The final, terminating, menu item triggers when the mouse button releases over the menu item.

How to Update a Menu Item Check

You can add a check mark next to a menu item to indicate that an option is enabled. In GUIDE, you can select **Check mark this item** in the Menu Editor to make the menu item checked by default. Each time the end user selects the menu item, the callback function can turn the check on or off.

This code shows how to change the check mark next to a menu item.

```matlab
if strcmp(get(hObject,'Checked'),'on')
    set(hObject,'Checked','off');
else
    set(hObject,'Checked','on');
end
```

The `strcmp` function compares two strings and returns `true` when they match. In this case, it returns `true` when the menu item's `Checked` property matches the string, `'on'`.

See "Create Menus for GUIDE UIs" on page 6-77 for more information about creating menu items in GUIDE. See "Create Menus for Programmatic UIs" on page 9-41 for more information about creating menu items programmatically.

Table

This code is an example of the table callback function, `CellSelectionCallback`. Associate this function with the table `CellSelectionCallback` property to make it execute when the end user selects cells in the table.

```matlab
function uitable1_CellSelectionCallback(hObject, eventdata, handles)
% hObject    handle to uitable1 (see GCBO)
% eventdata  structure with the following fields
%   Indices: row and column indices of the cell(s) currently selected
% handles    structure with handles and user data (see GUIDATA)
data = get(hObject,'Data');
indices = eventdata.Indices;
r = indices(:,1);
c = indices(:,2);
linear_index = sub2ind(size(data),r,c);
selected_vals = data(linear_index);
selection_sum = sum(sum(selected_vals))
```

When the end user selects cells in the table, this function performs the following tasks:

- Gets all the values in the table and stores them in the variable, `data`.
- Gets the indices of the selected cells. These indices correspond to the rows and columns in `data`.
- Converts the row and column indices into linear indices. The linear indices allow you to select multiple elements in an array using one command.
- Gets the values that the end user selected and stores them in the variable, `selected_vals`.

- Sums all the selected values and displays the result in the Command Window.

This code is an example of the table callback function, `CellEditCallback`. Associate this function with the table `CellEditCallback` property to make it execute when the end user edits a cell in the table.

```
function uitable1_CellEditCallback(hObject, eventdata, handles)
% hObject     handle to uitable1 (see GCBO)
% eventdata   structure with the following fields
% Indices: row and column indices of the cell(s) edited
% PreviousData: previous data for the cell(s) edited
% EditData: string(s) entered by the user
% NewData: EditData or its converted form set on the Data property.
% Empty if Data was not changed
% Error: error string when failed to convert EditData
data = get(hObject,'Data');
data_sum = sum(sum(data))
```

When the end user finishes editing a table cell, this function gets all the values in the table and calculates the sum of all the table values. The `ColumnEditable` property must be set to `true` in at least one column to allow the end user to edit cells in the table. For more information about creating tables and modifying their properties in GUIDE, see "Add Components to the GUIDE Layout Area" on page 6-13.

Axes

The code in this section is an example of an axes `ButtonDownFcn` that triggers when the end user clicks on the axes.
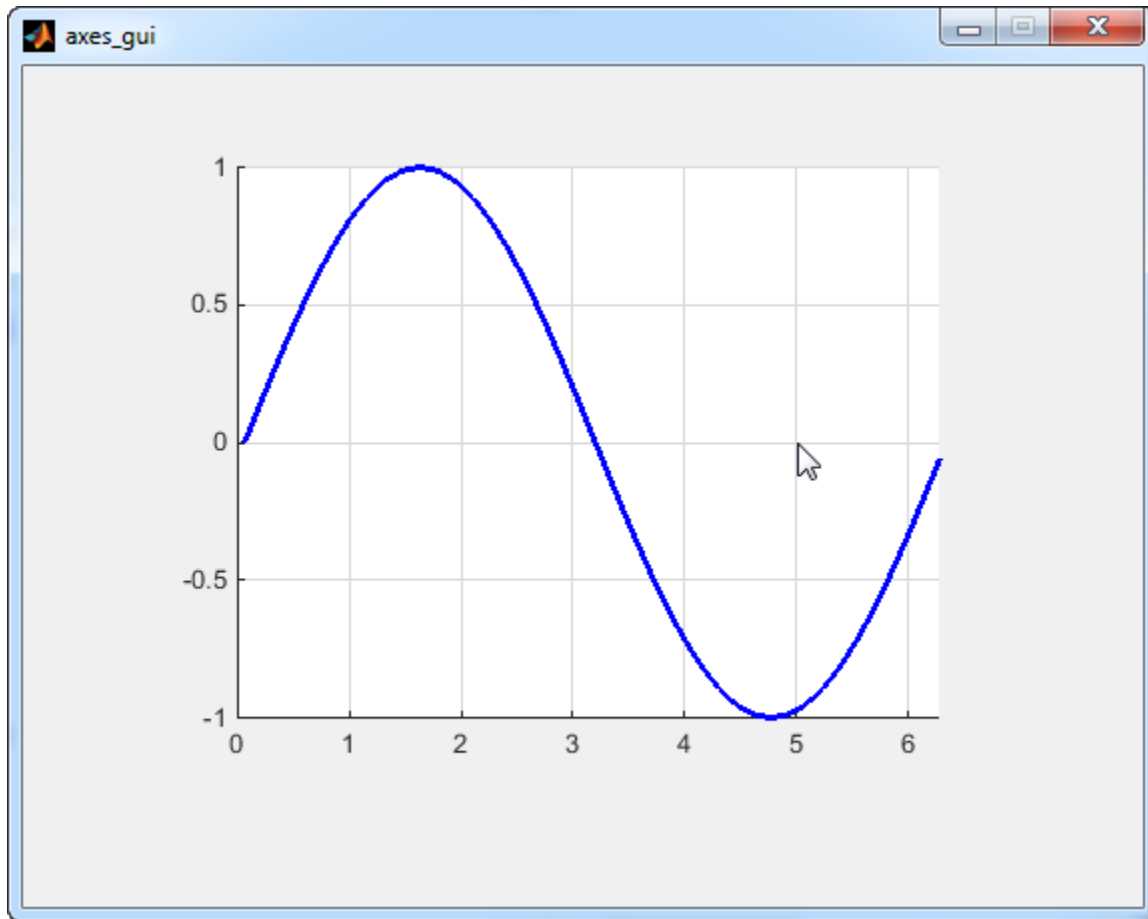
```matlab
function axes1_ButtonDownFcn(hObject, eventdata, handles)
% hObject    handle to axes1 (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)
pt = get(hObject,'CurrentPoint')
```

The coordinates of the pointer display in the MATLAB Command Window when the end user clicks on the axes (but not when that user clicks on another graphics object parented to the axes).

> **Note:** Most MATLAB plotting functions clear the axes and reset a number of axes properties, including the `ButtonDownFcn`, before plotting data. To create an interface that lets the end user plot data interactively, consider providing a component such as a push button to control plotting. Such components' properties are unaffected by the plotting functions. If you must use the axes `ButtonDownFcn` to plot data, then use functions such as `line`, `patch`, and `surface`.

More About

- "Write Callbacks in GUIDE" on page 7-2
- "Write Callbacks for Apps Created Programmatically" on page 10-5

Examples of GUIDE UIs

The following are examples that are packaged with MATLAB. The introductory text for most examples provides instructions on copying them to a writable folder on your system, so you can follow along.

- "Modal Dialog Box in GUIDE" on page 8-2
- "UI That Accepts Parameters and Generates Plots" on page 8-7
- "Synchronized Data Presentations in a GUIDE UI" on page 8-17
- "Interactive List Box in a GUIDE UI" on page 8-33
- "Plot Workspace Variables in a GUIDE UI" on page 8-39
- "Automatically Refresh Plot in a GUIDE UI" on page 8-44

8

Examples of GUIDE UIs

Modal Dialog Box in GUIDE

In this section...

"Create the Dialog Box" on page 8-2

"Create the Program That Opens the Dialog Box" on page 8-3

"Run the Program" on page 8-5

This example shows how to create a program that opens a modal dialog box when the user clicks a button. The dialog box contains two buttons, and the user must choose one of them. The program responds according to the user's selection in the dialog box.

Create the Dialog Box

1   On the **Home** tab, in the **Environment** section, click **Preferences** > **GUIDE** > **Show names in component palette**.

2   In the Command Window, type `guide`.

3   In the GUIDE Quick Start dialog box, select **Modal Question Dialog**. Then, click **OK**.

4   Right-click the text, "Do you want to create a question dialog?"

   Then, select **Property Inspector** from the context menu.

5   In the Property Inspector, select the `String` property. Then, change the existing value to: `Are you sure you want to close?`

   Then press **Enter**.
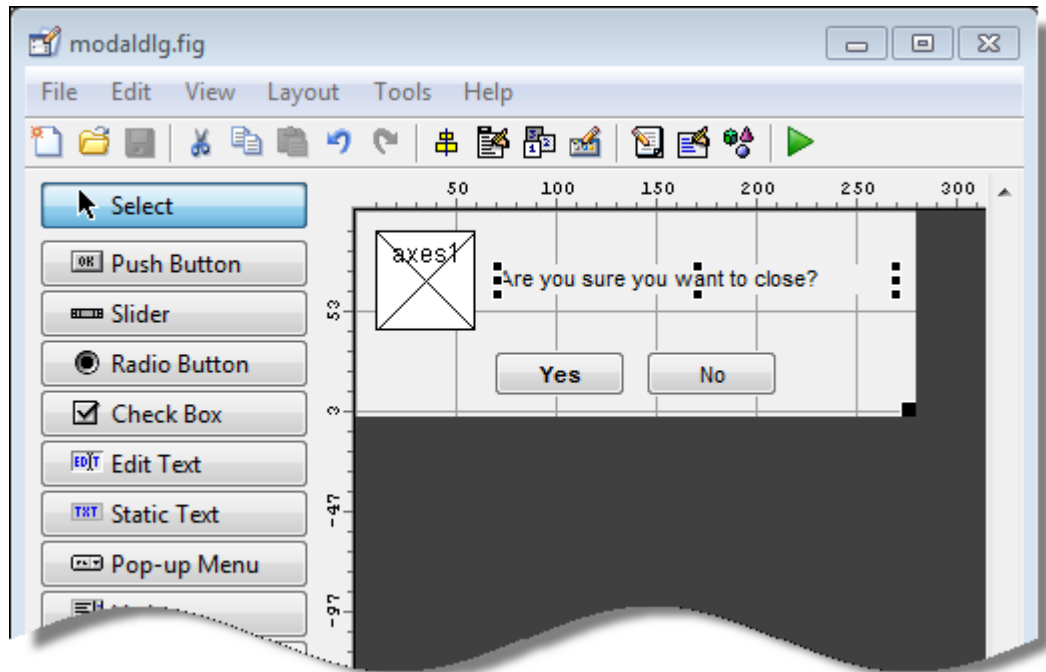
6   Select **File** > **Save As**.

7   In the Save As dialog box, in the **File name** field, type `modaldlg.fig`.

Create the Program That Opens the Dialog Box

Create a separate UI containing a **Close** button:

1   While still in GUIDE, select **File** > **New**.

2   In the GUIDE Quick Start dialog box, select **Blank GUI (Default)**. Then, click **OK**.

3   From the component palette on the left, drag a push button into the layout area.

4   Right-click the push button and select **Property Inspector**.

5   In the Property Inspector, select the `String` property. Then, change the existing value to `Close`. Then press **Enter**.
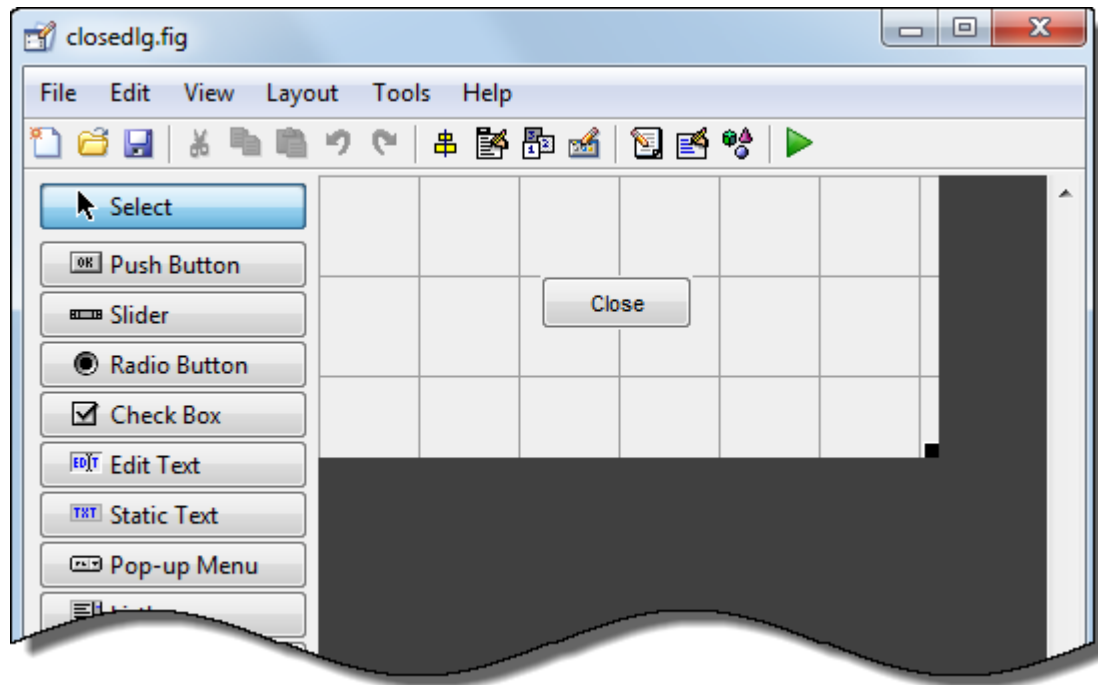
6. From the **File** menu, select **Save**.

7. In the Save dialog box, in the **File name** field, type `closedlg.fig`. Then, click **Save**.

   The code file, `closedlg.m`, opens in the Editor.

   On the **Editor** tab, in the **Navigate** section, click **Go To**, and then select `pushbutton1_Callback`.

   Then, locate the following generated code in the Editor:

   ```
   % --- Executes on button press in pushbutton1.
   function pushbutton1_Callback(hObject, eventdata, handles)
   % hObject    handle to close_pushbutton (see GCBO)
   % eventdata  reserved - to be defined in a future version of MATLAB
   % handles    structure with handles and user data (see GUIDATA)
   ```

8. Add the following code immediately after the comment that begins with `%` `handles...`.

```
% Get the current position from the handles structure
% to pass to the modal dialog.
pos_size = get(handles.figure1,'Position');

% Call modaldlg with the argument 'Position'.
user_response = modaldlg('Title','Confirm Close');
switch user_response
case 'No'
 % take no action
case 'Yes'
 % Prepare to close application window
 delete(handles.figure1)
end
```

When the user clicks the **Close** button in the closedlg window, the
pushbutton1_Callback function executes this command:

```
user_response = modaldlg('Title','Confirm Close');
```
Recall that the modaldlg function is coded in the other program file, modaldlg.m.
That function displays a second window: the Confirm Close dialog box. The return
argument, user_response, is the user's selection from that dialog box.

The switch command decides whether to close the first window (modaldlg) based on
the user's selection.

9   Save your code by pressing **Save** in the Editor Toolstrip.

Run the Program

1   In the Command Window, execute the command, closedlg.

2   MATLAB displays the closedlg window. Click the **Close** push button to execute
pushbutton1_Callback (in closedlg.m). That function calls modaldlg to display
the Confirm Close dialog box.
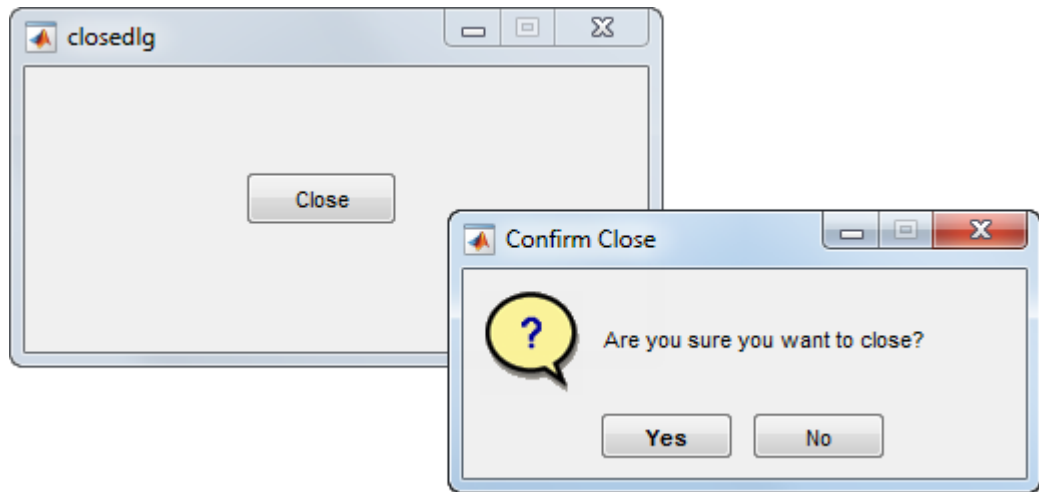
3 Click one of the buttons in the Confirm Close dialog box. When you click one of the buttons, `modaldlg.m` closes the Confirm Close dialog box and returns your selection to the calling function (`pushbutton1_Callback`). Then, the `switch` command in that function decides whether to close the remaining open window.

Related Examples

- "Create a Simple App Using GUIDE" on page 2-2

More About

- "Ways to Build Apps" on page 1-2
- "Dialog Boxes"
- "Write Callbacks in GUIDE" on page 7-2

UI That Accepts Parameters and Generates Plots

In this section...

About the Example

This example shows how to create a GUIDE UI that accepts input parameters and plots data in two axes. The parameters define a time-varying and frequency-varying signal. One plot displays the data in the time domain, and the other plot displays the data in the frequency domain.
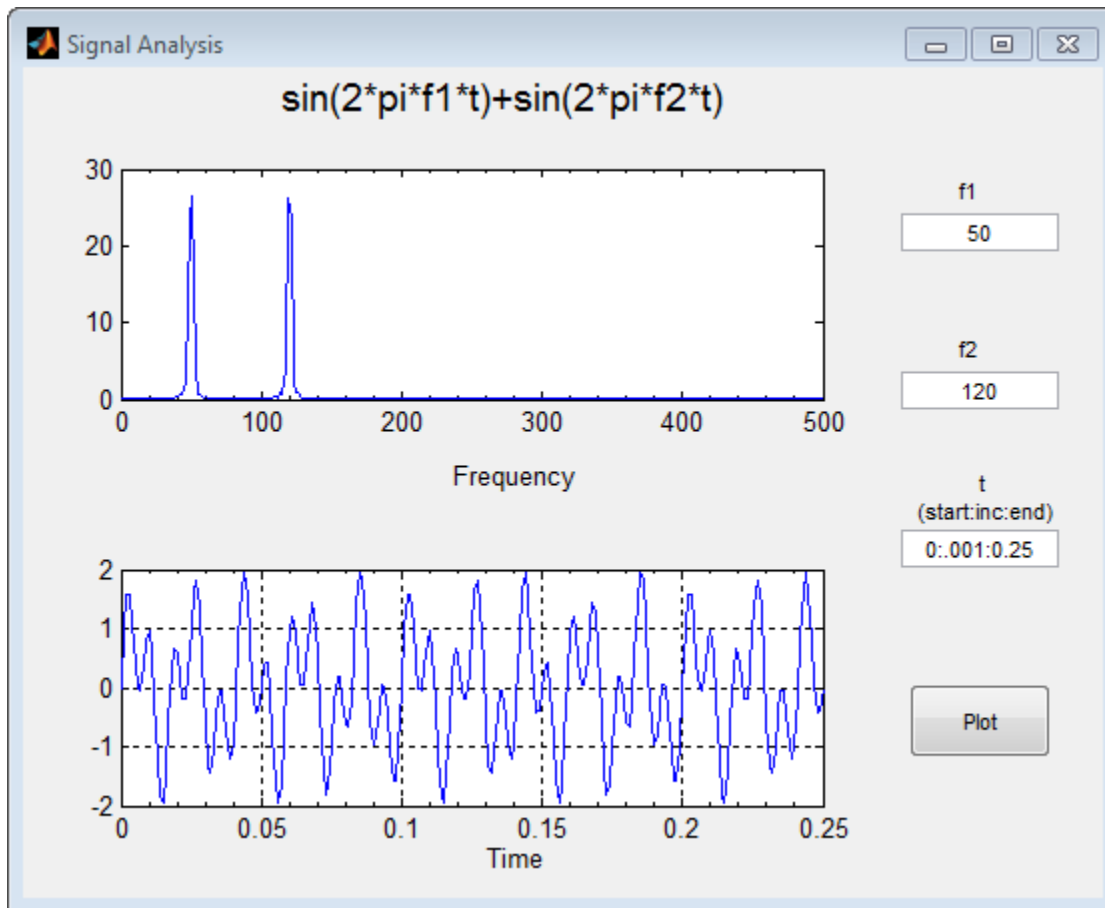
To get and view the example code:

1 Copy the example FIG-file and code file to your current (writeable) folder and open the FIG-file in GUIDE:

```
copyfile(fullfile(docroot, 'techdoc','creating_guis',...
'examples','two_axes*.*')), fileattrib('two_axes*.*', '+w')
guide two_axes.fig
```

2 From the GUIDE Layout Editor, click the Editor button 📄.

The two_axes.m code displays in the MATLAB Editor.

If you run the `two_axes` program and click the **Plot** button, the UI appears as shown in the preceding figure. The code evaluates the expression displayed at the top of the UI using parameters that you enter in the **f1**, **f2**, and **t** fields. The upper line graph displays a Fourier transform of the computed signal displayed in the lower line graph.

UI Design

This UI plots two graphs depicting three input values:

- Frequency one (`f1`)
- Frequency two (`f2`)
- A time vector (`t`)

When you click the **Plot** button, the program puts these values into a MATLAB expression that is the sum of two sine functions:

```
x = sin(2*pi*f1*t) + sin(2*pi*f2*t)
```

Then, the program calculates the FFT (fast Fourier transform) of `x` and plots the data in the frequency domain and the time domain in separate axes.

Default Values for Inputs

The program provides default values for the three inputs. This enables you to click the **Plot** button and see a result as soon as you run the program. The defaults indicate typical values.

The default values are created by setting the `String` property of the edit text. The following figure shows how the value was set for the time vector.
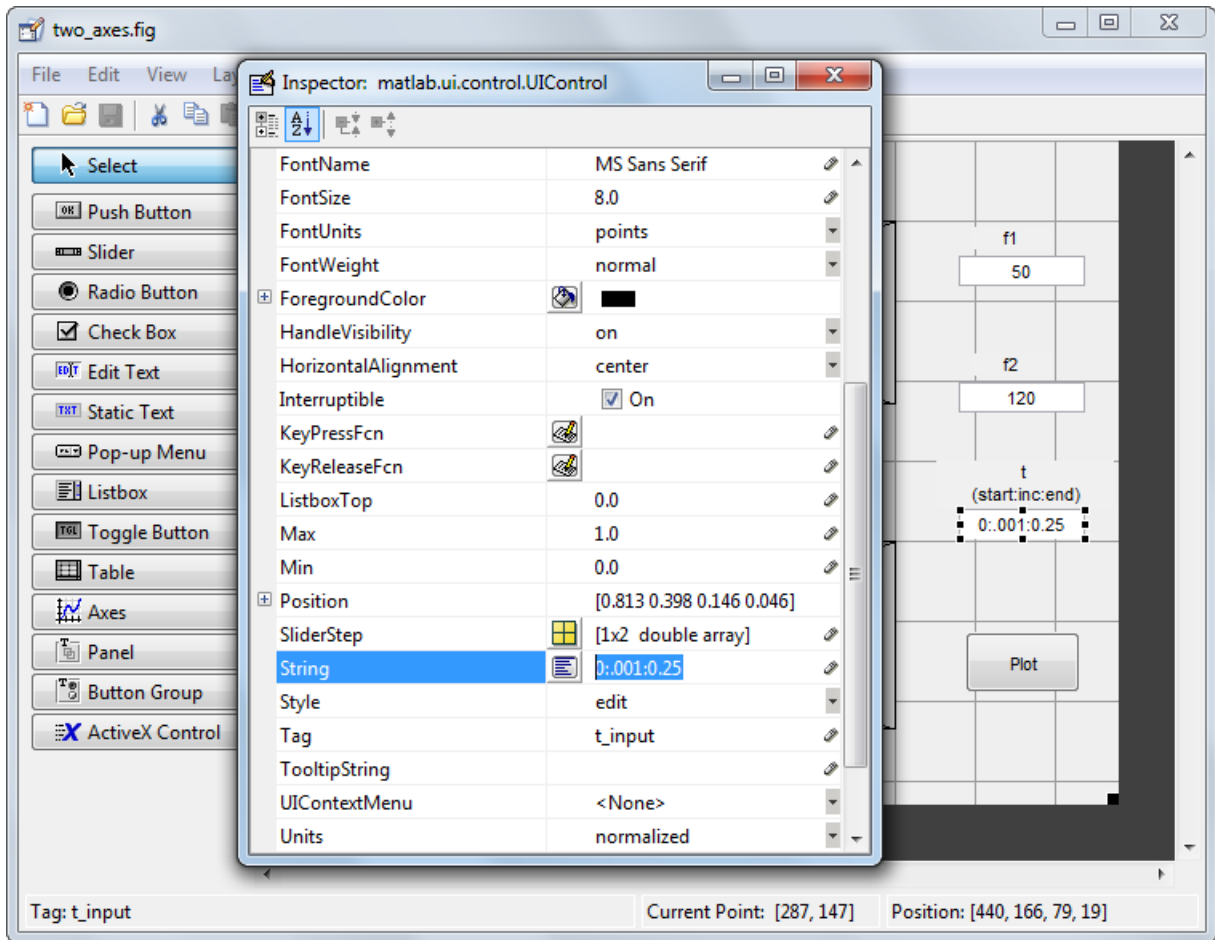
Identify the Axes

Since there are two axes in this UI, you must specify which one you want to target when plotting data. Use the `handles` structure to access the target axes in your code. All fields in the `handles` structure are named according to each object's `Tag` property value. In this case, `handles.frequency_axes` returns the top axes, and the `handles.time_axes` returns the bottom axes.

Validate Input as Numbers

When you UI displays, you can type parameters into three edit text fields as strings of text. If you type an invalid value, the graphs can fail to inform or even to generate. Preventing bad inputs from being processed is an important function of almost any UI that performs computations. This UI code validates these inputs:

- Ensure all three inputs are positive or negative real numbers
- Ensures that the `t` (time) input is a vector that increases monotonically and is not too long to display

Validate all three inputs are positive or negative real numbers

In this example, each edit text control callback validates its input. If the input fails validation, the callback disables the **Plot** button, changes its `String` to indicate the type of problem encountered, and restores focus to the edit text control, highlighting the erroneous input. When you enter a valid value, the **Plot** button reenables with its `String` set back to `'Plot'`. This approach prevents plotting errors and avoids the need for an error dialog box.

The `str2double` function validates most cases, returning `NaN` (Not a Number) for nonnumeric or nonscalar string expressions. An additional test using the `isreal` function makes sure that a text edit field does not contain a complex number, such as `'4+2i'`. The `f1_input_Callback` contains the following code to validate input for `f1` :

```
function f1_input_Callback(hObject, eventdata, handles)
% Validate that the text in the f1 field converts to a real number
f1 = str2double(get(hObject,'String'));
if isnan(f1) || ~isreal(f1)
    % isdouble returns NaN for non-numbers and f1 cannot be complex
    % Disable the Plot button and change its string to say why
    set(handles.plot_button,'String','Cannot plot f1')
    set(handles.plot_button,'Enable','off')
    % Give the edit text box focus so user can correct the error
    uicontrol(hObject)
else
    % Enable the Plot button with its original name
    set(handles.plot_button,'String','Plot')
    set(handles.plot_button,'Enable','on')
end
```

Similarly, `f2_input_Callback` code validates the `f2` input.

Validate the Time Input Vector

The time vector input, `t`, is more complicated to validate. As the `str2double` function does not operate on vectors, the `eval` function is called to convert the input string into

a MATLAB expression. Because you can type many things that `eval` cannot handle, the first task is to make sure that `eval` succeeded. The `t_input_Callback` uses `try, catch` blocks to do the following:

- Call `eval` with the `t_input` string inside the `try` block.
- If `eval` succeeds, perform additional tests within the `try` block.
- If `eval` generates an error, pass control to the `catch` block.
- In that block, the callback disables the **Plot** button and changes its `String` to `'Cannot plot t'`.

The remaining code in the `try` block makes sure that the variable `t` returned from `eval` is a monotonically increasing vector of numbers with no more than 1000 elements. If `t` passes all these tests, the callback enables **Plot** button and sets its `String` to `'Plot'`. If it fails any of the tests, the callback disables the **Plot** button and changes its `String` to an appropriate short message. Here are the `try` and `catch` blocks from the callback:

```
function t_input_Callback(hObject, eventdata, handles)
% Disable the Plot button ... until proven innocent
set(handles.plot_button,'Enable','off')
try
    t = eval(get(handles.t_input,'String'));
    if ~isnumeric(t)
        % t is not a number
        set(handles.plot_button,'String','t is not numeric')
    elseif length(t) < 2
        % t is not a vector
        set(handles.plot_button,'String','t must be vector')
    elseif length(t) > 1000
        % t is too long a vector to plot clearly
        set(handles.plot_button,'String','t is too long')
    elseif min(diff(t)) < 0
        % t is not monotonically increasing
        set(handles.plot_button,'String','t must increase')
    else
        % All OK; Enable the Plot button with its original name
        set(handles.plot_button,'String','Plot')
        set(handles.plot_button,'Enable','on')
        return
    end
    % Found an input error other than a bad expression
    % Give the edit text box focus so user can correct the error
    uicontrol(hObject)
 catch EM
    % Cannot evaluate expression user typed
    set(handles.plot_button,'String','Cannot plot t')
    % Give the edit text box focus so user can correct the error
    uicontrol(hObject)
end
```

The edit text callbacks execute when you enter text in an edit field and press **Return** or click elsewhere in the UI. Even if you immediately click the **Plot** button, the edit text callback executes before the **plot** button callback activates. When a callback receives invalid input, it disables the **Plot** button, preventing its callback from running. Finally, it restores focus to itself, selecting the text that did not validate so that you can re-enter a value.

For example, here is the response to input of a time vector, [1 2 6 4 5 7 9], that does not monotonically increase.
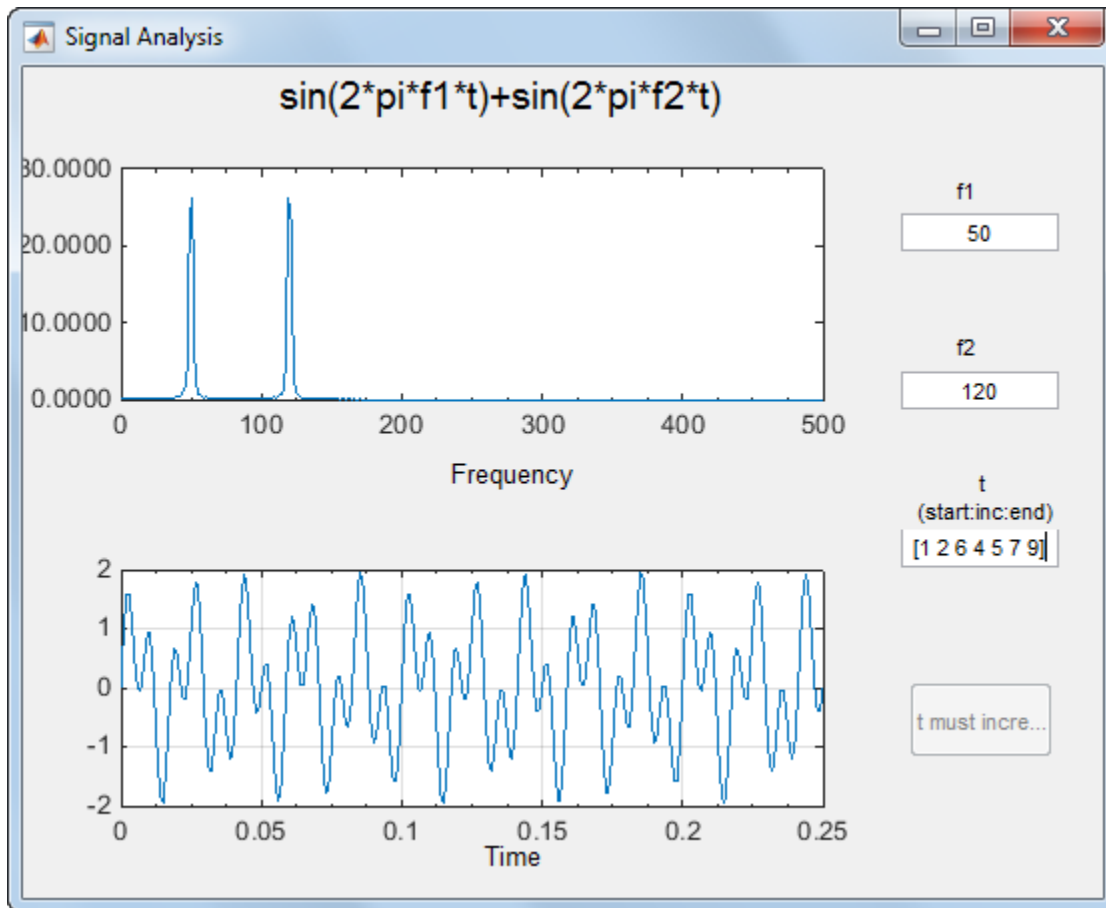
In this figure, the two plots reflect the last successful set of inputs, `f1 = 31.41`, `f2 = 120`, and `t = [1 2 3 4 5 7 9]`. The time vector `[1 2 6 4 5 7 9]` appears highlighted so that you can enter a new, valid, value. The highlighting results from executing the command `uicontrol(hObject)` in the preceding code listing.

Plot Push Button Behavior

When you click the **Plot** button, the `plot_button_Callback` performs three basic tasks: it gets input from the edit text components, calculates data, and creates the two plots.

Get Input

The first task for the `plot_button_Callback` is to read the input values. This involves:

- Reading the current values in the three edit text boxes using the `handles` structure to access the edit text handles.
- Converting the two frequency values (`f1` and `f2`) from strings to doubles using `str2double`.
- Evaluating the time string using `eval` to produce a vector `t`, which the callback used to evaluate the mathematical expression.

The following code shows how the `plot_button_Callback` obtains the input:

```
% Get user input
f1 = str2double(get(handles.f1_input,'String'));
f2 = str2double(get(handles.f2_input,'String'));
t = eval(get(handles.t_input,'String'));
```

Calculate Data

After constructing the string input parameters to numeric form and assigning them to local variables, the next step is to calculate data for the two graphs. The `plot_button_Callback` computes the time domain data using an expression of sines:

```
x = sin(2*pi*f1*t) + sin(2*pi*f2*t);
```

The callback computes the frequency domain data as the Fourier transform of the time domain data:

```
y = fft(x,512);
```

For an explanation of this computation, see the `fft` function.

Plot Data

The final task for the `plot_button_Callback` is to generate two plots. This involves:

- Targeting plots to the appropriate axes. For example, this code directs a graph to the time axes:

    `plot(handles.time_axes,t,x)`

- Providing the appropriate data to the `plot` function
- Turning on the axes grid, which the `plot` function automatically turns off

---

Note: Performing the last step is necessary because many plotting functions (including `plot`) clear the axes and reset properties before creating the graph. This means that you cannot use the Property Inspector to set the `XMinorTick`, `YMinorTick`, and grid properties in this example, because they are reset when the callback executes `plot`.

In the following code listing, notice how the `handles` structure provides access to the handle of the axes, when needed.

Plot_Button Callback

```
function plot_button_Callback(hObject, eventdata, handles, varargin)
% hObject    handle to plot_button (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)

% Get user input
f1 = str2double(get(handles.f1_input,'String'));
f2 = str2double(get(handles.f2_input,'String'));
t = eval(get(handles.t_input,'String'));

% Calculate data
x = sin(2*pi*f1*t) + sin(2*pi*f2*t);
y = fft(x,512);
m = y.*conj(y)/512;
f = 1000*(0:256)/512;

% Create frequency plot in proper axes
plot(handles.frequency_axes,f,m(1:257))
set(handles.frequency_axes,'XMinorTick','on')
grid on

% Create time plot in proper axes
plot(handles.time_axes,t,x)
set(handles.time_axes,'XMinorTick','on')
grid on
```

Resize and Command-Line Options

Select **Tools > GUI Options** to set these options in your UI:

• Resize behavior: **Proportional**

Selecting **Proportional** as the resize behavior enables you to resize the UI window. Using this option setting, when you resize the UI, everything expands or shrinks proportionately, except text.

• Command-line accessibility: **Callback**

When UIs include axes, their handles should be visible from other objects' callbacks. This enables you to use plotting commands like you would on the command line. **Callback** is the default setting for command-line accessibility.

For more information, see "GUIDE Options" on page 5-8.

Related Examples
• "Create a Simple App Using GUIDE" on page 2-2

More About
• "Write Callbacks in GUIDE" on page 7-2
• "Share Data Among Callbacks" on page 11-2

Synchronized Data Presentations in a GUIDE UI

About the Example

This example shows how to program a UI that has the following features:

- Initializes a table and a plot.
- Plots selected data in real time as you select data observations.
- Generates line graphs that display different views of data.

This UI plots different kinds of graphs into different axes for an entire data set or selections of it, and shows how Fourier transforms can identify periodicity in time series data.

You can use this UI to analyze and visualize time-series data containing periodic events.
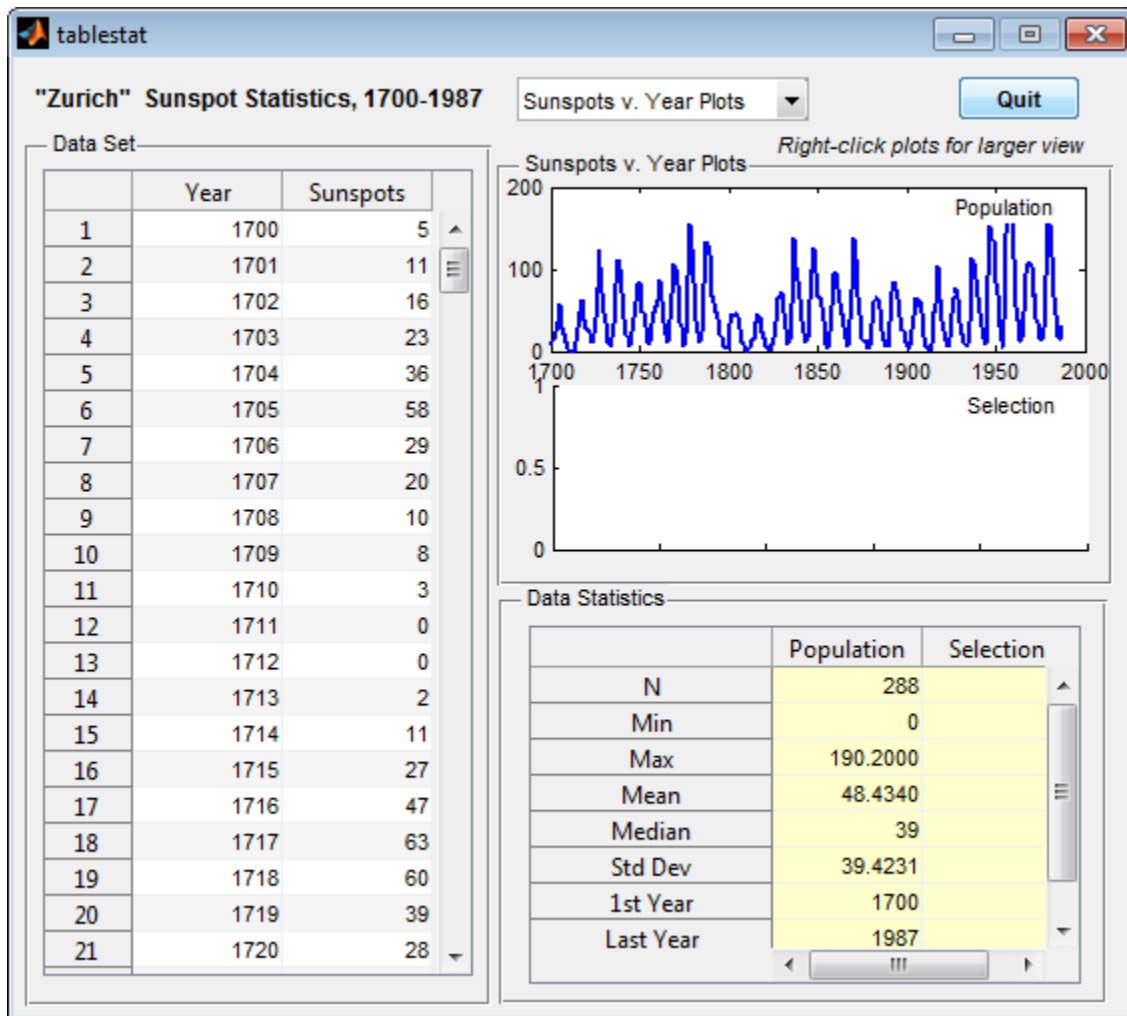
To get and view the example code:

1 Copy the example FIG-file and code file to your current (writeable) folder and open the FIG-file in GUIDE with the following commands:

```
copyfile(fullfile(docroot, 'techdoc','creating_guis',...
'examples','tablestat*.*')), fileattrib('tablestat*.*', '+w');
guide tablestat.fig;
```

2 In the GUIDE Layout Editor, click the Editor button ⬛.

The `tablestat.m` code file opens in the MATLAB Editor.

Recreate the UI

In the GUIDE Layout Editor, the `tablestat` UI looks like this.
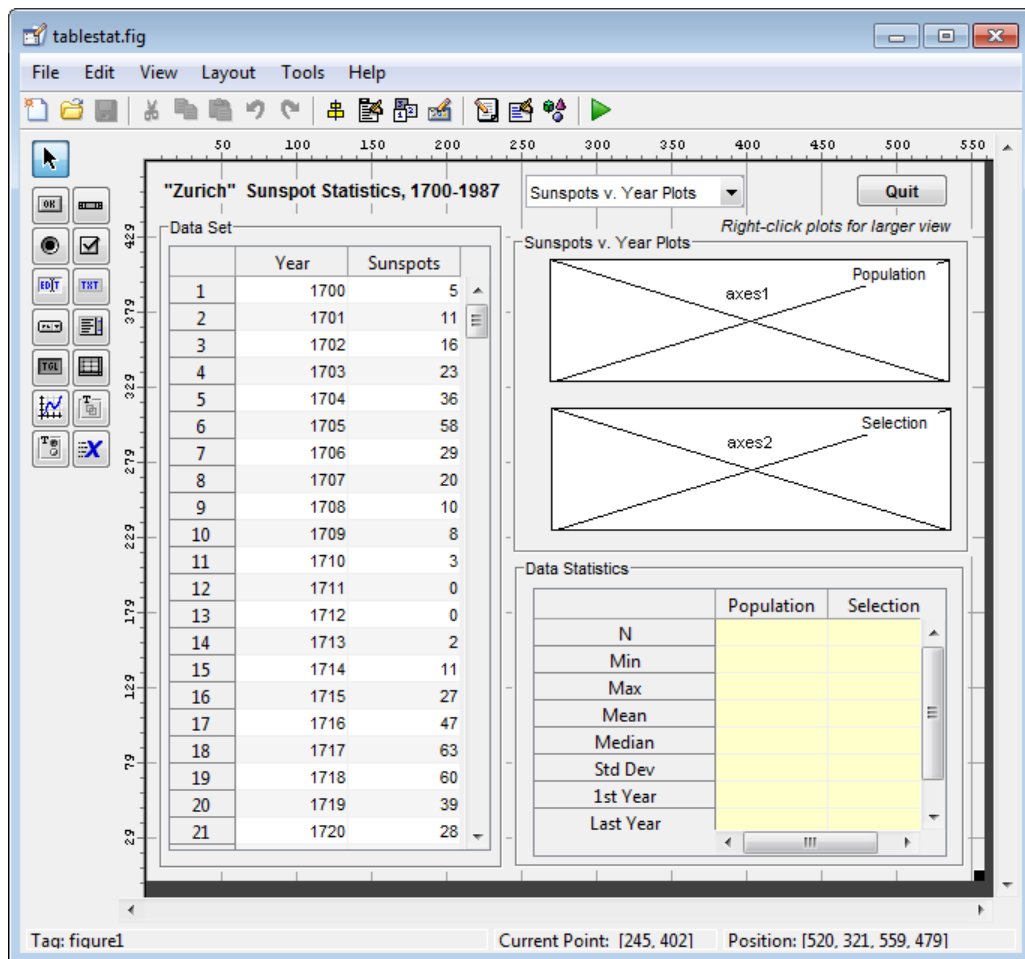
Perform the following steps in GUIDE and in the Property Inspector to generate the layout:

1 In the Command Window, type `guide`, select the **Blank GUI** template, and then click **OK**.

2 Use the Panel tool, , to drag out the three uipanels into the positions shown above. Keep the defaults for their `Tag` properties (which are `uipanel1`, `uipanel2`, and `uipanel3`). Create, in order:

    a A long panel on the left. In the Property Inspector, set its `Title` property value to `Data Set`.

    b A panel on the lower right, half the height of the first panel. In the Property Inspector, set its `Title` property value to `Data Statistics`.

    c A panel above the **Data Statistics** panel. In the Property Inspector, set its `Title` property to `Sunspots v. Year Plots`. This panel changes its name when the type of plot changes.

3 Use the Table tool, , to drag a uitable inside the **Data Set** panel. Use the Property Inspector to set the property values as follows:

    • `ColumnName`: `Year` and `Sunspot`.

    • `Data`: As described in "Initialize the Data Table" on page 8-23.

    • `Tag`: `data_table`.

    • `TooltipString`: `Drag to select a range of 11 or more observations`.

    • `CellSelectionCallback`: `data_table_CellSelectionCallback`.

    Click the pencil-and-paper icon to have GUIDE set this property value automatically and declare it in the code file.

4 Drag a second uitable inside the **Data Statistics** panel. Use the Table Property Editor to set row values as follows:

    a Double-click the **Data Statistics** table to open it in the Property Inspector.

    b In the Property Inspector, click the Table Property Editor icon to the right of the `RowName` property to open the Table Property Editor.

    c In the Table Property Editor, select **Rows** from the list in the left-hand column.

d Select the bottom radio button, **Show names entered below as row headers**.

e Type the nine strings listed in order on separate lines in the data entry pane, and then click **OK**.

- `BackgroundColor`: yellow (using the color picker).

- `ColumnName`: `Population` and `Selection`.

- `Tag`: `data_stats`.

- `TooltipString`: `statistics for table and selection`.

- `RowName` to nine strings: `N`, `Min`, `Max`, `Mean`, `Median`, `Std Dev`, `1st Year`, `Last Year`, and `Est. Period`.

The Table Property Editor looks like this before you close it.
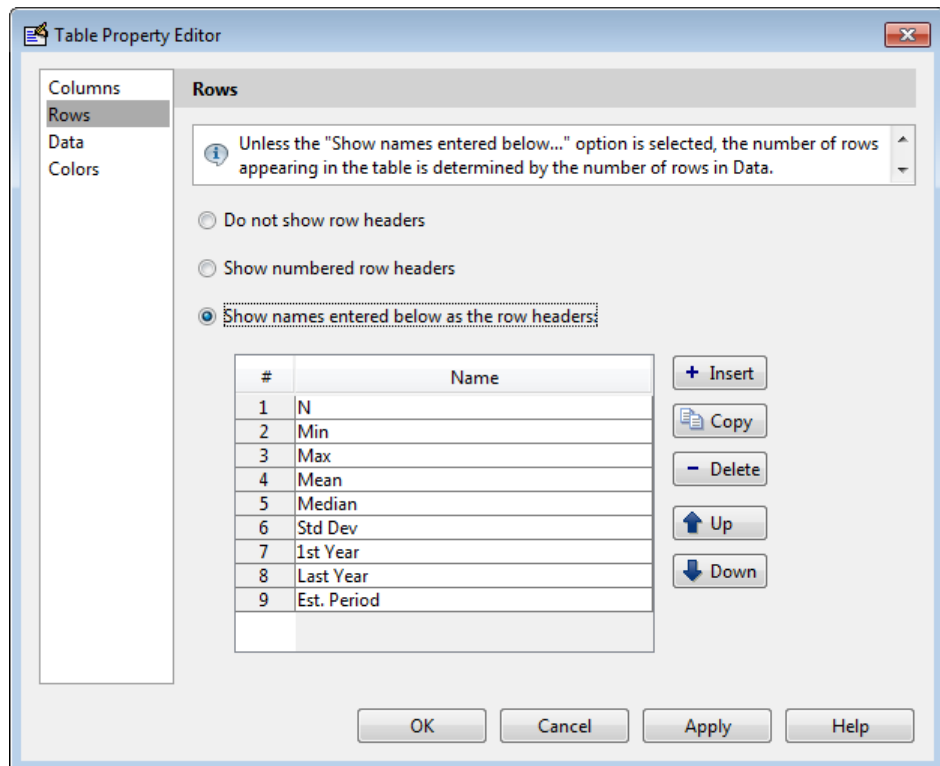
The **Data Statistics** table does not use any callbacks.

5   Use the Axes tool 🔛 to drag out an axes within the top half of the **Sunspots v. Year Plots** panel, leaving its name as `axes1`.

6   Drag out a second axes, leaving its name as `axes2` inside the **Sunspots v. Year Plots** panel, directly below the first axes.

  Leave enough space below each axes to display the *x*-axis labels.

7   Identify the axes with labels. Using the Text tool, drag out a small rectangle in the upper right corner of the upper axes (`axes1`). Double-click it, and in the Property Inspector, change its `String` property to `Population` and its `Tag` property to `poplabel`.

8   Place a second label in the lower axes (`axes2`), renaming this text object `Selection` and setting its `Tag` property to `sellabel`.

9   Create a title for the UI. Using the Text tool, drag out a static text object at the top left of the layout, above the data table. Double-click it, and in the Property Inspector, change its `String` property to `"Zurich" Sunspot Statistics, 1700-1987` and its `FontWeight` property to `bold`.

10 Add a prompt above the axes; place a text label just above the **Sunspots v. Year Plots** panel, near its right edge. Change its `Tag` property to `newfig`, its `String` property to `Right-click plots for larger view` and its `FontAngle` property to `Italic`.

11 Make a pop-up menu to specify the type of graph to plot. Using the Pop-up Menu tool 🔳, drag out a pop-up menu just above the **Sunspots v. Year** panel, aligning it to the panel's left edge. In the Property Inspector, set these properties:

  • `String`:

    ```
    Sunspots v. Year Plots
    FFT Periodogram Plots
    ```

  • `Tag`: `plot_type`

  • `Tooltip`: `Choose type of data plot`

  Then, click the `Callback` property's icon. This creates a declaration called `plot_type_Callback`, to which you add code later on.

12 Select the Push Button tool ⊞, and drag out a push button in the upper right of the figure. In the Property Inspector, rename it to **Quit** and set up its callback as follows:

- Double-click it and in the Property Inspector, set its `Tag` property to `quit` and its `String` property to `Quit`.

- Click the `Callback` property to create a callback for the button in the code file `tablestat.m`. GUIDE sets the `Callback` of the **Quit** item to `quit_Callback`.

- In the code file, for the `quit_Callback` function. enter:

```
close(ancestor(hObject,'figure'))
```

13 Save the UI in GUIDE, naming it `tablestat.fig`. This action also saves the code file as `tablestat.m`.

Initialize the Data Table

Although you can use the Opening Function to load data into a table, this example uses GUIDE to put data into the **Data Set** table. This way, the data becomes part of the figure after you save it. Initializing the table data causes the table to have the same number of rows and columns as the variable that it contains:

1 Access the sunspot example data set. In the Command Window, type:

```
load sunspot.dat
```
The variable `sunspot`, a 288-by-2 double array, displays in the MATLAB workspace.

2 Open the Property Inspector for the data table by double-clicking the **Data Set** table.

3 In the Property Inspector, click the Table Editor icon ⊞ to the right of the `Data` property to open the Table Property Editor.

4 In the Table Property Editor, select **Table** from the list in the left-hand column.

5 Select the bottom radio button, **Change data value to the selected workspace variable below**.

6 From the list of workspace variables in the box below the radio button, select `sunspot` and click **OK**.

GUIDE inserts the sunspot data in the table.

Compute the Data Statistics

The Opening Function retrieves the preloaded data from the data table and calls the local function, `setStats`, to compute population statistics, and then returns them. The `data_table_CellSelectionCallback` performs the same action when you select more than 10 rows of the data table. The only difference between these two calls is what input data is provided and what column of the **Data Statistics** table is computed. Here is the `setStats` function:

```
function stats = setStats(table, stats, col, peak)
% Computes basic statistics for data table.
%   table  The data to summarize (a population or selection)
%   stats  Array of statistics to update
%   col    Which column of the array to update
%   peak   Value for the peak period, computed externally

stats{1,col} =   size(table,1);      % Number of rows
stats{2,col} =    min(table(:,2));
stats{3,col} =    max(table(:,2));
stats{4,col} =   mean(table(:,2));
stats{5,col} = median(table(:,2));
stats{6,col} =    std(table(:,2));
stats{7,col} =        table(1,1);    % First row
stats{8,col} =        table(end,1);  % Last row
if ~isempty(peak)
    stats{9,col} = peak;             % Peak period from FFT
end
```

Note: When assigning data to a uitable, use a cell array, as shown in the code for `setStats`. You can assign data that you retrieve from a uitable to a numeric array, however, only if it is entirely numeric. Storing uitable data in cell arrays enables tables to hold numbers, strings of characters, or combinations of them.

The `stats` matrix is a 9-by-2 cell array in which each row is a separate statistic computed from the `table` argument. The last statistic is not computed by `setStats`; it comes from the `plotPeriod` function when it computes and plots the FFT periodogram and is passed to `setStats` as the `peak` parameter.

Specify the Type of Data Plot

From the UI, you can choose either of two types of plots to display from the `plot_type` pop-up menu:

- Sunspots v. Year Plots — Time-series line graphs displaying sunspot occurrences year by year (default).
- Periodogram Plots — Graphs displaying the FFT-derived power spectrum of sunspot occurrences by length of cycle in years.

When the plot type changes, one or both axes refresh. They always show the same kind of plot, but the bottom axes is initially empty and does not display a graph until you select at least 11 rows of the data table.

The `plot_type` control callback is `plot_type_Callback`. GUIDE generates it, and you must add code to it that updates plots appropriately. In the example, the callback consists of this code:

```
function plot_type_Callback(hObject, eventdata, handles)
% hObject    handle to plot_type (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)

% ---- Customized as follows ----
% Determine state of the pop-up and assign the appropriate string
% to the plot panel label
index = get(hObject,'Value');    % What plot type is requested?
strlist = get(hObject,'String');        % Get the choice's name
set(handles.uipanel3,'Title',strlist(index))  % Rename uipanel3

% Plot one axes at a time, changing data; first the population
table = get(handles.data_table,'Data'); % Obtain the data table
refreshDisplays(table, handles, 1)

% Now compute stats for and plot the selection, if needed.
% Retrieve the stored event data for the last selection
selection = handles.currSelection;
if length(selection) > 10  % If more than 10 rows selected
    refreshDisplays(table(selection,:), handles, 2)
else
    % Do nothing; insufficient observations for statistics
end
```

The function updates the Data Statistics table and the plots. To perform the updates, it calls the `refreshDisplays` function twice, which is a custom function added to the UI code file. In between the two calls, the `refreshDisplays` function retrieves row indices for the current selection from the `currSelection` member of the `handles` structure, where they were cached by the `data_table_CellSelectionCallback`.

You can see the effect of toggling the plot type in the two illustrations that follow. The one on the left shows the Sunspots v. Year plots, and the one on the right shows the FFT Periodograms Plots. The selection in both cases is the years 1901–1950.
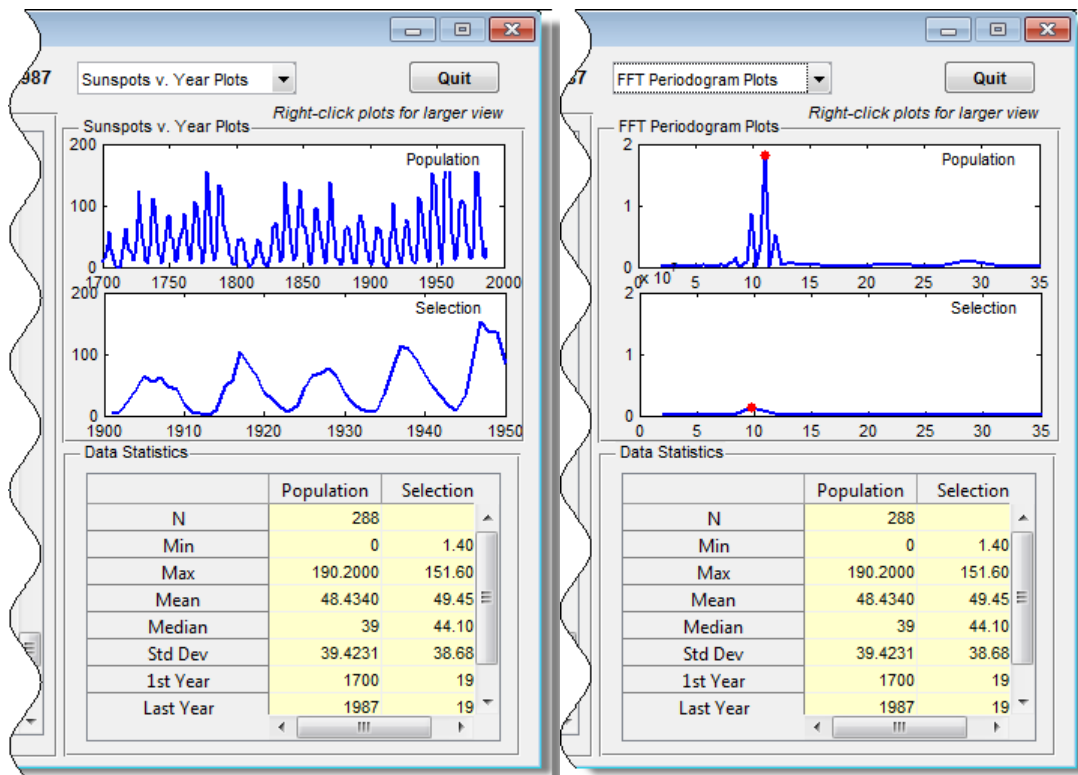
Respond to Data Selections

The **Data Set** table has two columns: **Year** and **Sunspots**. The data tables's Cell Selection Callback analyzes data from its second column, regardless of which columns you highlight. The `setStats` function (not generated by GUIDE) computes summary

statistics observations from the second column for insertion into the **Data Statistics** table on the right. The `plotPeriod` function (not generated by GUIDE) plots either the raw data or a Fourier analysis of it.

The `data_table_CellSelectionCallback` function manages the application's response to you selecting ranges of data. Ranges can be contiguous rows or separate groups of rows; holding down the **Ctrl** key lets you add discontiguous rows to a selection. Because the Cell Selection Callback is triggered as long as you hold the left mouse button down within the table, the selection statistics and lower plot are refreshed until selection is completed.

Selection data is generated during mouseDown events (mouse drags in the data table). The uitable passes this stream of cell indices (but not cell values) via the `eventdata` structure to the `data_table_CellSelectionCallback` callback. The callback's code reads the indices from the `Indices` member of the `eventdata`.

When the callback runs (for each new value of `eventdata`), it turns the event data into a set of rows:

```
selection = eventdata.Indices(:,1);
selection = unique(selection);
```

The event data contains a sequence of `[row, column]` indices for each table cell currently selected, one cell per line. The preceding code trims the list of indices to a list of selected rows, removing column indices. Then it calls the `unique` MATLAB function to eliminate any duplicate row entries, which arise whenever you select both columns. For example, suppose `eventdata.Indices` contains:

```
 1    1
 2    1
 3    1
 3    2
 4    2
```

This indicates that you selected the first three rows in column one (Year) and rows three and four in column two (Sunspots) by holding down the **Ctrl** key when selecting numbers in the second column. The preceding code transforms the indices into this vector:

```
1
2
3
4
```

This vector enumerates all the selected rows. If the selection includes less than 11 rows (as it does here) the callback returns, because computing statistics for a sample that small is not useful.

When the selection contains 11 or more rows, the data table is obtained, the selection is cached in the `handles` structure, and the `refreshDisplays` function is called to update the selection statistics and plot, passing the portion of the table that you selected:

```
table = get(hObject,'Data');
handles.currSelection = selection;
guidata(hObject,handles)
refreshDisplays(table(selection,:), handles, 2)
```

Caching the list of rows in the selection is necessary because changing plot types can force selection data to be replotted. As the `plot_type_Callback` has no access to the data table's event data, it requires a copy of the most recent selection.

Update the Statistics Table and the Graphs

The code must update the Data Statistics table and the graphs above it when:

- The UI is initialized, in its `tablestat_OpeningFcn`.
- You select cells in the data table, in its `data_table_CellSelectionCallback`.
- You select a different plot type, in the `plot_type_Callback`.

In each case, the `refreshDisplays` function is called to handle the updates. It in turn calls two other custom functions:

- `setStats` — Computes summary statistics for the selection and returns them.
- `plotPeriod` — Plots the type of graph currently requested in the appropriate axes.

The `refreshDisplays` function identifies the current plot type and specifies the axes to plot graphs into. After calling `plotPeriod` and `setStats`, it updates the **Data Statistics** table with the recomputed statistics. Here is the code for `refreshDisplays`:

```
function refreshDisplays(table, handles, item)
if isequal(item,1)
    ax = handles.axes1;
elseif isequal(item,2)
    ax = handles.axes2;
end
peak = plotPeriod(ax, table,...
             get(handles.plot_type,'Value'));
stats = get(handles.data_stats, 'Data');
stats = setStats(table, stats, item, peak);
set(handles.data_stats, 'Data', stats);
set(ax,'FontSize',7.0);
```

If you are reading this document in the MATLAB Help Browser, click the names of the functions underlined above to see their complete code (including comments) in the MATLAB Editor.

Display Graphs in New Figure Windows

The `tablestat` UI contains code to display either of its graphs in a larger size in a new figure window when you right-click either axes and selects the pop-up menu item, **Open plot in new window**. The static text string (tagged `newfig`) above the plot panel, **Right-click plots for larger view**, informs you that this feature is available.

The axes respond by:

1. Creating a new figure window.
2. Copying their contents to a new axes parented to the new figure.
3. Resizing the new axes to use 90% of the figure's width.
4. Constructing a title string and displaying it in the new figure.
5. Saving the figure and axes handles in the `handles` structure for possible later use or destruction.

Note: Handles are saved for both plots, but each time a new figure is created for either of them, the new handles replace the old ones, if any, making previous figures inaccessible from the UI.

Create Two Context Menus

To create the two context menus, from the GUIDE **Tools** menu, select the **Menu Editor**. After you create the two context menus, attach one to the each axes, `axes1` and `axes2`. In the Menu Editor, for each menu:

1. Click the **Context Menus** tab to select the type of menu you are creating.

2. Click the **New Context Menu** icon .

   This creates a context menu in the Menu Editor workspace called `untitled`. It has no menu items and is not attached to any UI object yet.

3. Select the new menu and in the **Tag** edit field in the **Menu Properties** panel, type `plot_axes1`.

4. Click the **New Menu Item** icon .

A menu item is displayed underneath the `plot_axes1` item in the Menu Editor workspace.

5   In the **Menu Properties** panel, type `Open plot in new window` for **Label** and `plot_ax1` for **Tag**. Do not set anything else for this item.

6   Repeat the last four steps to create a second context menu:

- Make the **Tag** for the menu `plot_axes2`.

- Create a menu item under it and make its **Label** `Open plot in new window` and assign it a **Tag** of `plot_ax2`.

7   Click **OK** to save your menus and exit the Menu Editor.

For more information about using the Menu Editor, see "Create Menus for GUIDE UIs" on page 6-77.

Attach Context Menus to Axes

Add the context menus you just created to the axes:

1   In the GUIDE Layout Editor, double-click `axes1` (the top axes in the upper right corner) to open it in the Property Inspector.

2   Click the right-hand column next to `UIContextMenu` to see a drop-down list.

3   From the list, select `plot_axes1`.

Perform the same steps for `axes2`, but select `plot_axes2` as its `UIContextMenu`.

Code Context Menu Callbacks

The two context menu items perform the same actions, but create different objects. Each has its own callback. Here is the `plot_ax1_Callback` callback for `axes1`:

```
function plot_ax1_Callback(hObject, eventdata, handles)
% hObject    handle to plot_ax1 (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)
%
% Displays contents of axes1 at larger size in a new figure

% Create a figure to receive this axes' data
axes1fig = figure;
% Copy the axes and size it to the figure
```

```
axes1copy = copyobj(handles.axes1,axes1fig);
set(axes1copy,'Units','Normalized',...
              'Position',[.05,.20,.90,.60])
% Assemble a title for this new figure
str = [get(handles.uipanel3,'Title') ' for ' ...
       get(handles.poplabel,'String')];
title(str,'Fontweight','bold')
% Save handles to new fig and axes in case
% we want to do anything else to them
handles.axes1fig = axes1fig;
handles.axes1copy = axes1copy;
guidata(hObject,handles);
```

The other callback, `plot_ax2_Callback`, is identical to `plot_ax1_Callback`, except that all instances of 1 in the code are replaced by 2, and `poplabel` is replaced with `sellabel`. The `poplabel` and `sellabel` objects are the **Population** and **Selection** labels on `axes1` and `axes2`, respectively. These strings are appended to the current `Title` for `uipanel3` to create a title for the plot in the new figure `axes1fig` or `axes2fig`.

Use Plot in New Window Feature

Whenever you right-click one of the axes in the UI and select `Open plot in new window`, a new figure is generated containing the graph in the axes. The callbacks do not check whether a graph exists in the axes (`axes2` is empty until you select cells in the **Data Set**) or whether a previously opened figure contains the same graph. A new figure is always created and the contents of `axes1` or `axes2` are copied into it. For example, you could right-click a periodogram in `axes1` and select `Open plot in new window`.

It is your responsibility to remove the new window when it is no longer needed. The context menus can be programmed to do this. Because their callbacks call `guidata` to save the handle of the last figure created for each of the axes, another callback can delete or reuse either figure. For example, the `plot_ax1_Callback` and `plot_ax2_Callback` callbacks could check `guidata` for a valid axes handle stored in `handles.axes1copy` or `handles.axes2copy`, and reuse the axes instead of creating a new figure.

Related Examples

- "Create a Simple App Using GUIDE" on page 2-2
- "Fast Fourier Transform (FFT)"
- "The FFT in One Dimension"

More About

- "Write Callbacks in GUIDE" on page 7-2
- "Share Data Among Callbacks" on page 11-2

Interactive List Box in a GUIDE UI

In this section...
"About the Example" on page 8-33
"Implement the List Box" on page 8-34

About the Example

This example shows how to create a list box to display the files in a folder. When you double click a list item, MATLAB opens the item:

- If the item is a file, MATLAB opens the file using the appropriate application.
- If the item is a folder, MATLAB reads the contents of that folder into the list box.
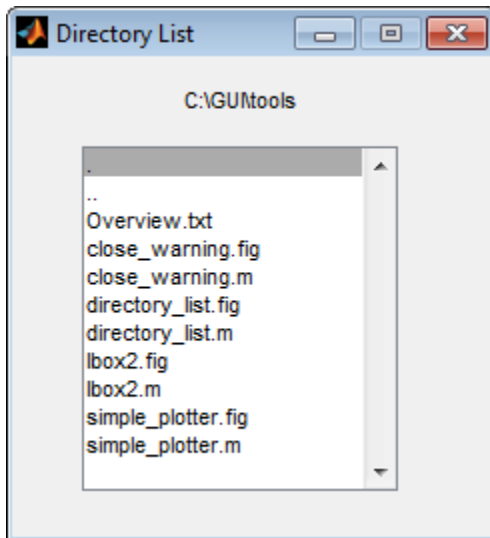
To get and view the example code:

1 Copy the example FIG-file and code file to your current (writeable) folder and open the FIG-file in GUIDE and the with the following commands:

```
copyfile(fullfile(docroot, 'techdoc','creating_guis',...
'examples','lbox2*.*')), fileattrib('lbox2*.*', '+w')
guide lbox2.fig
```

2　From GUIDE Layout Editor, click the Editor button 🖼.

　　The `lbox2.m` code displays in the MATLAB Editor.

Implement the List Box

The following sections describe the implementation:

- "Specify the Folder" on page 8-34 — shows how to pass a folder path as an input argument when the program runs.
- "Load the List Box" on page 8-35 — describes the local function that loads the contents of the folder into the list box. This local function also saves information about the contents of a folder in the `handles` structure.
- "Code List Box Behavior" on page 8-36 — describes how the list box is coded to respond to double clicks on items in the list box.

Specify the Folder

By default, UI code files generated by GUIDE open the UI when there are no input arguments, and call a local function when the first input argument is a character string. This example changes the behavior so that if you put the example files, `lbox2.m` and `lbox2.fig`, on the MATLAB path you can run the `lbox2` program to display the contents of a particular folder. To do so, pass the `dir` function as a string for the first input argument, and a string that specifies the path to the folder for the second input argument. For instance, from the Command Window, run the following to have the list box display the files in `C:\myfiles`:

```
lbox2('dir','C:\my_files')
```

The following code from `lbox2.m` shows the code for `lbox2_OpeningFcn`, which sets the list box folder to:

- The current folder, if no folder is specified.
- The specified folder, if a folder is specified.

```
function lbox2_OpeningFcn(hObject, eventdata, handles, varargin)
% This function has no output args, see OutputFcn.
% hObject    handle to figure
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)
% varargin   command line arguments to untitled (see VARARGIN)

% Choose default command line output for lbox2
handles.output = hObject;
```

```
% Update handles structure
guidata(hObject, handles);

if nargin == 3,
    initial_dir = pwd;
elseif nargin > 4
    if strcmpi(varargin{1},'dir')
        if exist(varargin{2},'dir')
            initial_dir = varargin{2};
        else
            errordlg({'Input argument must be a valid',...
                    'folder'},'Input Argument Error!')
            return
        end
    else
        errordlg('Unrecognized input argument',...
                'Input Argument Error!');
        return;
    end
end
% Populate the listbox
load_listbox(initial_dir,handles)
```

Load the List Box

A local function loads items into the list box. This local function accepts the path to a folder and the `handles` structure as input arguments and performs these steps:

- Changes to the specified folder so that the code can navigate up and down the tree, as required.
- Uses the `dir` command to get a list of files in the specified folder and to determine which name is a folder and which is a file. `dir` returns a structure (`dir_struct`) with two fields, `name` and `isdir`, containing this information.
- Sorts the file and folder names (`sortrows`) and saves the sorted names and other information in the `handles` structure so that this information can be passed to other functions.

  The `name` structure field is passed to `sortrows` as a cell array, which is transposed to get one file name per row. The `isdir` field and the sorted index values, `sorted_index`, are saved as vectors in the `handles` structure.
- Calls `guidata` to save the `handles` structure.
- Sets the list box `String` property to display the file and folder names and set the `Value` property to 1, ensuring that `Value` never exceeds the number of items in `String`, because MATLAB software updates the `Value` property only when a selection occurs; not when the contents of `String` changes.

- Displays the current folder in the text box by setting its `String` property to the output of the `pwd` command.

The `load_listbox` function is called by the opening function, as well as by the list box callback.

```
function load_listbox(dir_path, handles)
cd (dir_path)
dir_struct = dir(dir_path);
[sorted_names,sorted_index] = sortrows({dir_struct.name}');
handles.file_names = sorted_names;
handles.is_dir = [dir_struct.isdir];
handles.sorted_index = sorted_index;
guidata(handles.figure1,handles)
set(handles.listbox1,'String',handles.file_names,...
 'Value',1)
set(handles.text1,'String',pwd)
```

Code List Box Behavior

The listbox1_Callback code handles only one case: a double-click of an item. Double clicking is the standard way to open a file from a list box. If the selected item is a file, it is passed to the `open` command; if it is a folder, the program navigates to that folder and lists the contents.

- Define how to open file types

  The `open` command can handle a number of different file types, however, the callback treats FIG-files differently. Instead of opening the FIG-file as a standalone figure, it opens it with `guide` for editing.

- Determine which item was selected

  Since a single click of an item also invokes the list box callback, you must query the figure SelectionType property to determine when you have performed a double click. A double-click of an item sets the `SelectionType` property to `open`.

  All the items in the list box are referenced by an index from `1` to `n`. A value of `1` refers to the first item, and a value of `n` is the index of the `n`th item. The software saves this index in the list box `Value` property.

  The callback uses this index to get the name of the selected item from the list of items contained in the `String` property.

- Determine whether the selected item is a file or directory

  The `load_listbox` function uses the `dir` command to obtain a list of values that indicate whether an item is a file or folder. These values (1 for folder, 0 for file) are saved in the `handles` structure. The list box callback queries these values to determine if current selection is a file or folder and takes the following action:

  - If the selection is a folder — change to the folder (`cd`) and call `load_listbox` again to populate the list box with the contents of the new folder.
  - If the selection is a file — get the file extension (`fileparts`) to determine if it is a FIG-file, which is opened with `guide`. All other file types are passed to `open`.

  The `open` statement is called within a `try, catch` block to capture errors in an error dialog box (`errordlg`), instead of returning to the command line.

  You can extend the file types that the `open` command recognizes to include any file having a three-character extension. Do this by creating a MATLAB code file with the name `openxyz.m`. *xyz* is the file extension for the type of files to be handled. Do not, however, take this approach for opening FIG-files, because `openfig.m` is a MATLAB function which is needed to open UIs. For more information, see `open` and `openfig`.

listbox1_Callback code

```
function listbox1_Callback(hObject, eventdata, handles)
get(handles.figure1,'SelectionType');
% If double click
if strcmp(get(handles.figure1,'SelectionType'),'open')
    index_selected = get(handles.listbox1,'Value');
    file_list = get(handles.listbox1,'String');
    % Item selected in list box
    filename = file_list{index_selected};
    % If folder
    if  handles.is_dir(handles.sorted_index(index_selected))
        cd (filename)
        % Load list box with new folder.
        load_listbox(pwd,handles)
    else
        [path,name,ext] = fileparts(filename);
        switch ext
            case '.fig'
                % Open FIG-file with guide command.
                guide (filename)
            otherwise
```

```
                    try
                        % Use open for other file types.
                        open(filename)
                    catch ex
                        errordlg(...
                          ex.getReport('basic'),'File Type Error','modal')
                    end
            end
        end
end
```

Related Examples

- "Create a Simple App Using GUIDE" on page 2-2

More About

- "Write Callbacks in GUIDE" on page 7-2
- "Share Data Among Callbacks" on page 11-2

Plot Workspace Variables in a GUIDE UI

In this section...
"About the Example" on page 8-39
"Read Workspace Variables" on page 8-40
"Read Selections from List Box" on page 8-41

About the Example

This example shows how to create a UI that uses a list box to display names of variables in the base workspace, and then plot them. Initially, no variable names are selected in the list box. The UI contains controls that perform these tasks:

- Update the list.
- Select multiple variables in the list box. Exactly two variables must be selected.
- Create linear, `semilogx` and `semilogy` line graphs of selected variables.

The program evaluates the plotting commands in the base workspace. It does no validation before plotting. When you use this UI, you are responsible for selecting pairs of variables that can be plotted against one another. The top-most selection is used as the *x*-variable, the lower one as the *y*-variable.
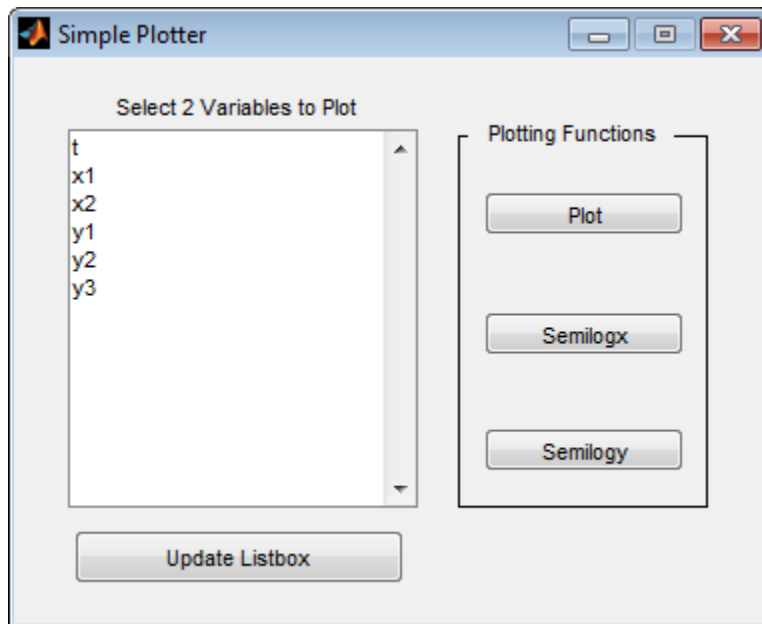
To get and view the example code:

1 Copy the example FIG-file and code file to your current (writeable) folder and open the FIG-file in GUIDE with the following commands:

```
copyfile(fullfile(docroot, 'techdoc','creating_guis',...
'examples','lb.*')), fileattrib('lb.*', '+w')
guide lb.fig
```

2 From GUIDE Layout Editor, click the Editor button 🗒.

The lb.m code displays in the MATLAB Editor.

Read Workspace Variables

When the UI initializes, it queries the workspace variables and sets the list box String property to display these variable names. Adding the following local function to the UI code, lb.m, accomplishes this using evalin to execute the who command in the base workspace. The who command returns a cell array of strings, which are used to populate the list box.

```
function update_listbox(handles)
vars = evalin('base','who');
set(handles.listbox1,'String',vars)
```

The function input argument is the handles structure set up by the GUIDE. This structure contains the handle of the list box, as well as the handles of all other components in the UI.

The callback for the **Update Listbox** push button also calls update_listbox.

Read Selections from List Box

To use the UI, select two variables from the workspace and then choose one of three plot commands to create a graph: plot, semilogx, or semilogy.

No callback for the list box exists in the code file. One is not needed because the plotting actions are initiated by push buttons.

Enable Multiple Selection

Use the Property Inspector to set these properties on the list box. To enable multiple selection in a list box, change the default values of the Min and Max properties so that Max - Min > 1.

Selecting Multiple Items

List box selection follows the standard for most systems:

- **Ctrl**+click left mouse button — noncontiguous multi-item selection
- **Shift**+click left mouse button — contiguous multi-item selection

Use one of these techniques to select the two variables required to create the plot.

Return Variable Names for the Plotting Functions

The local function, get_var_names, returns the two variable names that are selected when you click one of the three plotting buttons. The function does these tasks:

- Gets all the items in the list box from the String property.
- Gets the indices of the selected items from the Value property.

•   Returns two string variables, if there are two items selected. Otherwise `get_var_names` displays an error dialog box stating that you must select two variables.

Here is the code for `get_var_names`:

```
function [var1,var2] = get_var_names(handles)
list_entries = get(handles.listbox1,'String');
index_selected = get(handles.listbox1,'Value');
if length(index_selected) ~= 2
 errordlg('You must select two variables',...
   'Incorrect Selection','modal')
else
 var1 = list_entries{index_selected(1)};
 var2 = list_entries{index_selected(2)};
end
```

Callbacks for the Plotting Buttons

The callbacks for the plotting buttons call `get_var_names` to get the names of the variables to plot and then call `evalin` to execute the plot commands in the base workspace.

For example, here is the callback for the `plot` function:

```
function plot_button_Callback(hObject, eventdata, handles)
[x,y] = get_var_names(handles);
evalin('base',['plot(' x ',' y ')'])
```

The command to evaluate is created by concatenating the strings and variables, and looks like this:

```
try
    evalin('base',['semilogx(',x,',',y,')'])
catch ex
    errordlg(...
      ex.getReport('basic'),'Error generating semilogx plot','modal')
end
```

The `try/catch` block handles errors resulting from attempting to graph inappropriate data. When evaluated, the result of the command is:

```
plot(x,y)
```

The other two plotting buttons work in the same way, resulting in `semilogx(x,y)` and `semilogy(x,y)`.

Related Examples

- "Create a Simple App Using GUIDE" on page 2-2

More About

- "Write Callbacks in GUIDE" on page 7-2
- "Share Data Among Callbacks" on page 11-2

Automatically Refresh Plot in a GUIDE UI

In this section...
"About the Example" on page 8-44
"The Timer Implementation" on page 8-46

About the Example

This example shows how to refresh a display by incorporating a timer in a UI that updates data. Timers are MATLAB objects. Programs use their properties and methods to schedule tasks, update information, and time out processes. For example, you can set up a timer to acquire real-time data at certain intervals, which your UI then analyzes and displays.

The UI displays a surface plot of the `peaks` function and contains three uicontrols:

- The **Start Randomizing** push button — Starts the timer running, which executes at a rate determined by the slider control. At each iteration, random noise is added to the surface plot.
- The **Stop Randomizing** push button — Halts the timer, leaving the surface plot in its current state until you click the **Start Randomizing** button again.
- The **Timer Period** slider — Speeds up and slows down the timer, changing its period within a range of 0.01 to 2 seconds.
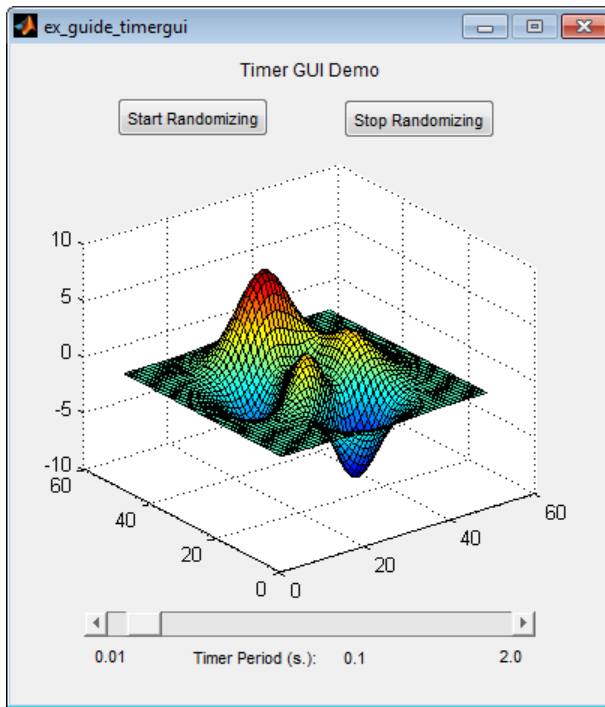
To get and view the example code:

1   Copy the example FIG-file and code file to your current (writeable) folder and open
    the FIG-file in GUIDE with the following commands:

    ```
    copyfile(fullfile(docroot, 'techdoc','creating_guis','examples',...
    'ex_guide_timergui*.*')), fileattrib('ex_guide_timergui*.*', '+w')
    guide ex_guide_timergui.fig
    ```

2   From the GUIDE Layout Editor, click the Editor button 📝.

    The ex_guide_timergui.m code displays in the MATLAB Editor.

The following figures show the UI when you run it. The left figure shows the initial state
of the UI. The right figure shows the UI after running the timer for 5 seconds at its
default period of 1 count per second.
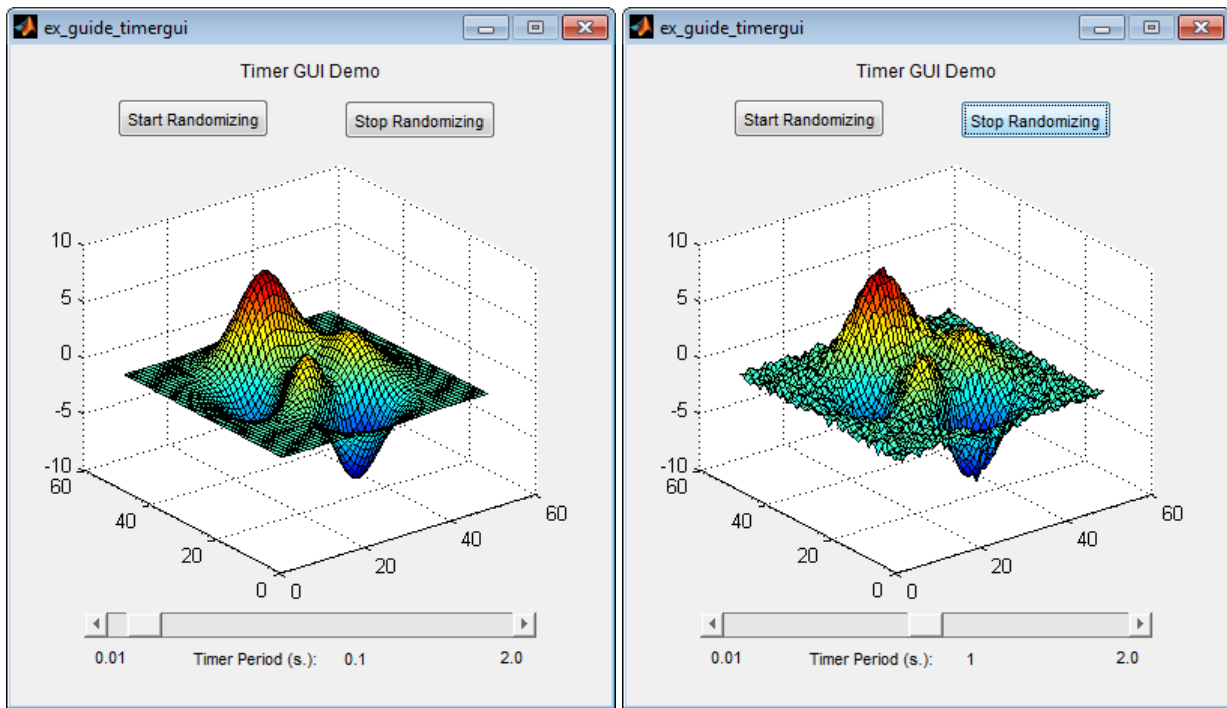
For details about timer properties, methods, and events, see "Use a MATLAB Timer Object" and the timer reference page.

The Timer Implementation

Each callback in the UI either creates, modifies, starts, stops, or destroys the timer object. The following sections describe what each callbacks does.

- "ex_guide_timergui_OpeningFcn" on page 8-47
- "startbtn_Callback" on page 8-47
- "stopbtn_Callback" on page 8-47
- "periodsldr_Callback" on page 8-47
- "update_display" on page 8-48
- "figure1_CloseRequestFcn" on page 8-48

ex_guide_timergui_OpeningFcn

`ex_guide_timergui_OpeningFcn` creates the timer using the following code:

```
handles.timer = timer(...
    'ExecutionMode', 'fixedRate', ...   % Run timer repeatedly
    'Period', 1, ...                    % Initial period is 1 sec.
    'TimerFcn', {@update_display,hObject}); % Specify callback
```

The opening function also initializes the slider `Min`, `Max`, and `Value` properties, and sets the slider label to display the value:

```
set(handles.periodsldr,'Min',0.01,'Max',2)
set(handles.periodsldr,'Value',get(handles.timer('Period'))
set(handles.slidervalue,'String',...
    num2str(get(handles.periodsldr,'Value')))
```

A call to `surf` renders the `peaks` data in the axes, adding the surfaceplot handle to the handles structure:

```
handles.surf = surf(handles.display,peaks);
```

Finally, a call to `guidata` saves the handles structure contents:

```
guidata(hObject,handles);
```

startbtn_Callback

`startbtn_Callback` calls timer `start` method if the timer is not already running:

```
if strcmp(get(handles.timer, 'Running'), 'off')
    start(handles.timer);
end
```

stopbtn_Callback

`stopbtn_Callback` calls the timer `stop` method if the timer is currently running:

```
if strcmp(get(handles.timer, 'Running'), 'on')
    stop(handles.timer);
end
```

periodsldr_Callback

`periodsldr_Callback` is called each time you move the slider. It sets the timer period to the slider current value after removing unwanted precision:

```
% Read the slider value
```

```
period = get(handles.periodsldr,'Value');
% Timers need the precision of periods to be greater than about
% 1 millisecond, so truncate the value returned by the slider
period = period - mod(period,.01);
% Set slider readout to show its value
set(handles.slidervalue,'String',num2str(period))
% If timer is on, stop it, reset the period, and start it again.
if strcmp(get(handles.timer, 'Running'), 'on')
    stop(handles.timer);
    set(handles.timer,'Period',period)
    start(handles.timer)
else              % If timer is stopped, reset its period only.
    set(handles.timer,'Period',period)
end
```

The slider callback must stop the timer to reset its period, because timer objects do not allow their periods to vary while they are running.

update_display

update_display is the callback for the timer object. It adds Gaussian noise to the ZData of the surface plot:

```
handles = guidata(hfigure);
Z = get(handles.surf,'ZData');
Z = Z + 0.1*randn(size(Z));
set(handles.surf,'ZData',Z);
```

Because update_display is not a GUIDE-generated callback, it does not include handles as one of its calling arguments. Instead, it accesses the handles structure by calling guidata. The callback gets the ZData of the surface plot from the handles.surf member of the structure. It modifies the Z matrix by adding noise using randn, and then resets the ZData of the surface plot with the modified data. It does not modify the handles structure.

figure1_CloseRequestFcn

MATLAB calls the figure1_CloseRequestFcn when you click the close box on the UI window. The callback cleans up the application before it exits, stopping and deleting the timer object and then deleting the figure window.

```
% Necessary to provide this function to prevent timer callback
% from causing an error after code stops executing.
% Before exiting, if the timer is running, stop it.
if strcmp(get(handles.timer, 'Running'), 'on')
    stop(handles.timer);
```

```
end
% Destroy timer
delete(handles.timer)
% Destroy figure
delete(hObject);
```

Related Examples

· "Create a Simple App Using GUIDE" on page 2-2

More About

· "Write Callbacks in GUIDE" on page 7-2
· "Share Data Among Callbacks" on page 11-2

Create UIs Programmatically

9

Lay Out a Programmatic UI

Structure of Programmatic UI Code Files

In this section...
"File Organization" on page 9-2
"File Template" on page 9-2
"Run the Program" on page 9-3

File Organization

Typically, a UI code file has the following ordered sections. You can help to maintain the structure by adding comments that name the sections when you first create them.

1 Comments displayed in response to the MATLAB `help` command.

2 Initialization tasks such as data creation and any processing that is needed to construct the components. See "Initialize a Programmatic UI" on page 10-2 for more information.

3 Construction of figure and components. For more information, see "Create Figures for Programmatic UIs" on page 9-4 and "Add Components to a Programmatic UI" on page 9-6.

4 Initialization tasks that require the components to exist, and output return. See "Initialize a Programmatic UI" on page 10-2 for more information.

5 Callbacks for the components. Callbacks are the routines that execute in response to user-generated events such as mouse clicks and key strokes. See "Write Callbacks for Apps Created Programmatically" on page 10-5 for more information.

6 Utility functions.

File Template

This is a template you can use to create a UI code file:

```
function varargout = myui(varargin)
% MYUI Brief description of program.
%       Comments displayed at the command line in response
%       to the help command.

% (Leave a blank line following the help.)
```

```
%  Initialization tasks

%  Construct the components

%  Initialization tasks

%  Callbacks for MYUI

%  Utility functions for MYUI

end
```

Save the file in your current folder or at a location that is on your MATLAB path.

Run the Program

You can display your UI at any time by executing the code file. For example, if your UI code file is `myui.m`, type

```
myui
```

at the command line. Provide run-time arguments as appropriate. The file must reside on your path or in your current folder.

When you execute the code, a fully functional copy of the UI displays on the screen. If the file includes code to initialize the UI and callbacks to service the components, you can manipulate components that it contains.

Related Examples

- "Create a Simple App Programmatically" on page 3-2

Create Figures for Programmatic UIs

In MATLAB software, a UI is a figure. Before you add components to it, create the figure explicitly and obtain a handle for it. In the initialization section of your file, use a statement such as the following to create the figure:

```
fh = figure;
```

where `fh` is the figure handle.

Note If you create a component when there is no figure, MATLAB creates a figure automatically but does not return the figure handle.

When you create the figure, you can also specify properties for the figure. The most commonly used figure properties are shown in the following table:

Property	Values	Description
MenuBar	figure, none. Default is figure.	Display or hide the MATLAB standard menu bar menus. If none and there are no user-created menus, the menu bar itself is removed.
Name	String	Title displayed in the figure window. If NumberTitle is on, this string is appended to the figure number.
NumberTitle	on, off. Default is on.	Determines whether the string 'Figure n' (where n is the figure number) is prefixed to the figure window title specified by Name.
Position	4-element vector: [distance from left, distance from bottom, width, height].	Size of the UI figure and its location relative to the lower-left corner of the screen.
Resize	on, off. Default is on.	Determines if the user can resize the figure window with the mouse.
Toolbar	auto, none, figure. Default is auto.	Display or hide the default figure toolbar.

Property	Values	Description
		• `none` — do not display the figure toolbar. • `auto` — display the figure toolbar, but remove it if a user interface control (`uicontrol`) is added to the figure. • `figure` — display the figure toolbar.
Units	`pixels`, `centimeters`, `characters`, `inches`, `normalized`, `points`, Default is `pixels`.	Units of measurement used to interpret position vector
Visible	`on`, `off`. Default is `on`.	Determines whether a figure is displayed on the screen.

For a complete list of properties and for more information about the properties listed in the table, see Figure Properties.

The following statement names the figure My UI, positions the figure on the screen, and makes the UI invisible so that the user cannot see the components as they are added or initialized. All other properties assume their defaults.

```
f = figure('Visible','off','Name','My UI',...
          'Position',[360,500,450,285]);
```

Related Examples

Add Components to a Programmatic UI

User interface controls are common UI components, such as buttons, check boxes, and sliders. Tables present data in rows and columns. Panels and button groups are containers in which you can group together related elements in your UI. ActiveX components enable you to display ActiveX controls.

In this section...
"User Interface Controls" on page 9-6
"Tables" on page 9-18
"Panels" on page 9-19
"Button Groups" on page 9-21
"Axes" on page 9-23
"ActiveX Controls" on page 9-25
"How to Set Font Characteristics" on page 9-25

User Interface Controls

Push Button

This code creates a push button:

```
f = figure;
pb = uicontrol(f,'Style','pushbutton','String','Button 1',...
               'Position',[50 20 60 40]);
```
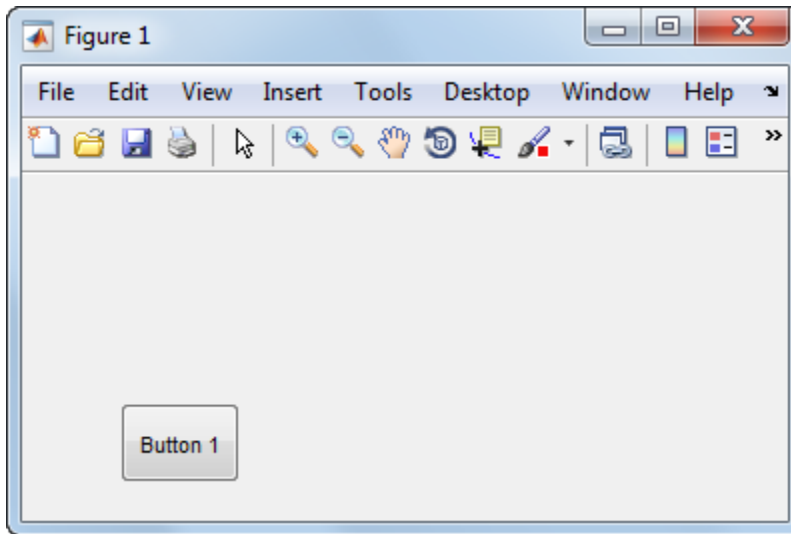
The first `uicontrol` argument, `f`, specifies the parent container. In this case, the parent is a figure, but you can also specify the parent to be any container, such as a panel or button group.

The name-value pair arguments, `'Style','pushbutton'`, the uicontrol to be a push button.

`'String','Button 1'` add the label, **Button 1** to the push button.

`'Position',[50 20 60 40]` specifies the location and size of the push button. In this example, the push button is 60 pixels wide and 40 high. It is positioned 50 pixels from the left of the figure and 20 pixels from the bottom.

Displaying an Icon on a Push Button

To add an icon to a push button, assign the button's `CData` property to be an m-by-n-by-3 array of RGB values that define a truecolor image.

Radio Button

This code creates a radio button:

```
f = figure;
r = uicontrol(f,'Style','radiobutton',...
```

```
                    'String','Indent nested functions.',...
                    'Value',1,'Position',[30 20 150 20]);
```
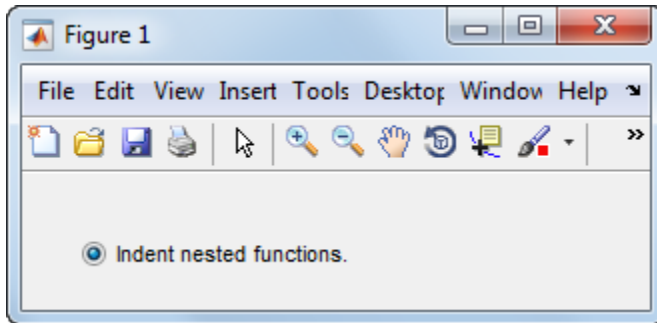

The first `uicontrol` argument, `f`, specifies the parent container. In this case, the parent is a figure, but you can also specify the parent to be any container, such as a panel or button group. If you have multiple radio buttons, you can manage their selection by specifying the parent to be a button group. See "Button Groups" on page 9-21 for more information.

The name-value pair arguments, `'Style','radiobutton'` specifies the uicontrol to a radio button.

`'String','Indent nested functions.'` specifies a label for the radio button.

`'Value',1` selects the radio button by default. Set the `Value` property to be the value of the `Max` property to select the radio button. Set the value to be the value of the `Min` property to deselect the radio button. The default values of `Max` and `Min` are 1 and 0, respectively.

`'Position',[30 20 150 20]` specifies the location and size of the radio button. In this example, the radio button is 150 pixels wide and 20 high. It is positioned 30 pixels from the left of the figure and 20 pixels from the bottom.

Toggle Button

This code creates a toggle button:

```
f = figure;
tb = uicontrol(f,'Style','togglebutton',...
                    'String','Left/Right Tile',...
                    'Value',0,'Position',[30 20 100 30]);
```
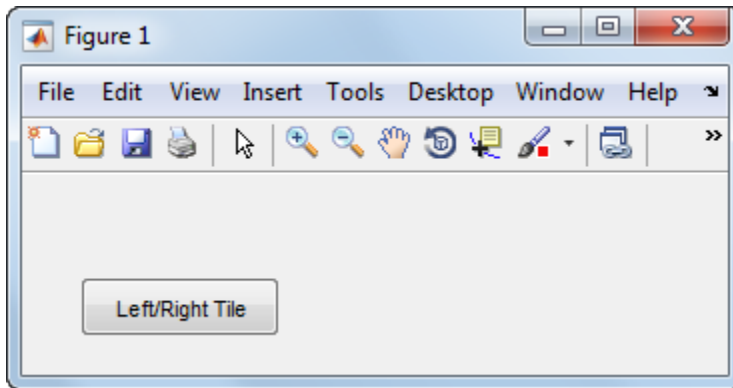
The first `uicontrol` argument, `f`, specifies the parent container. In this case, the parent is a figure, but you can also specify the parent to be any container, such as a panel or button group.

The name-value pair arguments, `'Style','togglebutton'`, specify the uicontrol to be a toggle button.

`'String','Left/Right Tile'` puts a text label on the toggle button.

The `'Value',0` deselects the toggle button by default. To select (raise) the toggle button, set `Value` equal to the `Max` property. To deselect the toggle button, set `Value` equal to the `Min` property. By default, `Min = 0` and `Max = 1`.

`'Position',[30 20 100 30]` specifies the location and size of the toggle button. In this example, the toggle button is 100 pixels wide and 30 pixels high. It is positioned 30 pixels from the left of the figure and 20 pixels from the bottom.

Note: You can also display an icon on a toggle button. See "Displaying an Icon on a Push Button" on page 9-7 for more information.

Check Box

This code creates a check box:

```
f = figure;
c = uicontrol(f,'Style','checkbox',...
                'String','Display file extension',...
```

```
                        'Value',1,'Position',[30 20 130 20]);
```
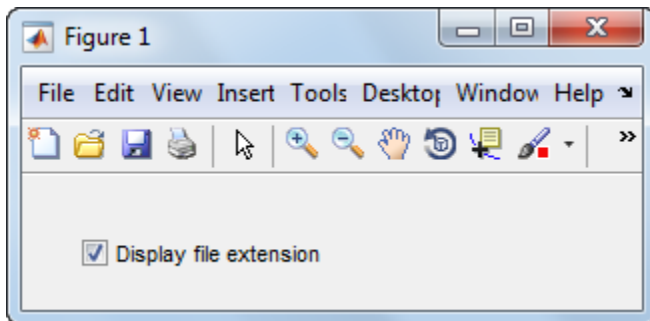

The first `uicontrol` argument, `f`, specifies the parent container. In this case, the parent is a figure, but you can also specify the parent to be any container, such as a panel or button group.

The name-value pair arguments, `'Style','checkbox'`, specify the uicontrol to be a check box.

The next pair, `'String','Display file extension'` puts a text label on the check box.

The `Value` property specifies whether the box is checked. Set `Value` to the value of the `Max` property (default is `1`) to create the component with the box checked. Set `Value` to `Min` (default is `0`) to leave the box unchecked. Correspondingly, when the user clicks the check box, MATLAB sets `Value` to `Max` when the user checks the box and to `Min` when the user unchecks it.

The `Position` property specifies the location and size of the check box. In this example, the check box is 130 pixels wide and 20 high. It is positioned 30 pixels from the left of the figure and 20 pixels from the bottom.

Slider

This code creates a slider:

```
f = figure;
s = uicontrol(f,'Style','slider',...
                'Min',0,'Max',100,'Value',25,...
                'SliderStep',[0.05 0.2],...
                'Position',[30 20 150 30]);
```
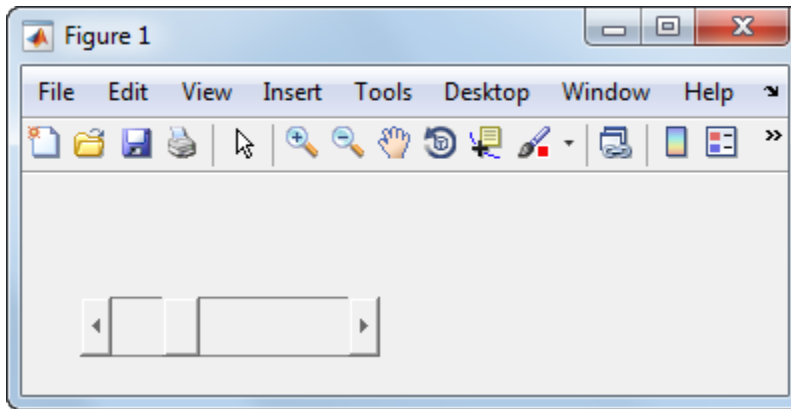
The first `uicontrol` argument, `f`, specifies the parent container. In this case, the parent is a figure, but you can also specify the parent to be any container, such as a panel or button group.

The name-value pair arguments, `'Style','slider'` specifies the uicontrol to be a slider.

`'Min',0` and `'Max',100` specify the range of the slider to be [0, 100]. The `Min` property must be less than `Max`.

`'Value',25` sets the default slider position to 25. The number you specify for this property must be within the range, [`Min`, `Max`].

`'SliderStep',[0.05 0.2]` specifies the fractional amount that the thumb moves when the user clicks the arrow buttons or the slider trough (also called the channel). In this case, the slider's thumb position changes by the smaller amount (5 percent) when the user clicks an arrow button. It changes by the larger amount (20 percent) when the user clicks the trough.

Specify `SliderStep` to be a two-element vector, [`minor_step major_step`]. The value of `minor_step` must be less than or equal to `major_step`. To ensure the best results, do not specify either value to be less than `1e-6`. Setting `major_step` to `1` or higher causes the thumb to move to `Max` or `Min` when the trough is clicked.

As `major_step` increases, the thumb grows longer. When `major_step` is 1, the thumb is half as long as the trough. When `major_step` is greater than 1, the thumb continues to grow, slowly approaching the full length of the trough. When a slider serves as a scroll

bar, you can uses this behavior to indicate how much of the document is currently visible by changing the value of `major_step`.
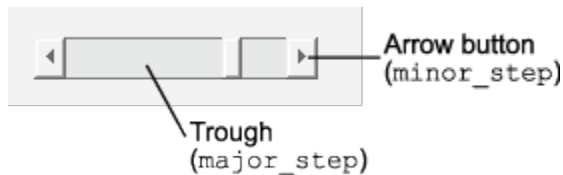
`'Position',[30 20 150 30]` specifies the location and size of the slider. In this example, the slider is 150 pixels wide and 30 high. It is positioned 30 pixels from the left of the figure and 20 pixels from the bottom.

Note: On Mac platforms, the height of a horizontal slider is constrained. If the height you set in the `Position` property exceeds this constraint, the displayed height of the slider is the maximum allowed by the system. However, the value of the `Position` property does not change to reflect this constraint.

Static Text

This code creates a static text component:

```
f = figure;
t = uicontrol(f,'Style','text',...
                'String','Select a data set.',...
                'Position',[30 50 130 30]);
```
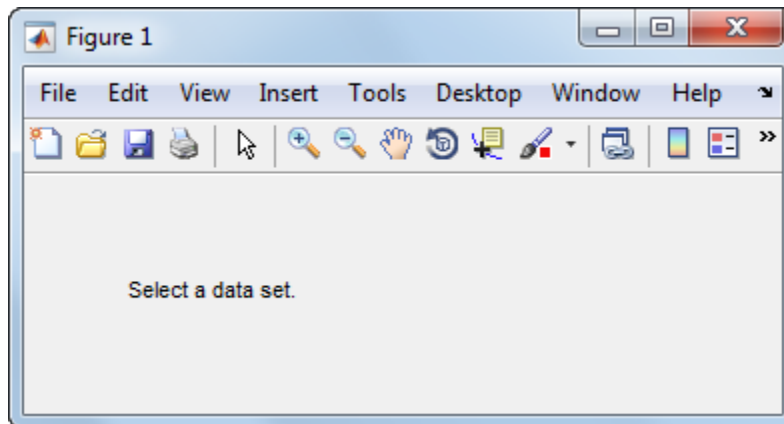
The first `uicontrol` argument, `f`, specifies the parent container. In this case, the parent is a figure, but you can also specify the parent to be any container, such as a panel or button group.

The name-value pair arguments, `'Style'`,`'text'` specify the uicontrol to be static text.

`'String'`,`'Select a set'` specifys the text that displays. If you specify a component width that is too small to accommodate the specified `String`, MATLAB wraps the string.

`'Position'`,`[30 50 130 30]` specifies the location and size of the static text. In this example, the static text is 130 pixels wide and 20 high. It is positioned 30 pixels from the left of the figure and 50 pixels from the bottom.

Editable Text Field

This code creates an editable text field, `txtbox`:

```
f = figure;
txtbox = uicontrol(f,'Style','edit',...
                'String','Enter your name here.',...
                'Position',[30 50 130 20]);
```
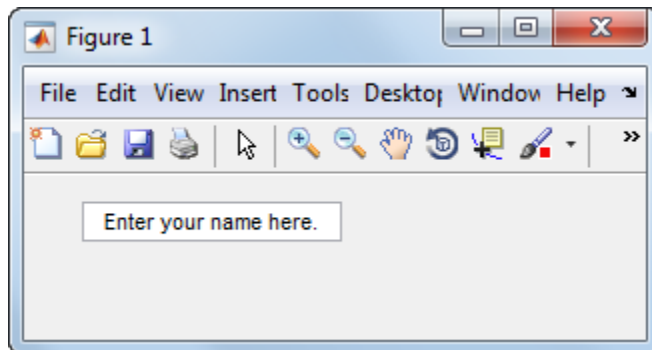

The first `uicontrol` argument, `f`, specifies the parent container. In this case, the parent is a figure, but you can also specify the parent to be any container, such as a panel or button group.

The name-value pair arguments, `'Style'`,`'edit'`, specify the style of the uicontrol to be an editable text field.

`'String','Enter your name here'`, specifies the default text to display.

The next pair, `'Position',[30 50 130 20]` specifies the location and size of the text field. In this example, the text field is 130 pixels wide and 20 pixels high. It is positioned 30 pixels from the left of the figure and 50 pixels from the bottom.

To enable multiple-line input, the value of `Max - Min` must be greater than 1, as in the following statement.

```
txtbox = uicontrol(f,'Style','edit',...
                'String','Enter your name and address here.',...
                'Max',2,'Min',0,...
                'Position',[30 20 130 80]);
```
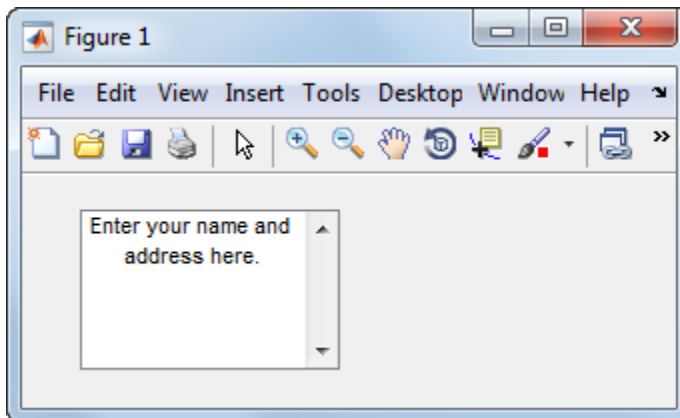

If the value of `Max - Min` is less than or equal to 1, the editable text field allows only a single line of input. If the width of the text field is too narrow for the string, MATLAB displays only part of the string. The user can use the arrow keys to move the cursor over the entire string.
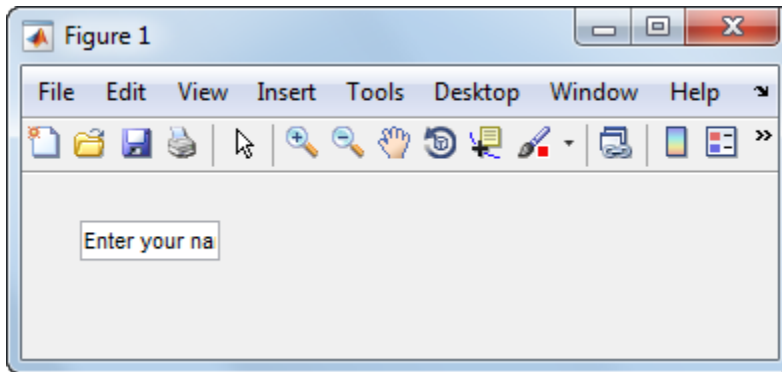
Pop-Up Menu

This code creates a pop-up menu:

```
f = figure;
pm = uicontrol(f,'Style','popupmenu',...
                  'String',{'one','two','three','four'},...
                  'Value',1,'Position',[30 80 130 20]);
```
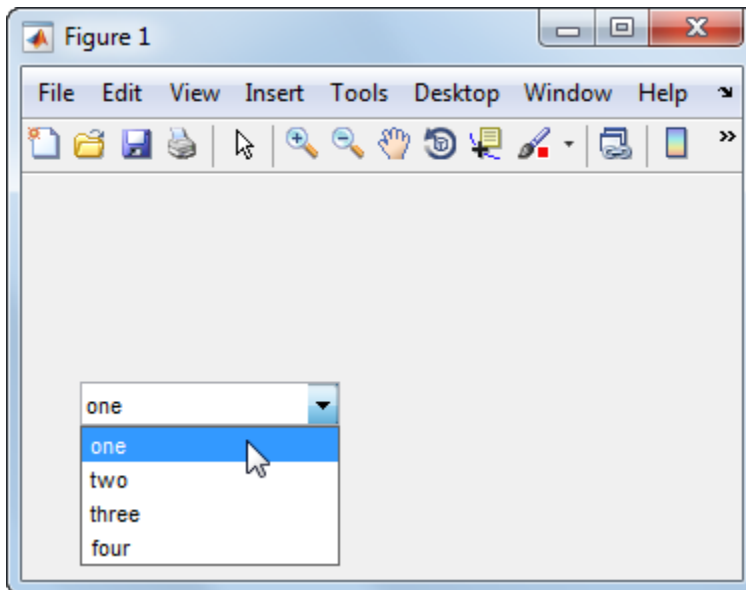
The first `uicontrol` argument, `f`, specifies the parent container. In this case, the parent is a figure, but you can also specify the parent to be any container, such as a panel or button group.

The name-value pair arguments, `Style,'popupmenu'`, specify the uicontrol to be a pop-up menu.

`'String',{'one','two','three','four'}` defines the menu items.

`'Value',1` sets the index of the item that is selected by default. Set `Value` to a scalar that indicates the index of the selected item. A value of 1 selects the first item.

`'Position',[30 80 130 20]` specifies the location and size of the pop-up menu. In this example, the pop-up menu is 130 pixels wide. It is positioned 30 pixels from the left of the figure and 80 pixels from the bottom. The height of a pop-up menu is determined by the font size; the height you set in the position vector is ignored.

List Box

This code creates a list box:

```
f = figure;
lb = uicontrol(f,'Style','listbox',...
                 'String',{'one','two','three','four'},...
                 'Position',[30 20 130 80],'Value',1);
```
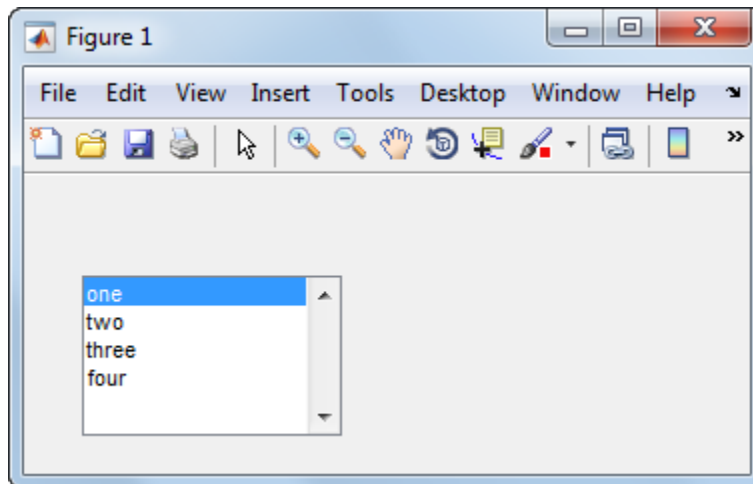
The first `uicontrol` argument, `f`, specifies the parent container. In this case, the parent is a figure, but you can also specify the parent to be any container, such as a panel or button group.

The name-value pair arguments, `'Style','listbox'`, specify the uicontrol to be a list box.

`'String',{'one','two','three','four'}` defines the list items.

`'Position',[30 20 130 80]` specifies the location and size of the list box. In this example, the list box is 130 pixels wide and 80 high. It is positioned 30 pixels from the left of the figure and 20 pixels from the bottom.

The final pair of arguments, `Value,1` sets the list selection to the first item in the list. To select a single item, set the `Value` property to be a scalar that indicates the position of the item in the list.

To select more than one item, set the `Value` property to be a vector of values. To enable your users to select multiple items, set the values of the `Min` and `Max` properties such that `Max - Min` is greater than 1. Here is a list box that allows multiple selections and has two items selected initially:

```
lb = uicontrol(f,'Style','listbox',...
               'String',{'one','two','three','four'},...
               'Max',2,'Min',0,'Value',[1 3],...
               'Position',[30 20 130 80]);
```
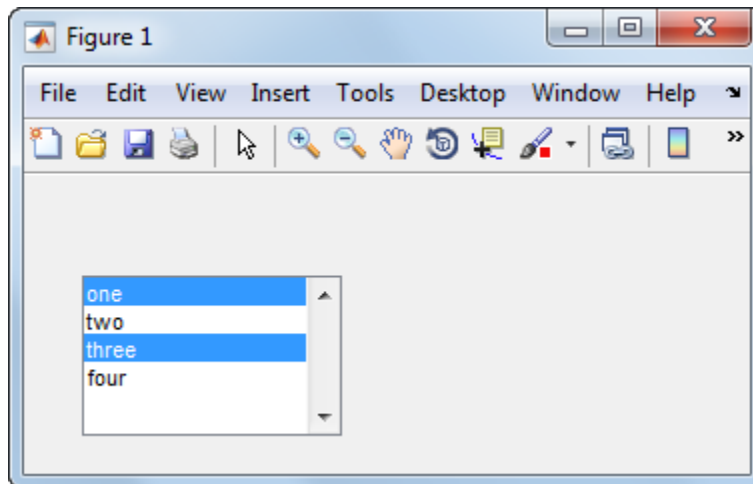
If you want no initial selection, set these property values:

- Set the `Max` and `Min` properties such that `Max - Min` is greater than 1.
- Set the `Value` property to an empty matrix `[]`.

If the list box is not large enough to display all list entries, you can set the `ListBoxTop` property to the index of the item you want to appear at the top when the component is created.

Tables

This code creates a table and populates it with the values returned by `magic(5)`.
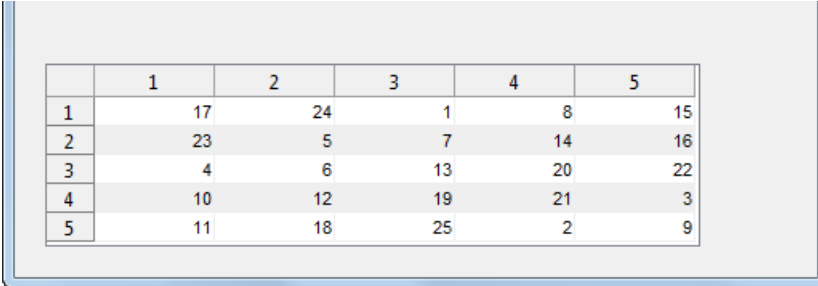
```
f = figure;
tb = uitable(f,'Data',magic(5));
```

The first `uitable` argument, `f`, specifies the parent container. In this case, the parent is a figure, but you can also specify the parent to be any container, such as a panel or button group.

The name-value pair arguments, `'Data',magic(5)`, specifies the table data. In this case, the data is the 5-by-5 matrix returned by the `magic(5)` command.

You can adjust the width and height of the table to accommodate the extent of the data. The uitable's `Position` property controls the outer bounds of the table, and the `Extent` property indicates the extent of the data. Set the last two values in the `Position` property to the corresponding values in the `Extent` property:

```
tb.Position(3) = tb.Extent(3);
tb.Position(4) = tb.Extent(4);
```

	1	2	3	4	5
1	17	24	1	8	15
2	23	5	7	14	16
3	4	6	13	20	22
4	10	12	19	21	3
5	11	18	25	2	9

You can change several other characteristics of the table by setting certain properties:

- To control the user's ability to edit the table cells, set the `ColumnEditable` property.
- To make your application respond when the user edits a cell, define a `CellEditCallback` function.
- To add or change row striping, set the `RowStriping` property.
- To specify row and column names, set the `RowName` and `ColumnName` properties.
- To format the data in the table, set the `ColumnFormat` property.

See Uitable Properties for the entire list of properties.

If you are building a UI using GUIDE, you can set many of the uitable properties using the **Table Property Editor**. For more information, see "Create a Table" on page 6-50.

Panels

This code creates a panel:

```
f = figure;
p = uipanel(f,'Title','My Panel',...
             'Position',[.25 .1 .5 .8]);
```
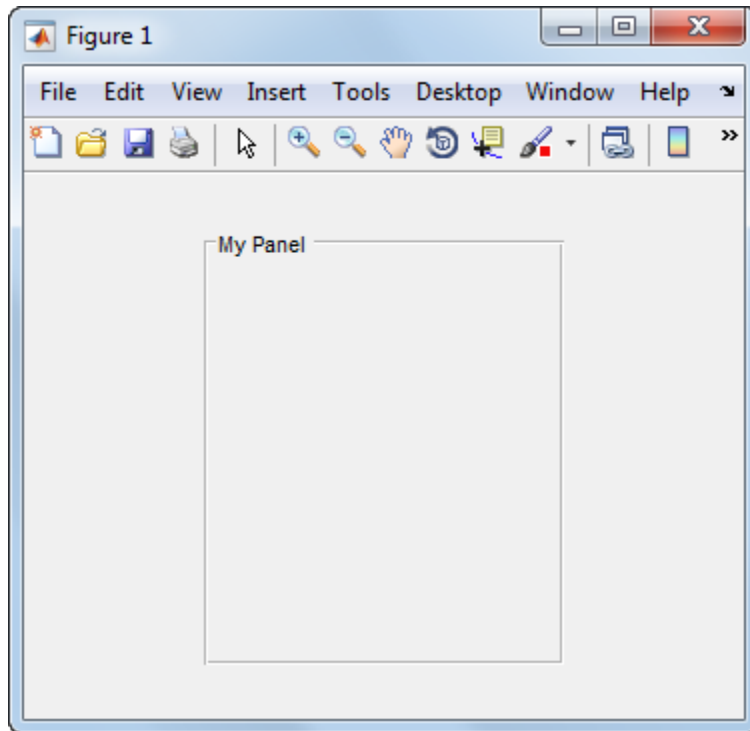
The first argument passed to `uipanel`, `f`, specifies the parent container. In this case, the parent is a figure, but you can also specify the parent to be any container, such as another panel or a button group.

`'Title','My Panel'` specifies a title to display on the panel.

`'Position',[.25 .1 .5 .8]` specifies the location and size of the panel as a fraction of the parent container. In this case, the panel is 50 percent of the width of the figure and 80 percent of its height. The left edge of the panel is located at 25 percent of the figure's width from the left. The bottom of the panel is located 10 percent of the figure's height from the bottom. If the figure is resized, the panel retains its original proportions.

The following commands add two push buttons to the panel. Setting the `Units` property to `'normalized'` causes the `Position` values to be interpreted as fractions of the parent panel. Normalized units allow the buttons to retain their original proportions when the panel is resized.

```
b1 = uicontrol(p,'Style','pushbutton','String','Button 1',...
                 'Units','normalized',...
                 'Position',[.1 .55 .8 .3]);
b2 = uicontrol(p,'Style','pushbutton','String','Button 2',...
                 'Units','normalized',...
                 'Position',[.1 .15 .8 .3]);
```
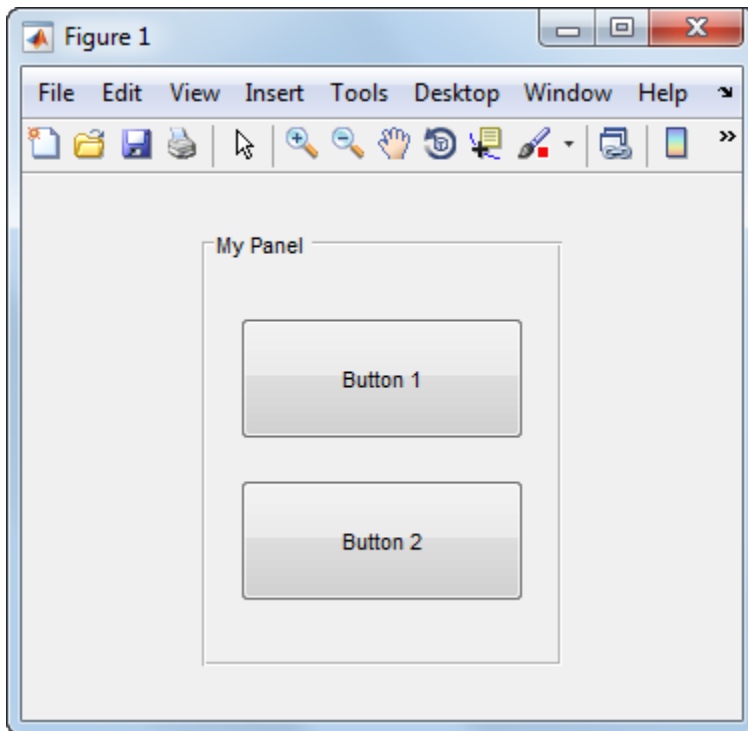

Button Groups

This code creates a button group:

```
f = figure;
bg = uibuttongroup(f,'Title','My Button Group',...
            'Position',[.1 .2 .8 .6]);
```
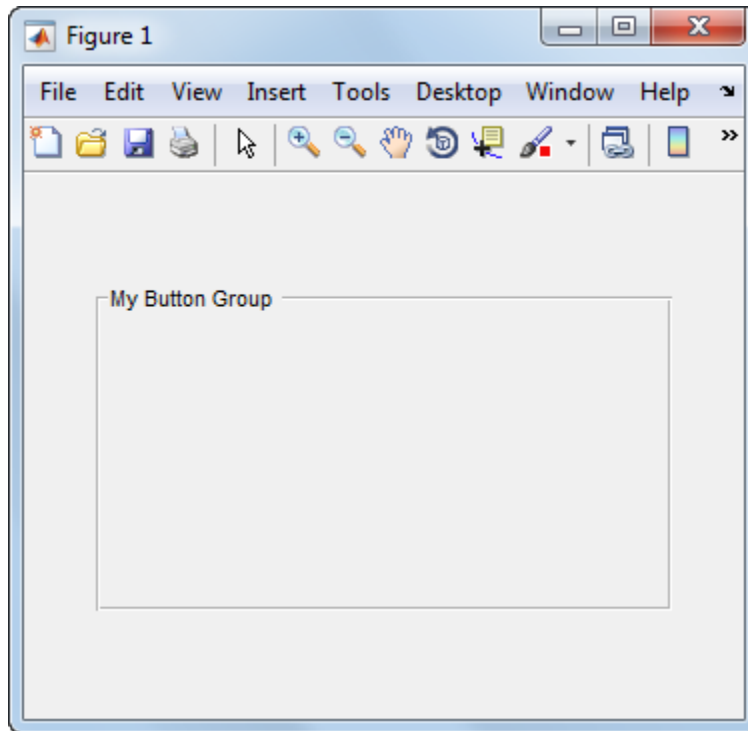
The first argument passed to uibuttongroup, f, specifies the parent container. In this case, the parent is a figure, but you can also specify the parent to be any container, such as a panel or another button group.

'Title','My Button Group' specifies a title to display on the button group.

'Position',[.1 .2 .8 .6] specifies the location and size of the button group as a fraction of the parent container. In this case, the button group is 80 percent of the width of the figure and 60 percent of its height. The left edge of the button group is located at 10 percent of the figure's width from the left. The bottom of the button group is located 20 percent of the figure's height from the bottom. If the figure is resized, the button group retains its original proportions.

The following commands add two radio buttons to the button group. Setting the Units property to 'normalized' causes the Position values to be interpreted as fractions of the parent panel. Normalized units allow the buttons to retain their original relative positions when the button group is resized.

```
rb1 = uicontrol(bg,'Style','radiobutton','String','Red',...
                'Units','normalized',...
                'Position',[.1 .6 .3 .2]);
rb2 = uicontrol(bg,'Style','radiobutton','String','Blue',...
                'Units','normalized',...
                'Position',[.1 .2 .3 .2]);
```
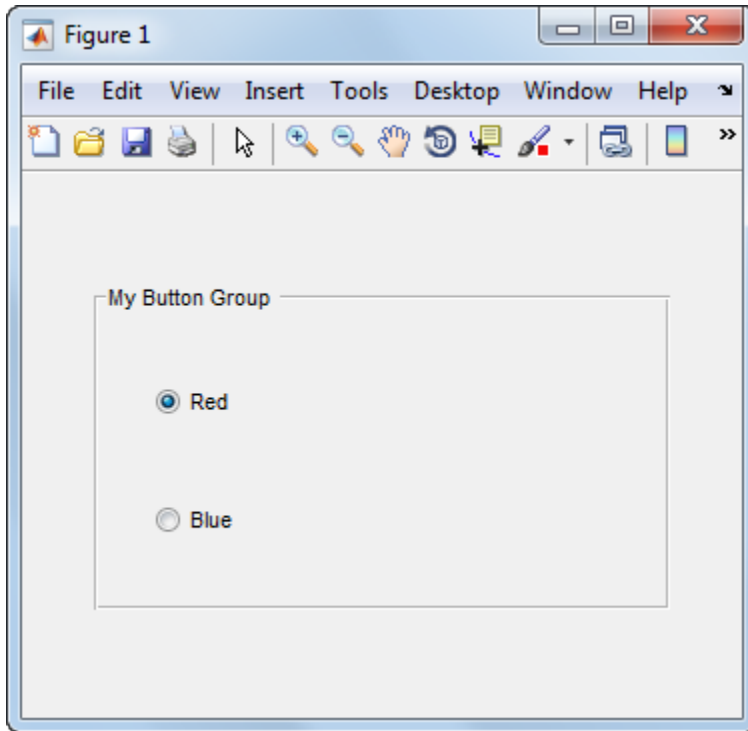

By default, the first radio button added to the uibuttongroup is selected. To override this default, set any other radio button's `Value` property to its `Max` property value.

Button groups manage the selection of radio buttons and toggle buttons by allowing only one button to be selected within the group. You can determine the currently selected button by querying the uibuttongroup's `SelectedObject` property.

Axes

This code creates an axes in a figure:

```
f = figure;
ax = axes('Parent',f,'Position',[.15 .15 .7 .7]);
```
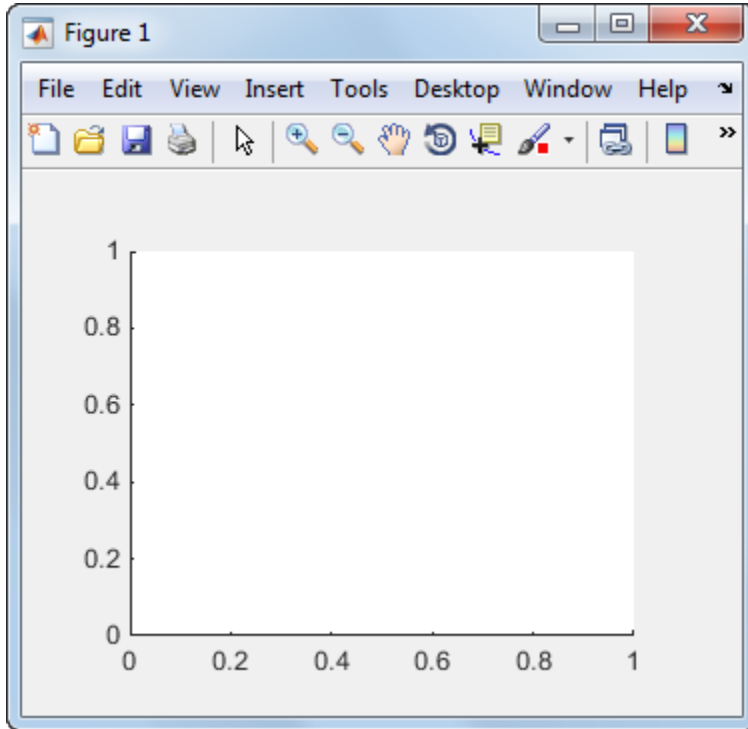

The first two arguments passed to the axes function, `'Parent',f` specify the parent container. In this case, the parent is a figure, but you can also specify the parent to be any container, such as a panel or button group.

`'Position',[.15 .15 .7 .7]` specifies the location and size of the axes as a fraction of the parent figure. In this case, the axes is 70 percent of the width of the figure and 70 percent of its height. The left edge of the axes is located at 15 percent of the figure's width from the left. The bottom of the axes is located 15 percent of the figure's height from the bottom. If the figure is resized, the axes retains its original proportions.

Prevent Customized Axes Properties from Being Reset

Data graphing functions, such as `plot`, `image`, and `scatter`, reset axes properties before they draw into an axes. This can be a problem when you want to maintain consistency of axes limits, ticks, colors, and font characteristics in a UI.

The default value of the NextPlot axes property, 'replace' allows the graphing functions to reset many property values. In addition, the 'replace' property value allows MATLAB to remove all callbacks from the axes whenever a graph is plotted. If you create an axes in a UI window, consider setting the NextPlot property to 'replacechildren'. You might need to set this property prior to changing the contents of an axes:

```
ax.NextPlot = 'replacechildren';
```

ActiveX Controls

ActiveX components enable you to display ActiveX controls in your UI. They are available only on the Microsoft Windows platform.

An ActiveX control can be the child only of a figure. It cannot be the child of a panel or button group.

See "Creating an ActiveX Control" about adding an ActiveX control to a figure. See "Create COM Objects" for general information about ActiveX controls.

How to Set Font Characteristics

Use the FontName property to specify a particular font for a user interface control, panel, button group, table, or axes.

Use the uisetfont function to display a dialog that allows you to choose a font, style, and size all at once:

```
myfont = uisetfont
```
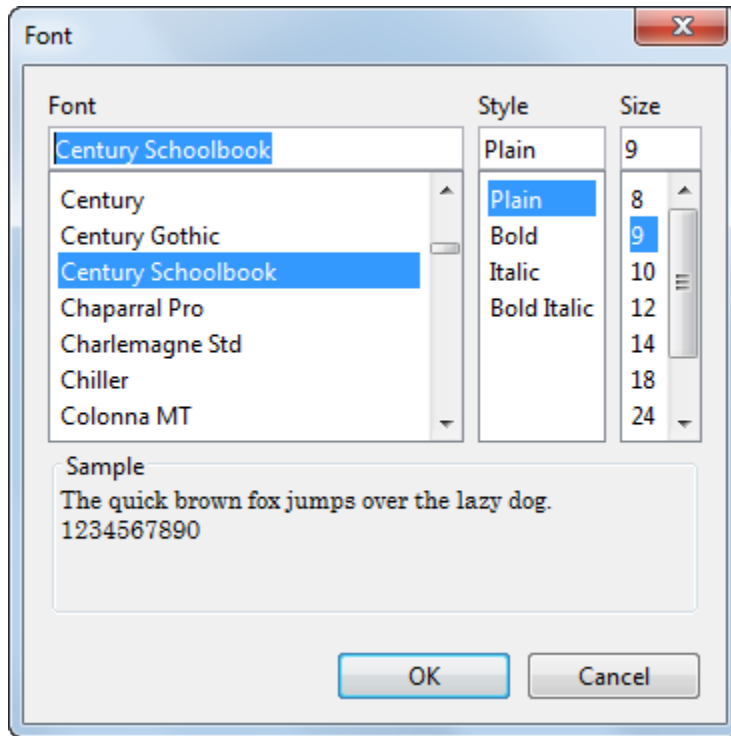
uisetfont returns the selections as a structure array:

```
myfont =
      FontName: 'Century Schoolbook'
    FontWeight: 'normal'
     FontAngle: 'normal'
      FontSize: 9
     FontUnits: 'points'
```

You can use this information to set font characteristics of a component in the UI:

```
btn = uicontrol;
btn.FontName = myfont.FontName;
btn.FontSize = myfont.FontSize;
```

Alternatively, you can set all the font characteristics at once:

```
set(btn,myfont);
```

Related Examples

- "Callbacks for Specific Components" on page 7-12
- "Write Callbacks for Apps Created Programmatically" on page 10-5

Lay Out a UI Programmatically

You can adjust the size and location of components, and manage front-to-back order of grouped components by setting certain property values. This topic explains how to use these properties to get the layout you want. It also explains how to use the `SizeChangedFcn` callback to control the UI's resizing behavior.

In this section...
"Component Placement and Sizing" on page 9-28
"Managing the Layout in Resizable UIs" on page 9-33
"Manage the Stacking Order of Grouped Components" on page 9-36

Component Placement and Sizing

A UI layout consists of a figure and one or more components that you place inside the figure. Accurate placement and sizing of each component involves setting certain properties and understanding how the inner and outer boundaries of the figure relate to each other.

Location and Size of Outer Bounds and Drawable Area

The area inside the figure, which contains the UI components, is called the *drawable area*. The drawable area is inside the outer bounds of the figure, but does not include the menu bar or tool bar. You can control the location and size of the drawable area by setting the figure's Position property as a four-element row vector. The first two elements of this vector specify the location. The last two elements specify the size. By default, the figure's `Position` values are in pixels.

This command creates a figure and sets the `Position` value. The left edge of the drawable area is 258 pixels from the left side of the screen. Its bottom edge is 132 pixels up from the bottom of the screen. Its size is 560 pixels wide by 420 pixels high:

```
f = figure('Position',[258 132 560 420]);
```
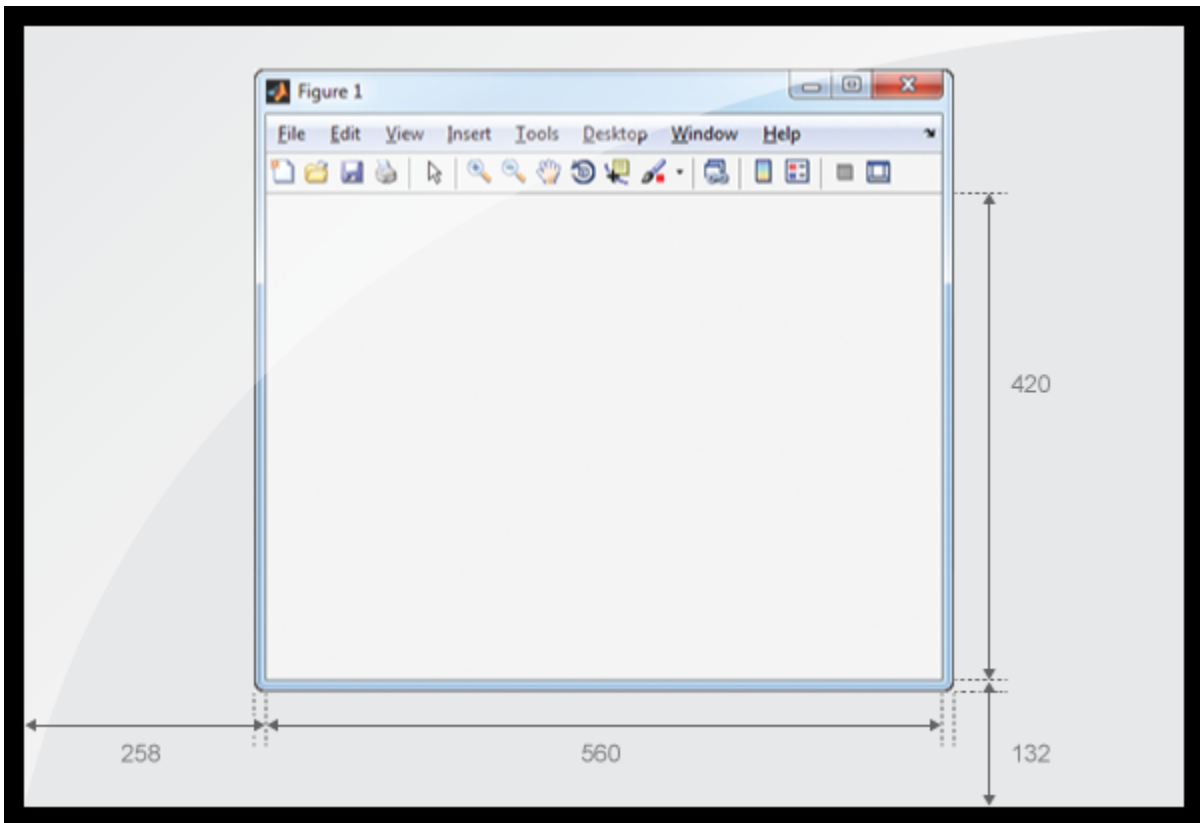
You can query or change the outer bounds of the figure by using the OuterPosition property. Like the `Position` property, the `OuterPosition` is a four element row vector:

```
f.OuterPosition
```

```
ans =
```

    250   124   576   512

The left outer edge of this figure is 250 pixels from the left side of the screen. Its bottom outer edge is 124 pixels up from the bottom of the screen. The area enclosed by the outer bounds of the figure is 576 pixels wide by 512 pixels high.
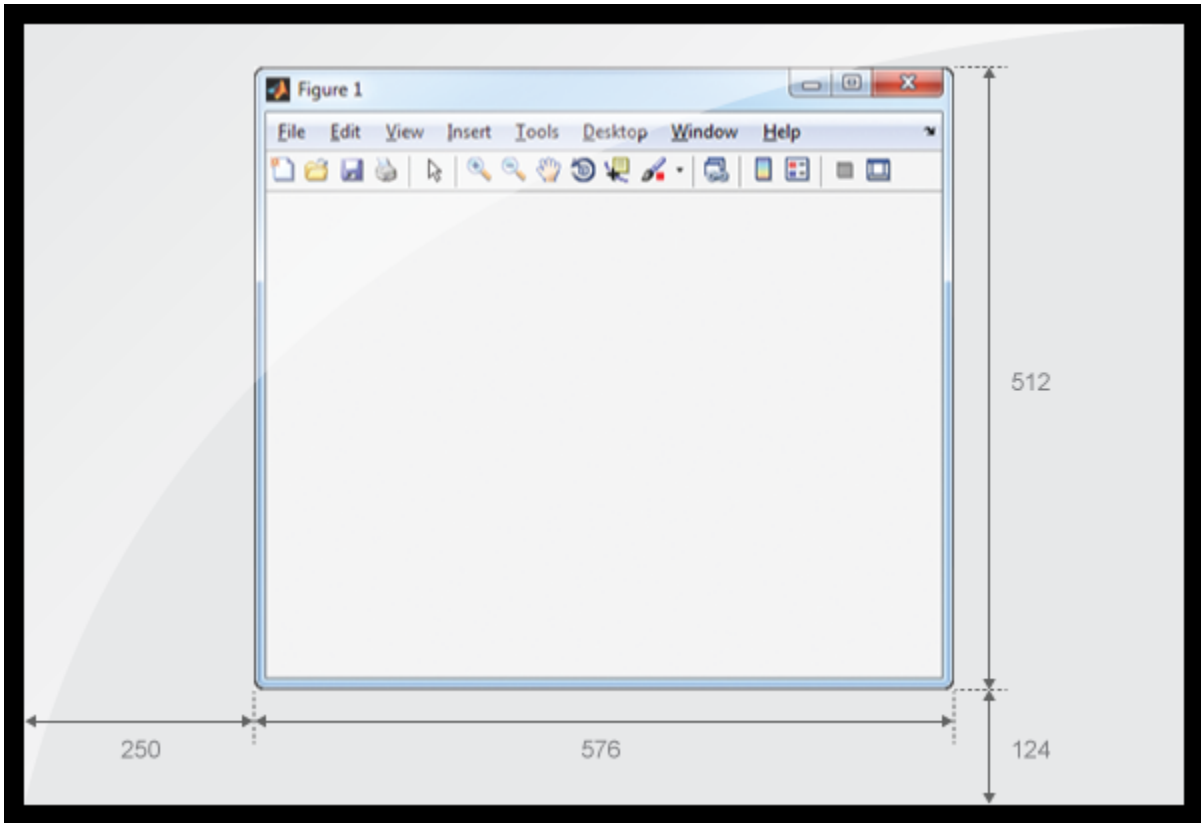
Explicitly changing the `Position` or `OuterPosition` causes the other property to change. For example, this is the current `Position` value of `f`:

```
f.Position
```

```
ans =
```

```
   258   132   560   420
```
Changing the `OuterPosition` causes the `Position` to change:

```
f.OuterPosition = [250 250 490 340];
f.Position
```

```
ans =
```

```
258    258    474    248
```

Other UI components, such as uicontrols, uitables, and uipanels have a `Position` property, which you can use to set their location and size.

Units of Measure

The default units associated with the `Position` property depend on the component you are placing. However, you can change the Units property to lay out your UI in the units of your choice. There are six different units of measure to choose from: inches, centimeters, normalized, points, pixels, and characters.

Always specify `Units` before `Position` for the most predictable results.

```
f = figure('Units','inches','Position',[4 3 6 5]);
```

Your choice of units can affect the appearance and resizing behavior of the UI:

- If you want the UI components to scale proportionally with the figure when the user resizes the figure, set the `Units` property of the components to `'normalized'`.
- UI Components do not scale proportionally inside the figure when their `Units` property is set to `'inches'`, `'centimeters'`, `'points'`, `'pixels'`, or `'characters'`.
- If you are developing a cross-platform UI, then set the `Units` property to `'points'` or `'characters'` to make the layout consistent across all platforms.

Example of a Simple Layout

Here is the code for a simple UI containing an axes and a button. To see how it works, copy and paste this code into the editor and run it.

```
function myui
   % Add the UI components
   hs = addcomponents;

   % Make figure visible after adding components
   hs.fig.Visible = 'on';

   function hs = addcomponents
      % add components, save handles in a struct
      hs.fig = figure('Visible','off',...
```

```
                    'Resize','off',...
                    'Tag','fig');
        hs.btn = uicontrol(hs.fig,'Position',[10 340 70 30],...
                    'String','Plot Sine',...
                    'Tag','button',...
                    'Callback',@plotsine);
        hs.ax = axes('Parent',hs.fig,...
                    'Position',[0.20 0.13 0.71 0.75],...
                    'Tag','ax');
    end

    function plotsine(hObject,callbackdata)
        theta = 0:pi/64:6*pi;
        y = sin(theta);
        plot(hs.ax,theta,y);
    end
end
```

This code performs the following tasks:

- The main function, myui, calls the addcomponents function. The addcomponents function returns a structure, hs, containing the handles to all the UI components.

- The addcomponents function creates a figure, an axes, and a button, each with specific Position values.

  - Notice that the Resize property of the figure is 'off'. This value disables the resizing capability of the figure.

  - Notice that the Visible property of the figure is 'off' inside the addcomponents function. The value changes to 'on' after addcomponents returns to the calling function. Doing this delays the figure display until after MATLAB adds all the components. Thus, the resulting UI has a clean appearance when it starts up.

- The plotsine function plots the sine function inside the axes when the user clicks the button.
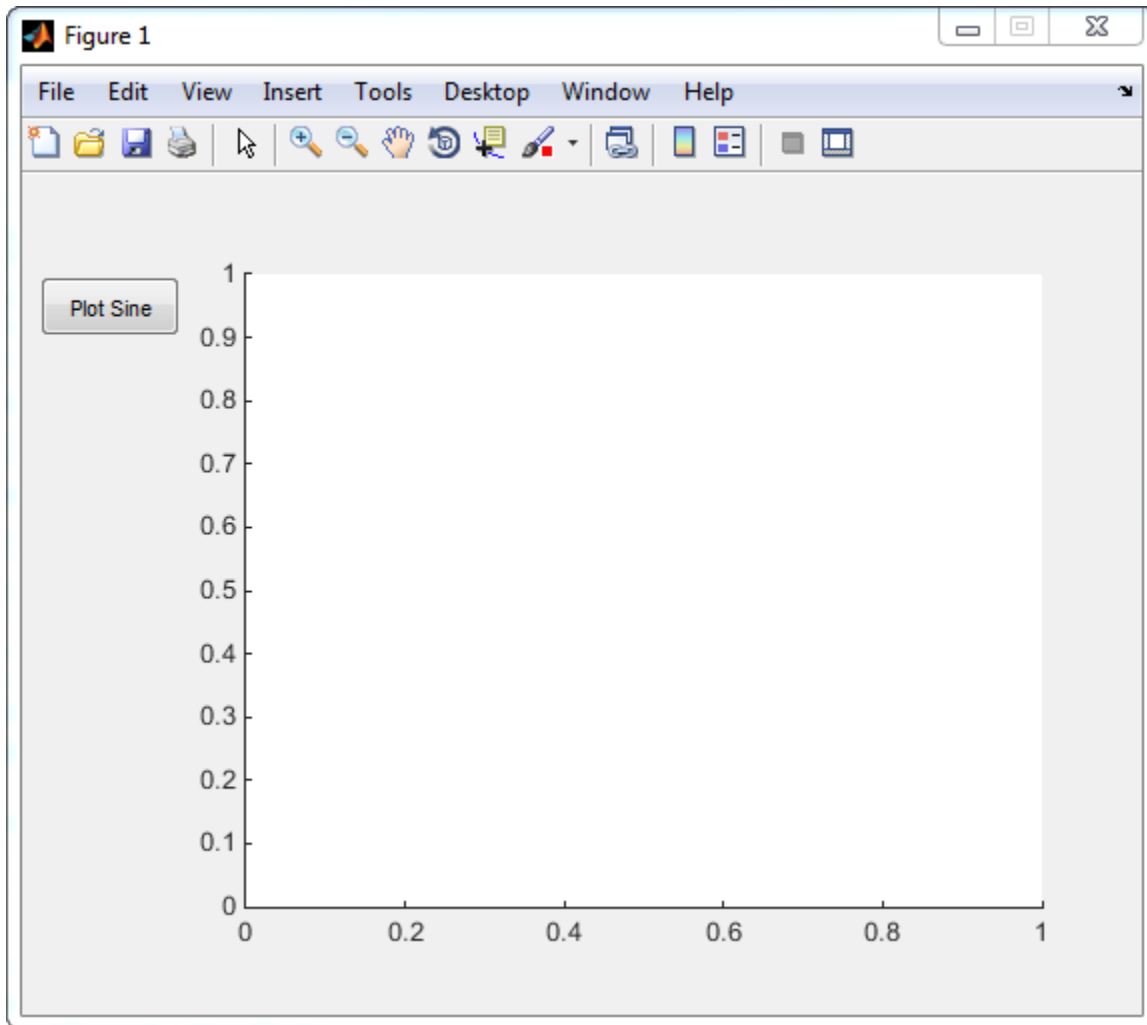
Managing the Layout in Resizable UIs

To create a resizable UI and manage the layout when the user resizes the window, set the figure's SizeChangedFcn property to be a handle to a callback function. Code the callback function to manage the layout when the window size changes.

If your UI has another container, such as a uipanel or uibuttongroup, you can manage the layout of the container's child components in a separate callback function that you assign to the `SizeChangedFcn` property.

The `SizeChangedFcn` callback executes only under these circumstances:

- The container becomes visible for the first time.
- The container is visible while its drawable area changes.
- The container becomes visible for the first time after its drawable area changes. This situation occurs when the drawable area changes while the container is invisible and becomes visible later.

Note: Typically, the drawable area changes at the same time the outer bounds change. However, adding or removing menu bars or tool bars to a figure causes the outer bounds to change while the drawable area remains constant. Therefore, the `SizeChangedFcn` callback does not execute when you add or remove menu bars or tool bars.

This UI is a resizable version of the simple UI defined in "Example of a Simple Layout" on page 9-31. This code includes a figure `SizeChangedFcn` callback called `resizeui`. The `resizeui` function calculates new `Position` values for the button and axes when the user resizes the window. The button appears to be stationary when the user resizes the window. The axes scales with the figure.

```
function myui
    % Add the UI components
    hs = addcomponents;

    % Make figure visible after adding components
    hs.fig.Visible = 'on';

    function hs = addcomponents
        % Add components, save handles in a struct
        hs.fig = figure('Visible','off',...
                   'Tag','fig',...
                   'SizeChangedFcn',@resizeui);
        hs.btn = uicontrol(hs.fig,'String',...
                   'Plot Sine',...
                   'Callback',@plotsine,...
                   'Tag','button');
        hs.ax = axes('Parent',hs.fig,...
```

```
                'Units','pixels',...
                'Tag','ax');
    end

    function plotsine(hObject,callbackdata)
        theta = 0:pi/64:6*pi;
        y = sin(theta);
        plot(hs.ax,theta,y);
    end

    function resizeui(hObject,callbackdata)

        % Get figure width and height
        figwidth = hs.fig.Position(3);
        figheight = hs.fig.Position(4);

        % Set button position
        bheight = 30;
        bwidth = 70;
        bbottomedge = figheight - bheight - 50;
        bleftedge = 10;
        hs.btn.Position = [bleftedge bbottomedge bwidth bheight];

        % Set axes position
        axheight = .75*figheight;
        axbottomedge = max(0,figheight - axheight - 30);
        axleftedge = bleftedge + bwidth + 30;
        axwidth = max(0,figwidth - axleftedge - 50);
        hs.ax.Position = [axleftedge axbottomedge axwidth axheight];
    end
end
```

The `resizeui` function sets the location and size of the button and axes whenever the user resizes the window:

- The button height, width, and left edge stay the same when the window resizes.

- The bottom edge of the button, `bbottomedge`, allows 50 pixels of space between the top of the figure and the top of the button.

- The value of the axes height, `axheight`, is 75% of the available height in the figure.

- The value of the axes bottom edge, `axbottomedge`, allows 30 pixels of space between the top of the figure and the top of the axes. In this calculation, the `max` function limits this value to nonnegative values.

- The value of the axes width, `axwidth`, allows 50 pixels of space between the right side of the axes and the right edge of the figure. In this calculation, the `max` function limits this value to nonnegative values.

Notice that *all* the layout code is inside the `resizeui` function. It is a good practice to put all the layout code inside the `SizeChangedFcn` callback to ensure the most accurate results.

Also, it is important to delay the display of the entire UI window until after all the variables that a `SizeChangedFcn` callback uses are defined. Doing so can prevent the `SizeChangedFcn` callback from returning an error. To delay the display of the window, set the `Visible` property of the figure to `'off'`. After you define all the variables that your `SizeChangedFcn` callback uses, set the `Visible` property to `'on'`.

Manage the Stacking Order of Grouped Components

The default front-to-back order, or stacking order, of components in a UI is as follows:

- Axes and other graphics objects appear behind other components. UI components and containers (uipanels, uibuttongroups, and uitabs) appear in front of them.
- UI components and containers appear in the order in which you create them. New components appear in front of existing components.

You can change the stacking order at any time, but there are some restrictions. Axes and other graphics objects can stack in any order with respect to each other. However, axes and other graphics objects cannot stack in front of UI components and containers. They always appear behind UI components and containers.

You can work around this restriction by grouping graphics objects into separate containers. Then you can stack those containers in any order. To group a graphics object into a container, set its `Parent` property to be that container. For example, you can group an axes into a uipanel by setting the `Parent` property of the axes to be the uipanel.

The `Children` property of a uipanel, uibuttongroup, or uitab lists the child objects inside the container according to their stacking order.

More About

- "DPI-Aware Behavior in MATLAB" on page 9-63

Customize Tabbing Behavior in a Programmatic UI

In this section...
"How Tabbing Works" on page 9-37
"Default Tab Order" on page 9-37
"Change the Tab Order in the uipanel" on page 9-39

How Tabbing Works

The tab order is the order in which components of the UI acquire focus when the user presses the keyboard **Tab** key. Focus is generally denoted by a border or a dotted border.

Tab order is determined separately for the children of each parent. For example, child components of the figure window have their own tab order. Child components of each panel or button group also have their own tab order.

If, in tabbing through the components at one level, a user tabs to a panel or button group, then the tabbing sequences through the components of the panel or button group before returning to the level from which the panel or button group was reached. For example, if a figure window contains a panel that contains three push buttons and the user tabs to the panel, then the tabbing sequences through the three push buttons before returning to the figure.

Note You cannot tab to axes and static text components. You cannot determine programmatically which component has focus.

Default Tab Order

The default tab order for each level is the order in which you create the components at that level.

The following code creates a UI that contains a pop-up menu with a static text label, a panel with three push buttons, and an axes.

```
fh = figure('Position',[200 200 450 270]);
pmh = uicontrol(fh,'Style','popupmenu',...
           'String',{'peaks','membrane','sinc'},...
```

```
            'Position',[290 200 130 20]);
sth = uicontrol(fh,'Style','text','String','Select Data',...
            'Position',[290 230 60 20]);
ph = uipanel('Parent',fh,'Units','pixels',...
            'Position',[290 30 130 150]);
ah = axes('Parent',fh,'Units','pixels',...
            'Position',[40 30 220 220]);
bh1 = uicontrol(ph,'Style','pushbutton',...
            'String','Contour','Position',[20 20 80 30]);
bh2 = uicontrol(ph,'Style','pushbutton',...
            'String','Mesh','Position',[20 60 80 30]);
bh3 = uicontrol(ph,'Style','pushbutton',...
            'String','Surf','Position',[20 100 80 30]);
```
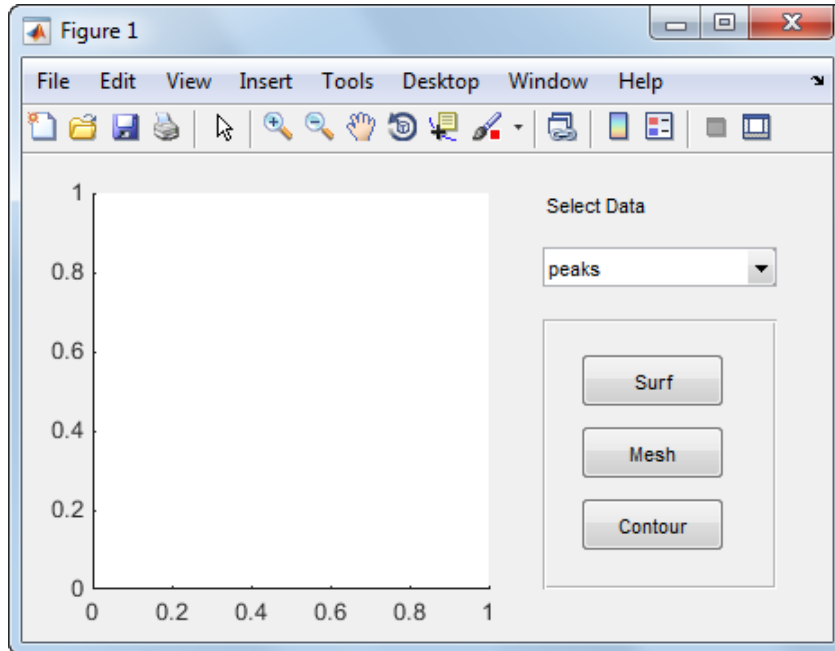

You can obtain the default tab order for a figure, panel, or button group by looking at its Children property. For the example, this command gets the children of the uipanel, ph.

```
ch = ph.Children
```

```
ch =
```

```
3x1 UIControl array:

UIControl    (Surf)
UIControl    (Mesh)
UIControl    (Contour)
```

The default tab order is the reverse of the child order: **Contour**, then **Mesh**, then **Surf**.

Note Displaying the children in this way shows only those children that have their `HandleVisibility` property set to `'on'`. Use `allchild` to retrieve children regardless of their handle visibility.

In this example, the default order is pop-up menu followed by the panel's **Contour**, **Mesh**, and **Surf** push buttons (in that order), and then back to the pop-up menu. You cannot tab to the axes component or the static text component.

Try modifying the code to create the pop-up menu following the creation of the **Contour** push button and before the **Mesh** push button. Now execute the code to create the UI and tab through the components. This code change does not alter the default tab order. This is because the pop-up menu does not have the same parent as the push buttons. The figure is the parent of the panel and the pop-up menu.

Change the Tab Order in the uipanel

Get the `Children` property of the uipanel, and then modify the order of the array elements. This code gets the children of the uipanel and stores it in the variable, `ch`.

```
ch = ph.Children

ch =

  3x1 UIControl array:

  UIControl    (Surf)
  UIControl    (Mesh)
  UIControl    (Contour)
```

Next, call the `uistack` function to change the tab order of buttons. This code moves the **Mesh** button up one level, making it the last item in the tab order.

```
uistack(ch(2),'up',1);
```

The tab order of the three buttons is now **Contour**, then **Surf**, then **Mesh**.

This command shows the new child order.

```
ph.Children

ans =

  3x1 UIControl array:

  UIControl    (Mesh)
  UIControl    (Surf)
  UIControl    (Contour)
```

Note Tab order also affects the stacking order of components. If components overlap, those that appear higher in the child order, display on top of those that appear lower in the order.

Create Menus for Programmatic UIs

In this section...
"Add Menu Bar Menus" on page 9-41
"Add Context Menus to a Programmatic UI" on page 9-48

Add Menu Bar Menus

Use the uimenu function to add a menu bar menu to your UI. A syntax for uimenu is

```
mh = uimenu(parent,'PropertyName',PropertyValue,...)
```

Where mh is the handle of the resulting menu or menu item. See the uimenu reference page for other valid syntaxes.

These topics discuss use of the MATLAB standard menu bar menus and describe commonly used menu properties and offer some simple examples.

- "Display Standard Menu Bar Menus" on page 9-41
- "Commonly Used Properties" on page 9-42
- "How Menus Affect Figure Docking" on page 9-43
- "Menu Bar Menu" on page 9-45

Display Standard Menu Bar Menus

Displaying the standard menu bar menus is optional. This figure's menu bar contains the standard menus.
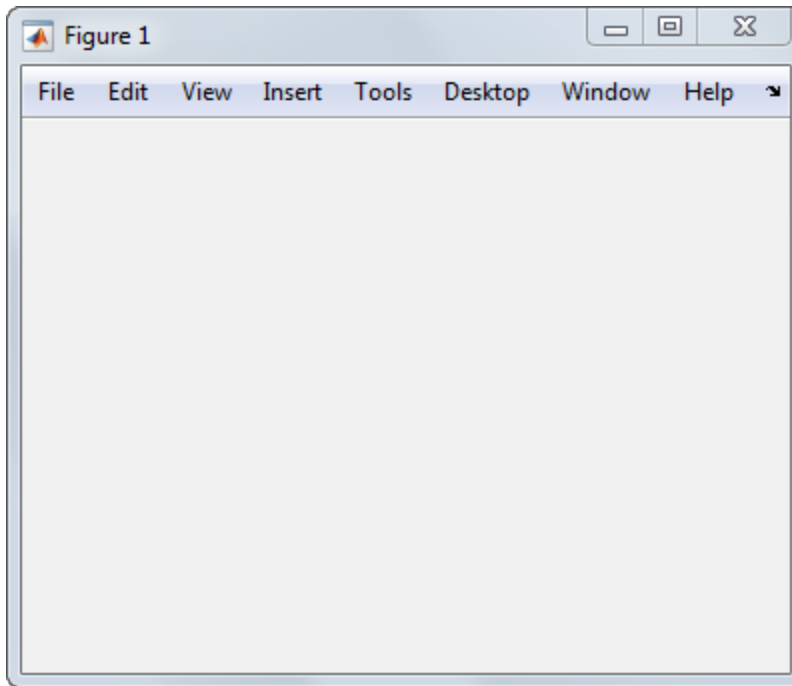
If you use the standard menu bar menus, any menus you create are added to it. If you choose not to display the standard menu bar menus, the menu bar contains only the menus that you create. If you display no standard menus and you create no menus, the menu bar itself does not display.

Use the figure `MenuBar` property to display or hide the MATLAB standard menu bar shown in the preceding figure. Set `MenuBar` to `figure` (the default) to display the standard menus. Set `MenuBar` to `none` to hide them.

```
fh.MenuBar = 'figure';  % Display standard menu bar menus.
fh.MenuBar = 'none';    % Hide standard menu bar menus.
```

In these statements, `fh` is the handle of the figure.

Commonly Used Properties

The most commonly used properties needed to describe a menu bar menu are shown in the following table.

Property	Values	Description
Accelerator	Alphabetic character	Keyboard equivalent. Available for menu items that do not have submenus.
Checked	off, on. Default is off.	Menu check indicator
Enable	on, off. Default is on.	Controls whether a menu item can be selected. When set to off, the menu label appears dimmed.
HandleVisibility	on, off. Default is on.	Determines if an object's handle is visible in its parent's list of children. For menus, set HandleVisibility to off to protect menus from operations not intended for them.
Label	String	Menu label. To display the & character in a label, use two & characters in the string. The words remove, default, and factory (case sensitive) are reserved. To use one of these as a label, prepend a backslash (\) to the string. For example, \remove yields **remove**.
Position	Scalar. Default is 1.	Position of a menu item in the menu.
Separator	off, on. Default is off.	Separator line mode

For a complete list of properties and for more information about the properties listed in the table, see Uimenu Properties.

How Menus Affect Figure Docking

When you customize the menu bar or toolbar, you can control the display of the window's docking controls by setting the DockControls property. You might not need menus for your UI, but if you want the user to be able to dock or undock the UI, it must contain a menu bar or a toolbar. This is because docking is controlled by the docking icon, a small curved arrow near the upper-right corner of the menu bar or the toolbar, as the following illustration shows.
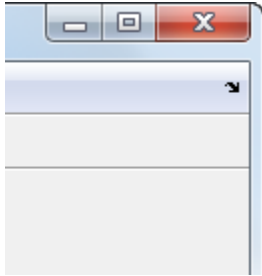
Figure windows with a standard menu bar also have a **Desktop** menu from which the user can dock and undock them.

To display the docking arrow and the **Desktop** > **Dock Figure** menu item, the figure property DockControls must be set to 'on'. You can set this property in the Property Inspector. In addition, the MenuBar and/or ToolBar figure properties must be set to 'on' to display docking controls.

The WindowStyle figure property also affects docking behavior. The default is 'normal', but if you change it to 'docked', then the following applies:

- The UI opens docked in the desktop when you run it.

- The DockControls property is set to 'on' and cannot be turned off until WindowStyle is no longer set to 'docked'.

- If you undock a UI created with WindowStyle set to 'docked', the window will have not have a docking arrow unless the figure displays a menu bar or a toolbar. When the window has no docking arrow, users can undock it from the desktop, but will be unable to redock it.

To summarize, you can display docking controls with the DockControls property as long as it is not in conflict with the figure's WindowStyle property.

Note: Modal dialogs (figures with the WindowStyle property set to 'modal') cannot have menu bars, toolbars, or docking controls.

For more information, see the DockControls, MenuBar, ToolBar, and WindowStyle property descriptions on the Figure Properties page.

Menu Bar Menu

The following statements create a menu bar menu with two menu items.

```
mh = uimenu(fh,'Label','My menu');
eh1 = uimenu(mh,'Label','Item 1');
eh2 = uimenu(mh,'Label','Item 2','Checked','on');
```

`fh` is the handle of the parent figure.

`mh` is the handle of the parent menu.

The `Label` property specifies the text that appears in the menu.

The `Checked` property specifies that this item is displayed with a check next to it when the menu is created.

If your UI displays the standard menu bar, the new menu is added to it.
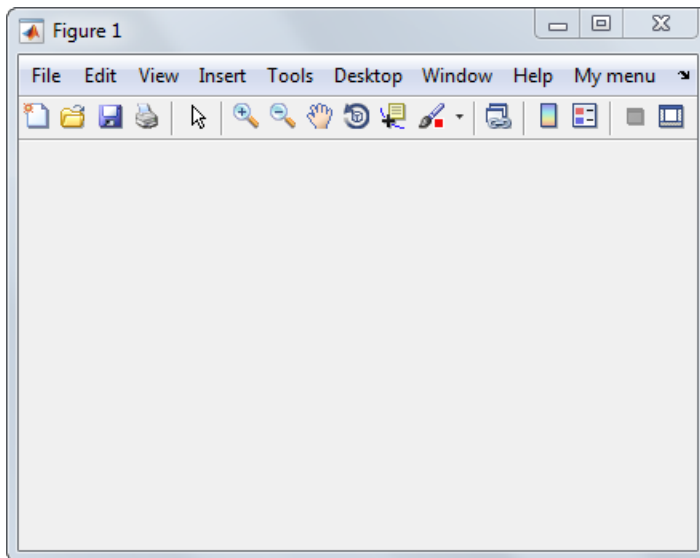
If your UI does not display the standard menu bar, MATLAB creates a menu bar if none exists and then adds the menu to it.
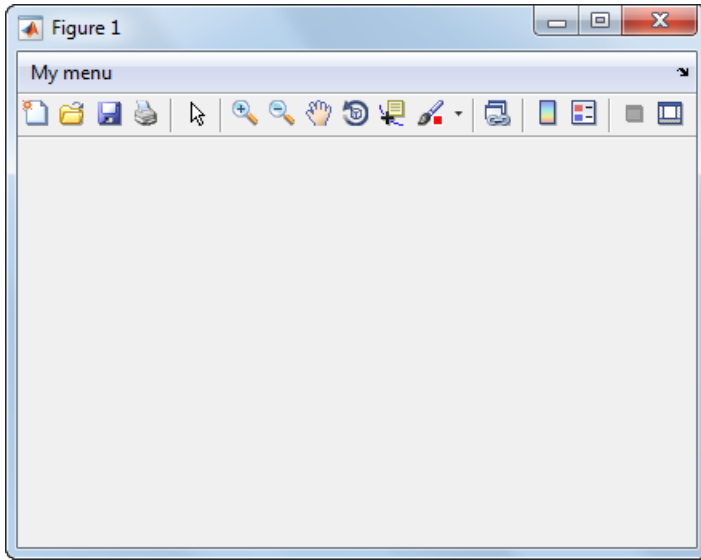
This command adds a separator line preceding the second menu item.

```
eh2.Separator = 'on';
```
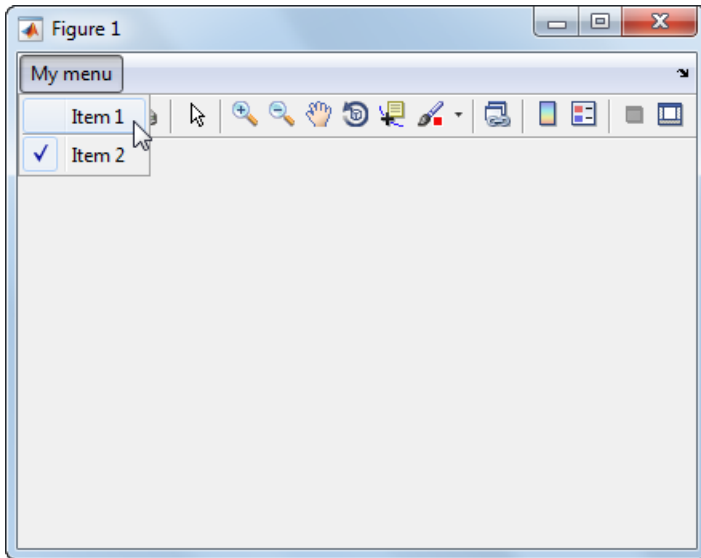
The following statements add two menu subitems to **Item 1**, assign each subitem a keyboard accelerator, and disable the first subitem.

```
seh1 = uimenu(eh1,'Label','Choice 1','Accelerator','C',...
              'Enable','off');
seh2 = uimenu(eh1,'Label','Choice 2','Accelerator','H');
```
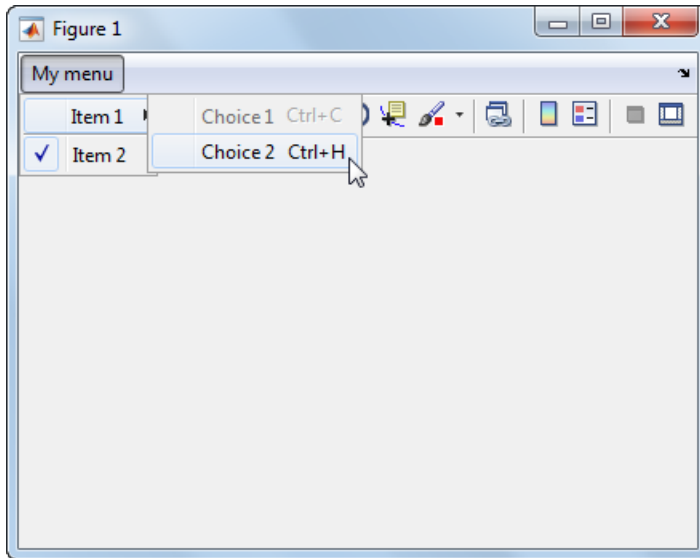

The `Accelerator` property adds keyboard accelerators to the menu items. Some accelerators may be used for other purposes on your system and other actions may result.

The `Enable` property disables the first subitem **Choice 1** so a user cannot select it when the menu is first created. The item appears dimmed.

Note After you have created all menu items, set their `HandleVisibility` properties `off` by executing the following statements:

```
menuhandles = findall(figurehandle,'type','uimenu');
menuhandles.HandleVisibility = 'off';
```

See the section, "Menu Item" on page 7-23, for information about programming menu items.

Add Context Menus to a Programmatic UI

Context menus appear when the user right-clicks on a figure or UI component. Follow these steps to add a context menu to your UI:

1 Create the context menu object using the `uicontextmenu` function.

2 Add menu items to the context menu using the `uimenu` function.

3 Associate the context menu with a graphics object using the object's `UIContextMenu` property.

Subsequent topics describe commonly used context menu properties and explain each of these steps:

Commonly Used Properties

The most commonly used properties needed to describe a context menu object are shown in the following table. These properties apply only to the menu object and not to the individual menu items.

Property	Values	Description
HandleVisibility	on, off. Default is on.	Determines if an object's handle is visible in its parent's list of children. For menus, set `HandleVisibility` to `off` to protect menus from operations not intended for them.
Parent	Figure handle	Handle of the context menu's parent figure.
Position	2-element vector: [distance from left, distance from bottom]. Default is [0 0].	Distances from the bottom left corner of the parent figure to the top left corner of the context menu. This property is used only when you programmatically set the context menu `Visible` property to `on`.

Property	Values	Description
`Visible`	`off`, `on`. Default is `off`	• Indicates whether the context menu is currently displayed. While the context menu is displayed, the property value is `on`; when the context menu is not displayed, its value is `off`. • Setting the value to `on` forces the posting of the context menu. Setting to `off` forces the context menu to be removed. The `Position` property determines the location where the context menu is displayed.

For a complete list of properties and for more information about the properties listed in the table, see Uicontextmenu Properties.

Create the Context Menu Object

Use the `uicontextmenu` function to create a context menu object. The syntax is

```
handle = uicontextmenu('PropertyName',PropertyValue,...)
```

The parent of a context menu must always be a figure. Use the `Parent` property to specify the parent of a uicontextmenu. If you do not specify the `Parent` property, the parent is the current figure as specified by the root `CurrentFigure` property.

The following code creates a figure and a context menu whose parent is the figure. At this point, the figure is visible, but not the menu.

```
fh = figure('Position',[300 300 400 225]);
cmenu = uicontextmenu('Parent',fh,'Position',[10 215]);
```

Note "Force Display of the Context Menu" on page 9-52 explains the use of the `Position` property.

Add Menu Items to the Context Menu

Use the `uimenu` function to add items to the context menu. The items appear on the menu in the order in which you add them. The following code adds three items to the context menu created above.

```
mh1 = uimenu(cmenu,'Label','Item 1');
mh2 = uimenu(cmenu,'Label','Item 2');
mh3 = uimenu(cmenu,'Label','Item 3');
```

You can specify any applicable Uimenu Properties when you define the context menu items. See the `uimenu` reference page and "Add Menu Bar Menus" on page 9-41 for information about using `uimenu` to create menu items. Note that context menus do not have an `Accelerator` property.

Note After you have created the context menu and all its items, set their `HandleVisibility` properties to `'off'` by executing the following statements:

```
cmenuhandles = findall(figurehandle,'type','uicontextmenu');
cmenuhandles.HandleVisibility = 'off';
menuitemhandles = findall(cmenuhandles,'type','uimenu');
menuitemhandles.HandleVisibility = 'off';
```

Associate the Context Menu with Graphics Objects

You can associate a context menu with the figure itself and with all components that have a `UIContextMenu` property. This includes axes, panel, button group, all user interface controls (uicontrols).

This code adds a panel and an axes to the figure. The panel contains a single push button.

```
ph = uipanel('Parent',fh,'Units','pixels',...
             'Position',[20 40 150 150]);
bh1 = uicontrol(ph,'String','Button 1',...
             'Position',[20 20 60 40]);
ah = axes('Parent',fh,'Units','pixels',...
             'Position',[220 40 150 150]);
```
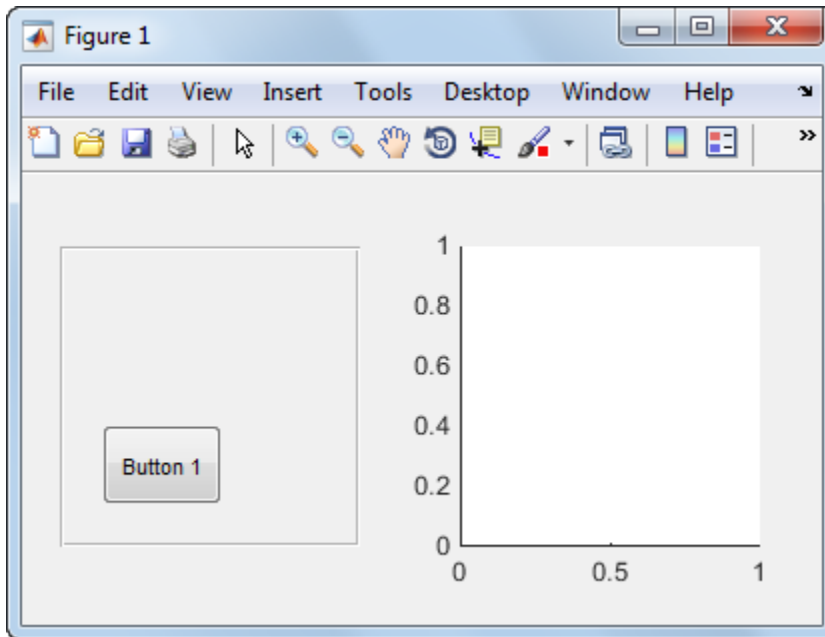
This code associates the context menu with the figure and with the axes by setting the UIContextMenu property of the figure and the axes to the handle cmenu of the context menu.

```
fh.UIContextMenu = cmenu;    % Figure
ah.UIContextMenu = cmenu;    % Axes
```

Right-click on the figure or on the axes. The context menu appears with its upper-left corner at the location you clicked. Right-click on the panel or its push button. The context menu does not appear.
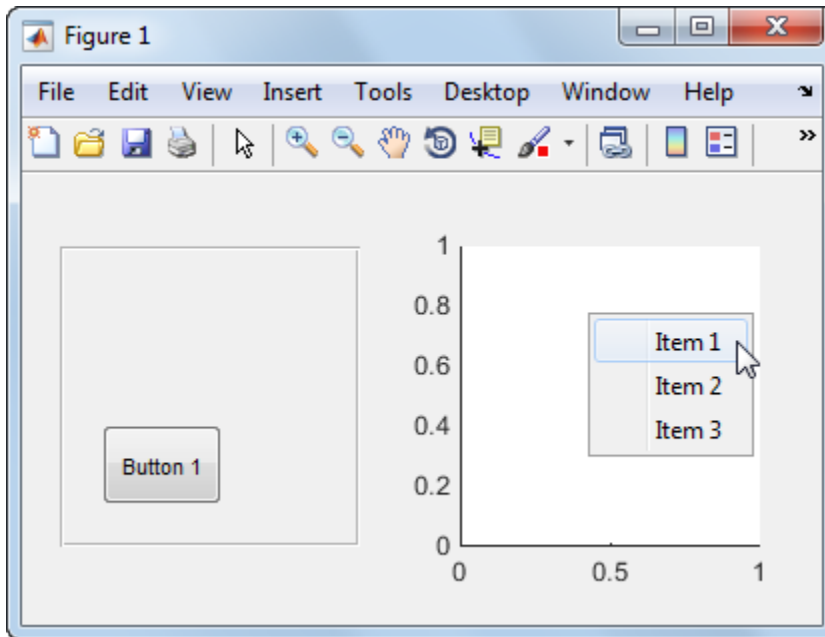
Force Display of the Context Menu

If you set the context menu `Visible` property `on`, the context menu is displayed at the location specified by the `Position` property, without the user taking any action. In this example, the context menu `Position` property is [10 215].

```
cmenu.Visible = 'on';
```

The context menu displays 10 pixels from the left of the figure and 215 pixels from the bottom.
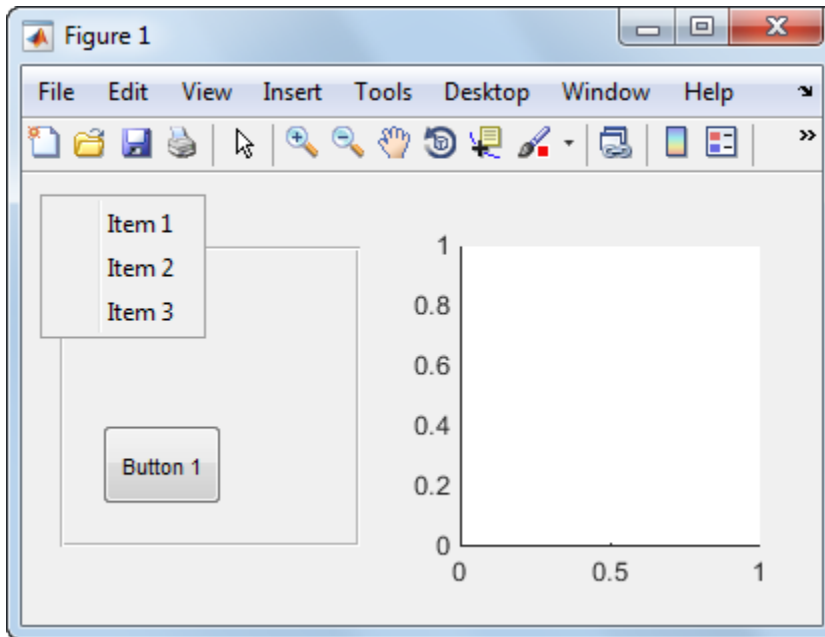
If you set the context menu `Visible` property to `off`, or if the user clicks outside the context menu, the context menu disappears.

More About

- "Create Toolbars for Programmatic UIs" on page 9-54

Create Toolbars for Programmatic UIs

In this section...
"Use the uitoolbar Function" on page 9-54
"Commonly Used Properties" on page 9-54
"Toolbars" on page 9-55
"Display and Modify the Standard Toolbar" on page 9-58

Use the uitoolbar Function

Use the `uitoolbar` function to add a custom toolbar to your UI. Use the `uipushtool` and `uitoggletool` functions to add push tools and toggle tools to a toolbar. A push tool functions as a push button. A toggle tool functions as a toggle button. You can add push tools and toggle tools to the standard toolbar or to a custom toolbar.

Syntaxes for the `uitoolbar`, `uipushtool`, and `uitoggletool` functions include the following:

```
tbh = uitoolbar(fh,'PropertyName',PropertyValue,...)
pth = uipushtool(tnh,'PropertyName',PropertyValue,...)
tth = uitoggletool(tbh,'PropertyName',PropertyValue,...)
```

The output arguments, `tbh`, `pth`, and `tth` are the handles, respectively, of the resulting toolbar, push tool, and toggle tool. See the `uitoolbar`, `uipushtool`, and `uitoggletool` reference pages for other valid syntaxes.

Subsequent topics describe commonly used properties of toolbars and toolbar tools, offer a simple example, and discuss use of the MATLAB standard toolbar:

Commonly Used Properties

The most commonly used properties needed to describe a toolbar and its tools are shown in the following table.

Property	Values	Description
CData	3-D array of values between 0.0 and 1.0	n-by-m-by-3 array of RGB values that defines a truecolor image

Property	Values	Description
		displayed on either a push button or toggle button.
HandleVisibility	on, off. Default is on.	Determines if an object's handle is visible in its parent's list of children. For toolbars and their tools, set HandleVisibility to off to protect them from operations not intended for them.
Separator	off, on. Default is off.	Draws a dividing line to left of the push tool or toggle tool
State	off, on. Default is off.	Toggle tool state. on is the down, or depressed, position. off is the up, or raised, position.
TooltipString	String	Text of the tooltip associated with the push tool or toggle tool.

For a complete list of properties and for more information about the properties listed in the table, see the Uitoolbar Properties, Uipushtool Properties, and Uitoggletool Properties.

Toolbars

The following statements add a toolbar to a figure, and then add a push tool and a toggle tool to the toolbar. By default, the tools are added to the toolbar, from left to right, in the order they are created.

```
% Create the toolbar
fh = figure;
tbh = uitoolbar(fh);

% Add a push tool to the toolbar
a = [.20:.05:0.95];
img1(:,:,1) = repmat(a,16,1)';
img1(:,:,2) = repmat(a,16,1);
img1(:,:,3) = repmat(flip(a),16,1);
pth = uipushtool(tbh,'CData',img1,...
          'TooltipString','My push tool',...
          'HandleVisibility','off');
```

```matlab
% Add a toggle tool to the toolbar
img2 = rand(16,16,3);
tth = uitoggletool(tbh,'CData',img2,'Separator','on',...
            'TooltipString','Your toggle tool',...
            'HandleVisibility','off');
```
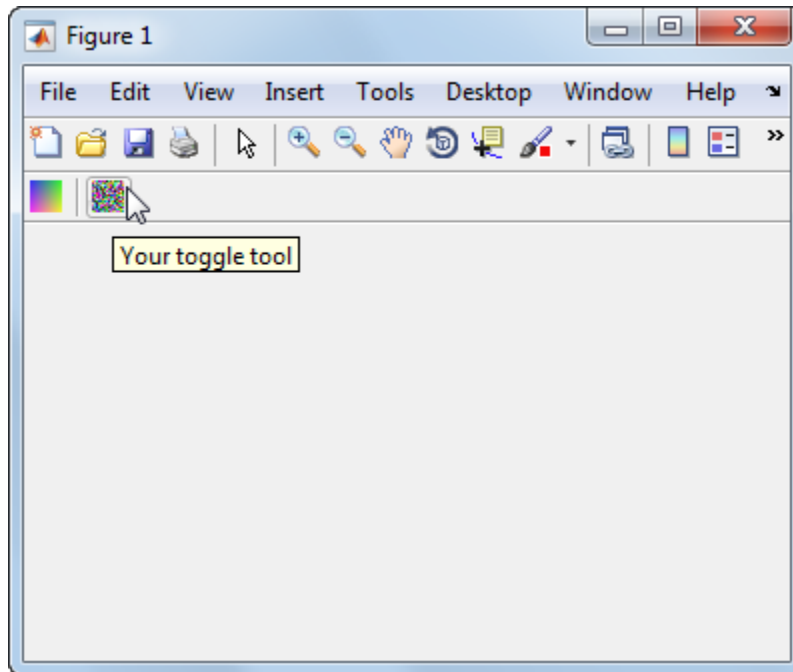

fh is the handle of the parent figure.

th is the handle of the parent toolbar.

CData is a 16-by-16-by-3 array of values between 0 and 1. It defines the truecolor image that is displayed on the tool. If your image is larger than 16 pixels in either dimension, it may be clipped or cause other undesirable effects. If the array is clipped, only the center 16-by-16 part of the array is used.

Note See the ind2rgb reference page for information on converting a matrix X and corresponding colormap, i.e., an (X, MAP) image, to RGB (truecolor) format.

TooltipString specifies the tooltips for the push tool and the toggle tool as My push tool and Your toggle tool, respectively.

In this example, setting the toggle tool Separator property to on creates a dividing line to the left of the toggle tool.

You can change the order of the tools by modifying the child vector of the parent toolbar. For this example, execute the following code to reverse the order of the tools.

```
oldOrder = allchild(tbh);
newOrder = flipud(oldOrder);
tbh.Children = newOrder;
```

This code uses flipud because the Children property is a column vector.
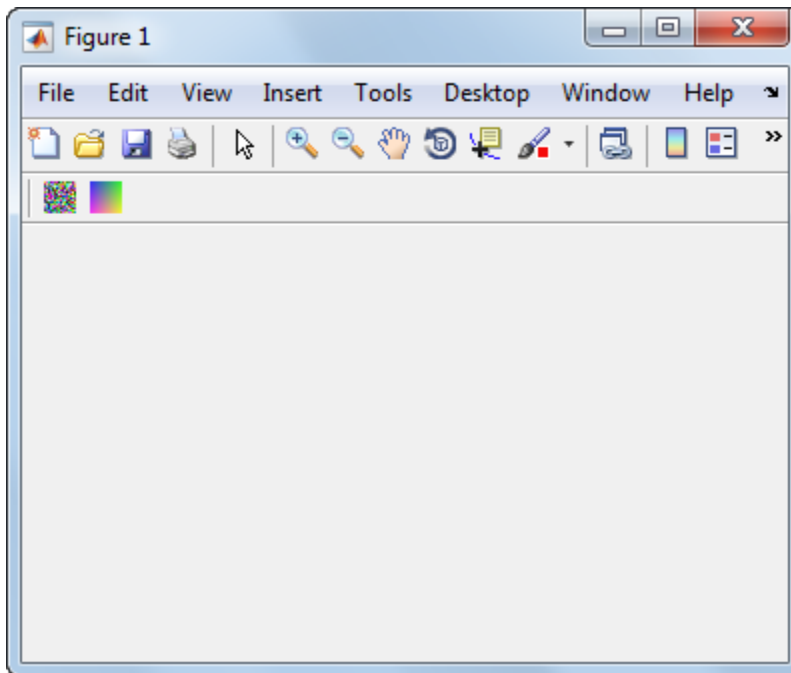
Use the delete function to remove a tool from the toolbar. The following statement removes the ▦ toggle tool from the toolbar. The toggle tool handle is tth.

```
delete(tth)
```

If necessary, you can use the `findall` function to determine the handles of the tools on a particular toolbar.

Note After you have created a toolbar and its tools, set their `HandleVisibility` properties `off` by executing statements similar to the following:

```
toolbarhandle.HandleVisibility = 'off';
toolhandles = toolbarhandle.Children;
toolhandles.HandleVisibility = 'off';
```

Display and Modify the Standard Toolbar

You can choose whether or not to display the MATLAB standard toolbar (highlighted in red below). You can also add or delete tools from the standard toolbar.
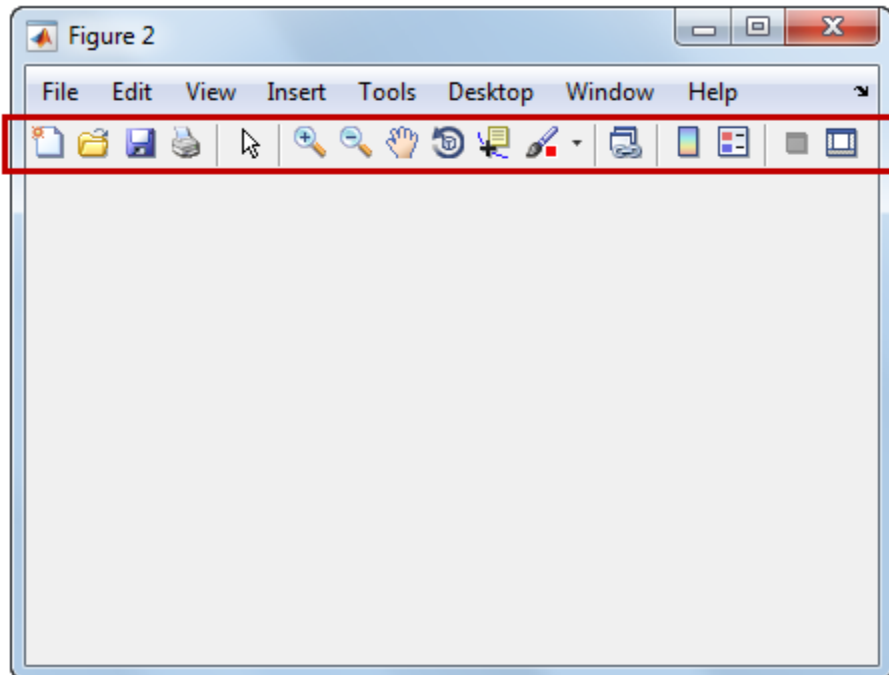
Display the Standard Toolbar

Use the figure `Toolbar` property to display or hide the standard toolbar. Set `Toolbar` to `'figure'` to display the standard toolbar. Set `Toolbar` to `'none'` to hide it.

```
fh.Toolbar = 'figure';  % Display the standard toolbar
fh.Toolbar = 'none';    % Hide the standard toolbar
```

In these statements, `fh` is the handle of the figure.

The default `ToolBar` value is `'auto'`, which uses the `MenuBar` property value.

Modify the Standard Toolbar

Once you have the handle of the standard toolbar, you can add tools, delete tools, and change the order of the tools.

Add a tool the same way you would add it to a custom toolbar. This code gets the handle of the standard toolbar and adds a toggle tool to it.

```
tbh = findall(fh,'Type','uitoolbar');
tth = uitoggletool(tbh,'CData',rand(20,20,3),...
            'Separator','on',...
            'HandleVisibility','off');
```
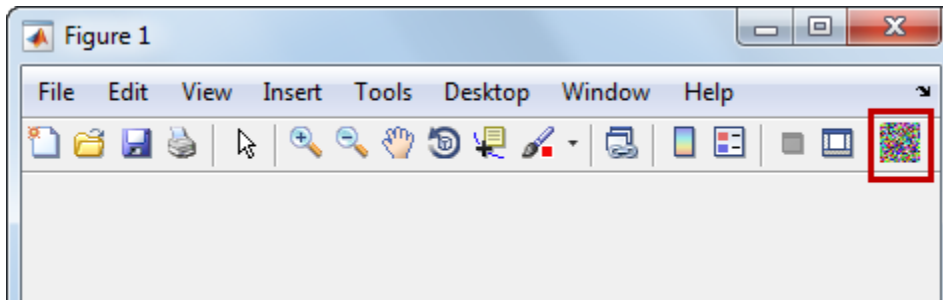

To remove a tool from the standard toolbar, determine the handle of the tool to be removed, and then use the `delete` function to remove it. The following code deletes the toggle tool that was added to the standard toolbar above.

```
delete(tth)
```

If necessary, you can use the `findall` function to determine the handles of the tools on the standard toolbar.

See Also

uipushtool | uitoggletool | uitoolbar

More About

- "Create Menus for Programmatic UIs" on page 9-41

Fonts and Colors for Cross-Platform Compatibility

In this section...
"Default System Font" on page 9-61
"Standard Background Color" on page 9-62

Default System Font

By default, user interface controls (uicontrols) use the default font for the platform on which they are running. For example, when displaying your UI on PCs, user interface controls use MS San Serif. When your UI runs on a different platform, they use that computer's default font. This provides a consistent look with respect to your UI and other applications on the same platform.

If you have set the FontName property to a named font and want to return to the default value, you can set the property to the string, `'default'`. This ensures that MATLAB software uses the system default at run-time.

```
pbh1.FontName = 'default';
```

Specify a Fixed-Width Font

If you want to use a fixed-width font for a user interface control, set its `FontName` property to the string, `'fixedwidth'`. This special identifier ensures that your UI uses the standard fixed-width font for the target platform.

You can find the name of the fixed-width font that is used on a given platform by querying the root `FixedWidthFontName` property.

```
get(groot,'FixedWidthFontName')
```

Use a Specific Font Name

You can specify an actual font name (such as Times or Courier) for the `FontName` property. However, doing so may cause your UI to appear differently than you intended when run on a different computer. If the target computer does not have the specified font, it substitutes another font that may not look good in your UI or may not be the standard font used on that system. Also, different versions of the same named font may have different size requirements for a given set of characters.

Standard Background Color

MATLAB uses the standard system background color of the system on which the UI is running as the default component background color. This color varies on different computer systems, e.g., the standard shade of gray on the PC differs from that on UNIX system, and may not match the default UI background color.

You can make the UI background color match the default component background color. This code gets the default component background color and assign it to the figure.

```
defaultBackground = get(groot,'defaultUicontrolBackgroundColor');
fh.Color = defaultBackground;
```

DPI-Aware Behavior in MATLAB

In this section...
"Visual Appearance" on page 9-63
"Using Object Properties" on page 9-65
"Using print, getframe, and publish Functions" on page 9-66

Starting in R2015b, MATLAB is DPI-aware, which means that it takes advantage of your full system resolution to draw graphical elements (fonts, UIs, and graphics). Graphical elements appear sharp and consistent in size on these high-DPI systems:

- Windows systems in which the display dots-per-inch (DPI) value is set higher than 96
- Macintosh systems with Apple Retina displays

DPI-aware behavior does not apply to Linux systems.

Previously, MATLAB allowed some operating systems to scale graphical elements. That scaling helped to maintain consistent appearance and functionality, but it also introduced undesirable effects. Graphical elements often looked blurry, and the size of those elements was sometimes inconsistent.

Visual Appearance

Here are the visual effects you might notice on high-DPI systems:

- The MATLAB desktop, graphics, fonts, and most UI components look sharp and render with full graphical detail on Macintosh and Windows systems.
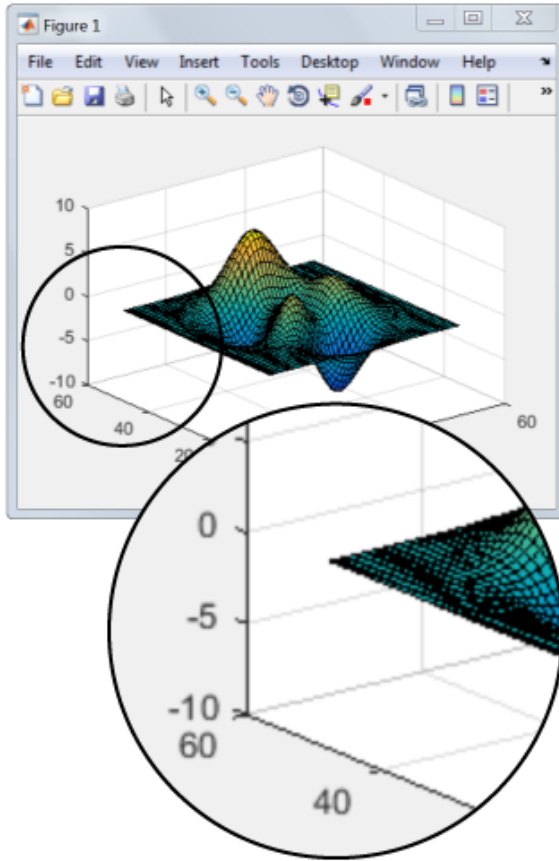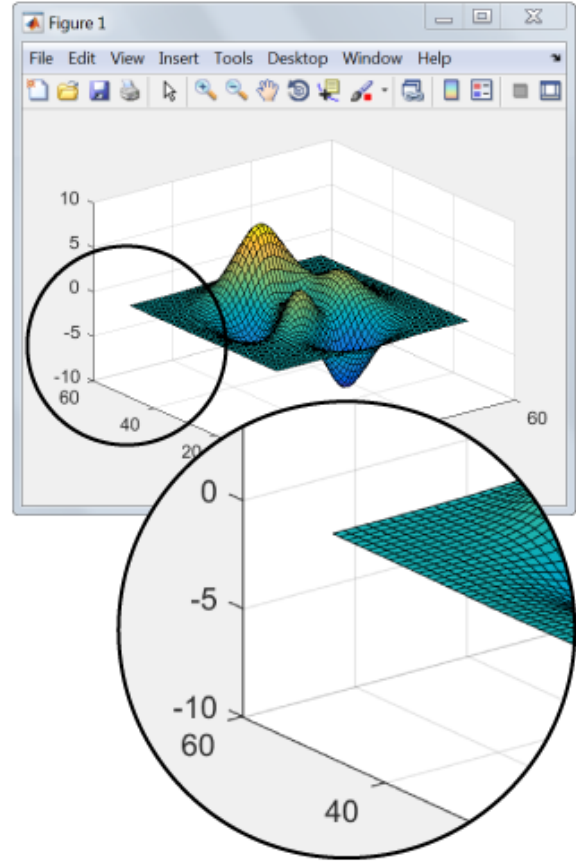
- When you create a graphics or UI object, and specify the Units as 'pixels', the size of that object is now consistent with the size of other objects. For example, the size of a push button (specified in pixels) is now consistent with the size of the text on that push button (specified in points).

- Elements in the MATLAB Toolstrip look sharper than in previous releases. However, icons in the Toolstrip might still look slightly blurry on some systems.

- On Windows systems, the MATLAB Toolstrip might appear larger than in previous releases.

- On Windows systems, the size of the Command Window fonts and Editor fonts might be larger than in previous releases. In particular, you might see a difference if you have nondefault font sizes selected in MATLAB preferences. You might need to adjust those font sizes to make them look smaller.

- You might see differences on multiple-display systems that include a combination of different displays (for example, some, but not all of the displays are high-DPI). Graphical elements might look different across displays on those systems.

Using Object Properties

These changes to object properties minimize the impact on your existing code and allow MATLAB to use the full display resolution when rendering graphical elements. All UIs you create in MATLAB are automatically DPI-aware applications.

Units Property

When you set the `Units` property of a graphics or UI object to `'pixels'`, the size of each pixel is now device-independent on Windows and Macintosh systems:

- On Windows systems, 1 pixel = 1/96 inch.
- On Macintosh systems, 1 pixel = 1/72 inch.
- On Linux systems, the size of a pixel is determined by the display DPI.

Your existing graphics and UI code will continue to function properly with the new pixel size. Keep in mind that specifying (or querying) the size and location of an object in pixels might not correspond to the actual pixels on your screen.

For example, each screen pixel on a 192-DPI Windows system is 1/192nd of an inch. In this case, twice as many screen pixels cover the same linear distance as the device-independent pixels do. If you create a figure, and specify its size to be 500-by-400 pixels, MATLAB reports the size to be 500-by-400 in the `Position` property. However, the display uses 1000-by-800 screen pixels to cover the same graphical region.

Note: Starting in R2015b, MATLAB might report the size and location of objects as fractional values (in pixel units) more frequently than in previous releases. For example, your code might report fractional values in the `Position` property of a figure, whereas previous releases reported whole numbers for that same figure.

Root ScreenSize Property

The ScreenSize property of the root object might not match the display size reported by high-DPI Windows systems. Specifically, the values do not match when the Units property of the root object is set to 'pixels'. MATLAB reports the value of the ScreenSize property based on device-independent pixels, not the size of the actual pixels on the screen.

Root ScreenPixelsPerInch Property

The ScreenPixelsPerInch property became a read-only property in R2015b. If you want to change the size of text and other elements on the screen, adjust your operating system settings.

Also, you cannot set or query the default value of the ScreenPixelsPerInch property. These commands now return an error:

```
get(groot,'DefaultRootScreenPixelsPerInch')
set(groot,'DefaultRootScreenPixelsPerInch')
```
The factory value cannot be queried either. This command returns an error as well:

```
get(groot,'FactoryRootScreenPixelsPerInch')
```

Using print, getframe, and publish Functions

getframe and print Functions

When using the getframe function (or the print function with the -r0 option) on a high-DPI system, the size of the image data array that MATLAB returns is larger than in previous releases. Additionally, the number of elements in the array might not match the figure size in pixel units. MATLAB reports the figure size based on device-independent pixels. However, the size of the array is based on the display DPI.

publish Function

When publishing documents on a high-DPI system, the images saved to disk are larger than in previous releases or on other systems.

See Also
Figure Properties | Root Properties

Code a Programmatic UI

Initialize a Programmatic UI

Programs that present a UI might perform these tasks when you launch them:

- Define default values
- Set UI component property values
- Process input arguments
- Hide the figure window until all the components are created

When you develop a UI, consider grouping these tasks together in your code file. If an initialization task involves several steps, consider creating a separate function for that task.

Examples

Declare Variables for Input and Output Arguments

These are typical declarations for input and output arguments.

```
mInputArgs = varargin; % Command line arguments

mOutputArgs = {};      % Variable for storing output
```

See the `varargin` reference page for more information.

Define Custom Property/Value Pairs

This example defines the properties in a cell array, `mPropertyDefs`, and then initializes the properties.

```
mPropertyDefs = {...
                 'iconwidth',  @localValidateInput, 'mIconWidth';
                 'iconheight', @localValidateInput, 'mIconHeight';
                 'iconfile',   @localValidateInput, 'mIconFile'};
mIconWidth = 16;   % Use input property 'iconwidth' to initialize
mIconHeight = 16;  % Use input property 'iconheight' to initialize
mIconFile = fullfile(matlabroot,'toolbox/matlab/icons/');
                   % Use input property 'iconfile' to initialize
```

Each row of the cell array defines one property. It specifies, in order, the name of the property, the routine that is called to validate the input, and the name of the variable that holds the property value.

The `fullfile` function builds a full filename from parts.

The following statements start the Icon Editor application. The first statement creates a new icon. The second statement opens existing icon file for editing.

```
cdata = iconEditor('iconwidth',16,'iconheight',25)
cdata = iconEditor('iconfile','eraser.gif');
```

`iconEditor` calls a routine, `processUserIputs`, during the initialization to accomplish these tasks:

- Identify each property by matching it to the first column of the cell array
- Call the routine named in the second column to validate the input
- Assign the value to the variable named in the third column

Make the Figure Invisible

When you create the figure window, make it invisible when you create it. Display it only after you have added all the UI components.

To make the window invisible, set the figure `Visible` property to `'off'` when you create the figure:

```
hMainFigure = figure(...
               'Units','characters',...
               'MenuBar','none',...
               'Toolbar','none',...
               'Position',[71.8 34.7 106 36.15],...
               'Visible','off');
```

After you have added all the components to the figure window, make the figure visible:

```
hMainFigure.Visible = 'on';
```

Most components have a `Visible` property. Thus, you can also use this property to make individual components invisible.

Return Output to the User

If your program allows an output argument, and the user specifies such an argument, then you want to return the expected output. The code that provides this output usually appears just before the program's main function returns.

In the example shown here,

1. A call to `uiwait` blocks execution until `uiresume` is called or the current figure is deleted.

2. While execution is blocked, the user creates the icon.

3. When the user clicks **OK**, that push button's callback calls the `uiresume` function.

4. The program returns the completed icon to the user as output.

```
% Make the window blocking.
uiwait(hMainFigure);

% Return the edited icon CData if it is requested.
mOutputArgs{1} = mIconCData;
if nargout>0
    [varargout{1:nargout}] = mOutputArgs{:};
end
```

`mIconData` contains the icon that the user created or edited. `mOutputArgs` is a cell array defined to hold the output arguments. `nargout` indicates how many output arguments the user has supplied. `varargout` contains the optional output arguments returned by the program. See the complete Icon Editor code file for more information.

Related Examples

- "Create a Simple App Programmatically" on page 3-2

Write Callbacks for Apps Created Programmatically

In this section...
"Callbacks for Different User Actions" on page 10-5
"How to Specify Callback Property Values" on page 10-7
"Callback Syntax" on page 10-9

Callbacks for Different User Actions

UI and graphics components have certain properties that you can associate with specific callback functions. Each of these properties corresponds to a specific user action. For example, a uicontrol has a property called `Callback`. You can set the value of this property to be a handle to a callback function, an anonymous function, or a string containing MATLAB commands. Setting this property makes your UI respond when the user interacts with the uicontrol. If the `Callback` property has no specified value, then nothing happens when the end user interacts with the uicontrol.

This table lists the callback properties that are available, the user actions that trigger the callback function, and the most common UI and graphics components that use them.

Callback Property	User Action	Components That Use This Property
`ButtonDownFcn`	End user presses a mouse button while the pointer is on the component or figure.	`axes`, `figure`, `uibuttongroup`, `uicontrol`, `uipanel`, `uitable`,
`Callback`	End user triggers the component. For example: selecting a menu item, moving a slider, or pressing a push button.	`uicontextmenu`, `uicontrol`, `uimenu`
`CellEditCallb`	End user edits a value in a table whose cells are editable.	`uitable`
`CellSelection`	End user selects cells in a table.	`uitable`
`ClickedCallba`	End user clicks the push tool or toggle tool with the left mouse button.	`uitoggletool`, `uipushtool`
`CloseRequestF`	The figure closes.	`figure`

Callback Property	User Action	Components That Use This Property
CreateFcn	Callback executes when MATLAB creates the object, but before it is displayed.	axes, figure, uibuttongroup, uicontextmenu, uicontrol, uimenu, uipushtool, uipanel, uitable, uitoggletool, uitoolbar
DeleteFcn	Callback executes just before MATLAB deletes the figure.	axes, figure, uibuttongroup, uicontextmenu, uicontrol, uimenu, uipushtool, uipanel, uitable, uitoggletool, uitoolbar
KeyPressFcn	End user presses a keyboard key while the pointer is on the object.	figure, uicontrol, uipanel, uipushtool, uitable, uitoolbar
KeyReleaseFcn	End user releases a keyboard key while the pointer is on the object.	figure, uicontrol, uitable
OffCallback	Executes when the State of a toggle tool changes to 'off'.	uitoggletool
OnCallback	Executes when the State of a toggle tool changes to 'on'.	uitoggletool
SizeChangedFcn	End user resizes a button group, figure, or panel whose Resize property is 'on'.	figure, uipanel, uibuttongroup
SelectionChanged	End user selects a different radio button or toggle button within a button group.	uibuttongroup
WindowButtonDown	End user presses a mouse button while the pointer is in the figure window.	figure
WindowButtonMo	End user moves the pointer within the figure window.	figure
WindowButtonUp	End user releases a mouse button.	figure
WindowKeyPres	End user presses a key while the pointer is on the figure or any of its child objects.	figure

Callback Property	User Action	Components That Use This Property
`WindowKeyRele`	End user releases a key while the pointer is on the figure or any of its child objects.	`figure`
`WindowScrollW`	End user turns the mouse wheel while the pointer is on the figure.	`figure`

How to Specify Callback Property Values

To associate a callback function with a UI component, set the value of one of the component's callback properties to be a reference to the callback function. Typically, you do this when you define the component, but you can change callback property values anywhere in your code.

Specify the callback property value in one of these ways:

- "Specify a Function Handle" on page 10-7.
- "Specify a Cell Array" on page 10-8. This cell array contains a function handle as the first element, followed by and any input arguments you want to use in the function.
- "Specify a String of MATLAB Commands (Not Recommended)" on page 10-9

Specify a Function Handle

Function handles provide a way for you to represent a function as a variable. You define the function as a local or nested function in the same file as the code, or you can write it in a separate program file that is on the MATLAB path. The function definition must define two input arguments, `hObject` and `callbackdata`. Handle Graphics automatically passes `hObject` and `callbackdata` when it calls the function. For more information about these arguments, see "Callback Syntax" on page 10-9.

This code creates a slider component and specifies its callback as a function handle. To see how it works, copy and paste this code into an editor and run it.

Note: This code uses dot notation to get object properties. Dot notation runs in R2014b and later. If you are using an earlier release, use the `get` function instead. For example, change `hObject.Value` to `get(hObject,'Value')`.

```matlab
function myui()
    figure
    uicontrol('Style','slider','Callback',@display_slider_value);
end
function display_slider_value(hObject,callbackdata)
    newval = num2str(hObject.Value);
    disp(['Slider moved to ' newval]);
end
```

This callback function displays the value of the slider when the end user adjusts it. A benefit of specifying callbacks as function handles is that MATLAB checks the function for syntax errors and missing dependencies when you assign the callback to the component. If there is a problem in the callback function, then MATLAB returns an error immediately instead of waiting for the end user to trigger the callback. This behavior helps you to find problems in your code before the end user encounters them.

Note: If you want to use an existing function that does not support the hObject and callbackdata arguments, then you can specify it as an anonymous function. For example,
uicontrol('Style','slider','Callback',@(hObject,callbackdata)myfunction(x));
In this case, x is an input argument to the function, myfunction. See "Anonymous Functions" for more information.

Specify a Cell Array

Use a cell array to specify a callback function that accepts input arguments that you want to use in the function. The first element in the cell array is a function handle. The other elements in the cell array are the input arguments to the function, separated by commas. For example, this code creates a push button component and specifies its callback to be a cell array containing the function handle, @pushbutton_callback, and one input argument, myvar. To see how it works, copy and paste this code into an editor and run it.

```matlab
function myui()
    myvar = 5;
    figure
    uicontrol('Style','pushbutton',...
              'Callback',{@pushbutton_callback,myvar});
end
function pushbutton_callback(hObject,callbackdata,x)
    display(x);
end
```

The function definition must define two input arguments, `hObject` and `callbackdata`, followed by any additional arguments the function uses. Handle Graphics automatically passes the `hObject` and `callbackdata` arguments when it calls the function. If you define additional input arguments, then the values you pass to the callback must exist in the workspace when the end user triggers the callback.

Like callbacks specified as function handles, MATLAB checks callbacks specified as cell arrays for syntax errors and missing dependencies when you assign the callback to the component. If there is a problem in the callback function, then MATLAB returns an error immediately instead of waiting for the end user to trigger the callback. This behavior helps you to find problems in your code before the end user encounters them.

Specify a String of MATLAB Commands (Not Recommended)

You can use string callbacks for a few simple commands, but the callback can become difficult to manage if it contains more than a few commands. The string you specify must consist of valid MATLAB expressions, which can include arguments to functions. For example:

```
hb = uicontrol('Style','pushbutton',...
               'String','Plot line',...
               'Callback','plot(rand(20,3))');
```

The callback string, `'plot(rand(20,3))'`, is a valid command, and MATLAB evaluates it when the end user clicks the button. If the callback string includes a variable, for example,

```
'plot(myvar)'
```

The variable, `myvar`, must exist in the base workspace when the end user triggers the callback, or it returns an error. The variable does not need to exist at the time you assign callback property value, but it must exist when the end user triggers the callback.

Unlike callbacks that are specified as function handles or cell arrays, MATLAB does *not* check string callbacks for syntax errors or missing dependencies. If there is a problem with the code in your string, it remains undetected until the end user triggers the callback.

Callback Syntax

All callback functions that you reference as function handles or cell arrays must accept the at least two input arguments, `hObject` and `callbackdata`. All other input arguments must appear after `hObject` and `callbackdata` in the function definition.

The `hObject` argument is a handle to the UI component that triggered the callback. The `callbackdata` argument provides additional information to certain callback functions. For example, if the end user triggers the `KeyPressFcn`, then MATLAB provides information regarding the specific key (or combination of keys) that the end user pressed. If `callbackdata` is not available to the callback function, then MATLAB passes it as an empty array. The following table lists the callbacks and components that use `callbackdata`.

Callback Property Name	Component
`WindowKeyPressFcn` `WindowKeyReleaseFcn` `WindowScrollWheel`	`figure`
`KeyPressFcn`	`figure`, `uicontrol`, `uitable`
`KeyReleaseFcn`	`figure`, `uicontrol`, `uitable`
`SelectionChangedFcn`	`uibuttongroup`
`CellEditCallback` `CellSelectionCallback`	`uitable`

Related Examples
- "Callbacks for Specific Components" on page 7-12

More About
- "Share Data Among Callbacks" on page 11-2
- "Interrupt Callback Execution" on page 12-2

Manage Application-Defined Data

Share Data Among Callbacks

In this section...
"Overview of Data Sharing Techniques" on page 11-2
"Store Data in UserData or Other Object Properties" on page 11-3
"Store Data as Application Data" on page 11-7
"Create Nested Callback Functions (Programmatic UIs)" on page 11-12
"Store Data Using the guidata Function" on page 11-13
"Sharing Data Among Multiple GUIDE UIs" on page 11-16

Overview of Data Sharing Techniques

Many UIs contain interdependent controls, menus, and graphics objects. Since each callback function has its own scope, you must explicitly share data with those parts of your UI that need to access it. The table below describes several different methods for sharing data within your UI.

Method	Description	Requirements and Trade-Offs
"Store Data in UserData or Other Object Properties" on page 11-3	Query or store property values directly though the component object. All UI components have a `UserData` property that can store any MATLAB data.	• Requires access to the component to set or retrieve the properties. • `UserData` holds only one variable at a time, but you can store multiple values as a `struct` array or cell array.
"Store Data as Application Data" on page 11-7	Associate data with a specific component using the `setappdata` function. You can access it later using the `getappdata` function.	• Requires access to the component to set or retrieve the application data. • Can share multiple variables.
"Create Nested Callback Functions (Programmatic	Nest your callback functions inside your main function. This gives your callback functions access to all the variables in the main function.	• Requires callback functions to be coded in the same file as the main function. • Not recommended for GUIDE UIs.

Method	Description	Requirements and Trade-Offs
UIs)" on page 11-12		• Can share multiple variables.
"Store Data Using the guidata Function" on page 11-13	Share data with the figure window using the guidata function.	• Stores or retrieves the data through any UI component. • Stores only one variable at a time, but you can store multiple values as a struct array or cell array.

Store Data in UserData or Other Object Properties

UI components contain useful information in their properties. For example, you can find the current position of a slider by querying its Value property. In addition, all components have a UserData property, which can store any MATLAB variable. All callback functions can access the value stored in the UserData property as long as those functions can access the component.

Share UserData in UIs Created Programmatically

Use dot notation, *component.propertyname*, to store or retrieve property values programmatically. Dot notation works in R2014b and later releases. This code queries and modifies the name of a figure.

```
hfig = figure;
figname = hfig.Name;
hfig.Name = 'My Window';
```
If you are using an earlier release, use the get and set functions instead:

```
hfig = figure;
figname = get(hfig,'Name');
set(hfig,'Name','My Window');
```

If your code does not have direct access to a component, use the findobj function to search for that component. If the search is successful, findobj returns the component as output. Then you can access the component's properties.

The following UI code uses the UserData property to share information about the slider. To see how it works, copy and paste this code into an editor and run it.

```
function my_slider()
```

```matlab
hfig = figure();
slider = uicontrol('Parent', hfig,'Style','slider',...
          'Units','normalized',...
          'Position',[0.3 0.5 0.4 0.1],...
          'Tag','slider1',...
          'UserData',struct('val',0,'diffMax',1),...
          'Callback',@slider_callback);

button = uicontrol('Parent', hfig,'Style','pushbutton',...
          'Units','normalized',...
          'Position',[0.4 0.3 0.2 0.1],...
          'String','Display Difference',...
          'Callback',@button_callback);
end

function slider_callback(hObject,eventdata)
 sval = hObject.Value;
 diffMax = hObject.Max - sval;
 data = struct('val',sval,'diffMax',diffMax);
 hObject.UserData = data;
 % For R2014a and earlier:
 % sval = get(hObject,'Value');
 % maxval = get(hObject,'Max');
 % diffMax = maxval - sval;
 % data = struct('val',sval,'diffMax',diffMax);
 % set(hObject,'UserData',data);

end

function button_callback(hObject,eventdata)
 h = findobj('Tag','slider1');
 data = h.UserData;
 % For R2014a and earlier:
 % data = get(h,'UserData');
 display([data.val data.diffMax]);
end
```

When the user moves the slider, the `slider_callback` uses these commands to store data in a structure:

- `data = struct('val',sval,'diffMax',diffMax)` stores the values, `sval` and `diffMax`, in a structure called `data`.

- `hObject.UserData = data` stores the value of `data` in the `UserData` property of the slider.

When the user clicks the push button, the `button_callback` uses these commands to retrieve the data:

- `h = findobj('Tag','slider1')` finds the slider component.
- `data = h.UserData` gets the value of the slider's `UserData` property.

Share UserData in GUIDE UIs

To set up a GUIDE UI for sharing slider data with the `UserData` property, perform these steps:

1 In the Command Window, type `guide`.

2 In the GUIDE Quick Start dialog box, select **Blank GUI (Default)**. Then, click **OK**.

3 Display the names of the UI components in the component palette:

   a Select **File** > **Preferences** > **GUIDE**.

   b Select **Show names in component palette**.

   c Click **OK**.

4 Select the push button tool from the component palette at the left side of the Layout Editor and drag it into the layout area.

5 Select the slider tool from the component palette at the left side of the Layout Editor and drag it into the layout area.

6 Select **File** > **Save**. Save the UI as `myslider.fig`. MATLAB opens the code file in the Editor.

7 Set the initial value of the `UserData` property in the opening function, `myslider_OpeningFcn`. This function executes just before the UI is visible to users.

   In `myslider_OpeningFcn`, insert these commands immediately after the command, `handles.output = hObject`.

```
data = struct('val',0,'diffMax',1);
set(handles.slider1,'UserData',data);
```
   After you add the commands, `myslider_OpeningFcn` looks like this.

```
function myslider_OpeningFcn(hObject, eventdata, handles, varargin)
% This function has no output args, see OutputFcn.
% hObject    handle to figure
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)
% varargin   command line arguments to junk (see VARARGIN)
```

```
% Choose default command line output for myslider
handles.output = hObject;
data = struct('val',0,'diffMax',1);
set(handles.slider1,'UserData',data);

% Update handles structure
guidata(hObject, handles);

% UIWAIT makes myslider wait for user response
% uiwait(handles.figure1);
```

Notice that `handles` is an input argument to `myslider_OpeningFcn`. The `handles` variable is a structure that contains all the components in the UI. Each field in this structure corresponds to a separate component. Each field name matches the `Tag` property of the corresponding component. Thus, `handles.slider1` is the slider component in this UI. The command, `set(handles.slider1,'UserData',data)` stores the variable, `data`, in the `UserData` property of the slider.

8 Add code to the slider callback for modifying the data. Add these commands to the end of the function, `slider1_Callback`.

```
maxval = get(hObject,'Max');
sval = get(hObject,'Value');
diffMax = maxval - sval;
data = get(hObject,'UserData');
data.val = sval;
data.diffMax = diffMax;
% Store data in UserData of slider
set(hObject,'UserData',data);
```

After you add the commands, `slider1_Callback` looks like this.

```
% --- Executes on slider movement.
function slider1_Callback(hObject, eventdata, handles)
% hObject     handle to slider1 (see GCBO)
% eventdata   reserved - to be defined in a future version of MATLAB
% handles     structure with handles and user data (see GUIDATA)

% Hints: get(hObject,'Value') returns position of slider
%        get(hObject,'Min') and get(hObject,'Max') to determine range of slider
maxval = get(hObject,'Max');
sval = get(hObject,'Value');
diffMax = maxval - sval;
data = get(hObject,'UserData');
```

```
data.val = sval;
data.diffMax = diffMax;
% Store data in UserData of slider
set(hObject,'UserData',data);
```
Notice that `hObject` is an input argument to the `slider1_Callback` function. `hObject` is always the component that triggers the callback (the slider, in this case). Thus, `set(hObject,'UserData',data)`, stores the `data` variable in the `UserData` property of the slider.

9 Add code to the push button callback for retrieving the data. Add these commands to the end of the function, `pushbutton1_Callback`.

```
% Get UserData from the slider
data = get(handles.slider1,'UserData');
currentval = data.val;
diffval = data.diffMax;
display([currentval diffval]);
```
After you add the commands, `pushbutton1_Callback` looks like this.

```
% --- Executes on button press in pushbutton1.
function pushbutton1_Callback(hObject, eventdata, handles)
% hObject    handle to pushbutton1 (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)

% Get UserData from the slider
data = get(handles.slider1,'UserData');
currentval = data.val;
diffval = data.diffMax;
display([currentval diffval]);
```

This code uses the handles structure to access the slider. The command, `data = get(handles.slider1,'UserData')`, gets the slider's `UserData` property. Then, the `display` function displays the stored values.

10 Save your code by pressing **Save** in the Editor Toolstrip.

Store Data as Application Data

To store application data, call the `setappdata` function:

```
setappdata(obj,name,value);
```

The first input, `obj`, is the component object in which to store the data. The second input, `name`, is a string that identifies the value. The third input, `value`, is the value you want to store.

To retrieve application data, use the `getappdata` function:

```
data = getappdata(obj,name);
```
The component, `obj`, must be the component object containing the data. The second input, `name`, must match the string you used to store the data. Unlike the `UserData` property, which only holds only one variable, you can use `setappdata` to store multiple variables.

Share Application Data in UIs Created Programmatically

This UI uses application data to share two values. To see how it works, copy and paste this code into an editor and run it.

```
function my_slider()
hfig = figure();
setappdata(hfig,'slidervalue',0);
setappdata(hfig,'difference',1);

slider = uicontrol('Parent', hfig,'Style','slider',...
        'Units','normalized',...
        'Position',[0.3 0.5 0.4 0.1],...
        'Tag','slider1',...
        'Callback',@slider_callback);

button = uicontrol('Parent', hfig,'Style','pushbutton',...
        'Units','normalized',...
        'Position',[0.4 0.3 0.2 0.1],...
        'String','Display Values',...
        'Callback',@button_callback);
end

function slider_callback(hObject,eventdata)
 diffMax = hObject.Max - hObject.Value;
 setappdata(hObject.Parent,'slidervalue',hObject.Value);
 setappdata(hObject.Parent,'difference',diffMax);
 % For R2014a and earlier:
 % maxval = get(hObject,'Max');
 % currval = get(hObject,'Value');
 % diffMax = maxval - currval;
 % parentfig = get(hObject,'Parent');
```

```
% setappdata(parentfig,'slidervalue',currval);
% setappdata(parentfig,'difference',diffMax);
end

function button_callback(hObject,eventdata)
 currentval = getappdata(hObject.Parent,'slidervalue');
 diffval = getappdata(hObject.Parent,'difference');
 % For R2014a and earlier:
 % parentfig = get(hObject,'Parent');
 % currentval = getappdata(parentfig,'slidervalue');
 % diffval = getappdata(parentfig,'difference');

 display([currentval diffval]);
end
```

When the user moves the slider, the `slider_callback` function calculates `diffMax`. Then, it uses these commands to modify the application data:

- `setappdata(hObject.Parent,'slidervalue',hObject.Value)` stores the current slider value in the figure using the name, `'slidervalue'`. In this case, `hObject.Parent` is the figure.

- `setappdata(parentfig,'difference',diffMax)` stores `diffMax` in the figure using the name, `'difference'`.

When the user clicks the push button, the `button_callback` function retrieves the data using these commands:

- `currentval = getappdata(hObject.Parent,'slidervalue')` retrieves the current slider value from the figure. In this case, `hObject.Parent` is the figure.

- `diffval = getappdata(hObject.Parent,'difference')` retrieve the difference value from the figure.

Share Application Data in GUIDE UIs

To set up a GUIDE UI for sharing application data, perform these steps:

1 In the Command Window, type `guide`.

2 In the GUIDE Quick Start dialog box, select **Blank GUI (Default)**. Then, click **OK**.

3 Display the names of the UI components in the component palette:

    a Select **File** > **Preferences** > **GUIDE**.

    b Select **Show names in component palette**.

    c Click **OK**.

4 Select the push button tool from the component palette at the left side of the Layout Editor and drag it into the layout area.

5 Select the slider tool from the component palette at the left side of the Layout Editor and drag it into the layout area.

6 Select **File** > **Save**. Save the UI as `myslider.fig`. MATLAB opens the code file in the Editor.

7 Set the initial value of the application data in the opening function, `myslider_OpeningFcn`. This function executes just before the UI is visible to users. In `myslider_OpeningFcn`, insert these commands immediately after the command, `handles.output = hObject`.

```
setappdata(handles.figure1,'slidervalue',0);
setappdata(handles.figure1,'difference',1);
```
After you add the commands, `myslider_OpeningFcn` looks like this.

```
function myslider_OpeningFcn(hObject,eventdata,handles,varargin)
% This function has no output args, see OutputFcn.
% hObject    handle to figure
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)
% varargin   command line arguments to junk (see VARARGIN)

% Choose default command line output for junk
handles.output = hObject;
setappdata(handles.figure1,'slidervalue',0);
setappdata(handles.figure1,'difference',1);

% Update handles structure
guidata(hObject, handles);

% UIWAIT makes junk wait for user response (see UIRESUME)
% uiwait(handles.figure1);
```
Notice that `handles` is an input argument to `myslider_OpeningFcn`. The `handles` variable is a structure that contains all the components in the UI. Each field in this structure corresponds to a separate component. Each field name matches the `Tag` property of the corresponding component. In this case, `handles.figure1` is the figure object. Thus, `setappdata` can use this figure object to store the data.

8 Add code to the slider callback for changing the data. Add these commands to the end of the function, `slider1_Callback`.

```
maxval = get(hObject,'Max');
currval = get(hObject,'Value');
diffMax = maxval - currval;
% Store application data
setappdata(handles.figure1,'slidervalue',currval);
setappdata(handles.figure1,'difference',diffMax);
```
After you add the commands, slider1_Callback looks like this.

```
function slider1_Callback(hObject, eventdata, handles)
% hObject    handle to slider1 (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)

% Hints: get(hObject,'Value') returns position of slider
%        get(hObject,'Min') and get(hObject,'Max') to determine range of slider
maxval = get(hObject,'Max');
currval = get(hObject,'Value');
diffMax = maxval - currval;
% Store application data
setappdata(handles.figure1,'slidervalue',currval);
setappdata(handles.figure1,'difference',diffMax);
```
This callback function has access to the handles structure, so the setappdata commands store the data in handles.figure1.

9 Add code to the push button callback for retrieving the data. Add these commands to the end of the function, pushbutton1_Callback.

```
% Retrieve application data
currentval = getappdata(handles.figure1,'slidervalue');
diffval = getappdata(handles.figure1,'difference');
display([currentval diffval]);
```
After you add the commands, pushbutton1_Callback looks like this.

```
% --- Executes on button press in pushbutton1.
function pushbutton1_Callback(hObject, eventdata, handles)
% hObject    handle to pushbutton1 (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)

% Retrieve application data
currentval = getappdata(handles.figure1,'slidervalue');
diffval = getappdata(handles.figure1,'difference');
display([currentval diffval]);
```

This callback function has access to the `handles` structure, so the `getappdata`
commands retrieve the data from `handles.figure1`.

10 Save your code by pressing **Save** in the Editor Toolstrip.

Create Nested Callback Functions (Programmatic UIs)

You can nest callback functions inside the main function of a programmatic UI. When
you do this, the nested callback functions share a workspace with the main function. As a
result, the nested functions have access to all the UI components and variables defined in
the main function. The following example code uses nested functions to share data about
the slider position. To see how it works, copy and paste this code into an editor and run
it.

```matlab
function my_slider()
 hfig = figure();
 data = struct('val',0,'diffMax',1);
 slider = uicontrol('Parent', hfig,'Style','slider',...
          'Units','normalized',...
          'Position',[0.3 0.5 0.4 0.1],...
          'Tag','slider1',...
          'Callback',@slider_callback);

 button = uicontrol('Parent', hfig,'Style','pushbutton',...
          'Units','normalized',...
          'Position',[0.4 0.3 0.2 0.1],...
          'String','Display Difference',...
          'Callback',@button_callback);

 function slider_callback(hObject,eventdata)
  sval = hObject.Value;
  diffMax = hObject.Max - sval;
  % For R2014a and earlier:
  % sval = get(hObject,'Value');
  % maxval = get(hObject,'Max');
  % diffMax = maxval - sval;

  data.val = sval;
  data.diffMax = diffMax;
 end

 function button_callback(hObject,eventdata)
  display([data.val data.diffMax]);
 end
```

```
end
```
The main function defines a `struct` array called `data`. When the user moves the slider, the `slider_callback` function updates the `val` and `diffMax` fields of the `data` structure. When the end user clicks the push button, the `button_callback` function displays the values stored in `data`.

Note: Nested functions are not recommended for GUIDE UIs.

Store Data Using the guidata Function

The `guidata` function provides a way to share data with the figure window. You can store or retrieve your data in any callback through the `hObject` component. This means that, unlike working with `UserData` or application data, you do not need access to one specific component to set or query the data. Call `guidata` with two input arguments to store data:

```
guidata(object_handle,data);
```
The first input, `object_handle`, is any UI component (typically `hObject`). The second input, `data`, is the variable to store. Every time you call `guidata` using two input arguments, MATLAB overwrites any previously stored data. This means you can only store one variable at a time. If you want to share multiple values, then store the data as a `struct` array or cell array.

To retrieve data, call `guidata` using one input argument and one output argument:

```
data = guidata(object_handle);
```
The component you specify to store the data does not need to be the same component that you use to retrieve it.

If your data is stored as a `struct` array or cell array, and you want to update one element without changing the other elements, then retrieve the data and replace it with the modified array:

```
data = guidata(hObject);
data.myvalue = 2;
guidata(hObject,data);
```

Use guidata in UIs Created Programmatically

To use `guidata` in a programmatic UI, store the data with some initial values in the main function. Then you can retrieve and modify the data in any callback function.

11-13

The following code is a simple example of a programmatic UI that uses `guidata` to share a structure containing two fields. To see how it works, copy and paste this code into an editor and run it.

```matlab
function my_slider()
hfig = figure();
guidata(hfig,struct('val',0,'diffMax',1));
slider = uicontrol('Parent', hfig,'Style','slider',...
        'Units','normalized',...
        'Position',[0.3 0.5 0.4 0.1],...
        'Tag','slider1',...
        'Callback',@slider_callback);

button = uicontrol('Parent', hfig,'Style','pushbutton',...
        'Units','normalized',...
        'Position',[0.4 0.3 0.2 0.1],...
        'String','Display Values',...
        'Callback',@button_callback);
end

function slider_callback(hObject,eventdata)
 data = guidata(hObject);
 data.val = hObject.Value;
 data.diffMax = hObject.Max - data.val;
 % For R2014a and earlier:
 % data.val = get(hObject,'Value');
 % maxval = get(hObject,'Max');
 % data.diffMax = maxval - data.val;

 guidata(hObject,data);
end

function button_callback(hObject,eventdata)
 data = guidata(hObject);
 display([data.val data.diffMax]);
end
```

When the user moves the slider, the `slider_callback` function executes these commands to retrieve and modify the stored data:

- `data = guidata(hObject)` retrieves the stored data as a structure.

- `data.diffMax = maxval - data.val` modifies the `diffMax` field in the structure.

- `guidata(hObject,data)` stores the modified structure.

When the user clicks the push button, the `button_callback` function calls `guidata` to retrieve a copy of the stored structure. Then it displays the two values stored in the structure.

Use guidata in GUIDE UIs

GUIDE uses the `guidata` function to store a structure called `handles`, which contains all the UI components. MATLAB passes the `handles` array to every callback function. If you want to use `guidata` to share additional data, then add fields to the `handles` structure in the opening function. The opening function is a function defined near the top of your code file that has `_OpeningFcn` in the name.

To modify your data in a callback function, modify the `handles` structure, and then store it using the `guidata` function. This slider callback function shows how to modify and store the `handles` structure in a GUIDE callback function.

```
function slider1_Callback(hObject, eventdata,handles)
% hObject    handle to slider1 (see GCBO)
% eventdata reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)

% Hints: get(hObject,'Value') returns position of slider
%        get(hObject,'Min') and get(hObject,'Max') to determine range
 handles.myvalue = 2;
 guidata(hObject,handles);
end
```

To see a full example of a GUIDE UI that uses the `guidata` function, follow these steps to copy and examine the code.

1 Set your current folder to one to which you have write access.

2 Copy the example code.

```
copyfile(fullfile(docroot, 'techdoc','creating_guis',...
   'examples','sliderbox_guidata.*')),...
    fileattrib('sliderbox_guidata.*', '+w');
```

3 Display this example in the GUIDE Layout Editor:

```
guide sliderbox_guidata.fig
```

4 View the code in the Editor by clicking the **Editor** button, .

5 Run the program by clicking the **Run Figure** button, .

Sharing Data Among Multiple GUIDE UIs

Example of Sharing Data Between Two UIs

To see a full-featured example that uses application data and the `guidata` function to share data between two separate UIs, use these steps to copy, run, and examine the code.

1 Set your current folder to one to which you have write access.

2 Copy the example code with this command:

```
copyfile(fullfile(docroot, 'techdoc','creating_guis','examples'...
  ,'changeme*.*')),fileattrib('changeme*.*', '+w');
```

3 Display the UIs in two GUIDE Layout Editors with these commands:

```
guide changeme_main
guide changeme_dialog
```

4 View the code in the editor by clicking the **Editor** button, 🖼, in both Layout Editors.

5 Run the `changeme_main` program by clicking **Run Figure** button, ▶, in that UI's Layout Editor.

Example of Sharing Data Among Three UIs

To see a full-featured example that uses `guidata` and `UserData` data to share data among three UIs, use these steps to copy, run, and examine the code.

1 Set your current folder to one to which you have write access.

2 Copy the example code with this command

```
copyfile(fullfile(docroot, 'techdoc','creating_guis',...
  'examples','guide*.*')),...
    fileattrib('guide*.*', '+w');
```

3 Display the UIs in three GUIDE Layout Editors with these commands:

```
guide guide_iconeditor;
guide guide_toolpalette;
guide guide_colorpalette;
```

4 View the code in the editor by clicking the **Editor** button, 🖼, in each Layout Editor.

5 Run the `guide_iconeditor` program by clicking **Run Figure** button, ▶, in that UI's Layout Editor.

More About

- "Nested Functions"
- "Interrupt Callback Execution" on page 12-2
- "Write Callbacks in GUIDE" on page 7-2
- "Write Callbacks for Apps Created Programmatically" on page 10-5

Manage Callback Execution

Interrupt Callback Execution

In this section...
"How to Control Interruption" on page 12-2
"Callback Behavior When Interruption is Allowed" on page 12-2
"Example" on page 12-3

MATLAB lets you control whether or not a callback function can be interrupted while it is executing. For instance, you can allow users to stop an animation loop by creating a callback that interrupts the animation. At other times, you might want to prevent potential interruptions, when the order of the running callback is important. For instance, you might prevent interruptions for a `WindowButtonMotionFcn` callback that shows different sections of an image.

How to Control Interruption

Callback functions execute according to their order in a queue. If a callback is executing and a user action triggers a second callback, the second callback attempts to interrupt the first callback. The first callback is the *running callback*. The second callback is the *interrupting callback*.

Two property values control the response to an interruption attempt:

- The `Interruptible` property of the object owning the running callback determines if interruption is allowed. A value of `'on'` allows the interruption. A value of `'off'` does not allow the interruption. The default value is `'on'`.

- If interruption is not allowed, then the `BusyAction` property (of the object owning the interrupting callback) determines if MATLAB enqueues or discards the interrupting callback. A value of `'queue'` allows the interrupting callback to execute after the running callback finishes execution. A value of `'cancel'` discards the interrupting callback. The default value is `'queue'`.

Callback Behavior When Interruption is Allowed

When an object's `Interruptible` property is set to `'on'`, its callback can be interrupted at the next occurrence of one of these commands: `drawnow`, `figure`, `getframe`, `waitfor`, or `pause`.

- If the running callback contains one of these commands, then MATLAB stops the execution of the running callback and executes the interrupting callback. MATLAB resumes executing the running callback when the interrupting callback completes.

- If the running callback does not contain one of these commands, then MATLAB finishes executing the callback without interruption.

For more details about the interruptible property and its effects, see the `Interruptible` property description on the Uicontrol Properties page.

Example

This example shows how to control callback interruption using the `Interruptible` and `BusyAction` properties.

Copy the Source File

1   In MATLAB, set your current folder to one in which you have write access.

2   Execute this MATLAB command:
```
copyfile(fullfile(docroot,
'techdoc','creating_guis','examples',...
'callback_interrupt.m')),fileattrib('callback_interrupt.m',
'+w');
```

Run the Example Code

Execute the command, `callback_interrupt`. The program displays two windows.
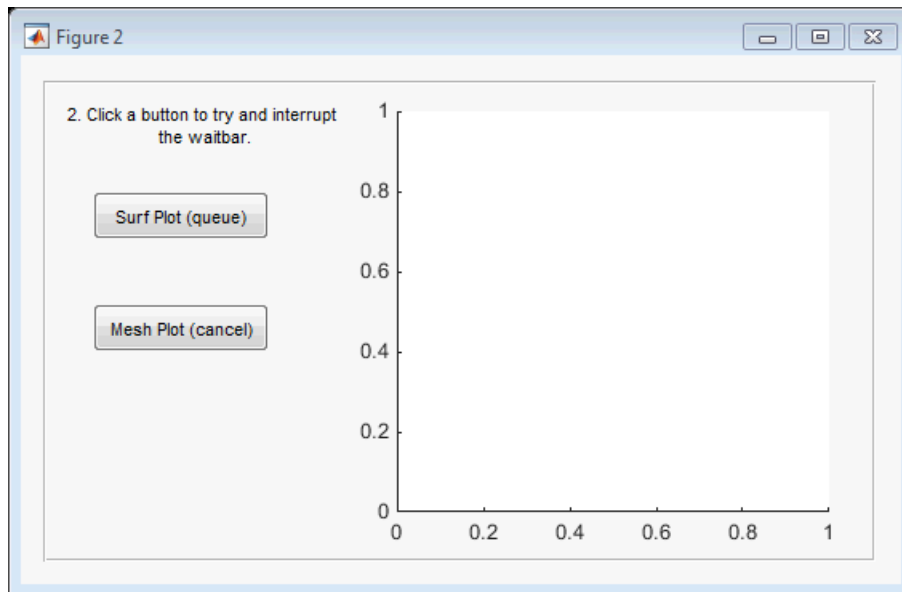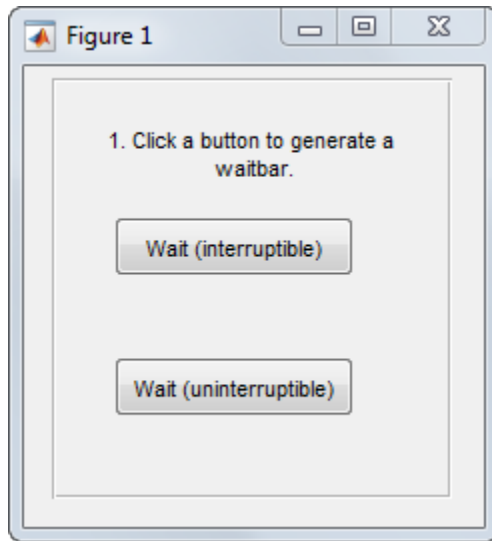
Clicking specific pairs of buttons demonstrates the effect of different property value combinations :

- *Callback interruption* — Click **Wait (interruptible)** immediately followed by either button in the second window: **Surf Plot (queue)** or **Mesh Plot (cancel)**. The wait bar displays, but is momentarily interrupted by the plotting operation.
- *Callback queueing* — Click **Wait (uninterruptible)** immediately followed by **Surf Plot (queue)**. The wait bar runs to completion. Then the surface plot displays.
- *Callback cancellation* — Click **Wait (uninterruptible)** immediately followed by **Mesh Plot (cancel)**. The wait bar runs to completion. No plot displays because MATLAB discards the mesh plot callback.

Examine the Source Code

The `Interruptible` and `BusyAction` properties are passed as input arguments to the `uicontrol` function when each button is created.

Here is the command that creates the **Wait (interruptible)** push button. Notice that the `Interruptible` property is set to `'on'`.

```
h_interrupt = uicontrol(h_panel1,'Style','pushbutton',...
                    'Position',[30,110,120,30],...
                    'String','Wait (interruptible)',...
                    'TooltipString','Interruptible = on',...
                    'Interruptible','on',...
                    'Callback',@wait_interruptible);
```

Here is the command that creates the **Wait (uninterruptible)** push button. Notice that the `Interruptible` property is set to `'off'`.

```
h_nointerrupt = uicontrol(h_panel1,'Style','pushbutton',...
                    'Position',[30,40,120,30],...
                    'String','Wait (uninterruptible)',...
                    'TooltipString','Interruptible = off',...
                    'Interruptible','off',...
                    'Callback',@wait_uninterruptible);
```

Here is the command that creates the **Surf Plot (queue)** push button. Notice that the `BusyAction` property is set to `'queue'`.

```
hsurf_queue = uicontrol(h_panel2,'Style','pushbutton',...
                    'Position',[30,200,110,30],...
                    'String','Surf Plot (queue)',...
                    'BusyAction','queue',...
                    'TooltipString','BusyAction = queue',...
                    'Callback',@surf_queue);
```

Here is the command that creates the **Mesh Plot (cancel)** push button. Notice that the
`BusyAction` property is set to `'cancel'`.

```
hmesh_cancel = uicontrol(h_panel2,'Style','pushbutton',...
                        'Position',[30,130,110,30],...
                        'String','Mesh Plot (cancel)',...
                        'BusyAction','cancel',...
                        'TooltipString','BusyAction = cancel',...
                        'Callback',@mesh_cancel);
```

See Also
drawnow | timer | uiwait | waitfor

Related Examples
- "Automatically Refresh Plot in a GUIDE UI" on page 8-44
- "Use a MATLAB Timer Object"

More About
- "Finding Code Bottlenecks"
- "Write Callbacks for Apps Created Programmatically" on page 10-5

App Designer

13

App Designer Basics

Create Simple App Using App Designer

> **Note:** This topic applies to apps you create using App Designer or the `uifigure` function. For alternative ways to build apps, see "Ways to Build Apps" on page 1-2.

This simple app displays the value of a slider in a semicircular gauge after the app user moves the slider bar to a new location. For more example apps, see "Simple Example Apps for App Designer" on page 16-31.

The running app looks like this image:
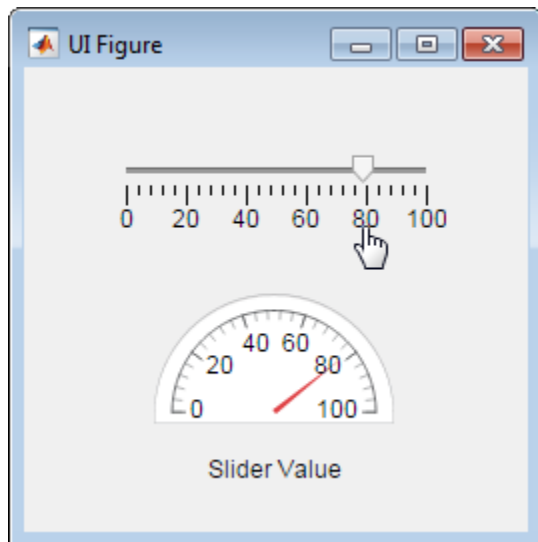
To create the app:

1   Open App Designer.

    In the MATLAB Command Window, type `appdesigner`.

2   Add the components to the design area.

    a   From the App Designer **Component Library** on the left, press **Ctrl** and drag a Slider component into the design area.

> **Tip** Pressing the **Ctrl** key as you drag a component into the design area
> suppresses automatic inclusion of a label with the component.

b From the App Designer **Component Library**, drag a **Semicircular Gauge**
component into the central design area.

By default, App Designer also adds a label component.

c Replace the default label text by clicking the label text, `Semicircular Gauge`,
and then typing `Slider Value`.

3 Add a slider callback. Callbacks are functions that you code to respond to app user
interactions with app components.

In the central design area, right-click the slider and select **Callbacks** > **Add
ValueChangedFcn callback**.
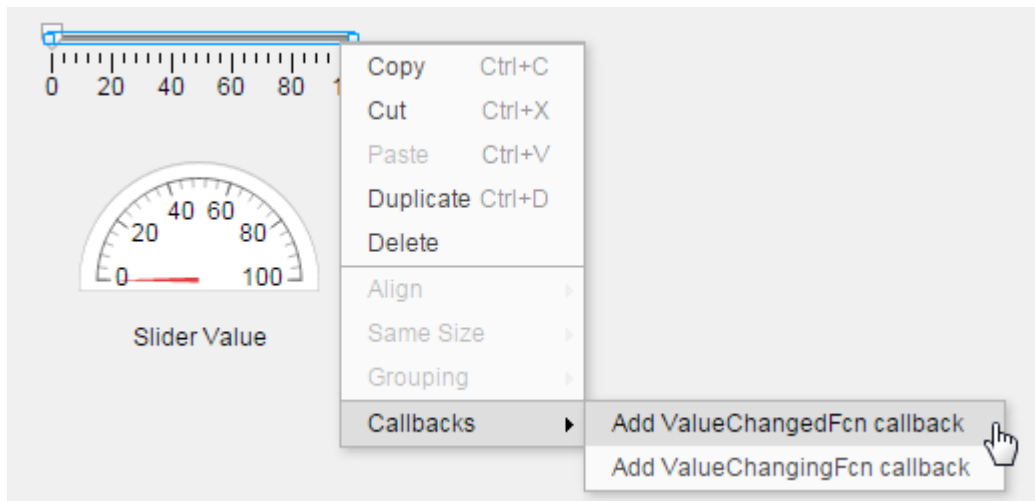
4 Accept the default callback name by clicking **OK** in the Add Callback Function dialog
box.

Although not done here (to keep the example simple), changing the function name
makes it easier to associate the callback code with a component. This practice is
especially effective when there are multiple instances of the same component in an
app.

5 In the code view that opens, notice the default code in the `SliderValueChanged` function.

```
 value = app.Slider.Value;
```

6 Replace the default code with code that sets the semicircular gauge value to the slider value.

To access or set a property value, specify `app`, the component, and the component property name using dot notation.

Replace the default callback code with this code.

```
app.SemicircularGauge.Value = app.Slider.Value;
```

7 Click **Run**.

Save the app when prompted.
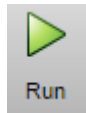
8 Test the app.

In the running app, move the slider and release the mouse button. The gauge value changes to reflect the new value.

See Also
"Code Response to Reflect Changing Slider Value" on page 16-42 | "Graphics Support in App Designer" on page 13-11

Open or Run App Designer Apps

In this section...
"Open App Designer" on page 13-5
"Open Existing App" on page 13-5
"Run Existing App" on page 13-5

Open App Designer

To open App Designer, use either of these methods:

- At the MATLAB command prompt, type `appdesigner`.
- On the MATLAB **Home** tab, select **New** > **App** > **App Designer**.

Open Existing App

To open an existing app, use either of these methods:

- On the App Designer toolstrip, click the **Designer** tab, and then click **Open**.
- At the MATLAB command prompt, type `appdesigner file-name`, where `file-name` is file name, full file path, or partial path of the MLAPP file.

Run Existing App

To run an existing app, use either of these methods:

- Type the file name at the MATLAB command prompt.

  The file must be in a folder on your MATLAB path.
- On the file in App Designer, and then click **Run** on the toolstrip.

Differences Between App Designer and GUIDE

App Designer is a design environment for building apps. Although it provides many of the same controls as GUIDE, the process for building apps is different. Most significantly, the graphics support, generated code, component access, callback coding, and plotting component are different. This table summarizes the differences.

What Is Different	In GUIDE	In App Designer
"Figure Support" on page 13-7	Use the `figure` function and Figure properties.	Use the `uifigure` function and UI Figure properties.
"Axes Support" on page 13-7	Use the `axes` function and Axes properties to access all the graphics functionality available in MATLAB.	Use the `uiaxes` function and UI Axes properties to plot 2-D line and scatter plots only.
"Code Structure" on page 13-7	Code is a series of local functions for callbacks and utility functions.	Code is a MATLAB class containing app components, callbacks, utility functions, and properties to manage and share data.
Code Editability	All code is editable.	Only callback code, utility functions, and user-defined properties are editable.
"Component Access and Configuration" on page 13-8	Use `get` and `set` functions.	Use dot notation.
"Callback Configuration" on page 13-8	Use the `Callback` property.	Use action-specific callbacks.
"Callback Arguments" on page 13-9	`hObject`, `eventdata`, and a `handles` structure are used.	`app` and `event` data (when needed) are used.
"Data Sharing" on page 13-9	Use the `UserData` property, or `guidata`, or `setappdata` functions	Use MATLAB class properties that you create.
"Component Creation" on page 13-10	Use the `uicontrol` function and uicontrol properties.	Use a component-specific function and its corresponding properties.

Figure Support

In both GUIDE and App Designer, a figure window contains UI components. GUIDE creates a figure window by calling the `figure` function. App Designer creates a UI figure that is equivalent to the window created by the `uifigure` function.

Some functions that you can use with `figure` windows are not supported for use with the UI figure window. In addition, some components are supported by only one type of figure window. For details, see "Graphics Support in App Designer" on page 13-11

Axes Support

In both GUIDE and App Designer, you plot data using an axes component. GUIDE calls the `axes` function to create an axes object. App Designer calls the `uiaxes` function to create a UIAxes object. UIAxes objects support a subset of plotting functions and features. For details, see "Graphics Support in App Designer" on page 13-11,

Code Structure

App Designer code has a different structure and characteristics from GUIDE code:

- MATLAB file structure

  App Designer code uses the MATLAB class structure, which is easy to read. App properties appear first, followed by user-defined functions, and then the code used for app initialization and component creation.

- Integrated editor

  You edit app code within App Designer using an integrated version of the MATLAB Editor. The integrated editor includes features such as debugging, smart indenting, and Code Analyzer indicators.

- Code editing

  To prevent you from accidentally overwriting code that App Designer manages, some code is not editable. Uneditable code is indicated in the editor by a light gray background. You can write and edit callback code, helper functions, and property declarations that you create.

- No default callback declarations

App Designer only creates a callback declaration when you explicitly request it. For example, if you right-click a component in the canvas, and then select a callback from the context menu, App Designer creates a callback declaration. This convention prevents empty callback declarations from cluttering your code.

- Default sample callback code

  When you explicitly add a callback declaration, App Designer adds sample code to the callback by default. When the app user triggers the callback by manipulating the component in the running app, the sample code accesses a component property value. The sample code provides an example of how to access a component and its properties from within a callback. You can replace this code with code that is appropriate for your app.

  Note: App Designer does not generate sample callback code when you create a callback for a button component.

- Single file

  App Designer creates a single MLAPP file that contains the information required to display and run your app. There is no associated FIG file.

For details see, "App Designer Code Generation" on page 16-2.

Component Access and Configuration

App Designer defines components as properties of the app. You can access components and their property values using dot notation. For example, this code sets the `Value` property of a numeric edit field named `Temperature` to 15:

```
app.Temperature.Value = 15;
```

As shown in the example, you always indicate that components are app object properties by including `app` in the dot notation. For a complete example, see "Create Simple App Using App Designer" on page 13-2.

Callback Configuration

GUIDE `uicontrol` components provide a `Callback` property that executes when the app user interacts with each component in a typical way (such as clicking a button). Most

App Designer components that support callbacks provide a `ValueChangedFcn` callback instead. For an example, see "Create Simple App Using App Designer" on page 13-2.

Some components support additional callbacks. For instance, sliders support a `ValueChangingFcn` callback that triggers repeatedly as the app user moves the slider. For an example, see "Code Response to Reflect Changing Slider Value" on page 16-42.

Callback Arguments

In GUIDE, all callback definitions pass `hObject`, `eventdata`, and a `handles` structure as arguments.

In App Designer, all callback definitions specify `app` as an input argument instead of `hObject` and a `handles` structure. All components are defined and accessed as properties of the `app` object. Event data is passed only when it is not empty.

For example, compare the callback definition for a check box that assigns the state of the check box to a variable, `value`.

In GUIDE, the check box callback definition is this:

```
function checkbox1_Callback(hObject,eventdata,handles)
value = get(hObject,'Value');
end
```

In App Designer, assuming the code name for the check box is `CheckBox`, the check box callback definition is this:

```
function CheckBoxValueChanged(app)
value = app.CheckBox.Value;
end
```

GUIDE always passes an empty `eventdata` argument to callbacks. App Designer only passes `event` data when the event value is not empty. For an example that uses the `event` argument, see "Code Response to Reflect Changing Slider Value" on page 16-42.

Data Sharing

In GUIDE, to share data across callbacks, you can use the `UserData` property or the `guidata` or `setappdata` functions.

In App Designer, to share data across callbacks, you create and use a property. For example, suppose that you create the `MyVal` property to share a value across your app callbacks. To set the property value, use dot notation:

```
app.MyVal = 15;
```

Then, if you want to set the `Value` property of a numeric edit field named `Temp` to the `MyVal` property value use this code:

```
app.Temp.Value = app.MyVal;
```

When referencing app properties, including components, always use `app` in the dot notation.

For more information and an example of data sharing using App Designer, see "About App Properties" on page 16-19.

Component Creation

GUIDE creates most components with the `uicontrol` function, and you use `uicontrol` properties to control component appearance and behavior. However, some `uicontrol` properties are applicable to only a subset of components.

App Designer code uses a different function for creating each type of component. For instance, `uilistbox` creates a list box and `uislider` creates a slider. Each component provides a set of properties designed specifically for that type of component. For example, when you select a slider in App Designer, the property editor and inspector present only slider options. For more information, see "Choose Components for Your App Designer App" on page 14-2 and "Customize App Designer Components" on page 14-11.

Graphics Support in App Designer

In this section...
"Graphics Support" on page 13-11
"Properties and Component Support" on page 13-12

The types of charts your app can support depend largely on the kind of figure that underlies the UI. Apps you create using GUIDE, and those you create programmatically with uicontrols, use traditional figures and axes. These apps support all of the graphics functionality available in MATLAB.

Apps you create using App Designer are based on a new kind of figure, called a *UI figure*. To display graphics in these apps, you must use a new type of axes object, called *UI axes*.

UI figures and UI axes are similar to traditional figures and axes, but there are some important differences to be aware of when deciding how to build your app.

Graphics Support

If you are creating an app that displays graphics, you must specify the target UI figure or UI axes when you call certain graphics functions. If you do not specify the target object, MATLAB uses `gcf` or `gca` to determine the target object, and the result might not be what you expected.

Furthermore, UI figures do not support most of the interactive functionality that traditional figures support. For example, UI figures do not support panning, zooming, mouse, or keyboard interactions.

The following table lists the graphics functions that UI figures and UI axes support in R2016a. All other graphics functions are unsupported.

Category	Supported Functions	Notes
Charting Functions and Graphics Objects	plot scatter line text title hold alpha	You must specify the UI axes as the target axes when using these functions. Otherwise, MATLAB uses `gca` to determine the target axes, and the result might not be what you expected.

Category	Supported Functions	Notes

	colormap	This code shows how to plot two lines in App Designer using the `hold` function:
		
		``
		plot(app.UIAxes,[1 2 3 4],'-r');
		hold(app.UIAxes);
		plot(app.UIAxes,[10 9 4 7],'--b');
		``

Coordinate Systems	xlim
	ylim
	xlabel
	ylabel
	grid
	box
		Specify the UI axes as the target axes when using these functions. For example, this code shows how to set the *x*-label on a plot in App Designer:
		
		``
		xlabel(app.UIAxes,'Frequency');
		``

Utilities	set
	get
	delete
	findobj
	reset
	ishandle
	isgraphics
	newplot
	ancestor
		Specify the UI figure or the UI axes as the target object when using these functions. For example, this code shows how to find a UI axes object in the figure, `app.UIFigure`. In this case, `findobj` searches the UI figure for an object whose `XLim` property is property is `[0 1]`.
		
		``
		ax = findobj(app.UIFigure,'XLim',[0 1]);
		``

Figures	close	Specify the UI figure as the target figure. For example:
		
		``
		f = uifigure;
		close(f);
		``

All other graphics functions that are not listed here are unsupported in App Designer, UI figures, and UI axes.		

Properties and Component Support

If you are moving code you wrote in previous releases into App Designer, you might encounter these limitations:

- UI figures support only a subset of the properties that traditional figures support. For example, UI figures do not support properties for printing, saving, callback execution, or custom mouse and keyboard interaction. For a full list of supported properties, see UI Figure (App Designer) Properties.

- UI axes support only a subset of the properties that traditional axes support. For example, UI axes do not support lighting and camera properties or properties for interactive control. For a full list of supported properties, see UI Axes (App Designer) Properties.

- UI figures support a different set of interactive components than traditional figures do. For example, UI figures do not support uicontrols or uimenus. For a full list of supported components, see "Components in App Designer".

See Also

UI Axes (App Designer) Properties | UI Figure (App Designer) Properties

More About

- "Differences Between App Designer and GUIDE" on page 13-6
- "Ways to Build Apps" on page 1-2

Component Choices and Customizations

Choose Components for Your App Designer App

App Designer provides many components for designing your app. The following table suggests which components to use based on what you want to accomplish with the component.

Your Goal	Suggested Components
"Graph Data" on page 14-2	Axes (Line and Scatter)
"Get Numeric Input" on page 14-3	Numeric edit field or spinner
"Get Text Input" on page 14-5	Text edit field or text area
"Allow Command Execution" on page 14-5	Button
"Allow Selection Between Mutually Exclusive States" on page 14-6	Check box, state button, or one of the switch components
"Allow Selection Among Two or More Options" on page 14-6	Drop-down component component, list box, radio or toggle button group, or discrete knob
"Indicate Status Visually" on page 14-8	Lamp
"Provide Numeric Display" on page 14-8	Gauge
"Organize Components" on page 14-9	Label
"Organize Components" on page 14-9	Tab group or panel

Graph Data

To graph data in a 2D line plot or scatter plot, use a UIAxes component. UIaxes components support a subset of graphics functionality. For details, see "Graphics Support in App Designer" on page 13-11.

Sample 2D Line and Scatter Plots
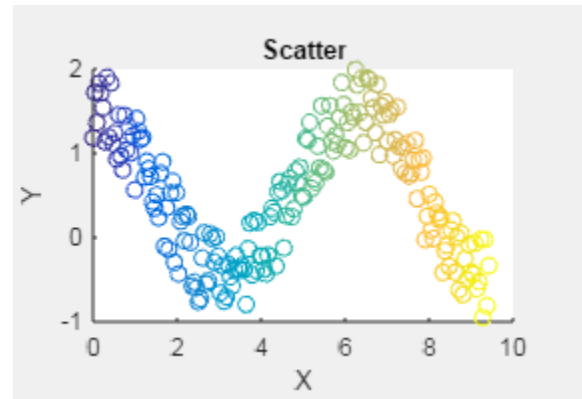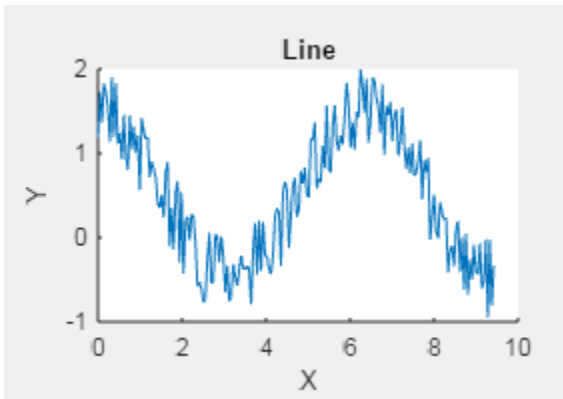
Get Numeric Input

You can use a component that enables the app user to either enter numeric data or select it. Numeric data entry allows the app user to specify a precise value. The component does not indicate the upper and lower limits. Numeric data selection does not allow the app user to specify precise values, but the component does indicate the upper and lower limits.

Numeric Data Entry

When you want your app to accept precise numeric data entry, use a numeric edit field or a spinner. A spinner enables the app user to change the numeric value in uniform increments (using arrow buttons).

Using these components, you can:

- Specify a limited or infinite range of values as valid.
- Display the value in a particular numeric display format, such as scientific notation, but store the value exactly as entered.
- Code these components to execute a function callback when the app user changes the numeric value and clicks outside the component.
- Give the app user the ability to specify precise values.

Sample Numeric Edit Fields

As shown in some of the samples that follow, if the app user enters an invalid value, the component flags the error. The component handles the error automatically and returns the value to the previous setting. *There is no need for you to code the validity checking. The component does it for you.*

| 0 | 0.000000e+00 | 0 | John◄ Value must be numeric |

↑
Value reverts to previous value
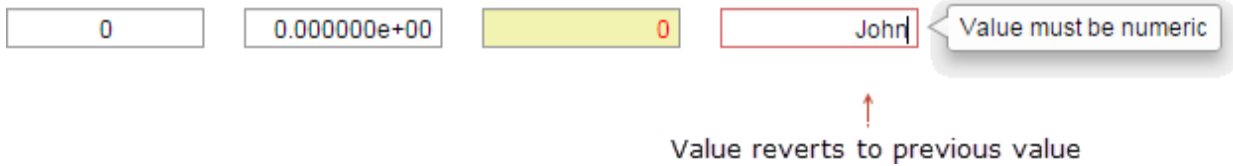
Sample Spinners

As shown in some of the samples that follow, if the app user enters an invalid value, the component flags the error. The component handles the error automatically and returns the value to the previous setting. *There is no need for you to code the validity checking. The component does it for you.*

| 0 ⬍ | 0 ⬍ | 0 ⬍ | 500 ⬍ ◄ Value must be between 0 and 100 |

↑
Value reverts to previous value
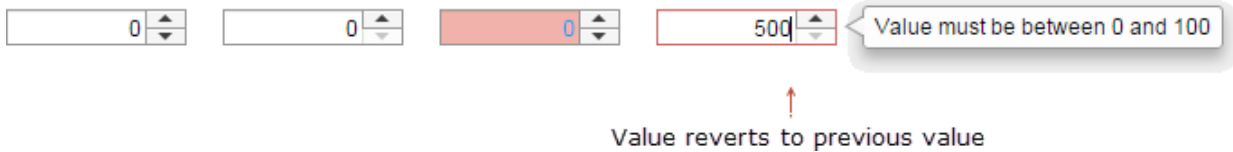
Numeric Data Selection

When you want the app user to select a numeric value from within a finite range of values, use a slider or a knob. These components provide a visual representation to the app user, showing where a value selection falls within a range. Consider using a slider when changing the slider orientation enables you to make best use of space in your app layout. Consider using a knob when you want a component to replicate an instrument visually. These components do not accommodate precise value input.

You can code these components to execute a function callback when the app user changes the component value, as the app user changes the value, or both.

Sample Sliders

Sample Knobs
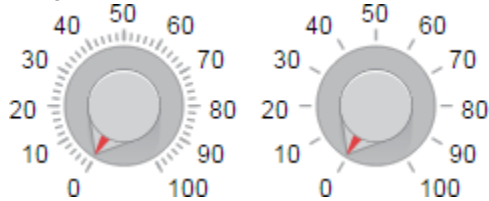
Get Text Input

To get text data, use a text edit field or a text area. To allow single-line text entry, use a text edit field. To allow multiline text entry, use a text area. A scroll bar appears when the app user hovers the mouse pointer over text that exceeds the boundaries of the text area.

Sample Text Edit Fields
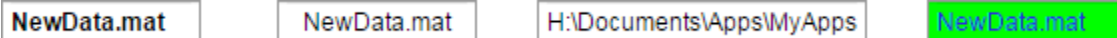
Sample Text Area Fields
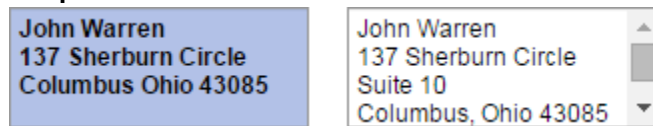
Allow Command Execution

Use a button to enable the app user to execute a command, such as to stop a simulation, or to open help.

You can code this component to execute a function callback when the app user clicks the button.

Sample Buttons
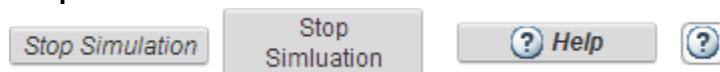
See also "Specify Multiline Text on Components" on page 14-14

Allow Selection Between Mutually Exclusive States

To allow the app user to select between two mutually exclusive states, use a check box, state button, or a switch (rocker switch, switch, or toggle switch). Consider using a check box when you want to provide a few words to describe the choice. Use a state button a when a short description or icon on the button, such as Play describes the purpose of the button. Use a switch when you want the component to replicate an instrument component visually.

You can code these components to execute a function callback when the app user:

- Selects or clears a check button
- Presses or releases a state button
- Toggles a switch

Sample Check Box

☐ reload files

Sample State Buttons

Sample Switches

## Allow Selection Among Two or More Options

To allow the app user to select from among two or more mutually exclusive options, use a drop-down component, button group, list box, or discrete knob. *There is no need for you to code the mutual exclusivity. The component manages it for you.*

You can code these components to execute a function callback when the app user commits a change to the component.

Requirement	Component to Consider
Allow app user to select from among a set of predefined choices.	Drop-down component, list box, button group, or discrete knob
Allow app user to enter a different value than the values presented	Drop-down component with the **Allow users to type in text** option selected. (`Enabled` property set to `'on'`.)
Provide a long list of options in a space-constrained area of the app. Add or remove options from the list programmatically, as the app is running.	List box or drop-down component A scroll bar is added to the list box automatically when the list of options exceeds the height of the list box.
Disable one or more options at design time or programmatically.	Radio button or toggle button group A disabled radio button appears dimmed. A disabled toggle button does not depress when the app user clicks it.
Allow app user to select multiple values.	List component (with the `Multiselect` property value `'on'`)
Have the component to replicate an instrument visually.	Discrete knob

Sample Drop-Down Components
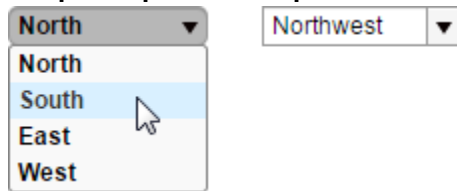
Sample List Box
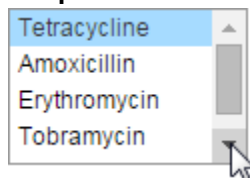
Sample Radio Button Group and Toggle Button Group
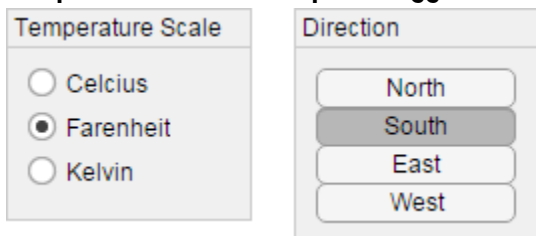
Sample Discrete Knobs
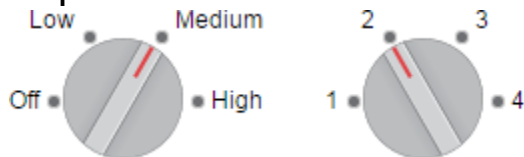
Indicate Status Visually

To indicate a status in your app, use a lamp. Typically, changes to another component trigger a callback to change the lamp color and indicate a change in status.

Sample Lamps

Provide Numeric Display

To display measurements in your app, such as the current temperature, use a gauge. Typically, another component callback changes the gauge value. App Designer provides circular, ninety-degree, semicircular, and linear gauges

Sample Gauges
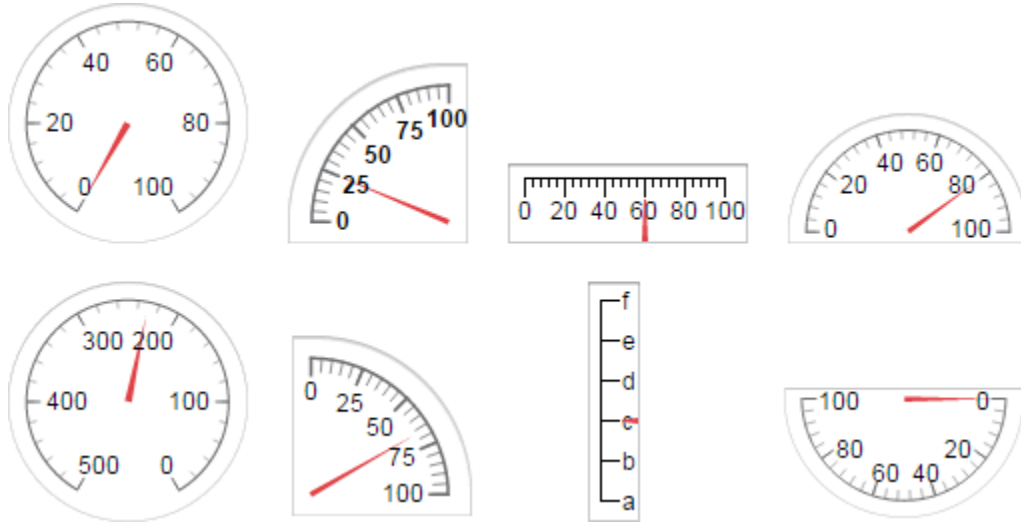
Identify Components

To convey information, such as the type of values you want the app user to specify or the units of measurement, use labels with other components.

Organize Components

To organize components into collections of components, use panels or tab groups. Consider using tab groups when you want to conserve app space and the app user does not need to see the content on multiple tabs concurrently.

Sample Panel

Sample Tab Groups

Temperature	Pressure

Input Air: [0]

Input Water: [0]

Output Air: [0]

Oput Waterut [0]

Input Air: [0]

Input Water: [0]

Output Air: [0]

Oput Waterut [0]

Temperature	Pressure

Customize App Designer Components

Set Commonly Used Component Properties

You can customize commonly used component properties in design view. The properties panel, by default, appears to the right of the design area. This panel presents commonly used properties for which you can adjust values.

After you add a component to the central design area, click it if it is not currently selected. Next, change the values on the **Configuration** tab in the property panel to the right of the design area. For instance, this image shows the property panel that displays when you select a button in the design area.
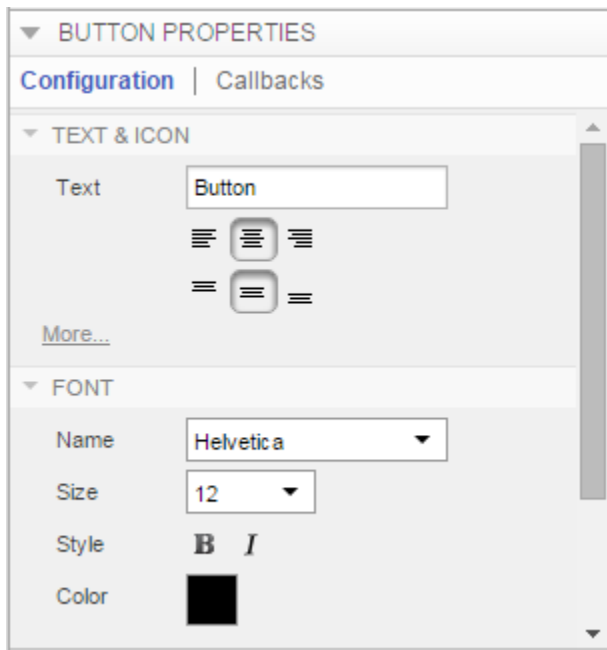
Changes you make are reflected in the app code (which you can view by clicking **Code View**).

Set Property Values for Multiple Components Simultaneously

You can set many property values for multiple components simultaneously. For instance, to make the font properties for all components that support font properties to be the same, follow these steps.

1   In the design area, multiselect the components by holding down the **Ctrl** key and clicking each component you want to select. Alternatively, left-click a blank spot in the design area and drag a selection border around the components you want to select.

   Selecting a design group is the same as individually selecting the components within the design group. For example, this image shows a selection that includes a set of grouped buttons and an ungrouped button. For information on creating design groups, see "Group Components for Layout Tasks in App Designer" on page 15-13.
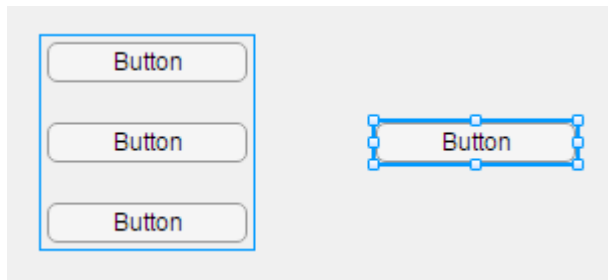
2 In the property panel to the right of the design area, on the **Configuration** tab, change the property values for the selected components.

Using the example from step 1, if you change the font style to bold in the property panel, then the text on all buttons changes to bold.
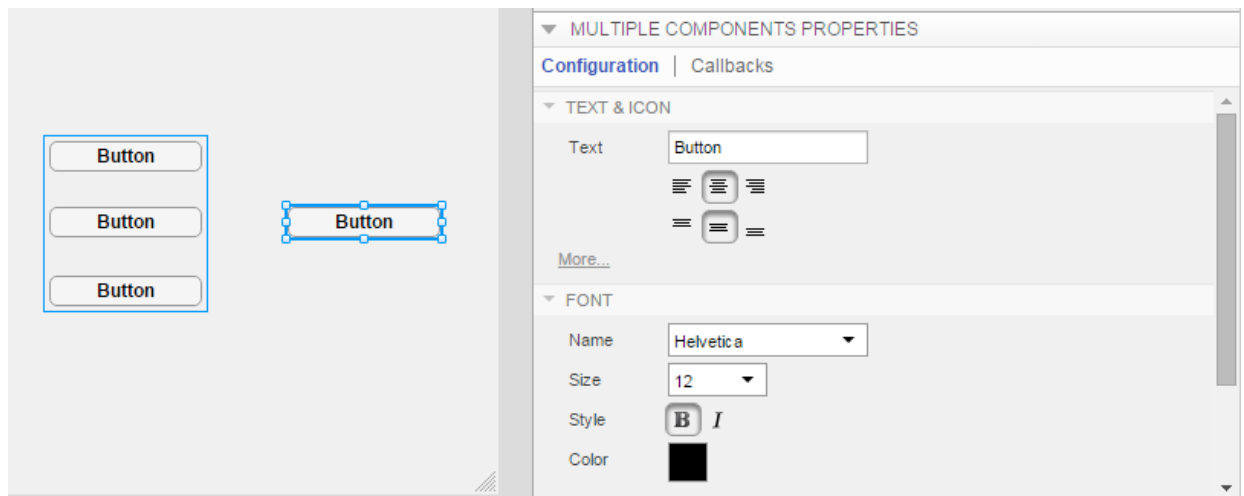
Be aware that:

- If you multiselect different types of components, then the properties presented on the **Configuration** tab are a subset of frequently used properties that are common to all the selected components.

  For instance, if you multiselect a text edit field and a lamp in the design area, then the only option presented on the **Configuration** tab is **Enable**.

- If you multiselect components of the same type, then most properties on the **Configuration** tab for that component are on display. The values presented in the property editor are those of the anchor component. The anchor component has a filled selection border. For example, the button labeled **High** in this image is the anchor component.
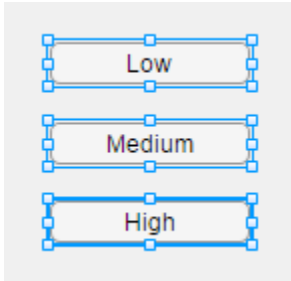
- Component options presented in a table cannot be edited for multiple components simultaneously. These tables appear dimmed on the **Configuration** tab when you multiselect such components. For instance, the **Major Tick** and **Major Tick Label** options cannot be edited for multiple gauge components simultaneously.

Change Text on Components in Design Area

Instead of using the property panel, you can quickly set `Text` or `Title` property values directly on a component in the design area. This approach is referred to as *in-place editing*.

Select the component, and then type the new text.

Specify Multiline Text on Components

To specify line breaks in text for multiline labels, buttons (including radio buttons), and check boxes, press **Shift + Enter**. For example, suppose that you want a label that looks like this image.

This chart shows
the function output.

1 Add a label to the design area.

2 In the design area, click the label text and type, `This chart shows`.

3 Press **Shift + Enter**.

   App Designer inserts a line break.

4 Type, `the function output`.

Set Any Writable Component Property

You can customize any writable component properties in code view.

1 Select a component in the **Component Browser**.

2 In the **Properties** panel, click the **Inspector** tab

3 In the field to the right of a property name, specify or select a new value.

To view what a property affects and how to specify the property value, click the property name or the property field.

The information appears at the bottom of the **Inspector** tab.
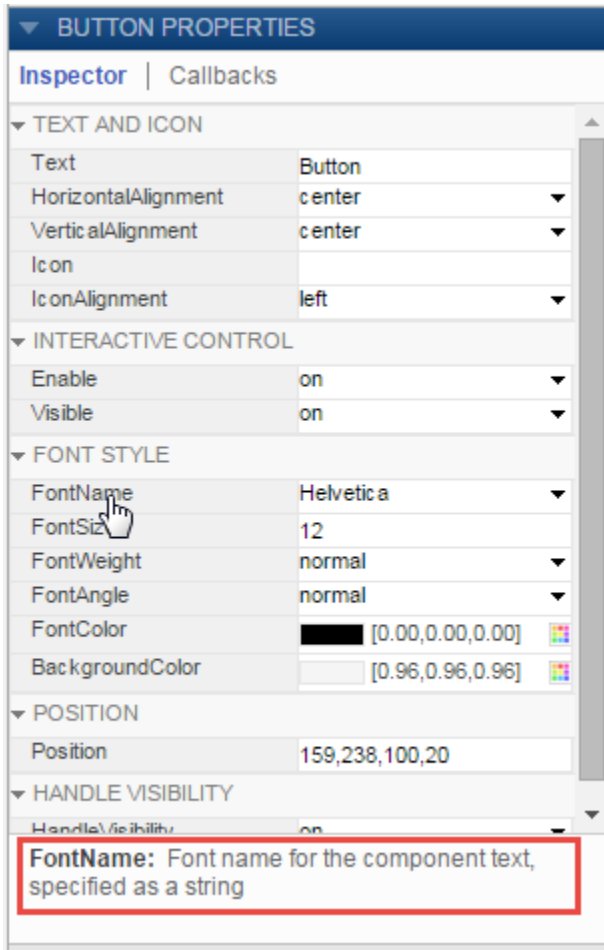
The only properties excluded from the inspector are `Parent`, `Child` and Callbacks. The following table describes how to view and change these properties.

Task	What to Do
See parent/child relationships.	Notice the indentation of component names in the **Component Browser**. Child components are indented below their parent component.

Task	What to Do
Change `Parent` and `Child` property values.	Move components in design view. For details, see "Parent and Reparent Components in App Designer" on page 15-15
Add a callback function.	In the **Component Browser** panel, select the component name. Then, in the component **Properties** panel, click the **Callbacks** tab. When you type a name for the callback, App Designer adds the callback function declaration to your code. For an alternate method for adding callbacks and details on coding callbacks, see "App Designer Callbacks" on page 16-8.
View or change callback function names.	In the **Component Browser** panel, select the component name. Then, in the component **Properties** panel, click the **Callbacks** tab. If you type a new name for an existing callback, App Designer renames the callback function and updates all references to it in your code.
View or change the code for a callback.	In the **Component Browser** panel, right-click the component name, and then click **Callbacks > Go to > callback-function-name callback**.

Set Window and Component Resize Behavior

By default, when the app user resizes the app window at run time, the components within the window resize automatically as the window size increases. Component alignment is maintained. Components resize based on their type.

It is a best practice is to make sure that all components are completely within the design area when you finalize your app design. If components are above or to the right of the central design area at design time, the automatic resize behavior does not initiate when resizing begins. Instead, it occurs only when the window is large enough to contain those components entirely.

For example, suppose that the design area appears as shown. When you save and run the app, automatic resizing is not initiated until the app window grows large enough to enclose the button (on the upper right) completely.
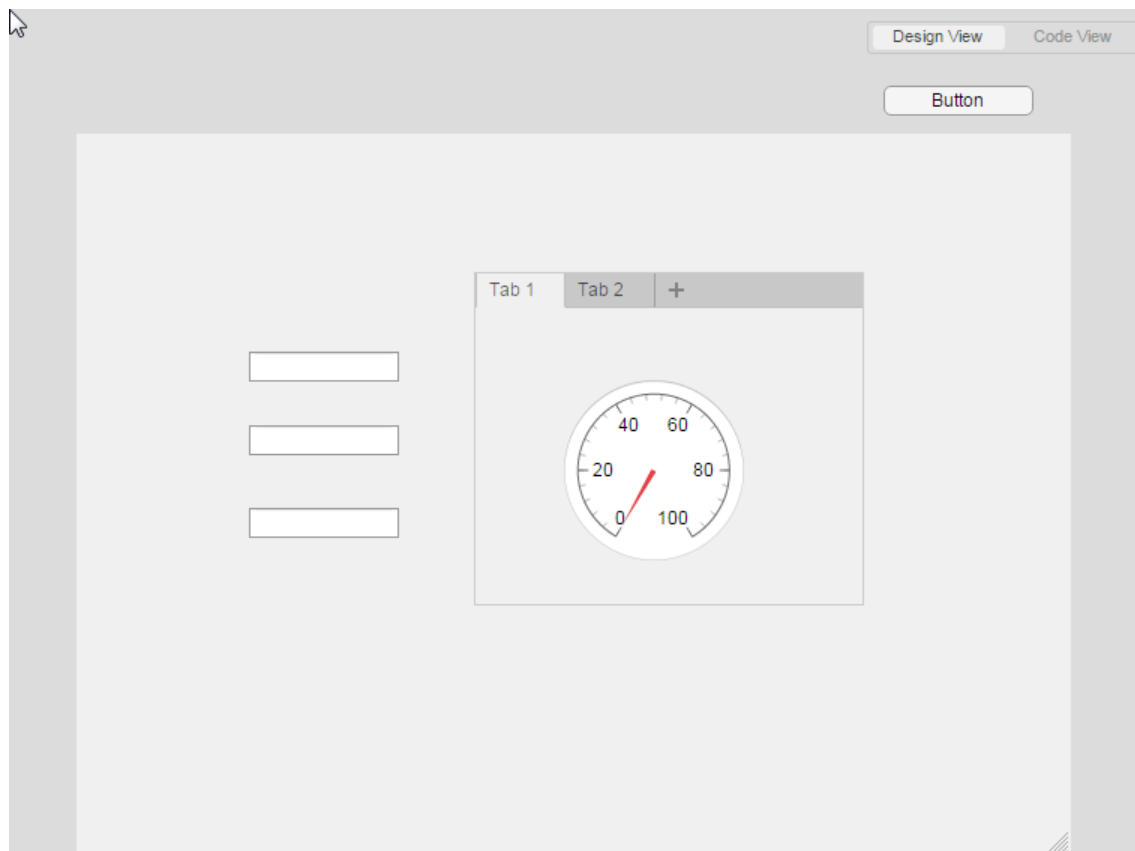
Components that you place above or to the right of the design area might become visible when the app user resizes the app window. If you have a good reason to keep components

there, but do not want the app user to be able to see them, disable the automatic resize behavior.

Disable Automatic Resize Behavior

You can disable automatic resize behavior explicitly or implicitly.

To disable automatic resize behavior explicitly:

1 In the design area, select click the figure name in the **Component Browser**.

   By default the component browser lists the figure as `app.UIFigure`.

2 In the **UIFigures Property** panel, clear **Resize components when app is resized**.

To disable automatic resize behavior implicitly, code the UI figure `SizeChangedFcn` callback. The callback code takes precedence over your selection of the **Automatically resize components when app is resized** option. For an example of manually setting the resize behavior of a component when the window resizes at run time, see "Control Component Resize Behavior When App Window Resizes" on page 16-48.

Tip If you create a UI figure `SizeChangedFcn` callback, and then decide

you want to use the automatic resize behavior, delete the *entire* function. Select the figure in the **Component Browser**, and then, on the **Callbacks** tab, clear the callback name. Otherwise, if you manually delete the function code from the code view editor, the function declaration remains and the autoresize behavior stays disabled.
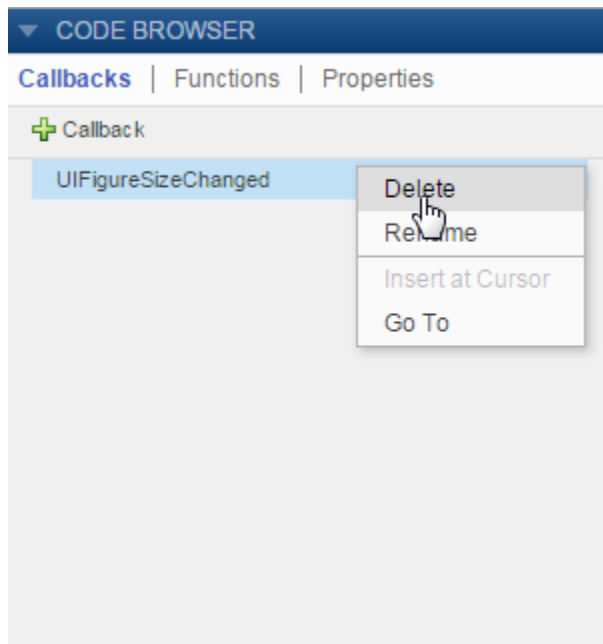
Specify Inclusiveness, Rounding, and Value Formatting for Numeric Components

- "Specify Inclusive and Exclusive Numeric Limit Values" on page 14-20
- "Round Numeric Values" on page 14-21
- "Adjust Value Formatting for Numeric Edit Fields and Spinners" on page 14-21

Specify Inclusive and Exclusive Numeric Limit Values

Spinners and numeric edit fields provide options that allow you to specify whether the minimum and maximum values are inclusive, exclusive, or a combination of both.

For instance, suppose that you add a numeric edit field to the design area. If you want to limit the values for the numeric edit field to greater than 5, but less than or equal to 10, follow these steps:

1 In the central design area, select the numeric edit field.

2   In the property panel, on the **Configuration** tab:

    a   In the **Minimum** field, type 5.

    b   In the **Maximum** field, type 10.

    c   Click **More...**.

    d   Clear **Include the minimum as a valid value**.

    e   Select **Include the maximum as a valid value**.

3   Run and test the app

When the app runs, you cannot enter 5 in the field. However, you can enter any value greater than 5, but less than or equal to 10.

Round Numeric Values

Spinners and numeric edit fields enable you to specify whether the displayed and stored value is rounded to the nearest integer. In the case where a numeral has a fractional part of 0.5, the value rounds to the integer with larger magnitude. For instance, 2.5 rounds to 3 and -2.5 rounds to -3.

1   In the central design area, select the numeric edit field or spinner.

2   In the property panel, on the **Configuration** tab, under the **Value** category, click **More...**.

3   Select **Round all values to the nearest integer**.

4   Run and test the app

Type values in the component and press return.

Adjust Value Formatting for Numeric Edit Fields and Spinners

Spinners and numeric edit fields provide the means to control the *display* of values in the running app. The actual value that MATLAB stores is unaffected.

For instance, suppose that you want App Designer to convert values that app user enters and display them using base 16.
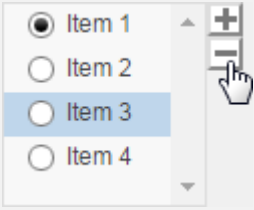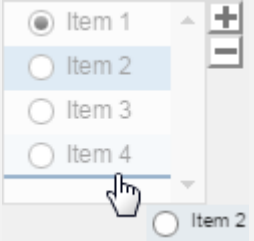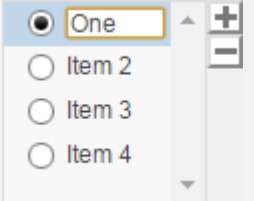
1   In the central design area, select the numeric edit field or spinner.

2   In the property panel, on the **Configuration** tab, scroll to the **Display** category.

3   Select **Custom**.

4   In the **Custom** field type: %x.

5 Run and test the app.

If you enter 20 in the numeric edit field and click away from the field, the value displays as 14. If you click into the field again, the stored value displays (20).

For a complete list of supported format strings, see num2str.

Adjust Drop-Down Component or List Box Rows

Goal	Steps	Visual
Add or remove rows.	Double-click the component in the central design area, select a component row, and then press the plus or minus button.	
Reorder rows.	Double-click the component in the cenral design area, and then drag and drop the row to a new location.	
Change row text.	Double-click the component in the central design area, click the existing text, and then type the new text.	

App Layout

Add and Delete Components Using App Designer

Note: For information on adding and deleting components in apps you create using GUIDE, see "Share Data Among Callbacks" on page 11-2.

When you first open App Designer, the central design area, which represents your app, is empty. To populate it with components, drag components from the **Component Library** on the left into the design area. For many components, App Designer adds a label component with the dragged component. They appear in the display area as grouped components. For instance, if you drag an edit field into the design area, App Designer adds a label component and groups it with the edit field by default.

To suppress default label inclusion, press and hold the **Ctrl** key as you drag the component into the design area.

For information on grouping components, see "Group Components for Layout Tasks in App Designer" on page 15-13.

Add Components

After adding components to the design area from the component library, add more components using any of these methods:

- Use a context menu option.

  Right-click a single component or a set of multiselected components in the design area, and then from the context menu, select **Duplicate**, **Copy**, or **Paste**.

- Drag-duplicate a component.

  While holding down the **Ctrl** key, left-click the component in the design area, and then drag the duplicate to a new location within the current parent container. (On Windows, you can also right-click and drag the duplicate component.)

To cancel an in-progress duplicate operation, press the **Esc** key.

- Use key combinations, as described in "App Designer Keyboard Shortcuts" on page 17-2.

You can copy or cut components from one app and paste them into another.

Delete Components

Delete components using any of these methods:

- Use a context menu option.

  Right-click a single component or a set of multiselected components, and then from the context menu, click **Delete**.

- Use key combinations, as described in "App Designer Keyboard Shortcuts" on page 17-2.

Note: If you delete a component grouped with a label, App Designer deletes the label too.

Multiselect Components

Multiselect components using any of these methods:

- In the design area, hold down the **Ctrl** or **Shift** key and click each component to include in the selection.
- Left-click a blank spot in the design area and drag a selection border around the components you want to select.
- Use key combinations, as described in "App Designer Keyboard Shortcuts" on page 17-2.

Note: You cannot multiselect tabs and you cannot multiselect components with different parents.

Component Copies and Property Values

When you duplicate or copy and paste components, the new components have the same property values as the copied or duplicated components, with these exceptions:

- Callback function
- `Position` property value
- Sometimes, the `Parent` property value.

  For information on how paste operations affect the `Parent` property value, see "Change Component Parent at Design Time" on page 15-16.

Effect of Component Deletion on Callback Function Code

When you delete a component, App Designer updates the code and the design area to remove the deleted component. If you have written callback code for the deleted component, that code is *not* deleted. For information on deleting the callback code, see "Delete Code from the App Designer Editor" on page 16-29.

Move Components in App Designer

Within the design area, you can move individual, multiselected, or grouped components manually (freehand) or by set increments, as follows:

- Freehand — Drag the selected components or groups.

  To assist you in aligning components you move freehand, use the snap-to-grid feature. On the **Canvas** tab, in the **View** section, select **Show grid** and **Snap to grid**.
- 1-pixel increments — Select the components or groups. Press a keyboard **Arrow** key in the direction that you want to move the component.
- 10-pixel increments — Select the components or groups. Press **Shift+Arrow** in the direction that you want to move the component.

To multiselect the components, hold down the **Ctrl** key or the **Shift** key and click each component in the design area. Alternatively, left-click a blank spot in the design area and drag a selection border around the components you want to select.

To cancel an in-process freehand move operation, press the **Esc** key.

For more information on aligning components, see "Align, Space, and Resize Components in App Designer" on page 15-6. For more information on keyboard shortcuts, see "App Designer Keyboard Shortcuts" on page 17-2.

Align, Space, and Resize Components in App Designer

You can direct App Designer to align and resize components automatically, or you can perform these operations manually. For instance, you can use the manual method for an initial layout, and then polish the layout using automatic alignment. In addition, you can add a grid to the display area to make layout easier.

In this section...
"Component Alignment" on page 15-6
"Space Components" on page 15-10
"Resize Design Area or Components" on page 15-11

Component Alignment

App Designer offers three techniques for positioning components:

- "Manual Alignment Using Alignment Hints" on page 15-6

  This method is useful when you want to arrange components in a visually appealing way, but are not concerned about precise measurements.

- "Manual Alignment Using a Grid" on page 15-7

  This method is useful when you want to position components at precise locations — for instance 30 pixels from the edge of the figure window.

- "Automatic Component Alignment" on page 15-9

  This method is useful for refining positioning after positioning components using alignment hints.

Manual Alignment Using Alignment Hints

By default, as you drag components in the design area, you see alignment hints that assist you in arranging the components.
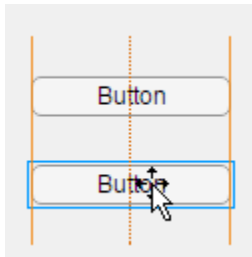
- Dashed orange lines indicate that components are center-aligned.
- Solid orange lines indicate that components are aligned along their edges (top, bottom, or sides).
- Perpendicular dashed red lines indicate that the component is centered in a container component or the UI figure window.

Alignment hints are not displayed if you select a component and then press arrow keys to move the component.

To disable alignment hints, on the **Canvas** tab, in the **View** section, clear **Show Alignment Hints**.

Manual Alignment Using a Grid

When positioning and sizing components, add a grid to the design area. By default, a snap-to-grid feature is enabled when you display the grid. The snap-to-grid feature ensures that when you add or move a component, the upper-left corner of the component is positioned at the intersection of two grid lines. Components do not snap to grid when you use arrow keys to move the component. This behavior allows you to fine-time component positions.
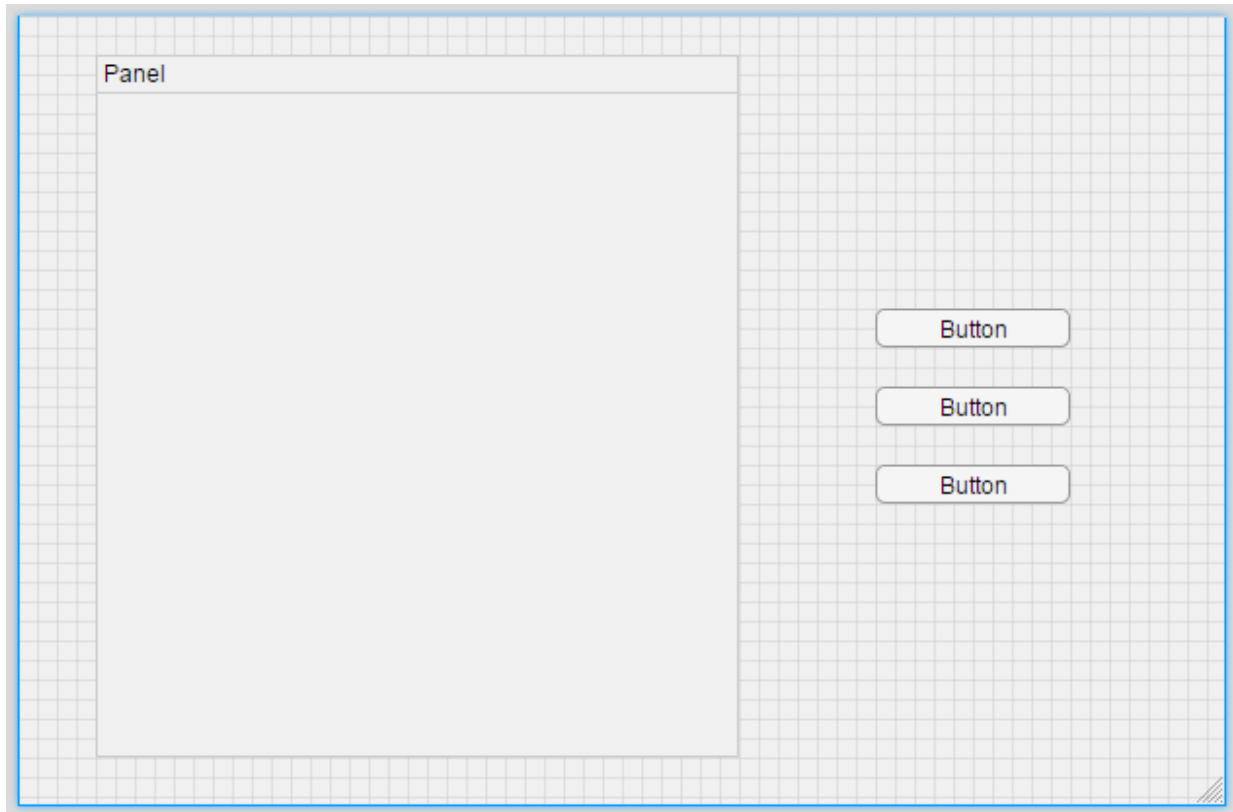
Goal	Instructions
Display or hide the grid in the design area	On the **Canvas** tab, in the **View** group, select or clear **Show grid**.
Change the grid line interval (in pixel units)	On the **Canvas** tab, in the **View** group, change the **Interval** value. The interval unit of measurement is pixels.
Enable or disable the snap to grid feature	On the **Canvas** tab, in the **View** group, select or clear **Snap to grid**.

Note: You cannot enable the snap to grid feature unless **Show grid** is selected.

Automatic Component Alignment

To use the automatic alignment feature:

1 Select two or more components.

2 Right-click the component with which you want the others to align, and then select an **Align** option.

In this image, button **C** is the component with which buttons **A** and **B** will align. Notice that App Designer indicates this state by enclosing button **C** in a heavier weight border than the other components.
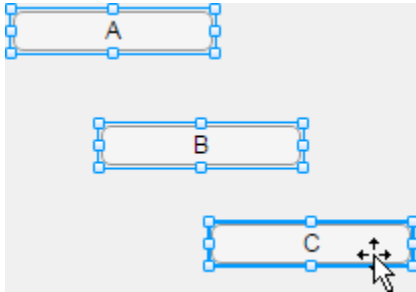
If you left align the components, they align along the left side of button **C**.
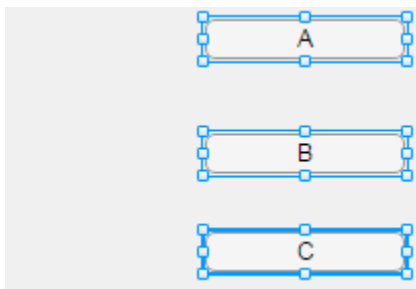
The component with which other components align is the **anchor** component. The technique you use to select components determines which component is the anchor:

- If you select components using **Ctrl + Click**, then the last component you select is the anchor component.

- If you drag-select components, then the last component you include in the drag rectangle is the anchor component.
- If you press **Ctrl + A** to select all components, then the last component you add to the design area is the anchor component. If the last component added is a direct child of the UI figure and is grouped, then the group is the anchor.

A group of components behaves as though it is a single component with respect to alignment and spacing operations.

Space Components

You can direct App Designer to space components automatically, or you can specify spacing values. Both approaches are presented in these steps:

1   In the design area, select the components that you want to space.

Select two or more components to space manually. Select three or more components to space automatically.

2   On the **Canvas** tab, in the **Space** section, do one of the following:

- To specify autospacing, select **Autospacing**.
- To specify a value by which to space components, select the **pixels** radio button, and optionally type a new value in the **pixels** field.
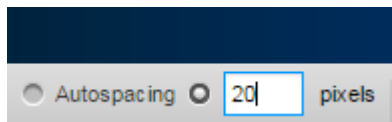
3   Select the direction in which you want to apply spacing by clicking a spacing application option, **Apply horizontally** or **Apply vertically**.

- If you selected **Autospacing**, App Designer evenly distributes the spacing equally among the selected components in the selected direction.
- If you specified the number of pixels, App Designer applies the spacing among the selected components as you specified in the selected direction.

Sometimes, the number of pixels between the two components prevents App Designer from making the space between each component identical. However, the spacing difference is never more than one pixel.

Resize Design Area or Components

Resize the design area or a component by clicking the design area or component edge and dragging the mouse in the direction you want to change. To change both height and width of a component simultaneously, click and drag a corner.

To see the change in size, on the **Canvas** tab, in the **View** section, choose **Show resizing hints**.
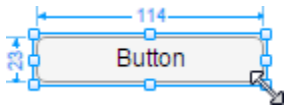
Although you can resize the components within a group, you cannot resize a group directly. The group automatically adjusts as you change the size or position of the components within it.

This table summarizes features that can make resizing components easier and faster.

Goal	Instructions
Maintain the aspect ratio as you resize.	Press and hold the **Shift** key as you drag the mouse.
Resize multiple components simultaneously.	Multiselect the components that you want to resize, and then drag the resize handle of one of the selected components. To maintain the aspect ratio, press and hold the **Shift** key as you drag the resize handle.
Make multiple components the same size.	Multiselect the components you want to make the same size, right-click, and select **Same Size** and then one of these context menu options: **Width & Height**, **Width**, or **Height** If you select **Width & Height**, but matching both dimensions would distort a component, then both dimensions are not honored. App Designer always maintains

Goal	Instructions
	the aspect ratio for components with a built-in aspect ratio. (Such components have 4 resize handles.) In addition, App Designer always maintains the height of a slider.
Cancel an in-process resize operation.	Press the **Esc** key.

For more keyboard shortcuts, see "Component Resize Shortcuts" on page 17-5.

For information on controlling run-time component resizing, see "Set Window and Component Resize Behavior" on page 14-17.

Group Components for Layout Tasks in App Designer

Create design groups to perform design operations on multiple components as a unit. For instance, group three buttons, so you can move them as a group. Unless otherwise noted, layout operations that you can perform on a single component, you can also perform on a design group.
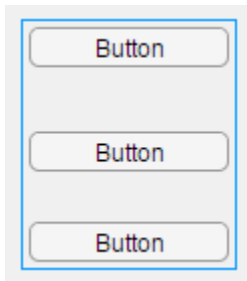
By default, when you drag a component, such as an edit field, from the **Component Library** into the design area, the component and its associated label component are added to the design area as a design group. Select a design group by clicking one of the components in the group or the empty area between components in the group.
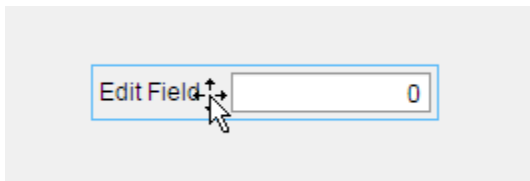
Design groups are design-time constructs only. They are not represented in the generated code nor seen in a running app.

Note: Do not confuse design groups with components that have the word group in their name, such as button groups, radio button groups, and tab groups.

Create and Manage Design Groups

Create a Design Group

Select two or more components in the design area, right-click and select **Grouping** > **Group**. If you group two groups, you create a nested group.

Add Component to a Design Group

Select the component or components and the design group to which you want to add them, right-click and select **Grouping** > **Group**.

Remove Component from a Design Group

Select the component within the design group by clicking the component twice. Then, right-click and select **Grouping** > **Remove from Group**. A group must contain three or more components to access this menu option.

Note: If you want to remove a component from a design group that contains only two components, select the design group and ungroup it.

Resize a Design Group

Resize a design group by adding or removing components, or by moving components. You cannot resize a design group directly.

Ungroup — Remove All Components from a Design Group

Select the design group, right-click, and select **Grouping** > **Ungroup**. The ungrouped components are *not* deleted.

Delete a Design Group and Contained Components

To delete a design group and all the components within it, select the design group, right-click and then click **Delete**.

App Designer deletes the group and the components within it. Any callbacks you wrote for the components remain in the code. For information on deleting callbacks, see "Delete Code from the App Designer Editor" on page 16-29.

Parent and Reparent Components in App Designer

When you drag a component from the component library and drop it in the design area, the drop location determines the initial parent/child relationship among components. For example, if you drag a panel onto the central design area, the parent of the panel is the UI figure window. If you drag a button onto that panel, the panel is the parent of the button (and the button is the child of the panel).

When you drag a component onto a container component, App Designer indicates which component will be the parent by highlighting the parent in blue. The dragged component must be completely within the target parent for the blue highlighting to display.
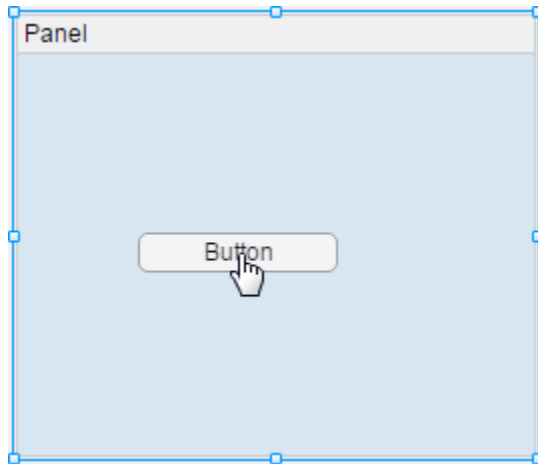
You can nest instances of the same container component, except for UI figures. For example, one panel can be the parent of another panel.

You can view the component parent-child relationships in the **Component Browser**.
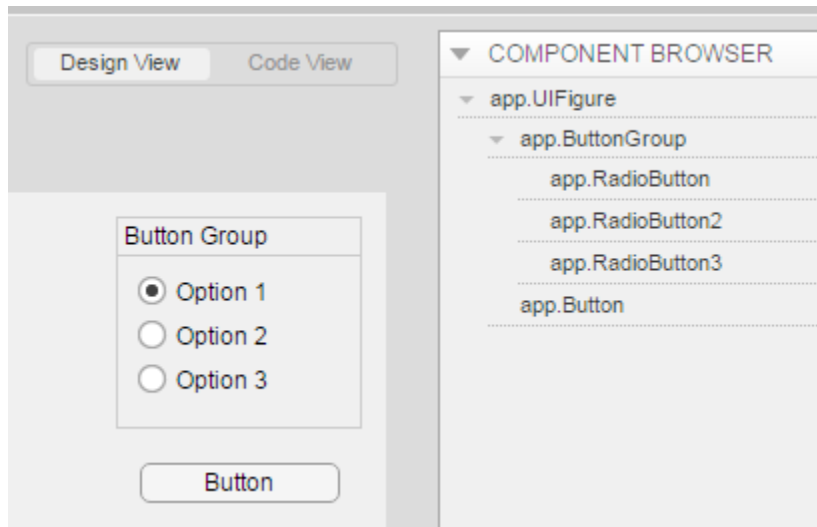
The indentation of component names reflects the parent-child relationships. In the preceding image, UIFigure is the parent of `ButtonGroup` and `Button`. `ButtonGroup` is the parent of `RadioButton`, `RadioButton2`, and `RadioButton3`.

Change Component Parent at Design Time

You can reparent a component or a group of components. Drag a component or a group of components from the current location in the design area onto a new container component.

When the dragged component or group of components is completely within the target container component, the target highlights blue. If it is not a valid parent, a prohibited icon 🚫 displays. If you release the mouse button over an invalid parent, a moved component or group snaps back to its original location.

The only valid container for radio buttons and toggle buttons are radio button groups and toggle button groups, respectively.

Components are not reparented to valid parent components under these circumstances:

- When you move a component using arrow keys
- When you attempt to drag-duplicate components into a new parent
- When you drag a grouped component into a different parent container than that to which the other components in the group are parented.

To change the parent of one component within a group of components, ungroup the components, and then drag the component to the new parent.

For information about creating component groups and ungrouping components, see "Group Components for Layout Tasks in App Designer" on page 15-13. Do not confuse grouped components with these container components that App Designer provides: radio button groups, toggle button groups, and tab groups.

16

App Programming

App Designer Code Generation

As you create an app, App Designer generates code to reflect your component layout and customizations. In addition, when requested, it generates the framework for the callback functions, which control app behavior. When you save your app, App Designer saves the code in an MLAPP file. When MATLAB runs your app, it executes this file.

App Designer automatically generates and manages the code for creating the app, its components, and the framework for callback functions. To prevent you from accidentally overwriting the code that App Designer manages, that code is not editable (indicated with a light gray background).

The code that App Designer generates is modularized. For example, when you add a button to the design area, the generated code is as shown in this image.

```matlab
1   classdef App1 < matlab.apps.AppBase
2
3       % Properties that correspond to app components
4       properties (Access = public)
5           UIFigure  % UI Figure
6           Button    % Button
7       end
8
9       methods (Access = private)
10
11          % Code that executes after component creation
12          function startup(app)
13
14          end
15      end
16
17      % App initialization and construction
18      methods (Access = private)
19
20          % Create UIFigure and components and specify parenting
21          function createAppComponents(app)
22
23              % Create UIFigure
24              app.UIFigure = uifigure;
25              app.UIFigure.Position = [100 100 640 480];
26              app.UIFigure.Name = 'UI Figure';
27              setAutoResize(app, app.UIFigure, true)
28
29              % Create Button
30              app.Button = uibutton(app.UIFigure, 'push');
31              app.Button.Position = [90 304 100 22];
32          end
33      end
34
35      methods (Access = public)
36
37          % Construct app
38          function app = App1()
39
40              % Create and configure components
41              createAppComponents(app)
42
43              % Process app after components are created
44              postCreateApp(app, @startup)
45
46              if nargout == 0
47                  clear app
48              end
49          end
50
51          % Code that executes before app deletion
52          function delete(app)
53
54              % Delete UIFigure when app is deleted
55              delete(app.UIFigure)
56          end
57      end
58  end
```

Marker 1 — lines 1–7
Marker 2 — lines 9–15
Marker 3 — lines 17–58

Notice that:

- The first section of code defines the app properties.

  Initially, this section contains the properties that correspond to the app and the app components. If you add properties to share data, App Designer adds your properties to this region. For information on adding properties, see "Share Data Across Callbacks in App Designer" on page 16-19.

- The second section of code contains the app functions.

  Initially, this section contains the framework for the startup function. If you add callbacks or utility functions to the app, App Designer adds them to this region. For information on adding functions, see "Code and Call App Functions in App Designer" on page 16-24 and"App Designer Callbacks" on page 16-8.

- The third section of code initializes and creates the app and its components.

  If you use App Designer property sheets or the Inspector to change component property values, those changes appear in this section of the code. For information on adding and customizing components, see "Add and Delete Components Using App Designer" on page 15-2 and "Customize App Designer Components" on page 14-11.

If you rename code elements, as described in "Rename Components, Properties, Callbacks, and Functions in App Designer" on page 16-5, App Designer updates the name in the read-only and read-write areas of the code.

Rename Components, Properties, Callbacks, and Functions in App Designer

App Designer references components, custom properties, callbacks, and functions in the code using default names. For instance, if there is more than one instance of a given component, App Designer appends the component name with a numeric value. If you include three buttons in your app, App Designer names them `Button`, `Button2`, and `Button3`.

The best practice is to replace each default name with a name that helps you to identify the code element when you read the code. For instance, if a button causes the app to display results, consider renaming it `ResultsButton`.

Rename Component Instances

To rename a component instance so that all references throughout your app code update, use the **Component Browser** as described in the steps that follow.

Note: By default, labels are not listed in the **Component Browser**. To list labels, right-click the name of any other component in the **Component Browser** and select **Include label components**.

---

1   In the **Component Browser**, double-click the name of the component instance to rename.
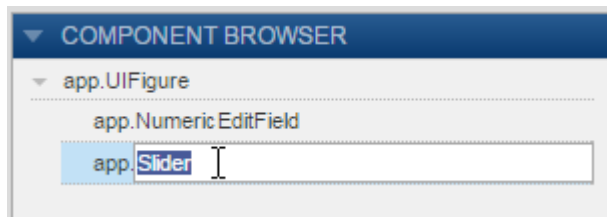
2   In the edit field that opens, type a new name.

App Designer updates the code to reflect the new name when you press **Enter** or click away from the edit field.

You cannot change the `app.` prefix. All app component names must have the `app.` prefix in code.

Tip If you are unsure which name corresponds to a particular component, use the **App Layout** panel. By default, **App Layout** appears in the lower left corner of code view.
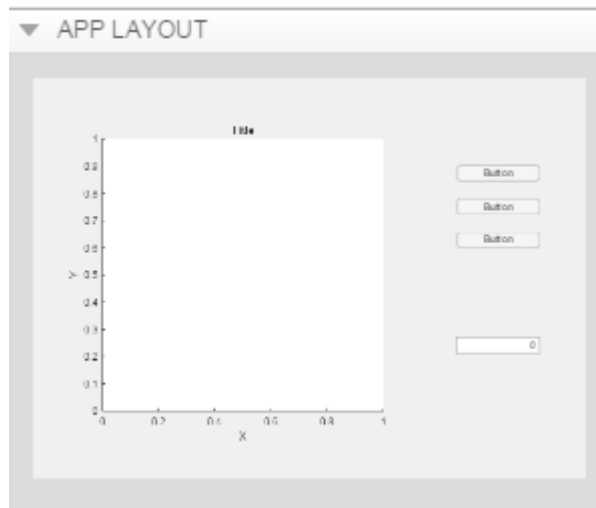
Click the component that you want to identify. App Designer highlights the name of that component in the code view **Component Browser**.

Rename Callback, Utility Function, or Property

To rename a callback, a utility function, or a property you created:

1   Click **Code View**.
2   In the **Code Browser** (to the left of the editor, by default), click the tab that corresponds to the element you want to rename.

    The tabs are labeled **Callbacks**, **Functions**, and **Properties**.
3   Double-click the callback, function, or property that you want to rename.
4   In the edit field that opens, type a new name.

App Designer updates the code to reflect the new name and all references to it when you press **Enter** or click away from the edit field.

App Designer Callbacks

> **Note:** For information on callbacks for apps you create with GUIDE or programmatically using the `figure` function, see "Write Callbacks in GUIDE" on page 7-2 or "Write Callbacks for Apps Created Programmatically" on page 10-5.

A callback is a function that executes in response to some predefined app user interaction with a component, such as pressing a button.

You can create a component callback from design view or code view. In the **Component Browser**, right-click the name of the component for which you want to create a callback. Then, select **Callbacks > Add** callback type.
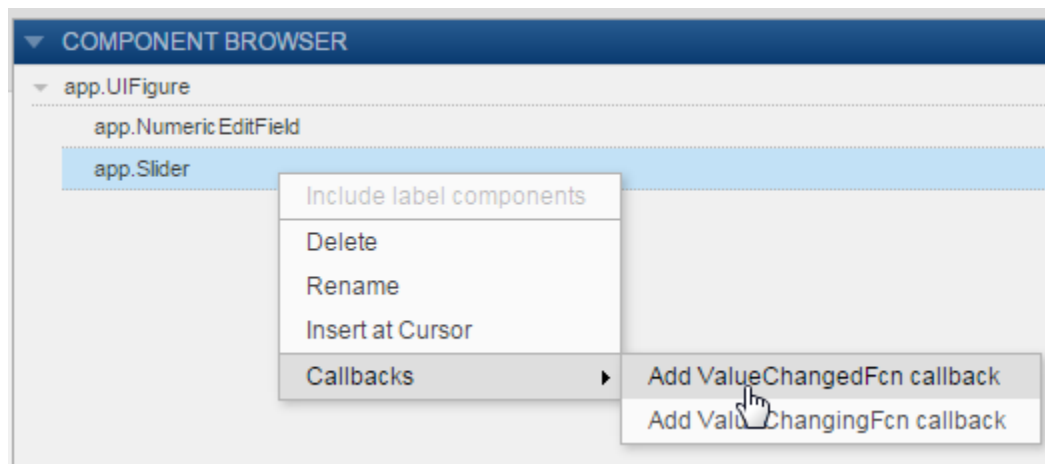
> **Tip** If a component does not support callbacks, the **Callbacks** option does not appear on the context menu for that component.

Then, follow these steps:

1 In the Add Callback Function dialog box, enter a name for your callback function.

A best practice is to specify a name for the callback, rather than accept the default name. This practice enables you to identify the control with which the callback is associated and what the callback does. For instance, if you intend for a button push to result in data being graphed, consider naming the button callback `GraphButtonPushed`.

2 Click **OK**.

App Designer opens code view with the cursor placed in the newly created callback function, below a line of sample code.

Note: Sample code is not included when you add a callback for a push-button component.

---

3 Modify or replace the sample code with the code that you want MATLAB to run when the app user interacts with the control.

Specify components and their properties using `app.` as a prefix for each component. This practice makes it possible to access properties directly using dot notation. For example, to set the `Enable` property of a button named `GraphButton` to `'off'`, use the following code:

```
app.GraphButton.Enable = 'off';
```

"App Designer Coding Tips" on page 16-16 describes ways that App Designer facilitates adding property names to your code.

4 Run your app.

Test to ensure that manipulating the component has the effect that you want. If your app returns errors, debug your code as described in "Detect and Correct Coding Errors Using App Designer" on page 16-13.

For examples, see "Simple Example Apps for App Designer" on page 16-31.

Note: Multiple components cannot share a callback. If you want to share code among callbacks, consider writing a utility function and calling it from multiple callbacks, as described in "Code and Call App Functions in App Designer" on page 16-24.

App Designer Startup Function

Use the app startup function to have MATLAB perform tasks when the app first opens. MATLAB runs the startup function after it creates the components.

For instance, use the startup function to plot data in an axes component when the app opens, or to initialize app properties.

To access the startup function in your app, from code view, on the **Editor** tab, click **Startup Function**.

Display Line Plot When an App Opens

This example shows how to create an app that opens with a plot in an axes component.
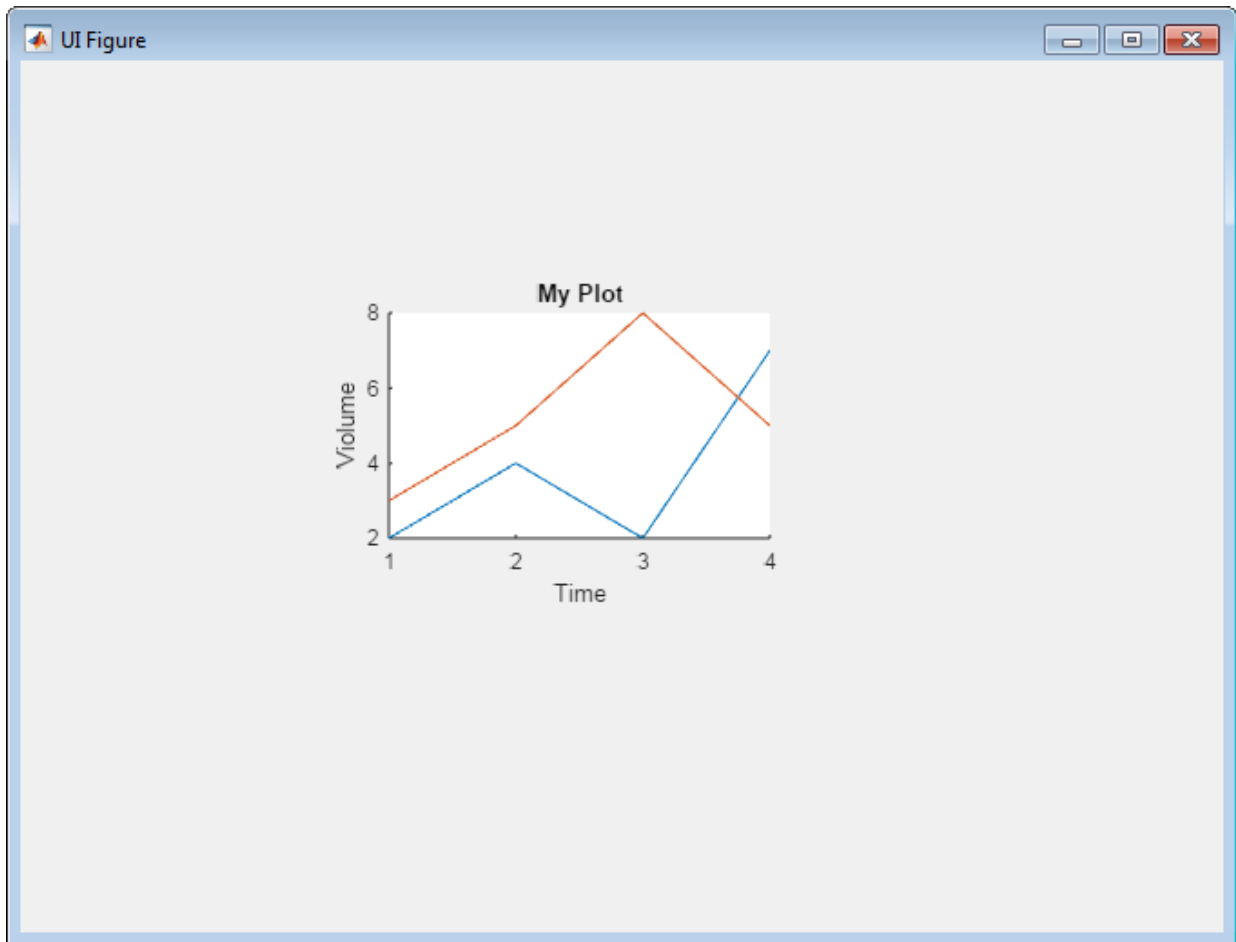
In App Designer:

1   In design view, drag an **Axes (Line and Scatter)** from the **Component Library** into the central design area.

2   In the property editor, set these values:

   · **Title**: My Plot
   · **X Label**: Time
   · **Y Label**: Volume

3 Click **Code View**.

4 On the **Editor** tab, click **Startup Function**.

App Designer places your cursor in the startup function.

5 To specify the data and plot it in the line plot, add this code.

```
d = [2 3; 4 5; 2 8; 7 5];
plot(app.UIAxes, d);
```

6 Save and run the app.

When the app opens, the app displays the line plot with the specified data plotted.

Detect and Correct Coding Errors Using App Designer

| In this section... |
| --- |
| "Error and Warning Detection During App Programming" on page 16-13 |
| "Error Detection at Run Time" on page 16-14 |
| "Debug App Code" on page 16-15 |
| "App Designer Coding Tips" on page 16-16 |

Error and Warning Detection During App Programming

App Designer provides two forms of error and warning detection during app programming. The first, app coding alerts, are specific to App Designer. Therefore it is best to address these errors and warnings first. The second, Code Analyzer messages, are applicable to all MATLAB code. Sometimes both forms of messages appear for the same line of code. Frequently, addressing an app coding alert resolves problems flagged for both App Designer and MATLAB.

App Coding Alerts

By default, as you write code, App Designer flags coding problems with coding alerts. For instance, if you refer to a property without specifying `app` in the dot notation, a warning alert icon displays. To view the alert message, click the icon. Click the icon again to close the message.
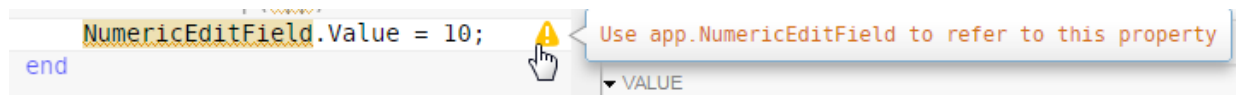
Notice that the MATLAB Code Analyzer indicators (orange wavy line and highlighting on `NumericEditField`) also appear. When you change `NumericEditField` to `app.NumericEditField`, both the coding alert and the Code Analyzer indicators disappear.

If you want to suppress coding alerts, on the **Editor** tab, in the **View** section, clear **Enable app coding alerts**.

Code Analyzer Messages

As with all MATLAB code, the Code Analyzer flags potential warning and error conditions. An orange wavy underline indicates a warning and a red underline indicates an error. Highlighted text indicates a problem that the Code Analyzer can fix for you.

To view the error or warning message, hover the cursor over a wavy line or highlight. The Code Analyzer message opens and any app coding alert that had been open, closes.
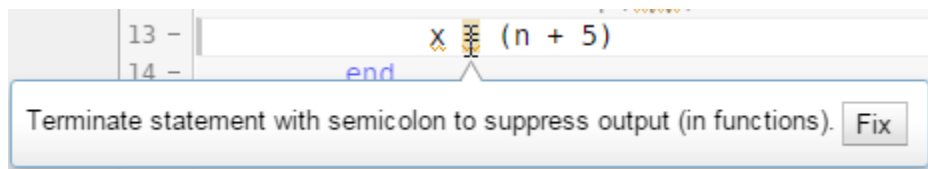
Adjusting your code in response to these messages can keep your code free of warnings and errors. If you allow errors and warnings to accumulate, it can be more difficult to debug your code later.

The App Designer editor does not provide all the Code Analyzer features that the MATLAB Editor does. For instance:

- Messages indicators do not appear in a column to the right of the code.
- Message details are not available.
- Adjusting Code Analyzer preferences has no effect on the App Designer editor.

Error Detection at Run Time

If your code contains an error detected at run time, App Designer flags the error in your code with an error alert icon .

The message for the most-recently thrown error opens automatically. To close it, click the icon. To open a different error, click that icon.
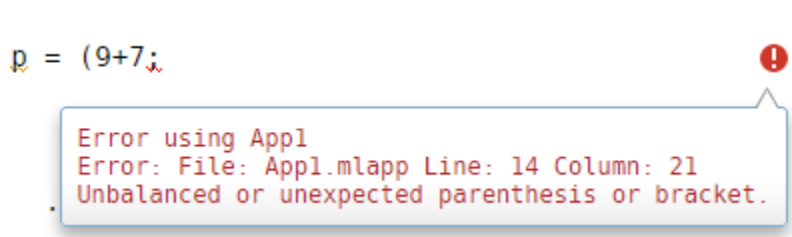
To clear the error alert, correct the error, and then rerun you app. Make sure that no new errors are revealed. Many errors are revealed only when a code branch runs, such as a particular case in a switch statement. So test your app thoroughly.

Debug App Code

Many of the debugging features available in the MATLAB Editor are also provided in the App Designer editor.
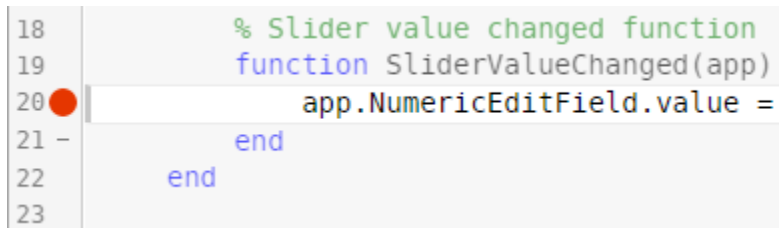
Note: Before you begin debugging, make sure that you save your MLAPP file.

In particular, you can:

• Set standard breakpoints.

 Save the file, and then click in the space between a line number and an *executable* line of code. Executable lines are preceded by a dash (–).

• Step through a file, pausing at points where you want to examine values.

 After you set one or more breakpoints and click **Run**, MATLAB starts running the code. When MATLAB hits a breakpoint, the **Editor** tab displays the **Continue**, **Step**, **Step In**, **Step Out**, and **Quit Debugging** buttons.

 If you step into a function that is in a file outside of your app, MATLAB opens the function file in the MATLAB Editor. It places the cursor at the first executable line of code.

• View variable values in the MATLAB Workspace browser.

• Clear a breakpoint by clicking it.

For details on these debugging features, see "Debug a MATLAB Program".

16-15

App Designer Coding Tips

- "List Label Components in the Component Browser" on page 16-16
- "Avoid Typographical Errors" on page 16-16
- "Reorganize App Designer Panels" on page 16-17
- "Quickly Locate Components in the Component Browser" on page 16-18

List Label Components in the Component Browser

By default, label components in the design area are not visible in the **Component Browser**. To make them visible, right-click any component name in the **Component Browser** and select **Include label components**. This option is disabled unless at least one label component is in the design area.

Note: If you delete a component while labels are hidden and a single label is grouped with the component, then the label is also deleted.
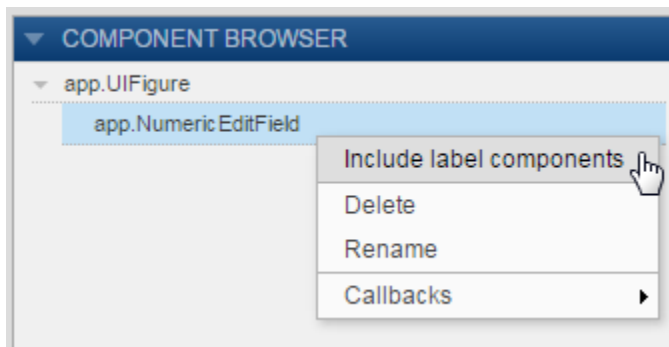
Avoid Typographical Errors

Instead of typing a component code name in your code, have App Designer do it for you.

1 Place the cursor in a writable area of the App Designer editor where you want the name to appear.

2 Drag the name from the **Component Browser** and drop it into the editor.

   The name drops at the cursor location.

Reorganize App Designer Panels

Click a panel title (such as **Component Browser**) and drag it. You can place panels on the right and left sides of App Designer only.
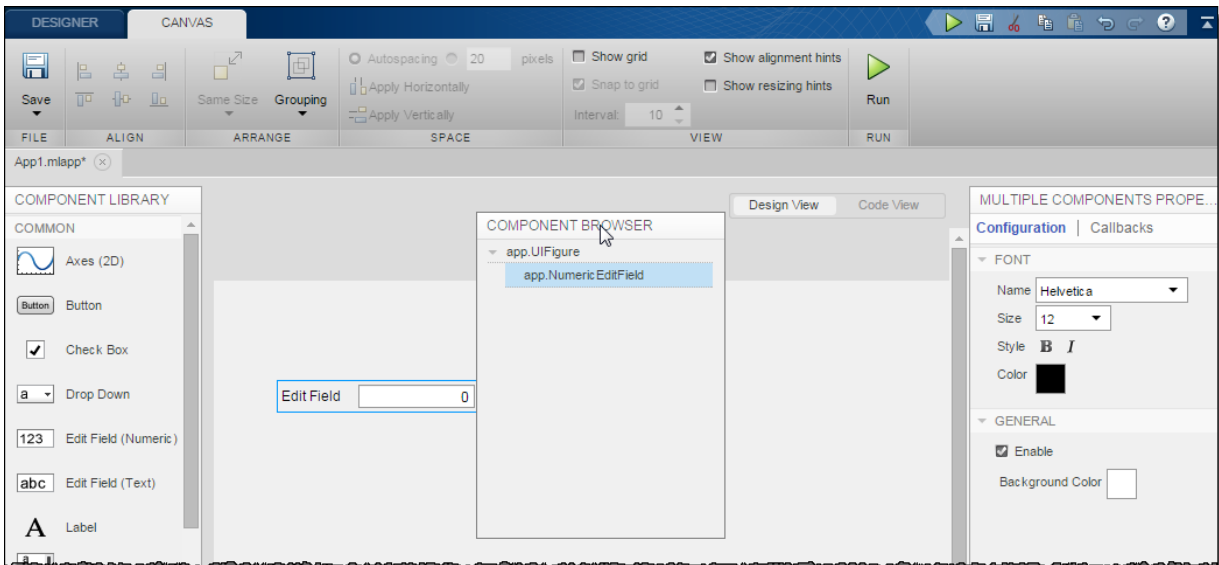
Quickly Locate Components in the Component Browser

In code view, click a component in the **App Layout** panel. The component name highlights in the **Component Browser**. By default, the **App Layout** panel is in the lower left corner of code view.

As shown in the image below, clicking the yellow lamp in the **App Layout** panel results in the Component Browser highlighting `app.LampSlow`.
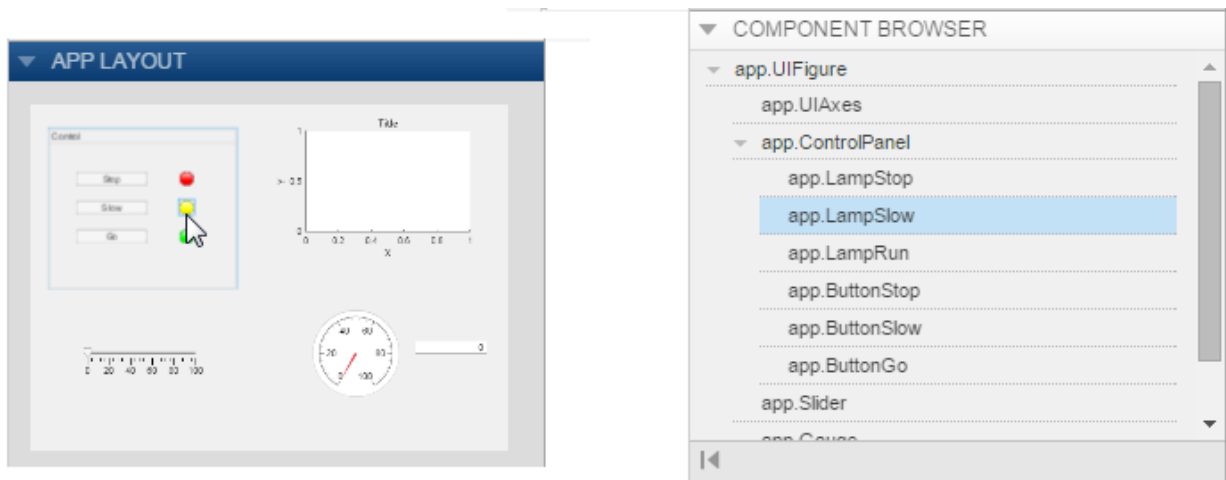
Share Data Across Callbacks in App Designer

Note: For information on sharing data across callbacks in apps you create using GUIDE, see "Share Data Among Callbacks" on page 11-2.

Create properties to store data that you intend to share across callbacks or the MATLAB workspace when you run your app. For instance, you can create a property to hold the result of a calculation and then use that property in one or more functions. App Designer adds an `app.` prefix to properties that you create. This practice makes it possible for you to access properties within the app using dot notation, similar to the way you access fields of a structure array.

About App Properties

There are two types of app properties:

- Properties that you create to share data among functions within your app

  This property is a *private* property that you define using App Designer. Private property values are not available in the MATLAB workspace. You can create a private property to hold the result of a calculation that an app function performs. Then, you can pass the calculation to another function within the app.

  For an example, see Share Data Among Functions in a Single App.

- Properties that you create to share data within and outside your app

  This property is a *public* property that you define using App Designer. For instance, suppose that you create a property to hold the result of a calculation that an app function performs. You want to make it possible for the app user to perform additional analysis on the result, outside the current app. Public properties are available in the MATLAB workspace while the app is running.

  For an example, see Share Data Among Multiple UI Figure Windows.

App Designer creates component properties. These are properties that control the display and behavior of a component. You can modify the property values, but you cannot create these properties. For instance, App Designer creates the `Text` property when you add a text edit field to the design area. You cannot change the property name, but you can change this property value.

Share Data Among Functions in a Single App

This example shows how to create an app that maintains a count of how many times a button is pressed during an app session. It shares values among functions within the app code using a private property.
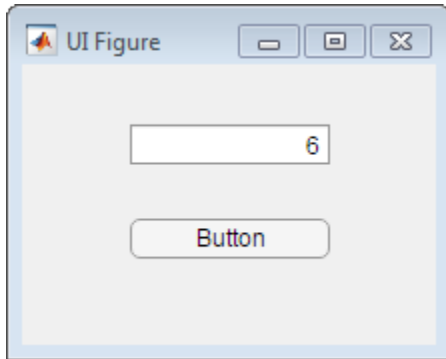
1 Hold down the **Ctrl** key and drag a **Numeric Edit Field** from the **Component Library** into the central design area.

   Holding down the **Ctrl** key prevents a label component from being added with the edit field.

2 Drag a **Button** into the design area.

3 Click **Code View**.

4 On the Editor tab click **Property** > **Insert Private Property** and replace the highlighted word, Property with Counter.

Note: If you code contains errors, the **Property** button is disabled. For information on debugging, see "Detect and Correct Coding Errors Using App Designer" on page 16-13.

---

5 Initialize the Counter property.

   a On the **Editor** tab, click **Startup Function**.

   b In the code, on the line following function startup(app) type:

      ```
      app.Counter = 0;
      ```

6 In the **Component Browser**, right-click **app.Button** and select **Callbacks** > **Add ButtonPushedFcn callback**.

7 In the Add Callback Function dialog box, click **OK**. (Accept the default callback function name.)

8 Add code to increment the counter and display the Counter property value in the numeric edit field each time the app user clicks the push button.

In the ButtonButtonPushed function, type:

```
app.Counter = app.Counter + 1;
app.NumericEditField.Value = app.Counter;
```

9 Save the app as buttonCounter.mlapp.

10 Run, test, and then close the app.

Share Data Among Multiple UI Figure Windows

This example shows how to create an app that gets app user input in one app, performs a calculation, and plots the results in another app. It shares a value across the apps by using a public property.
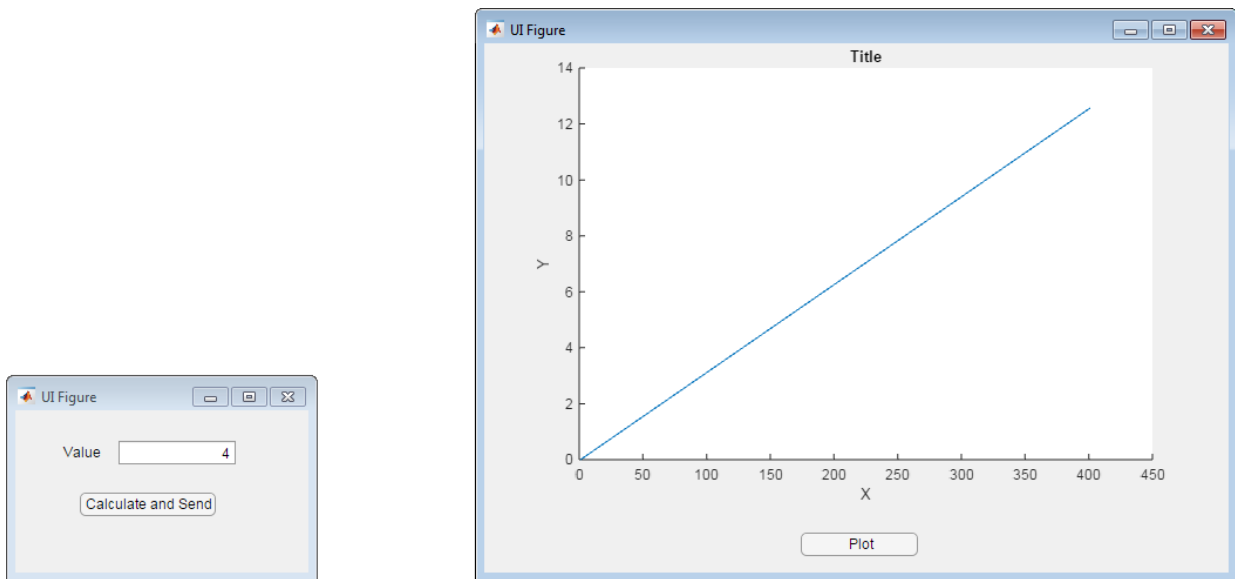
Lay out the app to get the app user value:

1   Drag a **Button** and an **Edit Field (Numeric)** including a label, from the
    Component Library into the central design area.

2   Double-click the numeric edit field label and change the text to `Value:`.

3   Double click the button text and change it to `Calculate and Send`.

4   Resize the UI figure to make it smaller.

5   Save the app as `calcandsend.mlapp`.

Next, Lay out and code the app to plot the results of the calculation:

1   On the **Designer** tab, click **New**.

2   Drag a **Button** from the **Component Library** to the bottom of the central design
    area.

3   Double-click the button text and change it to `Plot`.

4   Drag an **Axes (Line and Scatter)** from the **Component Library** into the central
    design area.

5   Resize the axes to fill most of the design area.

6   Click **Code View**.

7   On the **Editor** tab, click **Property** > **Insert Public Property**

8   In the code, replace the highlighted word `Property` with `Data`.

Note: If you code contains errors, the **Add Property** button is disabled. For
information on debugging, see "Detect and Correct Coding Errors Using App
Designer" on page 16-13.

---

9   On the **Editor** tab, click **Startup Function**.

10 Add this code to the startup function:

    ```
    app.UIFigure.Position = [500 100 635 450];
    ```

11 In the **Component Browser**, right-click **app.Button** and select **Callbacks** > **Add
    ButtonPushedFcn callback**, and then click **OK**.

12 Add this code to the ButtonButtonPushed callback function:

    ```
    y = app.Data;
    plot(app.UIAxes,y);
    ```

13 Save the app as `graphdata.mlapp`.

Code the first app, `calcandsend.mlapp`

1 Click the **calcandsend.mlapp** tab.

2 Click **Code View**.

3 In the **Component Browser**, right-click **app.Button** and select **Callbacks** > **Add ButtonPushedFcn callback**. Click **OK** to accept the default callback name.

4 Add this code to the `ButtonButtonPushed` function:

```
v = app.NumericEditField.Value;
y = 0:pi/100:v*pi;
g = graphdata;
g.Data = y;
```

The code calculates the value to be plotted. It then, calls the second app, and specifies the value calculated in the first app as the `Data` property value for the second app.

Run and test `calcandsend.mlapp`:

In the **Value** field, enter a numeric value, and then press **Calculate and Send**.

When the second app opens, click Plot. The data sent from the first app is plotted in the second app.

Code and Call App Functions in App Designer

There are four types of app functions:

- A function that runs when the app user starts the app.

  This is the *startup function* for the app. Use a startup function, for instance, to initialize properties.

  For an example, see "App Designer Startup Function" on page 16-10.

- A function that runs when the app user manipulates a control.

  This is a *callback function* for the component. Use a callback function, for instance, to display the value of a slider in a numeric edit field as the app user moves the slider.

  For an example, see "Code Response to Reflect Changing Slider Value" on page 16-42.

- A utility function that performs a task that you can reuse across your app, but not outside it.

  This is a *private utility function*. Use a private function, for instance, if you need to use the same code in multiple callbacks. You can code a private function containing that code, and then call the private function from each callback.

  For an example, see "Create Private and Public Utility Functions" on page 16-24.

- A utility function that performs a task that you can use outside (and across) your app.

  This is a *public utility function*. Use a public function, for instance, if you need to share the function with other apps, the MATLAB workspace, or devices attached to your computer system.

  For information on how to create a public function, see "Create Private and Public Utility Functions" on page 16-24.

Create Private and Public Utility Functions

Create a private or public utility function from App Designer code view, as follows:

1 On the **Editor** tab, click the **Function** down arrow, and then select **Insert Private Function** or **Insert Public Function**.

Unless you intend to use the function outside the current app, choose **Add Private Function**.

2   App Designer adds the framework for your function above the app startup function.

Except for the access specification, the framework for private and public functions is identical.

```
methods (Access = private)

    function results = func(app)

    end

end
```

The first time you add a private function, App Designer creates a private method block. If you add additional private functions, they are added to this method block. Similarly, App Designer creates a public method block the first time you add a public function to your app.

3   Replace the highlighted text, `func`, with a meaningful name for your function.

4   Optionally, replace `results` with one or more output arguments and add additional input arguments. For example:

```
methods (Access = private)

    function [a b] = PopulationEst(app,c)

    end

end
```

If your function does not require output or return a value, you can delete the `results =` syntax from the function declaration.

5   Add your function code.

Run your code and debug it, if needed, as described in "Detect and Correct Coding Errors Using App Designer" on page 16-13.

Create and Call a Private Utility Function

This example shows how to share a private utility function throughout an app. It uses a private function, `mycalc`, to calculate the sum and product of two values specified by the app user. When the app user changes either value, the app recalculates the sum and product by calling the `mycalc` function.
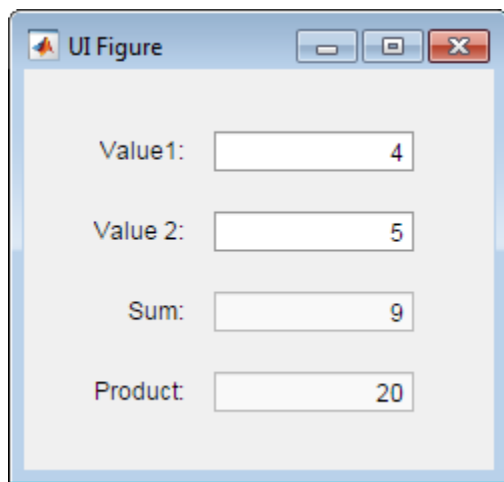
In App Designer:

1   Drag four numeric edit fields (and labels) from the **Component Library** on the left of App Designer into the design area. Arrange and customize them as shown in the preceding image and described in the table that follows. By default, when you drag a numeric edit field into the design area, an associated label is also added to the design area.

    To rename a component, select it in the design area, double-click the current component name in the **Component Browser**, and then type a new name.

Component	Label	Rename Numeric Edit Field Component	Nondefault Numeric Edit Field Property Values
Numeric Edit Field	Value 1:	Value1	None

Component	Label	Rename Numeric Edit Field Component	Nondefault Numeric Edit Field Property Values
Numeric Edit Field	Value 2:	`Value2`	None
Numeric Edit Field	Sum:	`Sum`	**Editable:** no
Numeric Edit Field	Product:	`Product`	**Editable:** no

2 Click **Code View**.

3 Create a function to calculate the sum and the product of the values that the app user enters for value 1 and value 2:

a On the **Editor** tab, click **Function** > **Insert Private Function**.

Note: If you code contains errors, the **Function** button is disabled. For information on debugging, see "Detect and Correct Coding Errors Using App Designer" on page 16-13.

---

b Replace the blue highlighted text, `func`, with a meaningful name for this function, `mycalc`.

c Delete the output argument, `results`.

d Code the function to get the app-user-entered values, perform calculations, and assign values to the function output arguments, `sum` and `product`.

The function appears as follows:

```
function mycalc(app)
    a = app.Value1.Value;
    b = app.Value2.Value;
    sum = a+b;
    prod = a*b;
  app.Sum.Value = sum;
  app.Product.Value = prod;
end
```

4 Code the callback for the `Value1` numeric edit field to call the `mycalc` function whenever the app user changes the field value:

a In the **Component Browser**, right-click **app.Value1** and select **Callbacks** > **Add ValueChangedFcn callback**.

b In the Add Callback Function dialog box, click **OK**.

   c   Replace the default `Value1ValueChanged` function code with the following code. The code calculates the sum and product and update the corresponding numeric edit fields:

      `mycalc(app);`

5   Code the callback for the `Value2` numeric edit field to call the `mycalc` function whenever the app user changes the field value:

   a   In the **Component Browser**, right-click **app.Value2** and select **Callbacks** > **Add ValueChangedFcn callback**.

   b   In the Add Callback Function dialog box, then click **OK**.

   c   Replace the default `Value2ValueChanged` function code with the following code:

      `mycalc(app);`

6   On the toolstrip, click **Run**.

     Save the app when prompted.

Test the app by entering numbers in the **Value 1** and **Value 2** fields and then clicking away from the fields.

Call a Public Utility Function

You can call a public utility function from a different app than the one in which it is defined. To do so, an instance of the app that defines the utility function must exist in the workspace of the app that calls the function.

For example, suppose that you create an app `getInput`, that contains a public function, `stats`. To call the `stats` function from a second app, `plotStats`, `plotStats` must contain code that calls `getInput` and assigns the output to a property.

The following `plotStats` code assigns the output from `getInput` to the property `getdata`.

```
app.getdata = getInput;
```

Then the `plotStats` app can call the `stats` function defined `getInput` by using a command such as this:

```
results = app.getdata.stats
```

Delete Code from the App Designer Editor

You can delete components, properties, and functions (including callbacks) from your app. You can delete any of these app elements from code view. From design view, you can delete app components and callbacks only.

To delete a component, in the **Component Browser**, right-click the code name. Then, select **Delete** from the context menu.
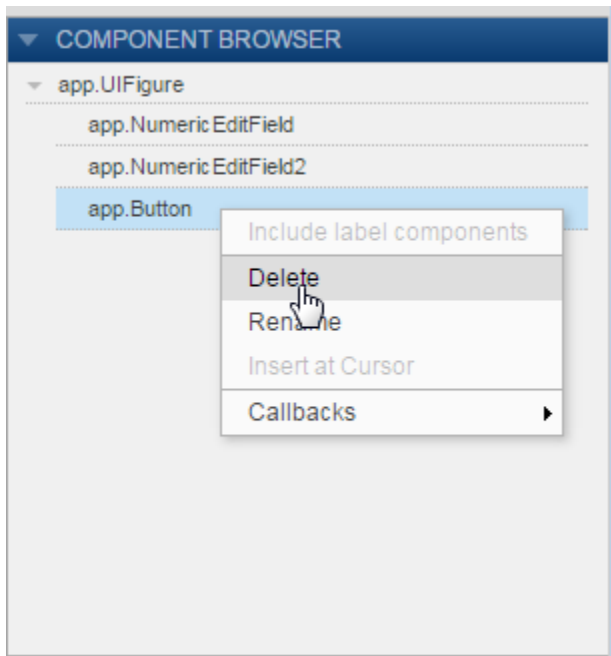

App Designer updates the code to remove the deleted component and its associated label component, if one exists. App Designer also removes the component and its label from design view.

Note: When you delete a component, callbacks created for that component are *not* deleted. Therefore, the code you wrote for the callback is preserved.

If you do not want to keep the callback code, in the **Code Browser** on the **Callbacks** tab, right-click the callback function name. Then, select **Delete** from the context menu.
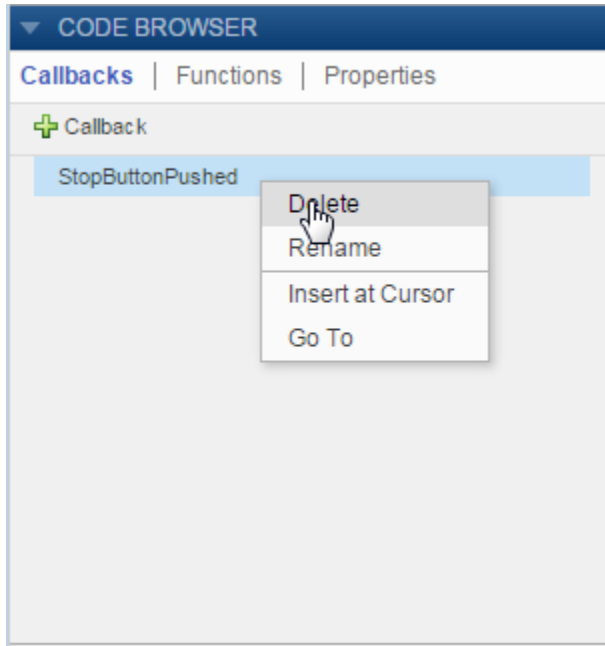
Similarly, to delete utility functions and properties you created, in the **Code Browser** on the **Functions** or **Properties** tab, right-click the function or property name. Then, select **Delete** from the context menu.

Simple Example Apps for App Designer

Each of these examples provides instructions on performing tasks frequently used in apps.

In this section...
"Display Plot in Axes Component" on page 16-31
"Display Multiple Plots in Axes Component" on page 16-34
"Display Plots in Multiple Axes Components" on page 16-39
"Code Response to Reflect Changing Slider Value" on page 16-42
"Code Response to Button Group Selection" on page 16-45
"Control Component Resize Behavior When App Window Resizes" on page 16-48

For more examples, see:

- "Use App Designer to Create a Simple Calculator App"
- "Use App Designer to Create a Data Analysis App"
- "Use App Designer to Create an App with Instrumentation Controls"

Display Plot in Axes Component

This example shows how to create an app that plots calculated data, based on an app user's input value. When the app user clicks a push button, the calculation results are plotted. The running app looks like this image.
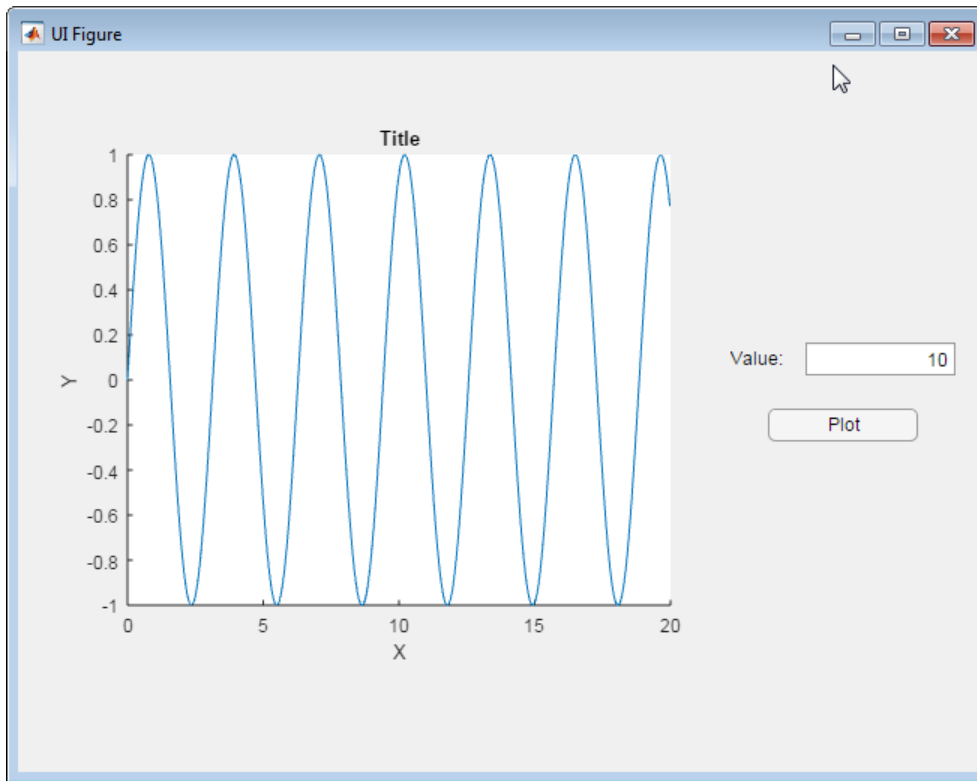
To create the app:

1 In App Designer, from the **Component Library** on the left, drag an **Axes (Line and Scatter)** component, an **Edit Field (Numeric)** component, and a **Button** component into the central design area.

2 Arrange the components as shown in the preceding image.

   Do not be concerned with component text in this step.

3 In the design area, double-click the `Edit Field` text and type `Value`.

4 In the design area, click the `Button` text and type `Plot`.

5 Above the design area, click **Code View**.
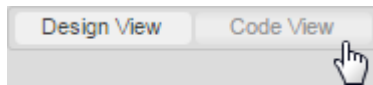
6   In the **Component Browser**, to the right of the editor, right-click **app.Button** and select **Callbacks** > **Add ButtonPushedFcn callback**.
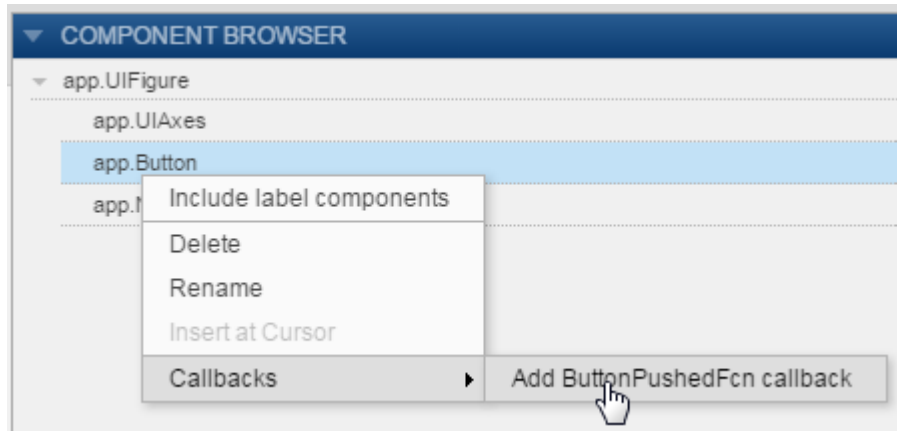
7   In the Add Callback Function dialog box, click **OK**. (Accept the default callback function name.)

8   Add this code to the callback function. The code uses the value of the numeric edit field in calculations and plots the results.

```
x = 0:pi/100:2*app.NumericEditField.Value;
y = sin(2*x);
plot(app.UIAxes,x,y);
```

9   Run the app.

On the toolstrip, click **Run**.

Save the app when prompted.
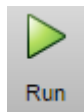
Test the app by typing a number in the **Value** field, and then clicking **Plot**.

Display Multiple Plots in Axes Component

This example shows how to create an app that plots data as specified by the app user's selection from a drop-down component. The running app looks like this image.
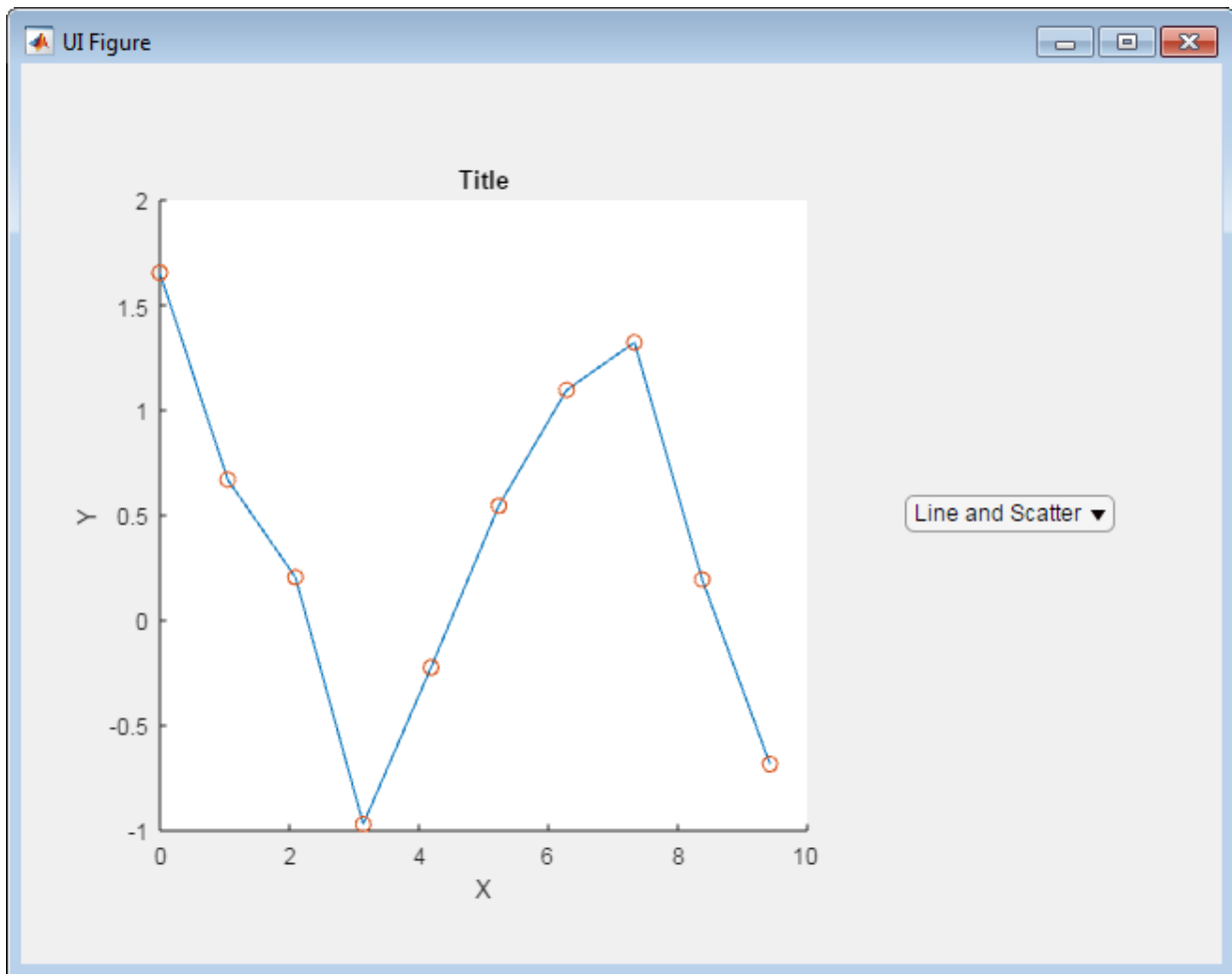
To create the app:

1   In App Designer, from the **Component Library** on the left, drag an **Axes (Line and Scatter)** into the central design area.

2 While holding down the **Crtl** key, drag a **Drop Down** into the central design area.

Holding down the **Crtl** key as you drag a component from the component library prevents App Designer from adding a label component along with the drop-down component.

3 Resize the axes component, and then arrange the components as shown in the preceding image.

Do not be concerned with axes labels or drop-down component text in this step.

4 Select the axes in the design area. Then, in the **Axes (Line and Scatter) Properties** panel to the right of the design area, clear the **Title**, **X Label**, and **Y Label** fields.
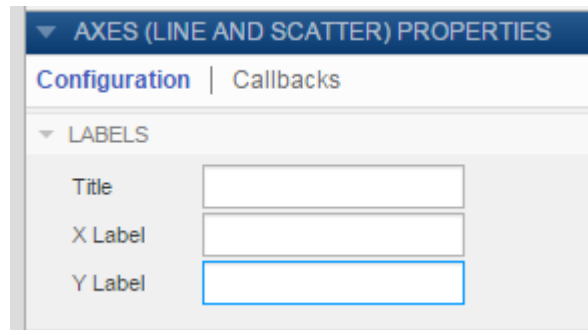
5 Specify the drop-down component text.

In the central design area, click the drop-down component, and then in the **Drop Down Properties** panel, set the values as shown. To edit a name in the **Items** column, double-click it. When you finish making text edits, make sure **Line Plot** is the selected item.
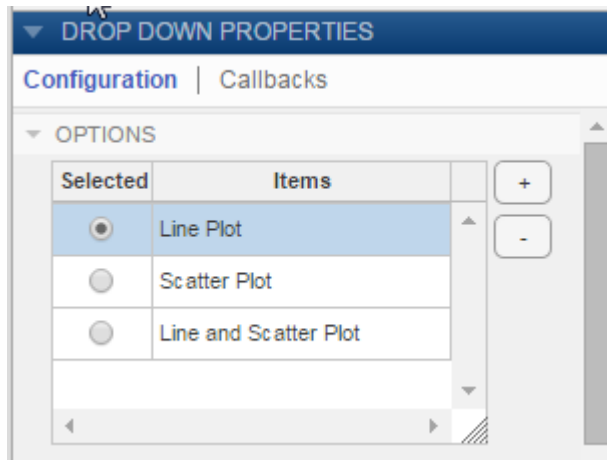
6 In the design area, resize the drop-down component to make it slightly wider. (Otherwise the `Line and Scatter Plot` string will be clipped in the running app.)

Tip To make the drop-down component the same size as shown in this example, on the toolstrip **Canvas** tab, select **Show resizing hints**.
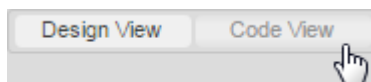
.

When you widen the drop-down component, App Designer displays the width in pixels. The width used in this example is 110 pixels.
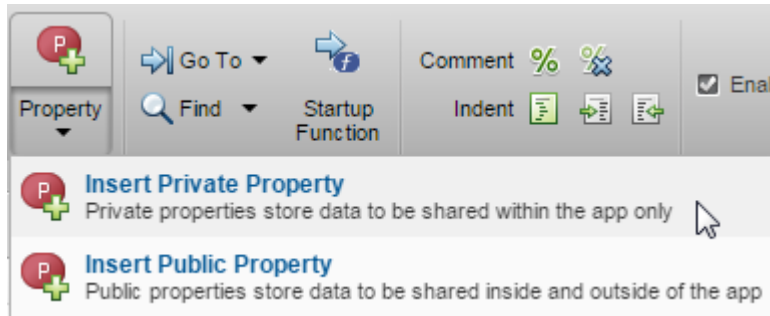

---

7 View the code created for your app.

Above the design area, click **Code View**.

8 Add a private property to hold plotting data for your app.

On the **Editor** tab, click the **Property** down arrow and select **Insert Private Property**.

9 In the code editor, replace the highlighted word `Property` with the property name, `xdata`.

When you refer to the property in a function, use dot notation, `app.xdata`.

10 Add another private property, `ydata`.

11 Specify the plot that you want to appear when the app first opens by adding code to the startup function.

On the **Editor** tab, click **Startup Function**. Add this code to the `startup` function to specify the data and the command to plot it.

```
app.xdata = linspace(0,3*pi,10);
app.ydata = cos(app.xdata) + rand(1,10);
plot(app.UIAxes,app.xdata,app.ydata);
```

12 Create a callback to control what the app does when an app user makes a selection from the drop-down component.

In the **Component Browser**, right-click **app.DropDown** and select **Callbacks** > **Add ValueChangedFcn callback**.
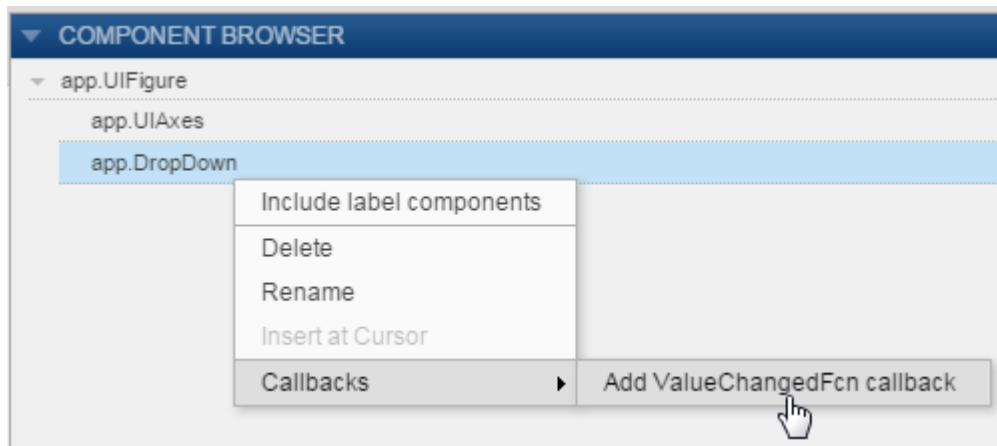
13 In the Add Callback Function dialog box, click **OK**. (Accept the default callback function name.)

App Designer adds sample code `value = app.DropDown.Value;` to the `DropDownValueChanged` function.

14 Update the callback code so it appears as follows. This code gets the option that the app user selected from the drop-down component, and then plots the data accordingly. The `hold` function keeps the scatter plot from replacing the line plot.

```
value = app.DropDown.Value;
if strcmp(value,'Line')
    plot(app.UIAxes,app.xdata,app.ydata);
elseif strcmp(value,'Scatter')
    scatter(app.UIAxes, app.xdata, app.ydata);
else
    plot(app.UIAxes,app.xdata,app.ydata);
    hold(app.UIAxes,'on');
    scatter(app.UIAxes, app.xdata, app.ydata);
end
```

15 Run the app.

On the **Editor** tab, click **Run**.

Save the app when prompted.

Run
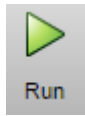
Test the app by selecting different options from the drop-down component.

Display Plots in Multiple Axes Components

This example shows how to create an app that accepts input parameters and plots data in two axes. The parameters define a time-varying and frequency-varying signal. One plot displays the data in the time domain. The other plot displays the data in the frequency domain. Both plots update when the app user clicks the push button. The running app looks like this image.
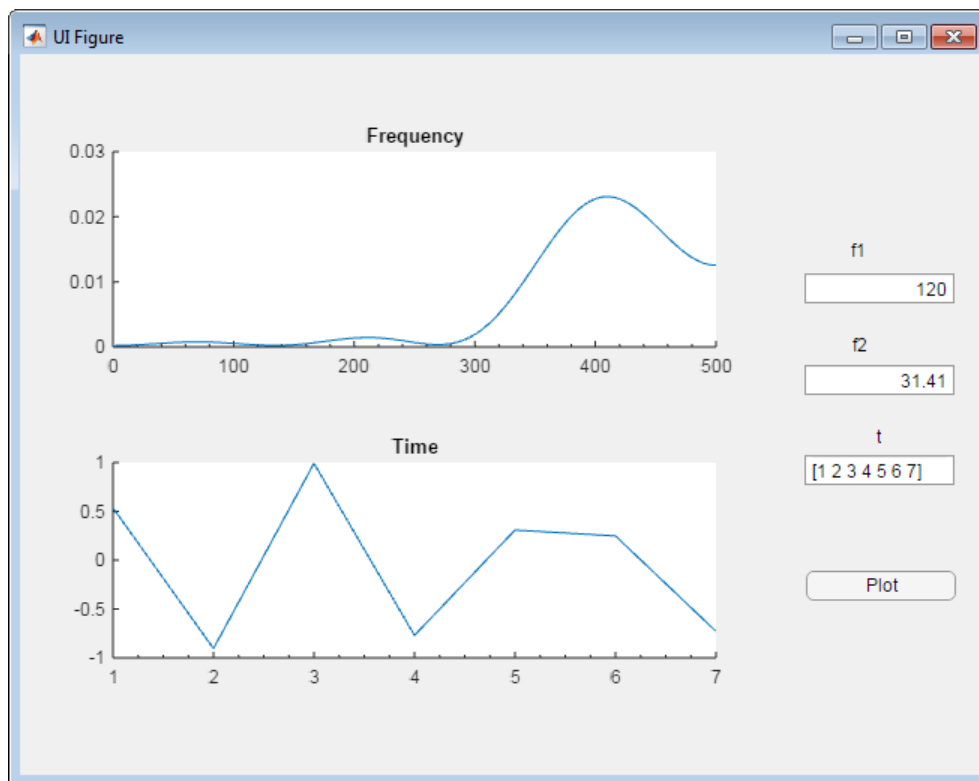
To create the app:

1 In App Designer, from the **Component Library** on the left, drag the components described in the following table into the central design area. Arrange the components as shown in the preceding image. Use the property sheet to the right of the design area to set component property values. By default, a label component is added when you drag an edit field into the design area. The **Label** column indicates the text to use for the label component associated with each edit field component that you add.

| Component	Name to Use in Code	Label	Property to Set
Axes (Line and Scatter)	UIAxesFreq	Not applicable	**Title:** Frequency
Axes (Line and Scatter)	UIAxesTime	Not applicable	**Title:** Time
Edit Field (Numeric)	NumericEditFieldf1	f1	**Maximum:** 500
Edit Field (Numeric)	NumericEditFieldf2	f2	**Maximum:** 500
Edit Field (Text)	EditFieldTime	t	Not Applicable
Button	Button	Not Applicable	**Text:** Plot

Tip To change the code name for a component, double-click the current component name in the **Component Browser**, and then type the new name.
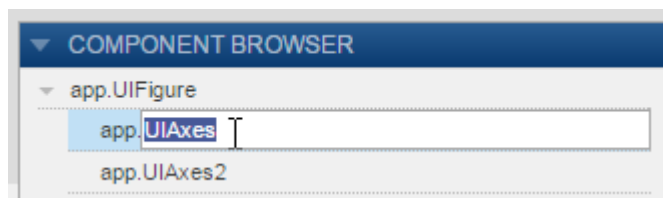
2 Add a callback function for the Plot button.

In the design area, right-click the **Plot** button and select **Callbacks** > **Add ButtonPushedFcn callback**.
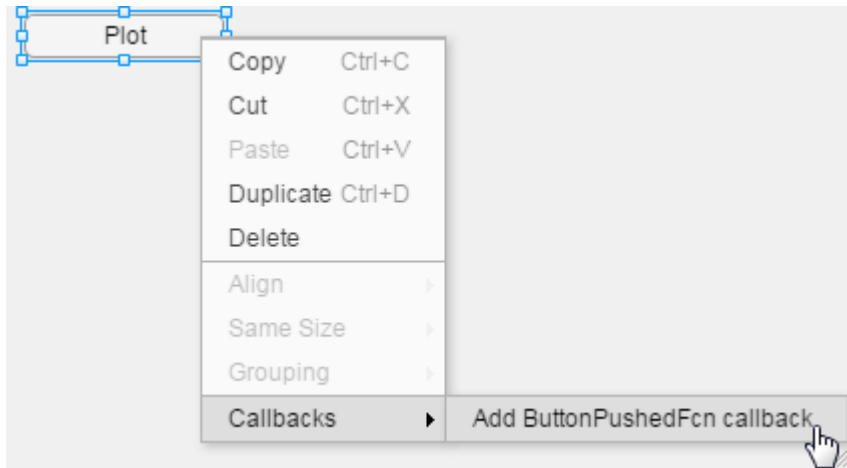
3 In the Add Callback Function dialog box, click **OK**. (Accept the default callback function name.)

App Designer opens code view.

4 Get the input values, perform calculations, and plot the data in the appropriate axes.

Add this code to `function ButtonButtonPushed`.

```
% Get app user input
f1 = app.NumericEditFieldf1.Value;
f2 = app.NumericEditFieldf2.Value;
t = str2num(app.EditFieldTime.Value);

% Calculate data
x = sin(2*pi*f1*t) + sin(2*pi*f2*t);
y = fft(x,512);
m = y.*conj(y)/512;
f = 1000*(0:256)/512;

% Create frequency plot in proper axes
plot(app.UIAxesFreq,f,m(1:257));
app.UIAxesFreq.XMinorTick = 'on';

% Create time plot in proper axes
plot(app.UIAxesTime,t,x);
app.UIAxesTime.XMinorTick = 'on';
```
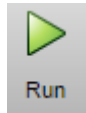
5 Run the app.

On the toolstrip, click **Run**.

Save the app when prompted.

Test the app by trying the various app controls in the running app.

- The numeric edit fields, `f1` and `f2`, display an error message if you enter a nonnumeric value or one that falls outside the limits that you set.

  This automatic input validation is an advantage to using a numeric edit field, rather than a text edit field to get numeric input.

- The edit field for the time vector does not prevent you from entering an invalid vector. To handle invalid time vector values, define a `ValueChangedFcn` callback function for the `EditFieldTime` component. Then code it to determine if the app user's input is valid.

Code Response to Reflect Changing Slider Value

This example shows how to create an app that displays a slider value in a semicircular gauge. The semicircular gauge is continuously updated to reflect the current slider value as the app user moves it. (Another example, "Create Simple App Using App Designer" on page 13-2 updates the semicircular gauge only when the app user releases the mouse button from the slider.) The running app looks like this image.
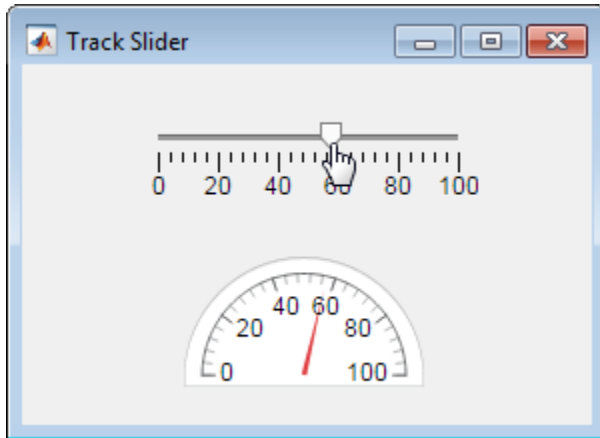
To create the app:

1. In App Designer, while holding down the **Ctrl** key, drag a **Slider** and a **Semicircular Gauge** from the **Component Library** into the central design area. Arrange the components as shown in the preceding image.

Tip Holding down the **Crtl** key as you drag a component from the component library prevents App Designer from adding a label component along with the component you drag in.

---

2. Specify a name to appear in the app title bar.

   In the **Component Browser**, click app.UIFigure, and then in the **UIFigure Properties** panel change the **Title** field to Track Slider.
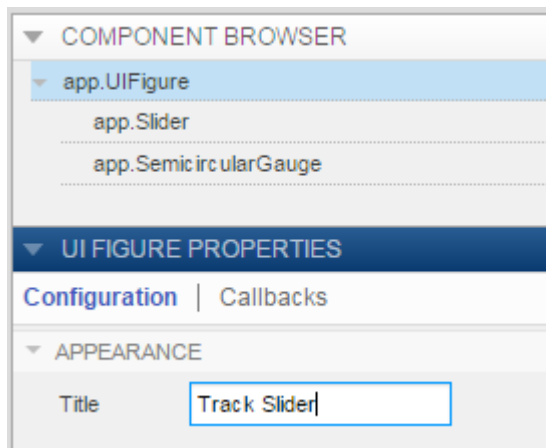
3 In the central design area, right-click the slider and select **Callbacks** > **Add ValueChangingFcn callback**.
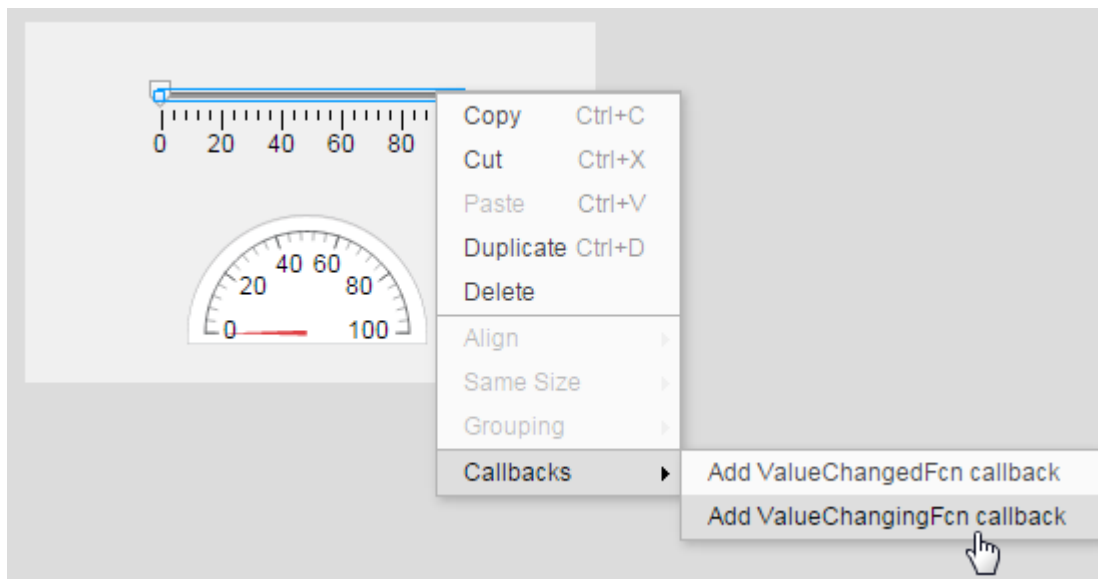
4 In the Add Callback Function dialog box, click OK. (Accept the default callback function name.)

Code view opens.

5 Code the callback to update the edit field when the app user drags the slider.

When the app user drags the slider, MATLAB generates `event` data and stores it in the slider event `Value` property. You can query the event data using dot notation.

Replace the default`SliderValueChanging` callback code, `changingValue = event.Value;`, with the following code:

```
app.SemicircularGauge.Value = event.Value;
```

6 Run the app.

On the toolstrip, click **Run**.

Save the app when prompted.
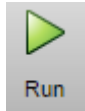
Test the app by performing these steps:

- Drag the slider to a new value.

  As you do so, the semicircular gauge is continuously updated to display the current slider value.

- Click the slider.

  The semicircular gauge is updated to display the new slider value.

Code Response to Button Group Selection

This example shows how to create an app that changes the color of a lamp based on which radio button the app user selects. The running app looks like this image.
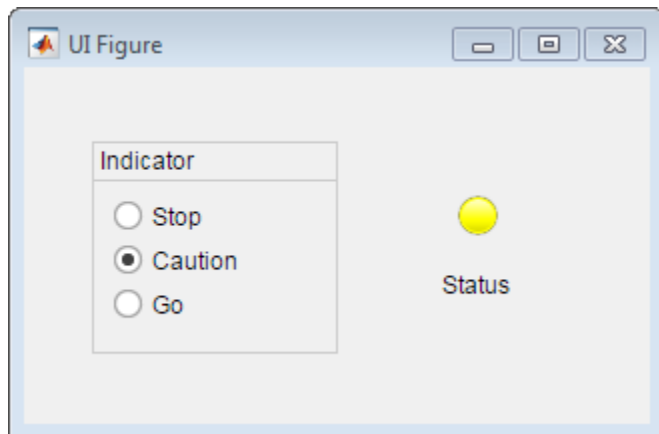
To create the app:

1   In App Designer, drag a **Radio Button Group** and a **Lamp** from the **Component Library** into the central design area. By default, a label is added with the lamp. Arrange and customize the components as shown in the preceding image and described in this table. Use the property sheet to the right of the design area to set property values.

Component	Name in Code	Property to Set
Radio Button Group	ButtonGroup (default)	**Title**: Indicator
RadioButton	StopRadioButton	**Text**: Stop
RadioButton	CautionRadioButton	**Text**: Caution
RadioButton	GoRadioButton	**Selected**: Yes **Text**: Go
Lamp	Lamp (default)	Not applicable
Label	Label (default)	**Text**: Status

Tip To change the code name for a component, double-click the current component name in the **Component Browser**, and then type the new name.
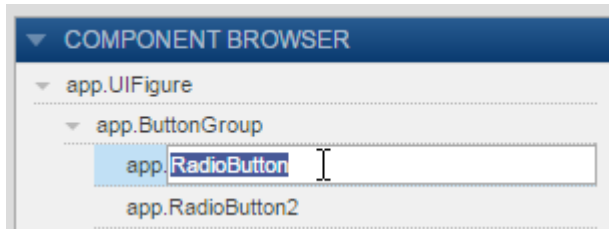
2 Above the design area, click **Code View**.
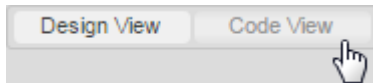
3 In the **Component Browser**, right-click **app.ButtonGroup**, and then select **Callbacks** > **SelectionChangeFcn callback**.
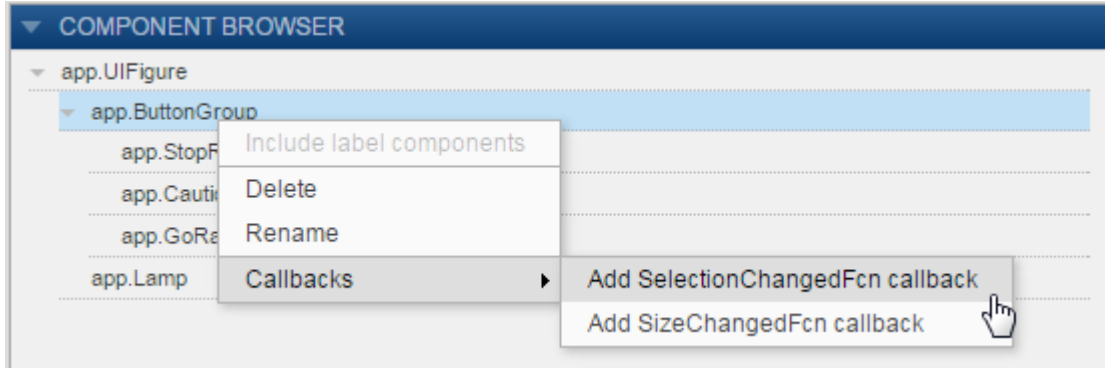
4 In the Add Callback Function dialog box, click **OK**. (Accept the default callback function name.)

Code view opens.

5 Specify code to determine which radio button is selected, and then set the lamp color accordingly.

Replace the default code in the `ButtonGroupSelectionChange` function with this code. The code changes the lamp color depending on the radio button selected by the app user.

```
                    switch app.ButtonGroup.SelectedObject
                        case app.GoRadioButton
                            app.Lamp.Color = 'green';
                        case app.CautionRadioButton
                            app.Lamp.Color = 'yellow';
                        case app.StopRadioButton
                            app.Lamp.Color = 'red';
                    end
```

6 Run the app.

On the toolstrip, click **Run**.

Save the app when prompted.
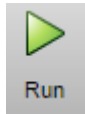
Test the app by selecting each radio button.

Control Component Resize Behavior When App Window Resizes

This example shows how to code the callback for an UI figure window containing a slider so that when the app user resizes the window, the slider size changes and remains centered in the window.

By default, components resize when the app user resizes the app window. If you code the UI figure `SizeChangedFcn` callback, your callback code takes precedence.

Note: Only consider coding the figure window `SizeChangedFcn` callback when you want components to resize, but do not like the default resize behavior.

The running app looks like this image
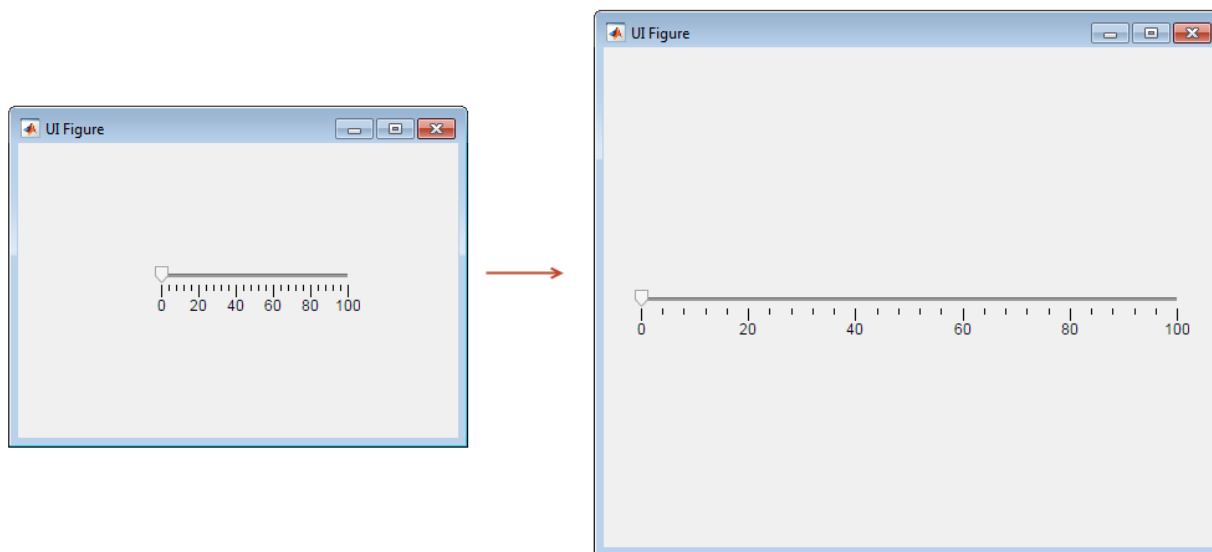
To create the app:

1 In App Designer, hold down the **Ctrl** key as you drag a **Slider** from the **Component Library** into the central design area.

Holding down the **Ctrl** key prevents a label component from being added with the slider.

2 Center the slider in the design area.

As you move the slider, App Designer displays hints to help you center the component.
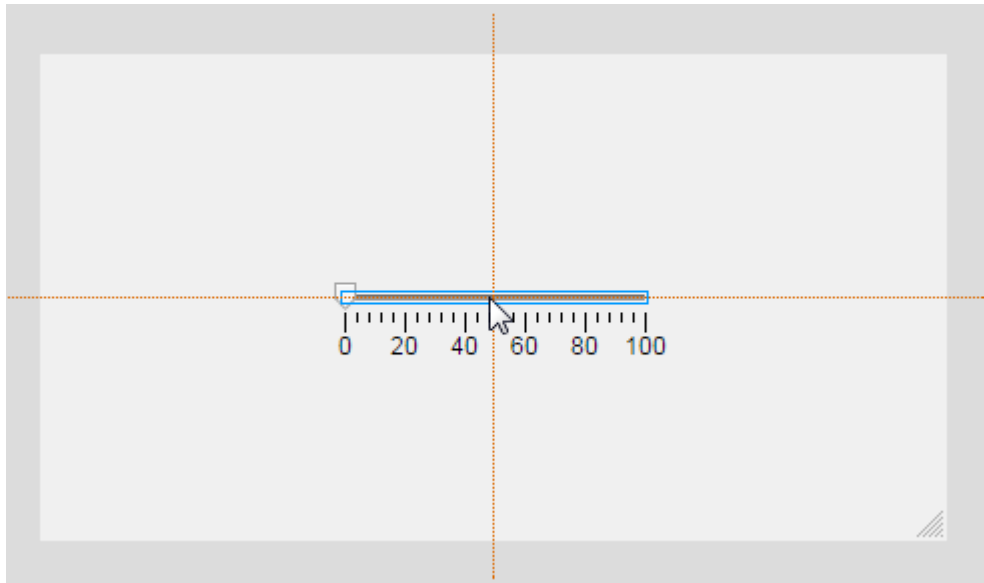
3   Above the design area, click **Code View**.
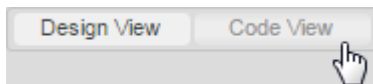
4   In the **Component Browser**, right-click **app.UIFigure**, and then select **Callbacks** > **Add SizeChangedFcn callback**.
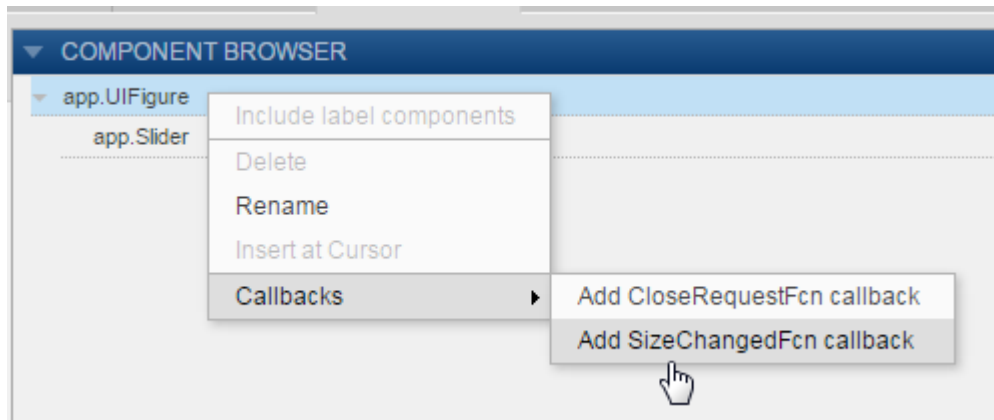
5 In the Add Callback Function dialog box, click **OK**. (Accept the default callback function name.)

6 Update the callback code as shown here.

```
position = app.UIFigure.Position;
app.Slider.Position = [30, (position(4)-6)/2, position(3)-60, 6];
```

This code gets the UI Figure window width and height after a resize operation and updates the location and size of the slider to keep it centered in the window. The slider will be 30 pixels from the left and right sides of the window. The code subtracts the slider height (6 pixels) from the window height and divides by 2. The code subtracts 60 from the window width to account for 30 pixels on either side of the slider.

7 Run the app.

On the toolstrip, click **Run**.

Save the app when prompted.
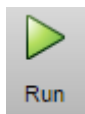
Test the app by resizing the UI Figure window.

Note: Code you specify for the figure `SizeChangedFcn` callback takes precedence over the resize behavior you select in design view with the **UIFigure Properties** panel.
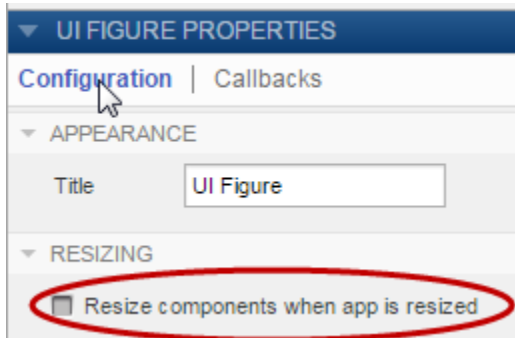
Keyboard Shortcuts

App Designer Keyboard Shortcuts

In this section...
"Shortcuts Available Throughout App Designer" on page 17-2
"Component Browser Shortcuts" on page 17-2
"Design View Shortcuts" on page 17-3
"Code View Shortcuts" on page 17-7

Shortcuts Available Throughout App Designer

Action	Key or Keys
Run the active app.	**F5**
Save the active app.	**Ctrl+S**
Save the active app, allowing you to specify a new file name. (Save as)	**Ctrl+Shift+S**
Open a previously saved app.	**Ctrl+O**
Redo an undone modification, returning it to the changed state.	**Ctrl+Y** or, in the design area only, **Ctrl+Shift+Z**
Undo a modification, returning it to the previous state.	**Ctrl+Z**
Alternate between design and code view.	**Shift + F7** If debugging is in progress, this shortcut does not change the view.
Quit App Designer.	**Ctrl+Q**

Component Browser Shortcuts

These shortcuts are available in the **Component Browser**, in both code view and design view

Action	Key or Keys
Select multiple components.	Hold down the **Ctrl** key as you click each component that you want to include in the multiselection.

Action	Key or Keys
Deselect a component from multiselection.	Hold down the **Ctrl** key as you click each component that you want to remove from a multiselection.
Navigate from clicked component to the previous or next component listed in the code browser.	**Up Arrow** and **Down Arrow**
Edit code name of clicked component in the code browser.	**F2** on Windows and Linux **Enter** on Mac

Design View Shortcuts

These shortcuts are available from the App Designer design view only.

Add Component Shortcuts

Action	Shortcut
Add component and associated label (if any) to central design area.	**Click** the component and hold down the mouse key to drag the component from the **Component Library** on the left into the design area.
Add component only to central design area.	Hold down the **Ctrl** key, click the component, and drag it from the **Component Library** on the left into the design area.

Component, Group, and Text Selection Shortcuts

Action	Key or Keys
Move the selection to the next component, or container in the design area tab key navigation sequence.	**Tab**
Move the selection to the previous component or container in the design area tab key navigation sequence.	**Shift+Tab**
Selects all components for which the `Parent` property value is the UI figure object, with one exception. If any of those components are grouped, the group is selected, not the individual components within the grouping.	**Ctrl+A**
Clear a component selection. Press again to reselect the component.	**Shift+Click** or **Ctrl+Click**
In the property editor or in-place editing, select all text in a text input field.	**Ctrl+A**
Select group containing a component.	**Alt + Click** a component

Group and Ungroup Components Shortcuts

Select the components that you want to group, and then press **Ctrl + G**. All components to be grouped must have the same parent component.

Action	Key or Keys
Group selected components.	**Ctrl+G**
Ungroup components in selected group.	**Ctrl+Shift+G**

Component and Group Move Shortcuts

This table summarizes the keyboard shortcuts for moving selected components and groups.

Action	Key or Keys
Move down 1 pixel.	**Down Arrow**
Move left 1 pixel.	**Left Arrow**

Action	Key or Keys
Move right 1 pixel.	**Right Arrow**
Move up 1 pixel.	**Up Arrow**
Move down 10 pixels.	**Shift+Down Arrow**
Move left 10 pixels.	**Shift+Left Arrow**
Move right 10 pixels.	**Shift+Right Arrow**
Move up 10 pixels.	**Shift+Up Arrow**
Cancel an in-progress operation.	**Escape**

Component Resize Shortcuts

Action	Key
Resize component while maintaining aspect ratio.	Press and hold down the **Shift** key before you begin to drag the component expander.
Resize component while keeping center location unchanged.	Press and hold down the **Ctrl** key before you begin to drag the component expander.
Resize component while maintaining aspect ration and keeping center location unchanged.	Press and hold down the **Ctrl** and **Shift** keys before you begin to drag the component expander.
Cancel an in-progress resize operation.	**Escape**

Component Copy, Duplicate, and Delete Shortcuts

Action	Key or Keys
Copy selected components and groups to the clipboard.	**Ctrl+C**
Duplicate the selected components and groups (without copying them to the clipboard).	**Ctrl+D**
Cut the selected components and groups from the design area onto the clipboard.	**Ctrl+X**
Delete the selected components and groups from the design area.	**Backspace** or **Delete**
Paste components and groups from the clipboard into the design area or a	**Ctrl+V**

Action	Key or Keys
container component (panel, tab, or button group). Radio buttons and toggle buttons can only be pasted into radio button groups or toggle button groups, respectively.	

Design Area Grid Shortcuts

Action	Keys
Toggle grid on and off.	**Alt+G**
Toggle snap to grid on and off.	**Alt+P**
Increase grid interval by 5 pixels.	**Alt+Page Up**
Decrease grid interval by 5 pixels.	**Alt+Page Down**

Component Alignment Shortcuts

Action	Keys
Align selected components and groups on their left edges.	**Ctrl+Alt+1**
Align selected components and groups on their horizontal centers.	**Ctrl+Alt+2**
Align selected components and groups on their right edges.	**Ctrl+Alt+3**
Align selected components and groups on their top edges.	**Ctrl+Alt+4**
Align selected components and groups on their vertical middle.	**Ctrl+Alt+5**
Align selected components and groups on their bottom edges.	**Ctrl+Alt+6**

Change Font Characteristics Shortcuts

Action	Key or Keys
Toggle the font weight of selected components *and their children* between normal and bold.	**Ctrl+B**

Action	Key or Keys
Decrease the value of the `FontSize` property of the selected components *and their children* by one step.	**Ctrl+[**
Increase the value of the `FontSize` property of the selected components *and their children* by one step.	**Ctrl+]**

Font size steps are: 8, 9, 10, 11, 12, 14, 16, 18, 20, 22, 24, 26, 28, 36, 48, 72.

Code View Shortcuts

These shortcuts are available only from the App Designer code view, within the editor.

- "Code Indenting Shortcuts" on page 17-7
- "Cut, Copy, and Paste Code Shortcuts" on page 17-7
- "Find Code Shortcuts" on page 17-8
- "Other App Designer Code Editor Shortcuts" on page 17-8

Code Indenting Shortcuts

Action	Key or Keys
Smart indent selected code.	**Ctrl+I**
Increase indent on current line of code or currently selected code.	**Ctrl+]**
Decrease indent on current line of code or currently selected code.	**Ctrl+[**

Cut, Copy, and Paste Code Shortcuts

Action	Key or Keys
Cut selected code.	**Ctrl+X**
Copy selected code.	**Ctrl+C**
Paste selected code.	**Ctrl+V**

Find Code Shortcuts

Action	Key or Keys
Find.	**Ctrl+F**
Find next.	**F3**
Find previous.	**Shift+F3**
Find selection.	**Ctrl+F3**

Other App Designer Code Editor Shortcuts

Action	Key or Keys
Add comment to selected code.	**Ctrl+R**
Evaluate selection.	**F9**
Open selection.	**Ctrl+D**
Go to specified line number.	**Ctrl+G**
Set or clear breakpoint.	**F12**

App Packaging

18

Packaging GUIs as Apps

Apps Overview

What Is an App?

A MATLAB app is a self-contained MATLAB program with a user interface that automates a task or calculation. All the operations required to complete the task — getting data into the app, performing calculations on the data, and getting results are performed within the app. Apps are included in many MATLAB products. In addition, you can create your own apps. The **Apps** tab on the MATLAB Toolstrip displays all currently installed apps when you click the down arrow on the far right of the toolstrip.
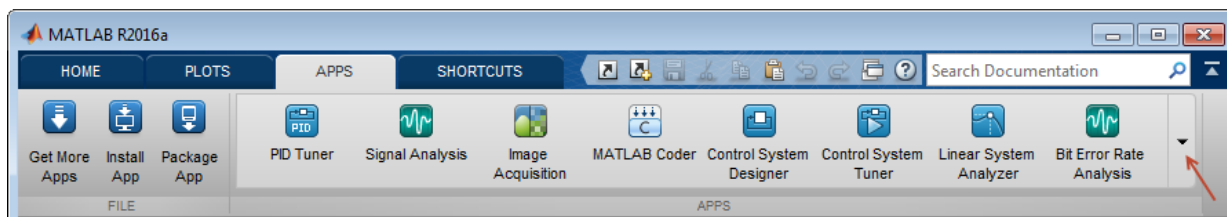
Note: You cannot run MATLAB apps using the MATLAB Runtime. Apps are for MATLAB to MATLAB deployment. To run code using the MATLAB Runtime, the code must be packaged using MATLAB Compiler™.

Where to Get Apps

There are three key ways to get apps:

- MATLAB Products

  Many MATLAB products, such as Curve Fitting Toolbox™, Signal Processing Toolbox™, and Control System Toolbox™ include apps. In the apps gallery, you can see the apps that come with your installed products.

- Create Your Own

  You can create your own MATLAB app and package it into a single file that you can distribute to others. The apps packaging tool automatically finds and includes all the files needed for your app. It also identifies any MATLAB products required to run your app.

You can share your app directly with other users, or share it with the MATLAB user community by uploading it to the MATLAB File Exchange. When others install your app, they do not need to be concerned with the MATLAB search path or other installation details.

Watch this video for an introduction to creating apps:

Packaging and Installing MATLAB Apps (2 min, 58 sec)

- Add-Ons

  Apps (and other files) uploaded to the MATLAB File Exchange are available from within MATLAB:

  1 On the **Home** tab, in the **Environment** section, click the **Add-Ons** arrow button.

  2 Click **Get Add-Ons**.

  3 Search for apps by name or a descriptive string.

Why Create an App?

When you create an app package, MATLAB creates a single app installation file (`.mlappinstall`) that enables you and others to install your app easily.

In particular, when you package an app, the app packaging tool:

- Performs a dependency analysis that helps you find and add the files your app requires.
- Reminds you to add shared resources and helper files.
- Stores information you provide about your app with the app package. This information includes a description, a list of additional MATLAB products required by your app, and a list of supported platforms.
- Automates app updates (versioning).

In addition when others install your app:

- It is a one-click installation.
- Users do not need to manage the MATLAB search path or other installation details.
- Your app appears alongside MATLAB toolbox apps in the apps gallery.

Best Practices and Requirements for Creating an App

Best practices:

- Write the app as an interactive application with a user interface written in the MATLAB language.
- All interaction with the app is through the user interface.
- Make the app reusable. Do not make it necessary for a user restart the app to use different data or inputs with it.
- Ensure the main function returns the handle of the main figure. (The main function created by GUIDE returns the figure handle by default.)

  Although not a requirement, doing so enables MATLAB to remove the app files from the search path when users exit the app.

- If you want to share your app on MATLAB File Exchange, you must release it under a BSD license. In addition, there are restrictions on the use of binary files such as MEX-files, p-coded files, or DLLs.

Requirements:

- The main file must be a function (not a script).
- Because you invoke apps by clicking an icon in the apps gallery, the main function cannot have input arguments. (It can, however, take optional arguments.)

Package Apps

Package apps you create into an app package for sharing with others. When you create an app package, MATLAB creates a single app installation file (`.mlappinstall`). The installation file enables you and others to install your app and access it from the apps gallery without concern for installation details or the MATLAB path.

Note: As you enter information in the Package Apps dialog box, MATLAB creates and saves a `.prj` file continuously. A `.prj` file contains information about your app, such as included files and a description. Therefore, if you exit the dialog box before clicking the **Package** button, the `.prj` file remains, even though a `.mlappinstall` file is not created. The `.prj` file enables you to quit and resume the app creation process where you left off.

To create an app installation file:

1   On the desktop Toolstrip, on the **Home** tab, click the **Add-Ons** down-arrow.

2   Click **Package App**.

3   In the Package App dialog box, click **Add main file** and specify the file that you use to run the app you created.

The main file must be callable with no input, and must be a function or method, not a script. MATLAB analyzes the main file to determine other MATLAB files on which the main file depends. For more information, see "Dependency Analysis" on page 18-11.

Tip   The main file must return the figure handle of your app for MATLAB to remove your app files from the search path when users exit the app. For more information, see "What Is the MATLAB Search Path?"

(Functions created by GUIDE return the figure handle.)

---

4   Add additional files required to run your app, by clicking **Add files/folders**.

Such files are data, image, and other files that were not included via dependency analysis.

You can include external interfaces, such as MEX-files, ActiveX, or Java® in the `.mlappinstall` file, although doing so can restrict the systems on which your app can run.

5 Describe your app

   a In the **App Name** field, type an app name.

      If you install the app, MATLAB uses the name for the `.mlappinstall` file and to label your app in the apps gallery.

   b Optionally, specify an app icon.

      Click the icon to the left of the **App Name** field to select an icon for your app or to specify a custom icon. MATLAB automatically scales the icon for use in the Install dialog box, App gallery, and quick access toolbar.

   c Optionally, select a previously saved screen shot to represent your app.

   d Optionally, specify author information.

   e In the **Description** field, describe your app so others can decide if they want to install it.

   f Identify the products on which your app depends.

      Click the plus button on the right side of the **Products** field, select the products on which your app depends, and then click **Apply Changes**.

After you create the package, when you select a `.mlappinstall` file in the Current Folder browser, MATLAB displays the information you provided (except your email address and company name) in the Current Folder browser **Details** panel. If you share you app in the MATLAB Central File Exchange, the same information also displays there. The screen shot you select, if any, represents your app in File Exchange.

6 Click **Package**.

As part of the app packaging process, MATLAB creates a `.prj` file that contains information about your app, such as included files and a description. The `.prj` file enables you to update the files in your app without requiring you to respecify descriptive information about the app.

7 In the Build dialog box, note the location of the installation file (`.mlappinstall`), and then click **Close**.

For information on installing the app, see "Install an Add-On Manually".

Modify Apps

When you update the files included in a `.mlappinstall` file, you recreate and overwrite the original app. You cannot maintain two versions of the same app.

To update files in an app you created:

1   In the Current Folder browser, navigate to the folder containing the project file (`.prj`) that MATLAB created when you packaged the app.

    By default, MATLAB writes the `.prj` file to the folder that was the current folder when you packaged the app.

2   From the Current Folder browser, double-click the project file for your app package, *appname*`.prj`

    The Package App dialog box opens.

3   Adjust the information in the dialog box to reflect your changes by doing any or all of the following:

    •   If you made code changes, add the main file again, and refresh the files included through analysis.

    •   If your code calls additional files that are not included through analysis, add them.

    •   If you want anyone who installs your app over a previous installation to be informed that the content is different, change the version.

        Version numbers must be a combination of integers and periods, and can include up to three periods — `2.3.5.2`, for example.

        Anyone who attempts to install a revision of your app over another version is notified that the version number is changed. The user can continue or cancel the installation.

    •   If your changes introduce different product dependencies, adjust the product list in the **Products** field.

4   Click **Package**.

Share Apps

To share your app with others, give them the `.mlappinstall` file. All files you added when you packaged the app are included in the `.mlappinstall` file. When the recipients install your app, they do not need to be concerned with the MATLAB path or other installation details. The `.mlappinstall` file manages that for them.

You can share your app with others by attaching the `.mlappinstall` file to an email message, or using any other method you typically use to share files.

If you upload your app to MATLAB Central File Exchange, your users can download the app from within MATLAB, as follows:

1 In MATLAB, on the **Home** tab, in the **Environment** section, click the **Add-Ons** arrow button.

2 Click **Get Add-Ons**.

3 Search for your app by name or a descriptive string.

Provide instructions on installing your app by referring your users to "Install an Add-On Manually".

Note: While `.mlappinstall` files can contain any files you specify, MATLAB Central File Exchange places additional limitations on submissions. Your app cannot be submitted to File Exchange when it contains any of the following:

- MEX-files

- Other binary executable files, such as DLLs or ActiveX controls. (Data and image files are typically acceptable.)

MATLAB App Installer File — mlappinstall

A MATLAB app installer file, `.mlappinstall`, is an archive file for sharing an app you created using MATLAB. A single app installer file contains everything necessary to install and run an app: the source code, supporting data, information (such as product dependencies), and the app icon.

An `.mlappinstall` file is a compressed package that conforms to the Open Packaging Conventions (OPC) interoperability standard. You can search for and install `.mlappinstall` files using your operating system file browser. When you select an `.mlappinstall` file in Windows Explorer or Quick Look (Mac OS), the browser displays properties for the file, such as `Authors` and `Release`. Use these properties to search for `.mlappinstall` files. Use the `Tags` property to add custom searchable text to the file.

For information on creating an app installer file for an app you created, see "Package Apps" on page 18-5.

Dependency Analysis

When you create an app package, MATLAB analyzes your main file and attempts to include all the MATLAB files that it uses in the app package. However, MATLAB does not guarantee to find every dependent file. MATLAB does not find files for functions that your code references as strings (for instance, as arguments to `eval`, `feval`, and callback functions). In addition, MATLAB can include some files that the main file never calls when it runs.

Dependency analysis searches for executable files, such as:

- MATLAB files
- P-files
- `.fig` files
- MEX-files

Dependency analysis does not search for data, image, or other binary files, such as Java classes and `.jar` files. Add such files manually when you package your app. The Package Apps dialog box provides an option for doing so.