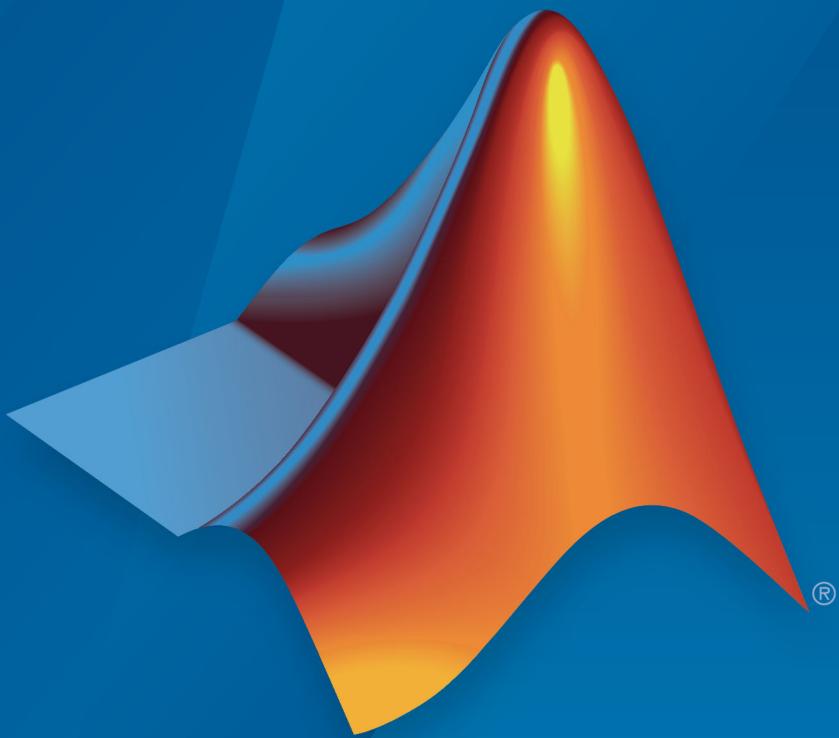


MATLAB®

App Building



MATLAB®

R2018b

 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

MATLAB® App Building

© COPYRIGHT 2000–2018 by The MathWorks, Inc.

The software described in this document is furnished under a license agreement. The software may be used or copied only under the terms of the license agreement. No part of this manual may be photocopied or reproduced in any form without prior written consent from The MathWorks, Inc.

FEDERAL ACQUISITION: This provision applies to all acquisitions of the Program and Documentation by, for, or through the federal government of the United States. By accepting delivery of the Program or Documentation, the government hereby agrees that this software or documentation qualifies as commercial computer software or commercial computer software documentation as such terms are used or defined in FAR 12.212, DFARS Part 227.72, and DFARS 252.227-7014. Accordingly, the terms and conditions of this Agreement and only those rights specified in this Agreement, shall pertain to and govern the use, modification, reproduction, release, performance, display, and disclosure of the Program and Documentation by the federal government (or other entity acquiring for or through the federal government) and shall supersede any conflicting contractual terms or conditions. If this License fails to meet the government's needs or is inconsistent in any respect with federal procurement law, the government agrees to return the Program and Documentation, unused, to The MathWorks, Inc.

Trademarks

MATLAB and Simulink are registered trademarks of The MathWorks, Inc. See www.mathworks.com/trademarks for a list of additional trademarks. Other product or brand names may be trademarks or registered trademarks of their respective holders.

Patents

MathWorks products are protected by one or more U.S. patents. Please see www.mathworks.com/patents for more information.

Revision History

November 2000	Online Only	New for MATLAB 6.0 (Release 12)
June 2001	Online Only	Revised for MATLAB 6.1 (Release 12.1)
July 2002	Online Only	Revised for MATLAB 6.6 (Release 13)
June 2004	Online Only	Revised for MATLAB 7.0 (Release 14)
October 2004	Online Only	Revised for MATLAB 7.0.1 (Release 14SP1)
March 2005	Online Only	Revised for MATLAB 7.0.4 (Release 14SP2)
September 2005	Online Only	Revised for MATLAB 7.1 (Release 14SP3)
March 2006	Online Only	Revised for MATLAB 7.2 (Release 2006a)
May 2006	Online Only	Revised for MATLAB 7.2
September 2006	Online Only	Revised for MATLAB 7.3 (Release 2006b)
March 2007	Online Only	Revised for MATLAB 7.4 (Release 2007a)
September 2007	Online Only	Revised for MATLAB 7.5 (Release 2007b)
March 2008	Online Only	Revised for MATLAB 7.6 (Release 2008a)
October 2008	Online Only	Revised for MATLAB 7.7 (Release 2008b)
March 2009	Online Only	Revised for MATLAB 7.8 (Release 2009a)
September 2009	Online Only	Revised for MATLAB 7.9 (Release 2009b)
March 2010	Online Only	Revised for MATLAB 7.10 (Release 2010a)
September 2010	Online Only	Revised for MATLAB 7.11 (Release 2010b)
April 2011	Online Only	Revised for MATLAB 7.12 (Release 2011a)
September 2011	Online Only	Revised for MATLAB 7.13 (Release 2011b)
March 2012	Online Only	Revised for MATLAB 7.14 (Release 2012a)
September 2012	Online Only	Revised for MATLAB 8.0 (Release 2012b)
March 2013	Online Only	Revised for MATLAB 8.1 (Release 2013a)
September 2013	Online Only	Revised for MATLAB 8.2 (Release 2013b)
March 2014	Online Only	Revised for MATLAB 8.3 (Release 2014a)
October 2014	Online Only	Revised for MATLAB 8.4 (Release 2014b)
March 2015	Online Only	Revised for MATLAB 8.5 (Release 2015a)
September 2015	Online Only	Revised for MATLAB 8.6 (Release 2015b)
March 2016	Online Only	Revised for MATLAB 9.0 (Release 2016a)
September 2016	Online Only	Revised for MATLAB 9.1 (Release 2016b)
March 2017	Online Only	Revised for MATLAB 9.2 (Release 2017a)
September 2017	Online Only	Revised for MATLAB 9.3 (Release 2017b)
March 2018	Online Only	Revised for MATLAB 9.4 (Release 2018a)
September 2018	Online Only	Revised for MATLAB 9.5 (Release 2018b)

Introduction to Creating UIs

About Apps in MATLAB Software

1

Ways to Build Apps	1-2
Use App Designer	1-2
Use GUIDE	1-3
Use MATLAB Functions to Create Apps	
Programmatically	1-4

How to Create a App with GUIDE

2

Create a Simple App Using GUIDE	2-2
Open a New UI in the GUIDE Layout Editor	2-3
Set the Window Size in GUIDE	2-5
Layout the UI	2-6
Code the Behavior of the App	2-16
Run the App	2-22
Files Generated by GUIDE	2-24
Code Files and FIG-Files	2-24
Code File Structure	2-24
Adding Callback Templates to an Existing Code File	2-25
About GUIDE-Generated Callbacks	2-26

A Simple Programmatic App

3

Create a Simple App Programmatically	3-2
Create a Code File	3-3
Create the Figure Window	3-3
Add Components to the UI	3-4
Code the App's Behavior	3-7
Verify Code and Run the App	3-10

Create UIs with GUIDE

What Is GUIDE?

4

GUIDE: Getting Started	4-2
UI Layout	4-2
UI Programming	4-2

GUIDE Preferences and Options

5

GUIDE Preferences	5-2
Set Preferences	5-2
Confirmation Preferences	5-2
Backward Compatibility Preference	5-4
All Other Preferences	5-4
GUIDE Options	5-8
The GUI Options Dialog Box	5-8
Resize Behavior	5-9
Command-Line Accessibility	5-9
Generate FIG-File and MATLAB File	5-10
Generate FIG-File Only	5-12

GUIDE Templates	6-2
Access the Templates	6-2
Template Descriptions	6-3
Set the UI Window Size in GUIDE	6-11
Prevent Existing Objects from Resizing with the Window	6-11
Set the Window Position or Size to an Exact Value	6-12
Maximize the Layout Area	6-12
Add Components to the GUIDE Layout Area	6-13
Place Components	6-13
User Interface Controls	6-19
Panels and Button Groups	6-40
Axes	6-45
Table	6-49
ActiveX Component	6-60
Resize GUIDE UI Components	6-62
Align GUIDE UI Components	6-66
Align Objects Tool	6-66
Property Inspector	6-69
Grid and Rulers	6-72
Guide Lines	6-73
Customize Tabbing Behavior in a GUIDE UI	6-75
Create Menus for GUIDE Apps	6-78
Menus for the Menu Bar	6-78
Context Menus	6-88
Create Toolbars for GUIDE UIs	6-95
Toolbar and Tools	6-95
Editing Tool Icons	6-103
Design Cross-Platform UIs in GUIDE	6-107
Default System Font	6-107
Standard Background Color	6-108
Cross-Platform Compatible Units	6-108

Programming a GUIDE App

7

Write Callbacks in GUIDE	7-2
Callbacks for Different User Actions	7-2
GUIDE-Generated Callback Functions and Property Values	7-4
GUIDE Callback Syntax	7-5
Renaming and Removing GUIDE-Generated Callbacks	7-6
Initialize UI Components in GUIDE Apps	7-8
Opening Function	7-8
Output Function	7-10
Callbacks for Specific Components	7-12
How to Use the Example Code	7-12
Push Button	7-12
Toggle Button	7-13
Radio Button	7-13
Check Box	7-14
Edit Text Field	7-15
Slider	7-16
List Box	7-16
Pop-Up Menu	7-18
Panel	7-20
Button Group	7-21
Menu Item	7-22
Table	7-25
Axes	7-26
Examples of GUIDE Apps	7-29

Examples of GUIDE UIs

8

Modal Dialog Box in GUIDE	8-2
Create the Dialog Box	8-2
Create the Program That Opens the Dialog Box	8-3
Run the Program	8-5

GUIDE App With Parameters for Displaying Plots	8-7
Open and Run the Example	8-7
Examine the Code	8-8
GUIDE App Containing Tables and Plots	8-12
Open and Run the Example	8-12
Examine the Code	8-13
Interactive List Box App in GUIDE	8-16
Open and Run The Example	8-16
Examine the Layout and Callback Code	8-18
Plot Workspace Variables in a GUIDE App	8-21
Open and Run the App	8-21
Examine the Code	8-22
Automatically Refresh Plot in a GUIDE App	8-24
Open and Run the Example	8-24
Examine the Code	8-25

Create UIs Programmatically

Lay Out a Programmatic UI

9

Structure of Programmatic App Code Files	9-2
File Organization	9-2
File Template	9-2
Run the Program	9-3
Add Components to a Programmatic App	9-4
User Interface Controls	9-4
Tables	9-15
Panels	9-16
Button Groups	9-18
Axes	9-20
ActiveX Controls	9-22
How to Set Font Characteristics	9-22

Lay Out a UI Programmatically	9-25
Component Placement and Sizing	9-25
Managing the Layout in Resizable UIs	9-30
Manage the Stacking Order of Grouped Components	9-33
Customize Tabbing Behavior in a Programmatic App	9-34
How Tabbing Works	9-34
Default Tab Order	9-34
Change the Tab Order in the uipanel	9-36
Create Menus for Programmatic Apps	9-38
Add Menu Bar Menus	9-38
Add Context Menus to a Programmatic App	9-46
Create Toolbars for Programmatic Apps	9-51
Use the uitoolbar Function	9-51
Commonly Used Properties	9-51
Toolbars	9-52
Display and Modify the Standard Toolbar	9-55
DPI-Aware Behavior in MATLAB	9-58
Visual Appearance	9-58
Using Object Properties	9-60
Using print, getframe, and publish Functions	9-61

Code a Programmatic App

10

Initialize a Programmatic App	10-2
Examples	10-2
Write Callbacks for Apps Created Programmatically	10-5
Callbacks for Different User Actions	10-5
How to Specify Callback Property Values	10-7

Manage Application-Defined Data

11

Share Data Among Callbacks	11-2
Overview of Data Sharing Techniques	11-2
Store Data in UserData or Other Object Properties	11-3
Store Data as Application Data	11-8
Create Nested Callback Functions (Programmatic Apps)	11-12
Store Data Using the guidata Function	11-13
GUIDE Example: Share Slider Data Using guidata	11-16
GUIDE Example: Share Data Between Two Apps	11-16
GUIDE Example: Share Data Among Three Apps	11-17

Manage Callback Execution

12

Interrupt Callback Execution	12-2
How to Control Interruption	12-2
Callback Behavior When Interruption is Allowed	12-2
Example	12-3

Examples of Programmatic Apps

13

Programmatic App that Displays a Table	13-2
---	-------------

App Designer

App Designer Basics

14

Create and Run a Simple App Using App Designer	14-2
Run the Tutorial	14-2
Tutorial Steps for Creating the App	14-2
App Designer Versus GUIDE	14-6
Migrate GUIDE Apps to App Designer	14-6
Differences Between App Designer and GUIDE	14-6
Displaying Graphics in App Designer	14-9
Calling Graphics Functions	14-9
Displaying Plots Using Other Types of Axes	14-10
Unsupported Functionality	14-11
App Designer Preferences	14-14

Component Choices and Customizations

15

App Designer Components	15-2
Common Components	15-3
Containers and Figure Tools	15-6
Instrumentation	15-7
Create Menus for App Designer Apps	15-10
Create and Arrange Menus	15-10
Add Callbacks to Menu Items	15-12
Create Keyboard Shortcuts	15-13
Use Check Marks to Indicate Status	15-15
Table Array Data Types in App Designer Apps	15-17
Logical Data	15-17
Categorical Data	15-18

Datetime Data	15-18
Duration Data	15-19
Nonscalar Data	15-20
Missing Data Values	15-21
Example: App that Displays a Table Array	15-22

Add UI Components to App Designer

Programmatically	15-24
Create the Component and Assign the Callback	15-24
Write the Callback	15-25
Example: Confirmation Dialog Box with a Close Function	15-25
Example: App that Populates Tree Nodes Based on a Data File	15-26

App Layout

16

Lay Out Apps in App Designer	16-2
Customizing Components	16-3
Aligning and Spacing Components	16-4
Grouping Components	16-6
Arranging Components in Containers	16-7
Managing Resizable Apps in App Designer	16-9
Using Grid Layout Managers	16-12
Example: Hide Rows Based on Run-Time Conditions ..	16-16

App Programming

17

Managing Code in App Designer Code View	17-2
Managing Components, Functions, and Properties	17-2
Identifying Editable Sections of Code	17-3
Programming Your App	17-4
Fixing Coding Problems and Run-Time Errors	17-7

Startup Tasks and Input Arguments in App Designer	17-8
Create a StartupFcn Callback	17-8
Define Input App Arguments	17-9
Creating Multiwindow Apps in App Designer	17-12
Overview of the Process	17-12
Send Information to the Dialog Box	17-13
Return Information to the Main App	17-14
Manage Windows When They Close	17-15
Example: Plotting App That Opens a Dialog Box	17-16
Write Callbacks in App Designer	17-18
Create a Callback Function	17-18
Using Callback Function Input Arguments	17-20
Searching for Callbacks in Your Code	17-21
Deleting Callbacks	17-22
Example: App with a Slider Callback	17-22
Create Helper Functions in App Designer	17-24
Create a Helper Function	17-24
Managing Helper Functions	17-25
Example: Helper Function that Initializes and Updates Two Plots	17-26
Share Data Within App Designer Apps	17-28
Example: Share Plot Data and a Drop-Down List Selection	17-30
Compatibility Between Different Releases of App Designer	17-32
Save Copy As Versus Save As	17-33
Use One Callback for Multiple App Designer Components	17-35
Example of a Shared Callback	17-35
Change or Disconnect a Callback	17-37

App Designer Examples

18

Mortgage Calculator App in App Designer	18-2
Data Analysis App in App Designer	18-4
App with Instrumentation Controls in App Designer ...	18-6
App That Displays a Hierarchical Tree in App Designer	18-8
Image Histogram App in App Designer	18-10
Polar Plotting App in App Designer	18-12
App with an Interactive Table in App Designer	18-14
Memory Monitor That Uses Timer Object in App Designer	18-16
Wind Speed App That Uses Timer Object in App Designer	18-18
Multiwindow App in App Designer	18-20

Keyboard Shortcuts

19

App Designer Keyboard Shortcuts	19-2
Shortcuts Available Throughout App Designer	19-2
Component Browser Shortcuts	19-2
Design View Shortcuts	19-3
Code View Shortcuts	19-8

Apps Overview	20-2
What Is an App?	20-2
Where to Get Apps	20-2
Why Create an App?	20-3
Best Practices and Requirements for Creating an App	20-4
Package Apps From the MATLAB Toolstrip	20-5
Package Apps in App Designer	20-8
Modify Apps	20-11
Ways to Share Apps	20-13
Share MATLAB Files Directly	20-13
Package Your App	20-14
Create a Deployed Web App	20-15
Create a Standalone Desktop Application	20-16
MATLAB App Installer File — mlappinstall	20-17
Dependency Analysis	20-18

Introduction to Creating UIs

About Apps in MATLAB Software

Ways to Build Apps

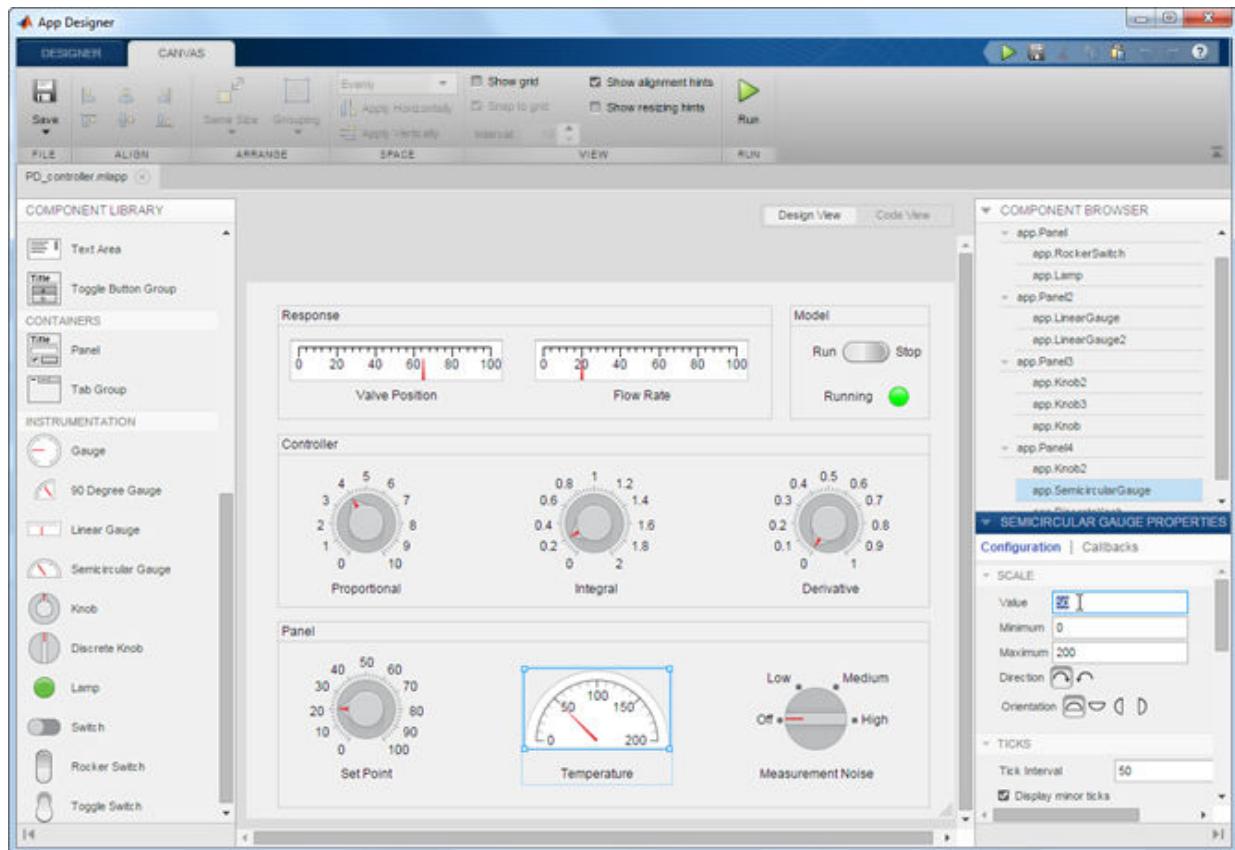
There are different ways to build MATLAB apps:

- “Use App Designer” on page 1-2
- “Use GUIDE” on page 1-3
- “Use MATLAB Functions to Create Apps Programmatically” on page 1-4

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.

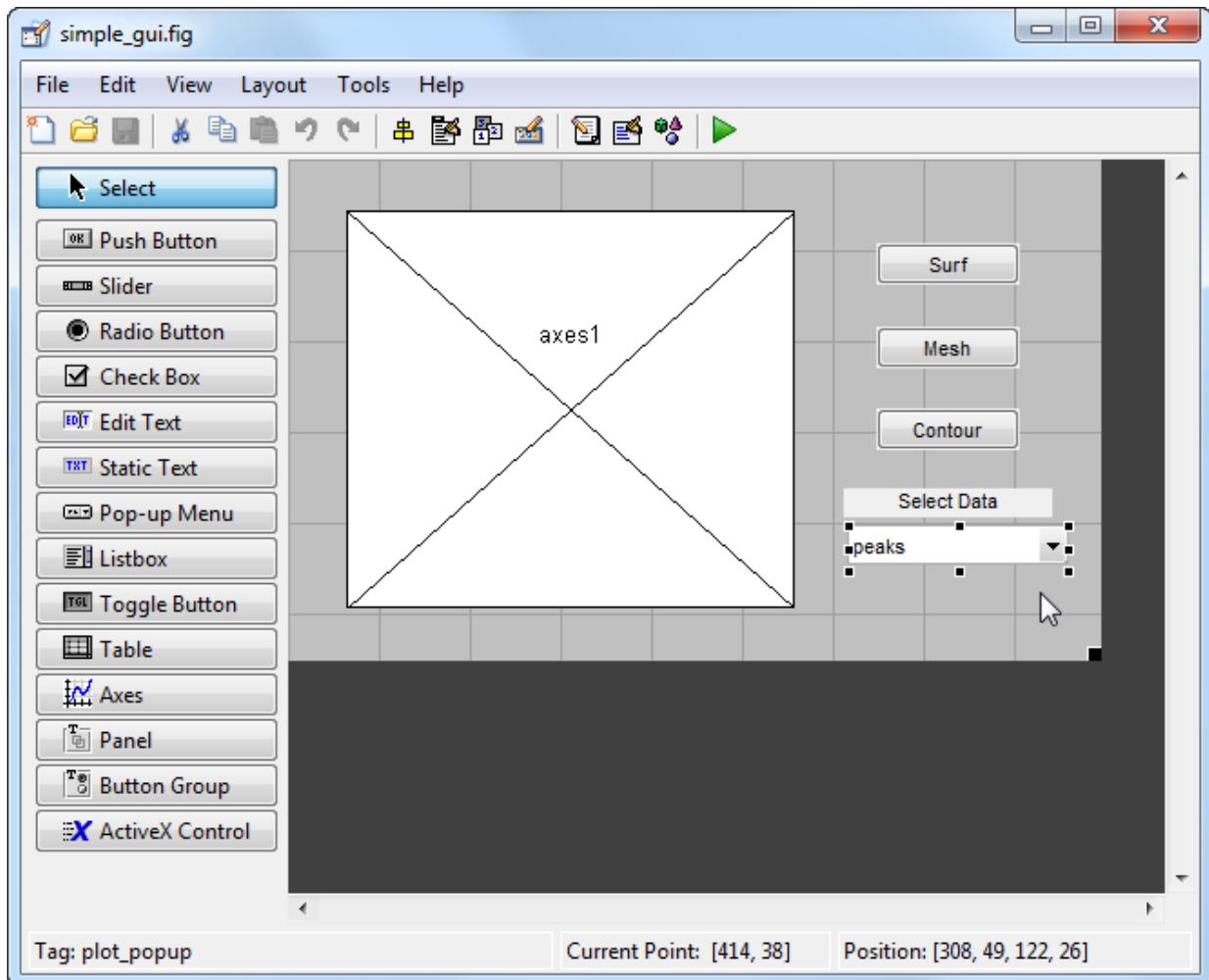
Use App Designer

App Designer is a rich drag-and-drop environment introduced in R2016a, and it is the recommended environment for building most apps. 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 is available, including gauges, lamps, knobs, and switches. Most graphics functionality is supported.



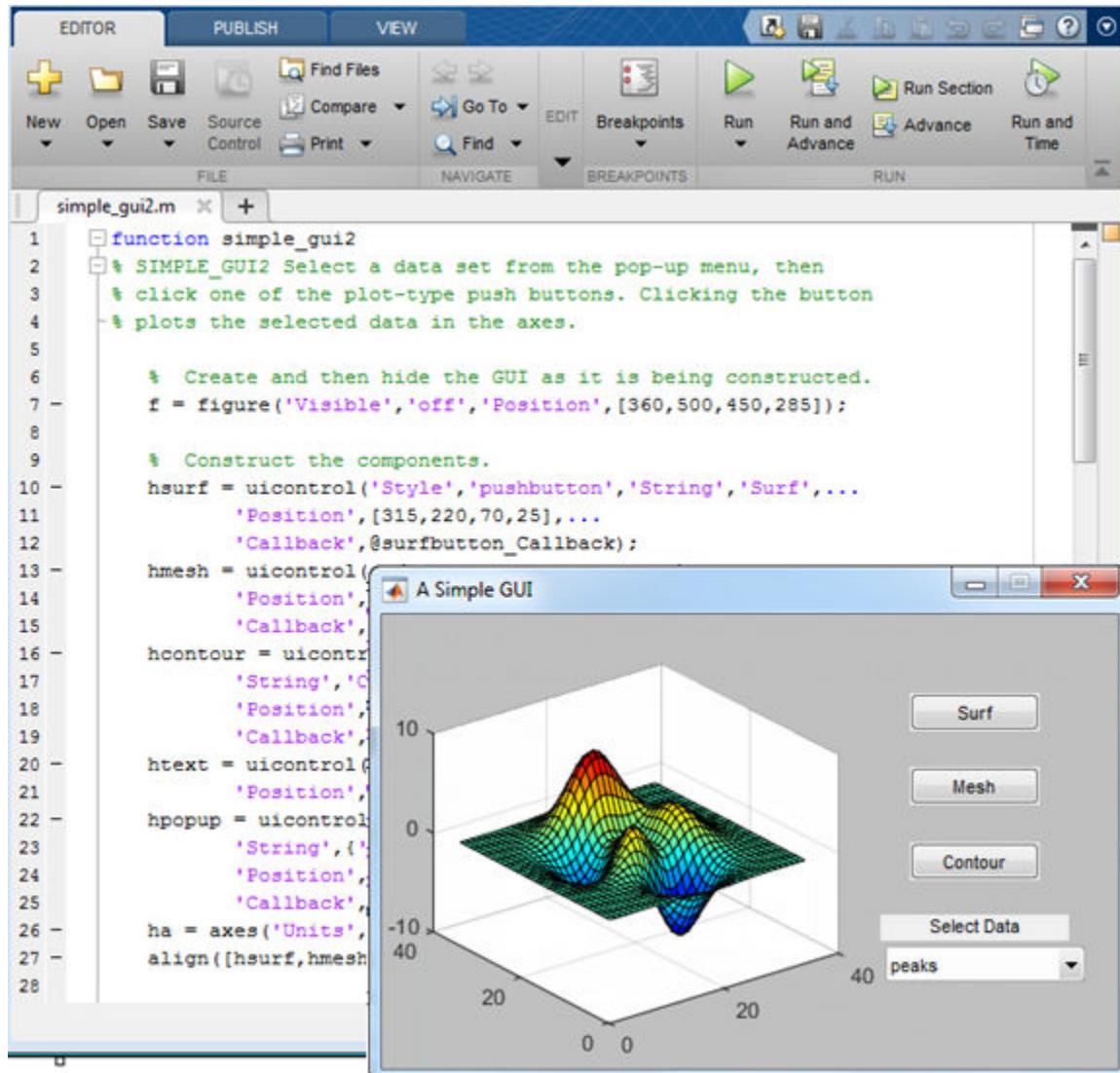
Use GUIDE

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 created with GUIDE are compatible with almost all other releases, and they support all the graphics functionality in MATLAB.



Use MATLAB Functions to Create Apps Programmatically

You can also 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.



See Also

Related Examples

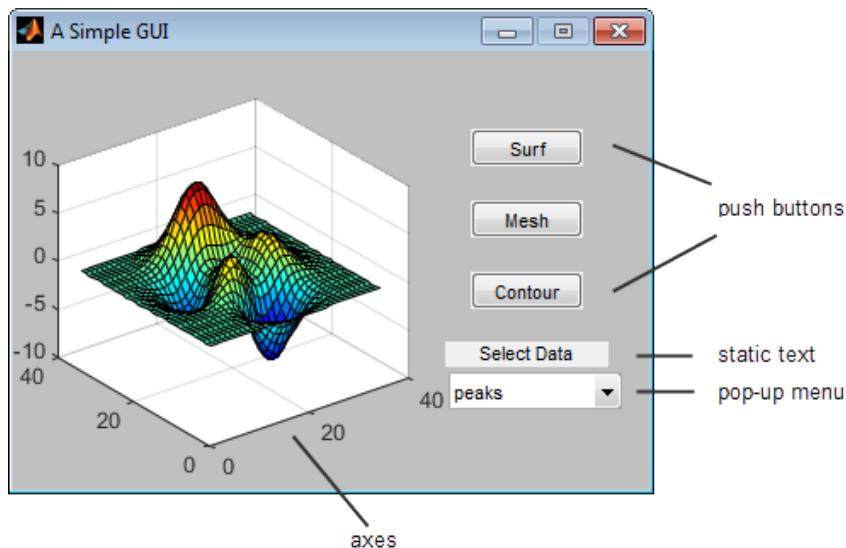
- “App Designer Versus GUIDE” on page 14-6
- “Create a Simple App Using GUIDE” on page 2-2
- “Create a Simple App Programmatically” on page 3-2
- “Create and Run a Simple App Using App Designer” on page 14-2
- “Displaying Graphics in App Designer” on page 14-9

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.

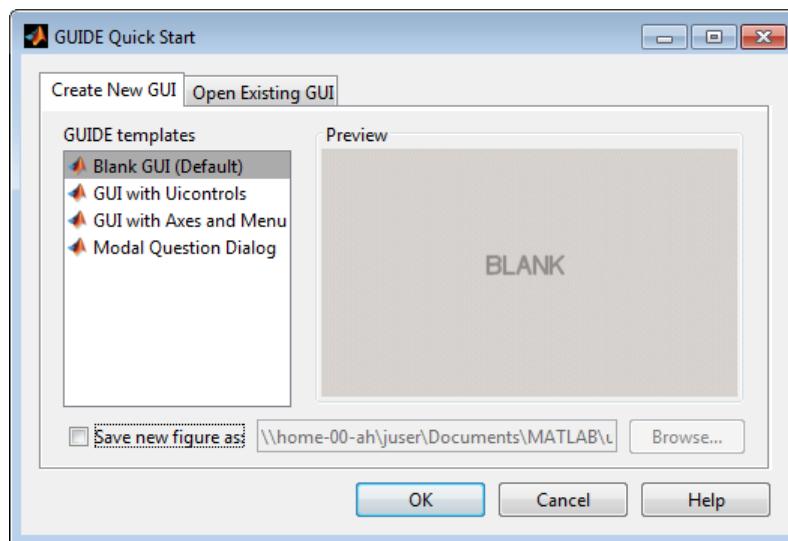
If you only want to view and run the code that created this app, set your current folder to one to which you have write access. Copy the example code and open it in the Editor by issuing the following MATLAB commands:

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

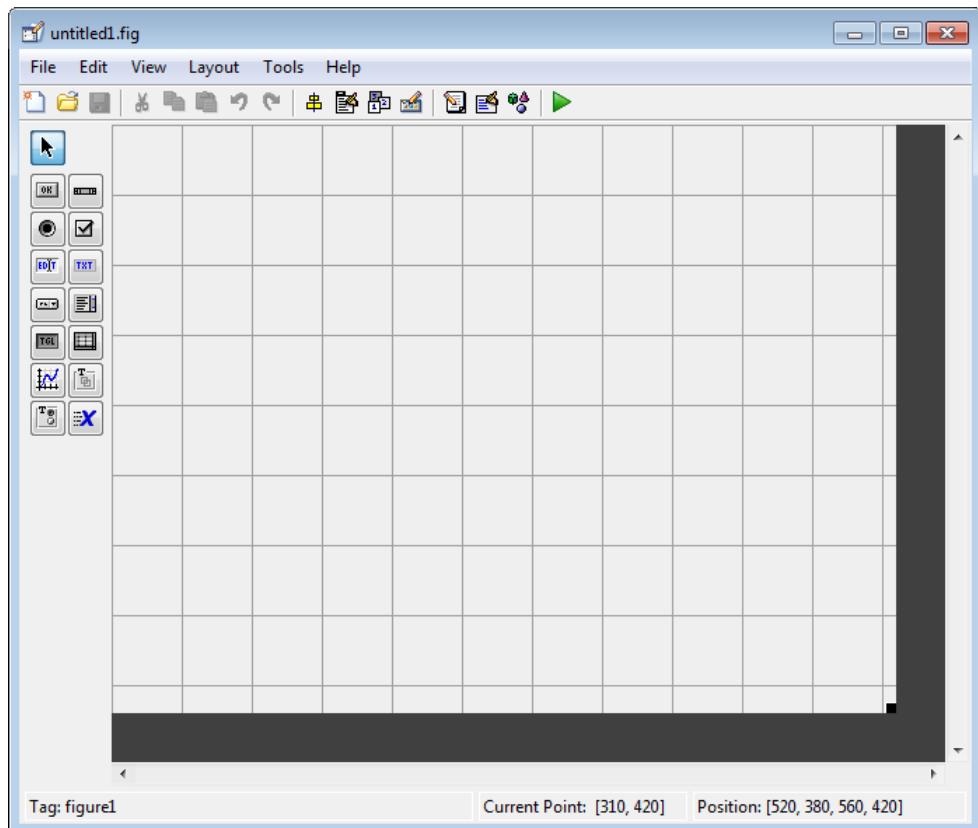
Click the **Run** ➤ button to run the app.

Open a New UI in the GUIDE Layout Editor

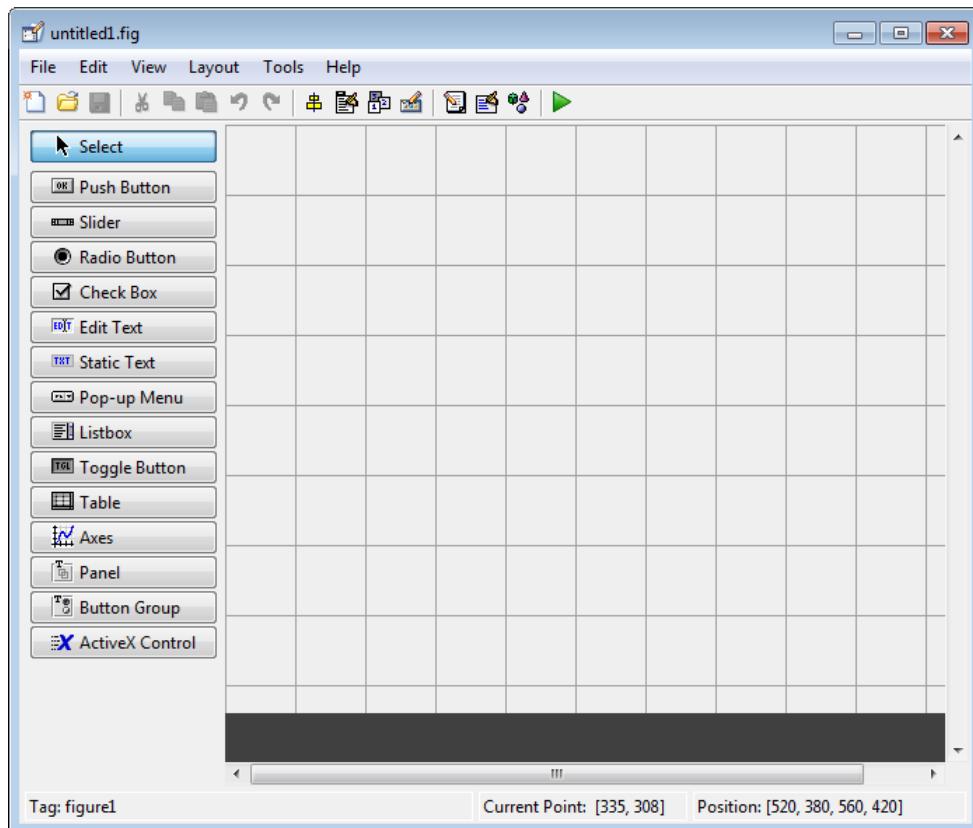
- 1 Start GUIDE by typing guide at the MATLAB prompt.



- 2 In the GUIDE Quick Start dialog box, select the **Blank GUI (Default)** template, and then click **OK**.

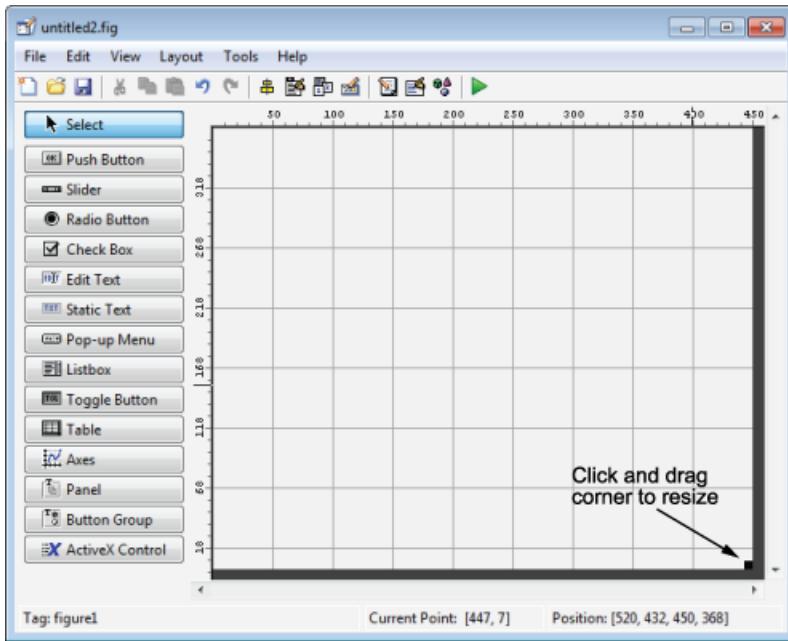


- 3 Display the names of the components in the component palette:
 - a Select **File > Preferences > GUIDE**.
 - b Select **Show names in component palette**.
 - c Click **OK**.



Set the Window Size in GUIDE

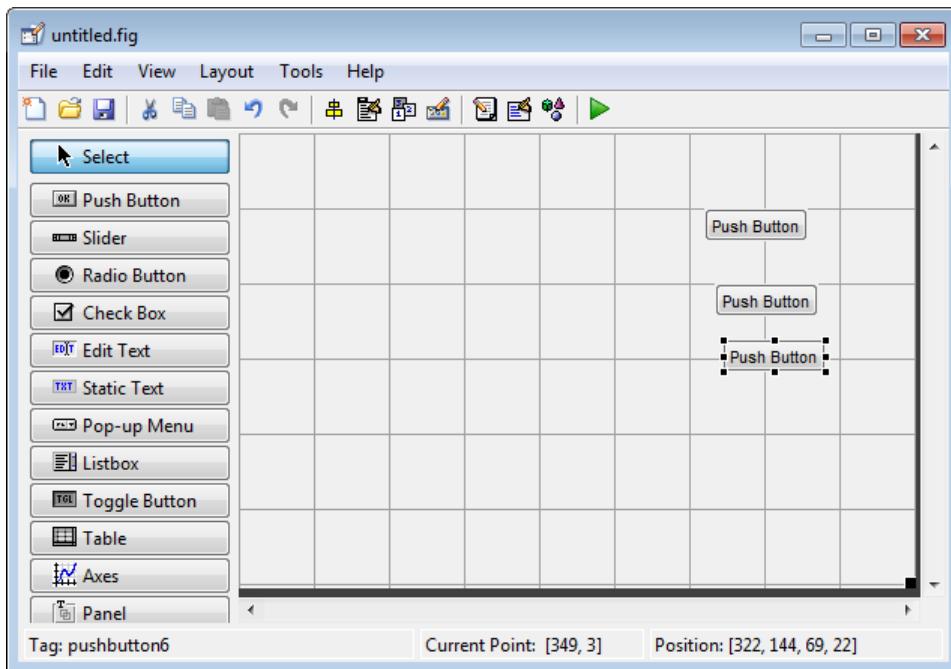
Set the size of the window by resizing the grid area in the Layout Editor. Click the lower-right corner and drag it until the canvas is approximately 3 inches high and 4 inches wide. If necessary, make the canvas larger.



Layout the UI

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

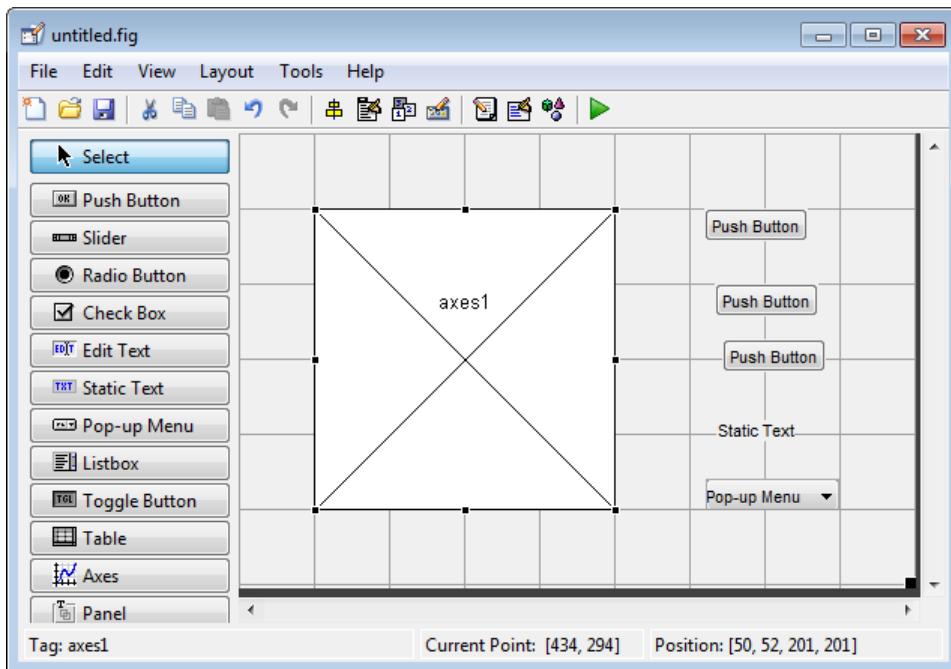
- 1 Add the three push buttons to the UI. Select the push button tool from the component palette at the left side of the Layout Editor and drag it into the layout area. Create three buttons, positioning them approximately as shown in the following figure.



2 Add the remaining components to the UI.

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

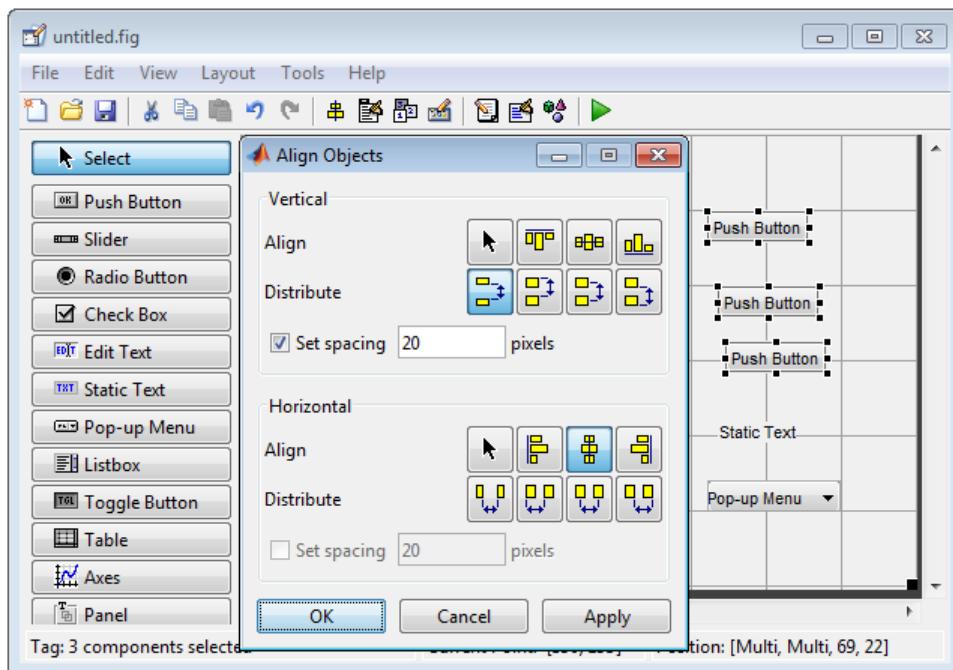
Arrange the components as shown in the following figure. Resize the axes component to approximately 2-by-2 inches.



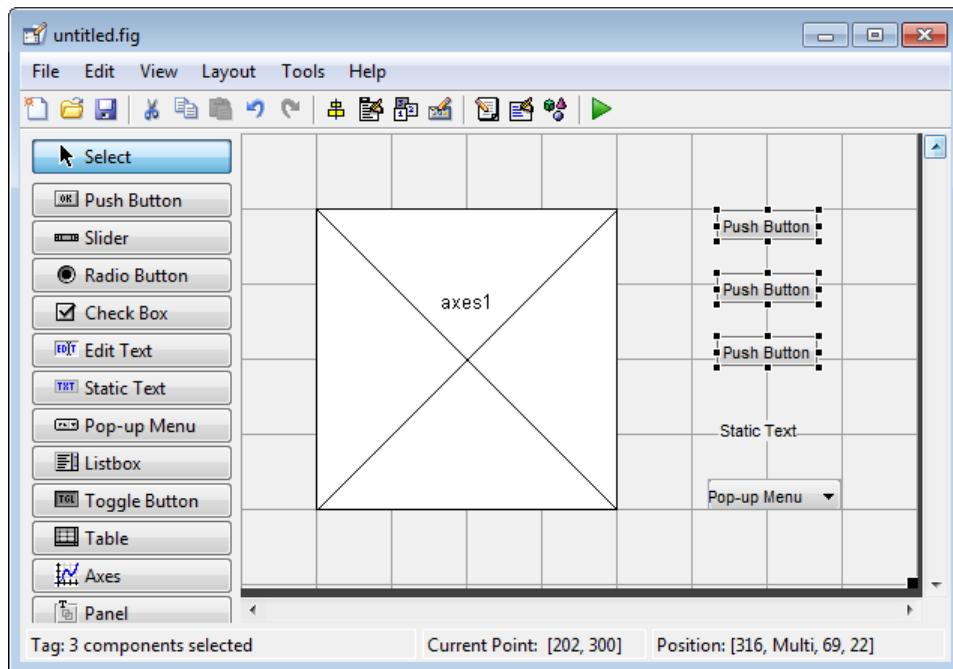
Align the Components

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.



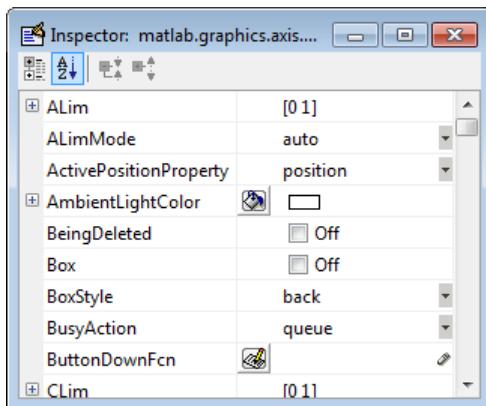
4 Click **OK**.



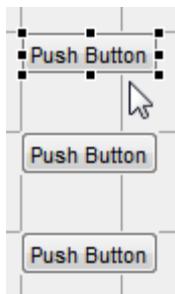
Label the Push Buttons

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 Press the **Enter** key. The push button label changes to **Surf**.

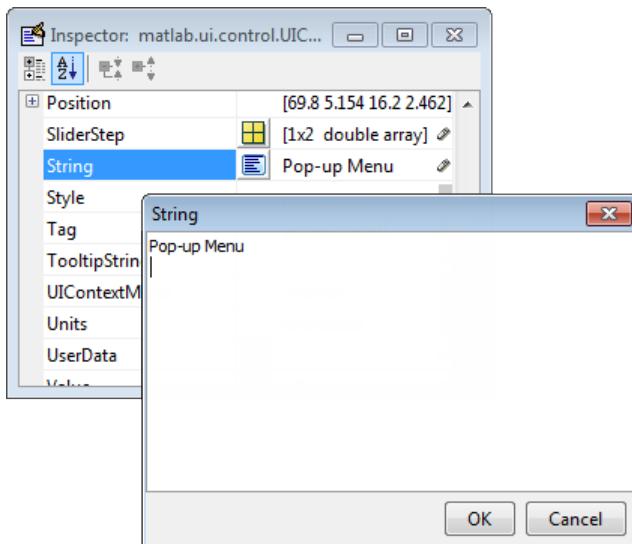


- 5 Click each of the remaining push buttons in turn and repeat steps 3 and 4. Label the middle push button **Mesh**, and the bottom button **Contour**.

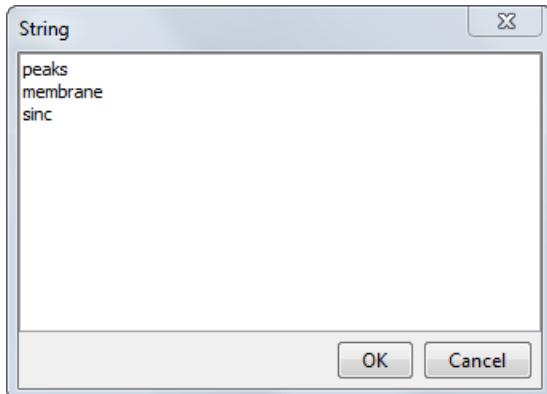
List Pop-Up Menu Items

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.

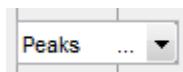


- 3 Replace the existing text with the names of the three data sets: peaks, membrane, and sinc. Press **Enter** to move to the next line.



- 4 When you finish editing the items, click **OK**.

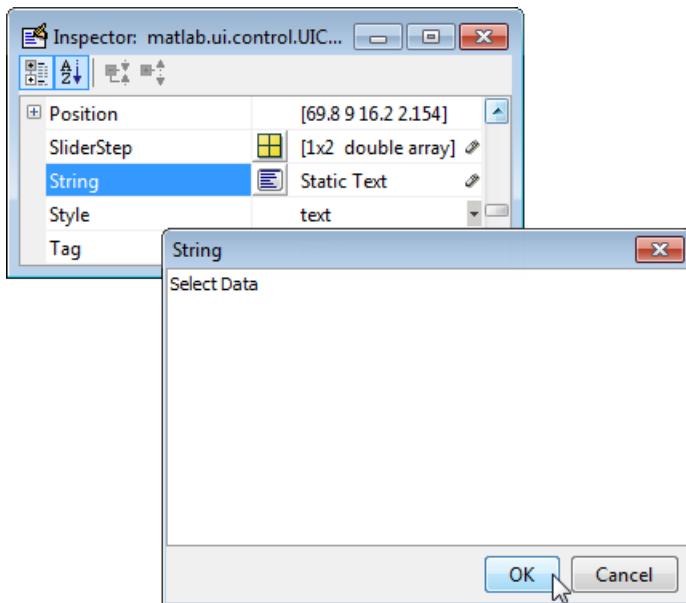
The first item in your list, **peaks**, appears in the pop-up menu in the layout area.



Modify the Static Text

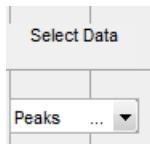
In this UI, the static text serves as a label for the pop-up menu. This section shows you how to change the static text to read **Select Data**.

- 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.

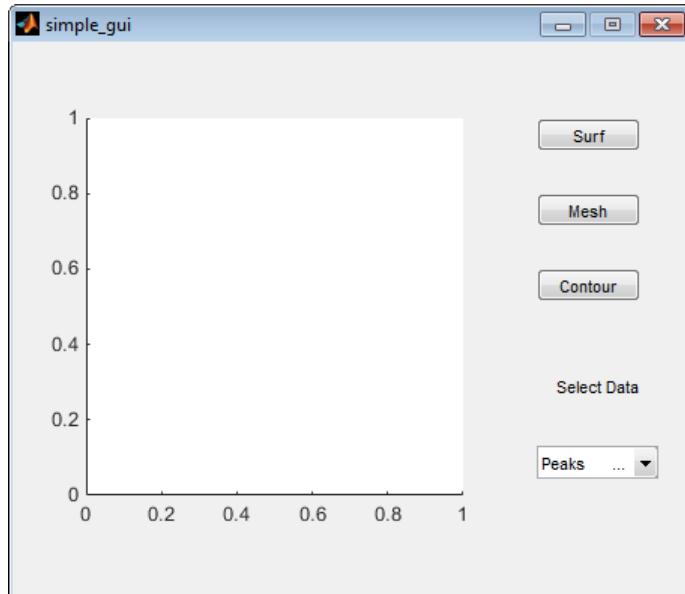
- 1 Save and run your program by selecting **Tools > Run**.
- 2 GUIDE displays a dialog box displaying: "Activating will save changes to your figure file and MATLAB code. Do you wish to continue?"

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)
```

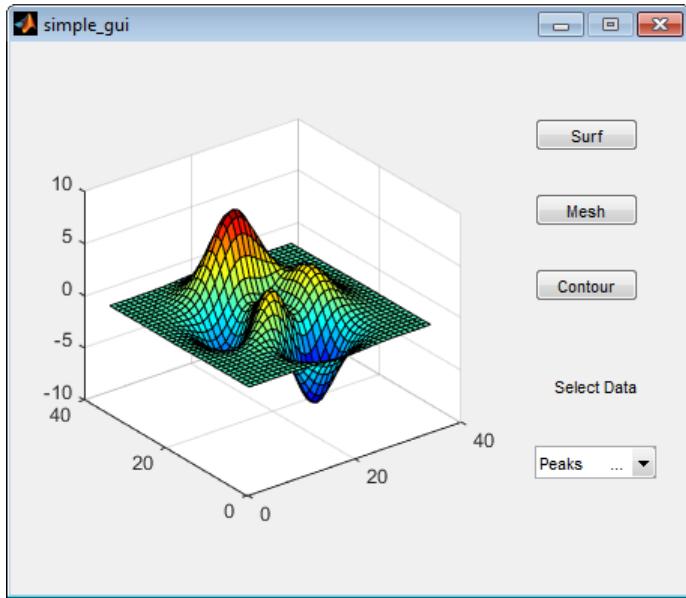
```
% This function has no output args, see OutputFcn.  
% hObject    handle to figure  
% eventdata   reserved - to be defined in a future version of MATLAB  
% handles     structure with handles and user data (see GUIDATA)  
% varargin    command line arguments to simple_g(see VARARGIN)  
  
% Choose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

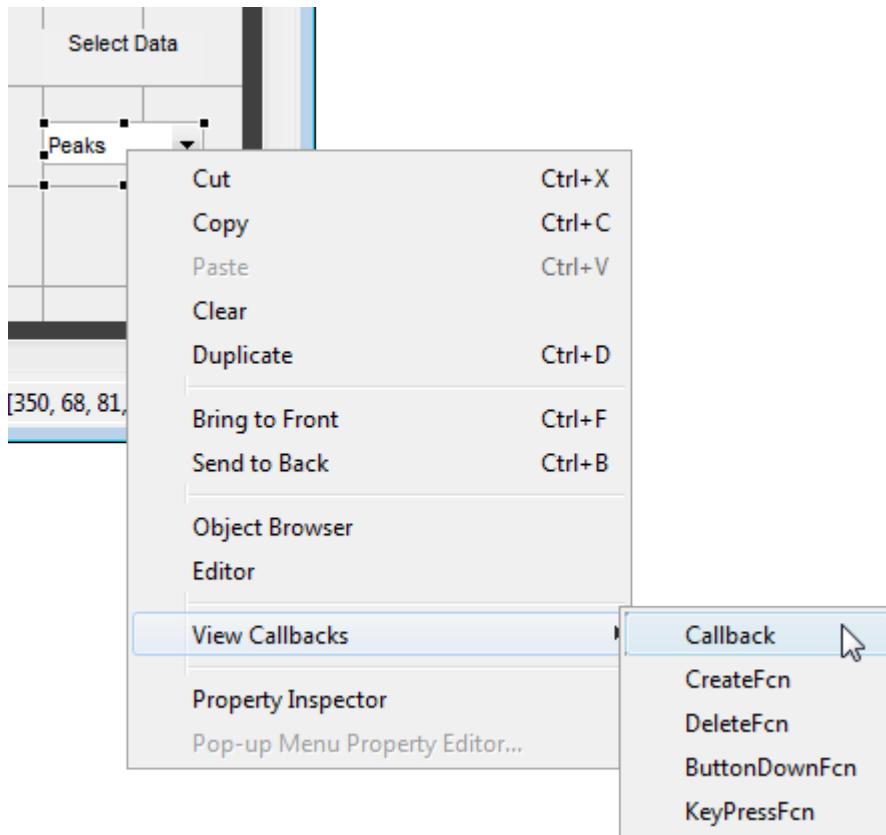
The last two lines create a current data value and set it to `peaks`, and then display the `surf` plot for `peaks`. The following figure shows how the app looks when it first displays.



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 menu item. 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

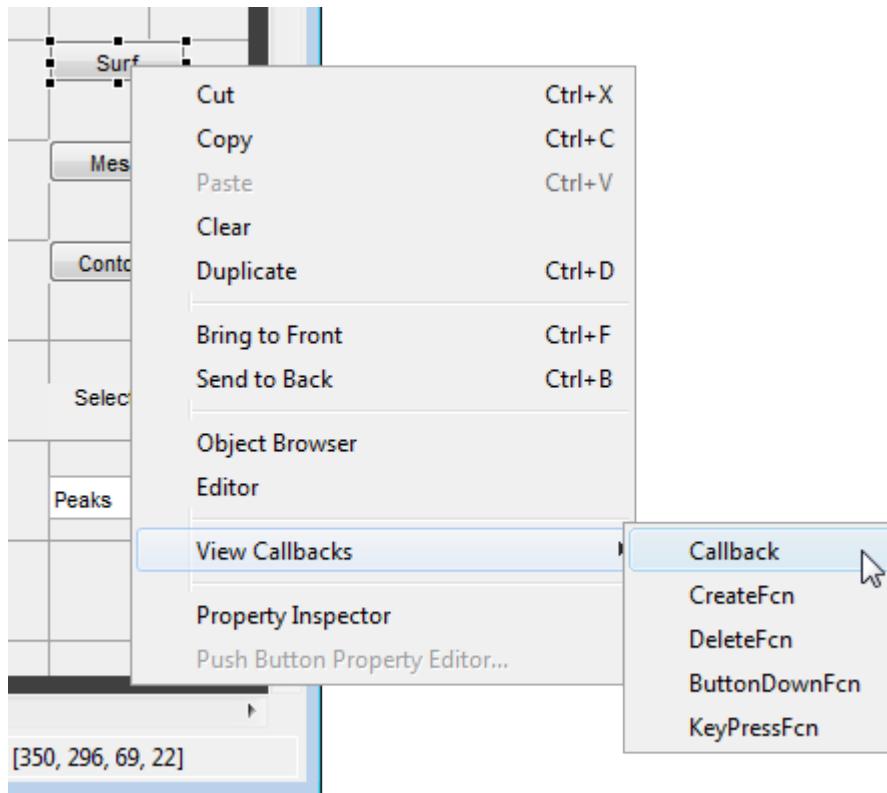
The code then uses a **switch** statement to make the selected data set the current data. The last statement saves the changes to the **handles** structure.

```
% 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`:

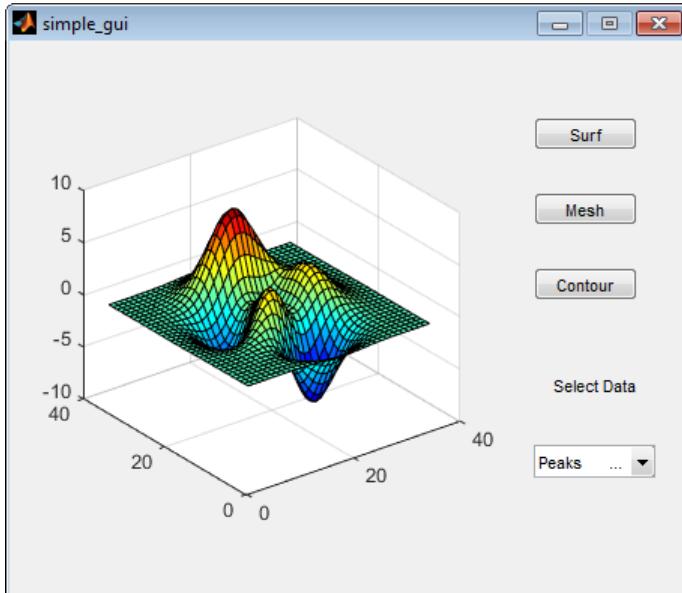
```
% Display contour plot of the currently selected data.  
contour(handles.current_data);
```

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 app displays a mesh plot of the MathWorks® L-shaped Membrane logo.
- 3 Try other combinations before closing the window.

See Also

Related Examples

- “Ways to Build Apps” on page 1-2
- “Create a Simple App Programmatically” on page 3-2
- “Create and Run a Simple App Using App Designer” on page 14-2

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 app, 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 app for 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 app.

When you lay out the app in the Layout Editor, your components and layout is stored in the FIG-file. When you program the app, your code is stored in the corresponding code file.

If your app 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 app 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 <code>help</code> command.
Initialization	GUIDE initialization tasks. <i>Do not edit this code.</i>
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 app, 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  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 on page 7-5 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.

See Also

Related Examples

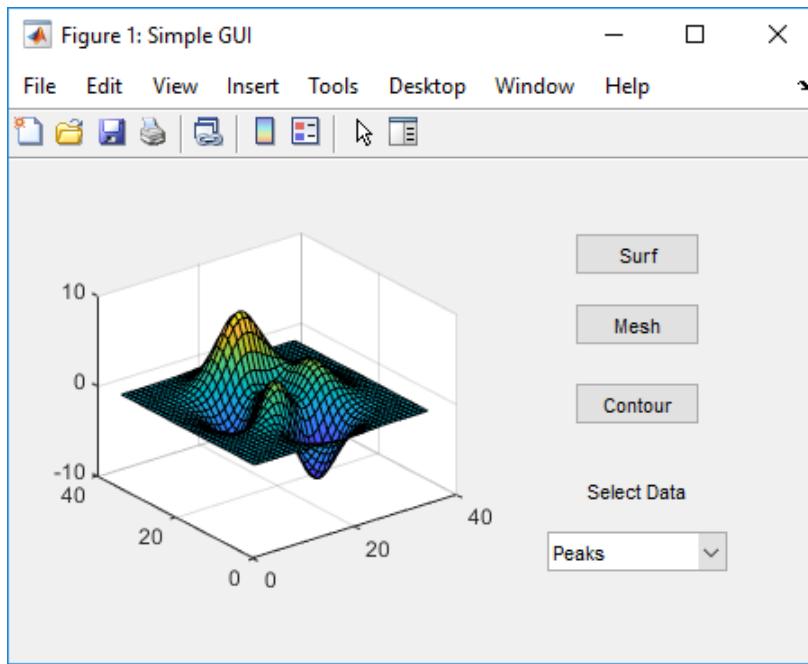
- “Create a Simple App Using GUIDE” on page 2-2
- “Write Callbacks in GUIDE” on page 7-2

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 is set to 'off' to make the window invisible as components are added or initialized. The window becomes visible when the UI has all its components and is initialized.
- The **Position** property is set to 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 a text label by adding these statements to the code file following the push button definitions. The first statement creates the label. The second statement creates the **popup menu**.

```
htext  = uicontrol('Style','text','String','Select Data',...
                   'Position',[325,90,60,15]);  
hpopup = uicontrol('Style','popupmenu',...
                   'Position',[325,115,60,15]);
```

```
'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]);

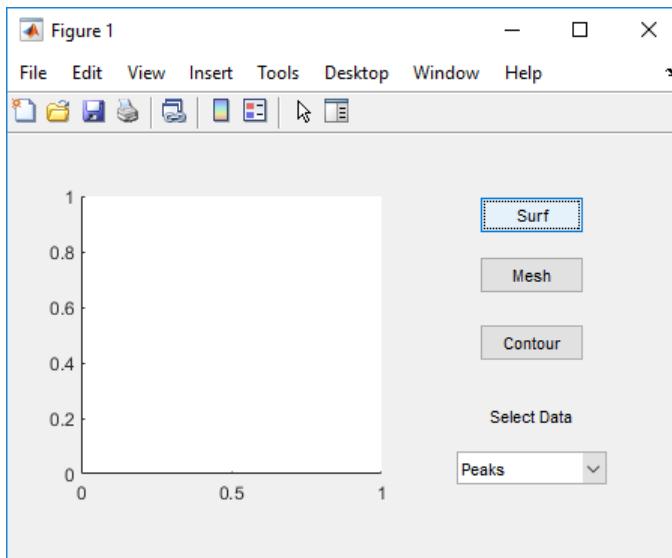
% Construct the components.
hsurf = uicontrol('Style','pushbutton','String','Surf',...
    'String',{ 'Peaks','Membrane','Sinc'},...
    'Position',[300,50,100,25]);
```

```
'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 menu item. 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
% 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.

```
% 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:

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

with this code:

```
% 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';
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

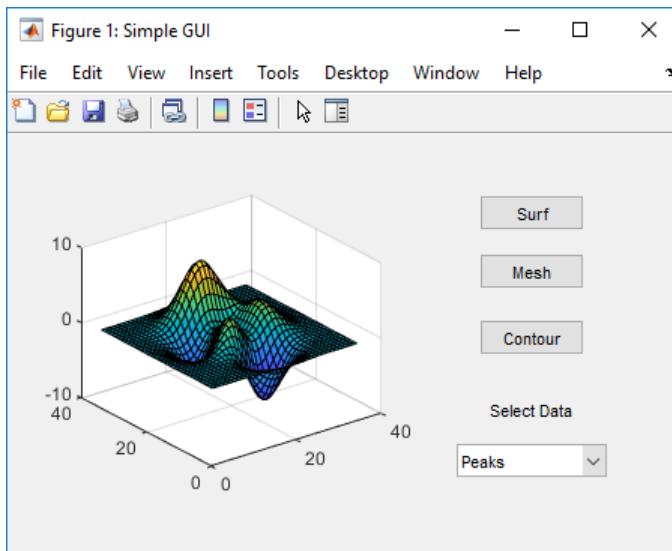
% Push button callbacks. Each callback plots current_data in the
% specified plot type.

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
```

- 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.
```

See Also

Related Examples

- “Ways to Build Apps” on page 1-2
- “Create a Simple App Using GUIDE” on page 2-2
- “Create and Run a Simple App Using App Designer” on page 14-2

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 Preferences and Options

- “GUIDE Preferences” on page 5-2
- “GUIDE Options” on page 5-8

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.

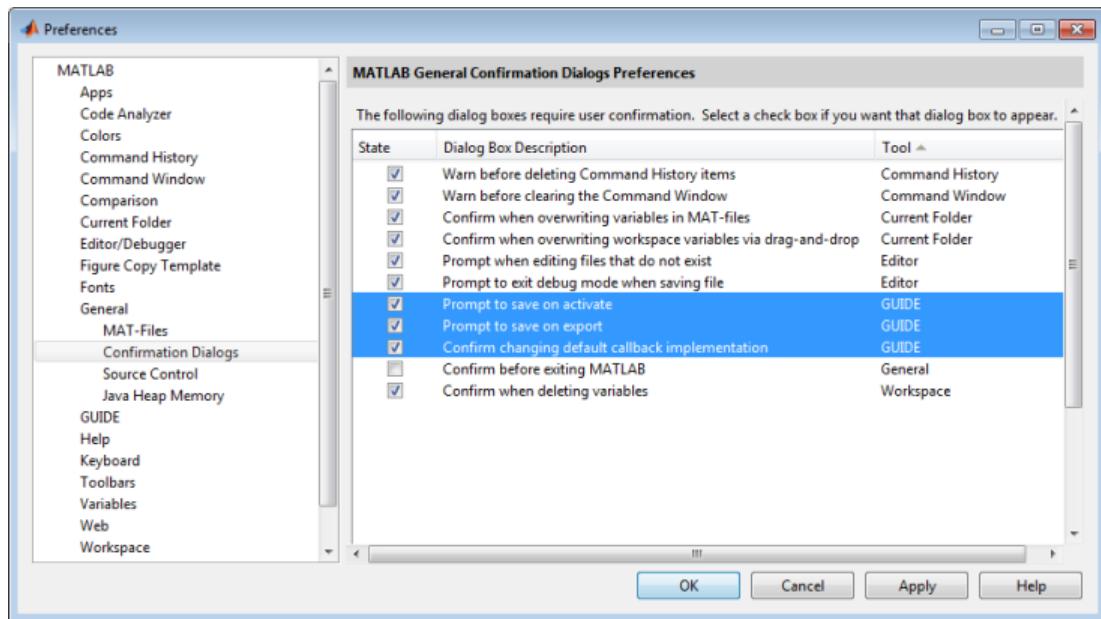
The preferences are in different locations within the Preferences dialog box:

Confirmation Preferences

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

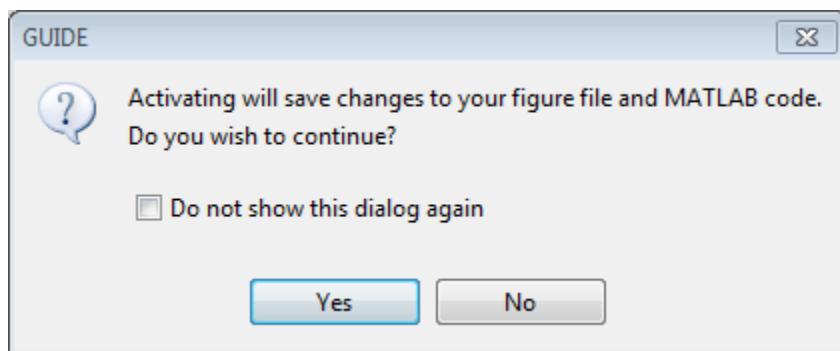
- Activate a UI from GUIDE.
- Export a UI from GUIDE.
- Change a callback signature generated by GUIDE.

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.



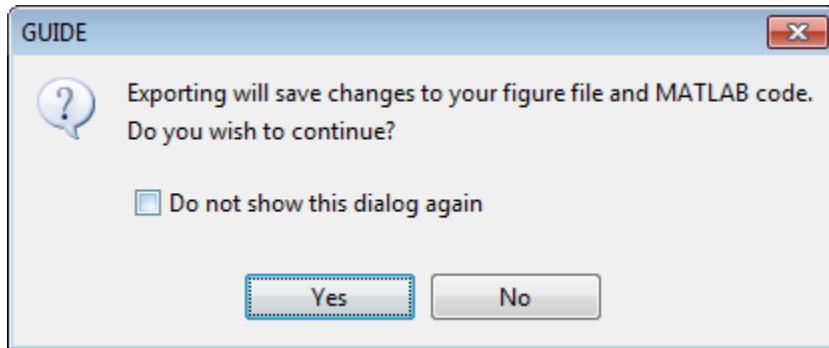
Prompt to Save on Activate

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

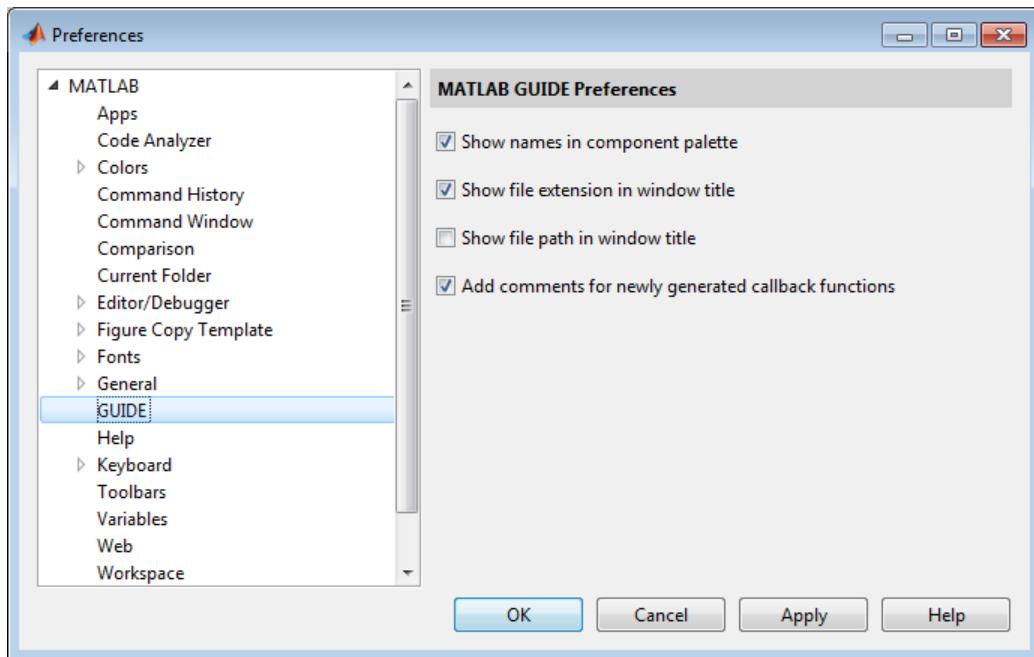
UI FIG-files created or modified with MATLAB 7.0 or a later version are not automatically compatible with Version 6.5 and earlier versions. GUIDE automatically generates FIG-files, which are binary files that contain the UI layout information.

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.

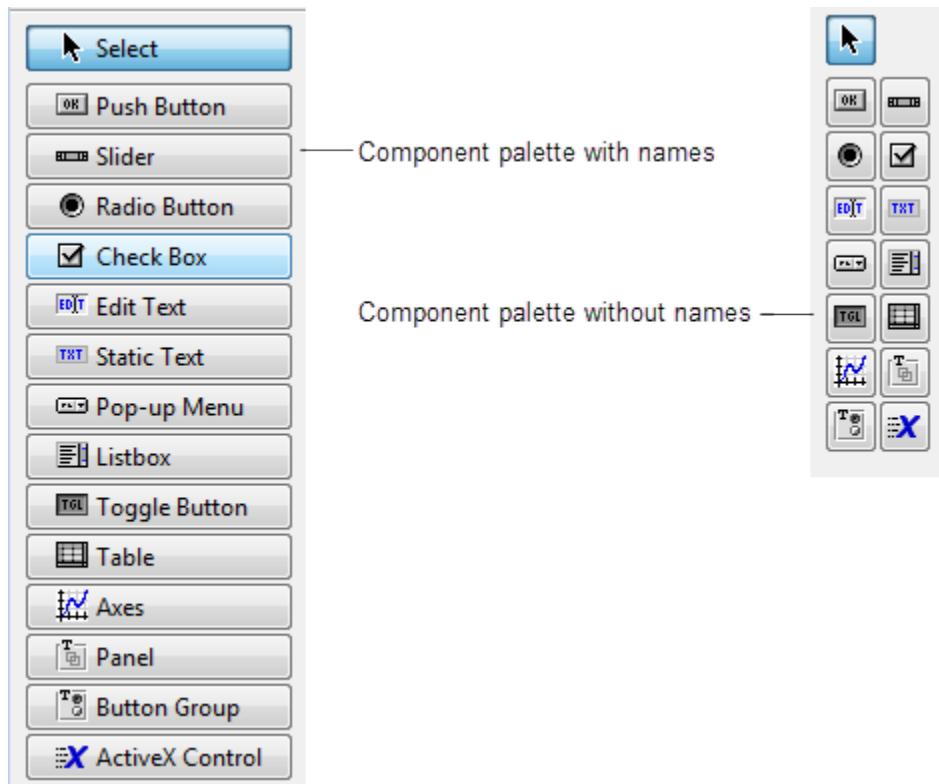


The following topics describe the preferences in this dialog:

- “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.

See Also

Related Examples

- “GUIDE Options” on page 5-8

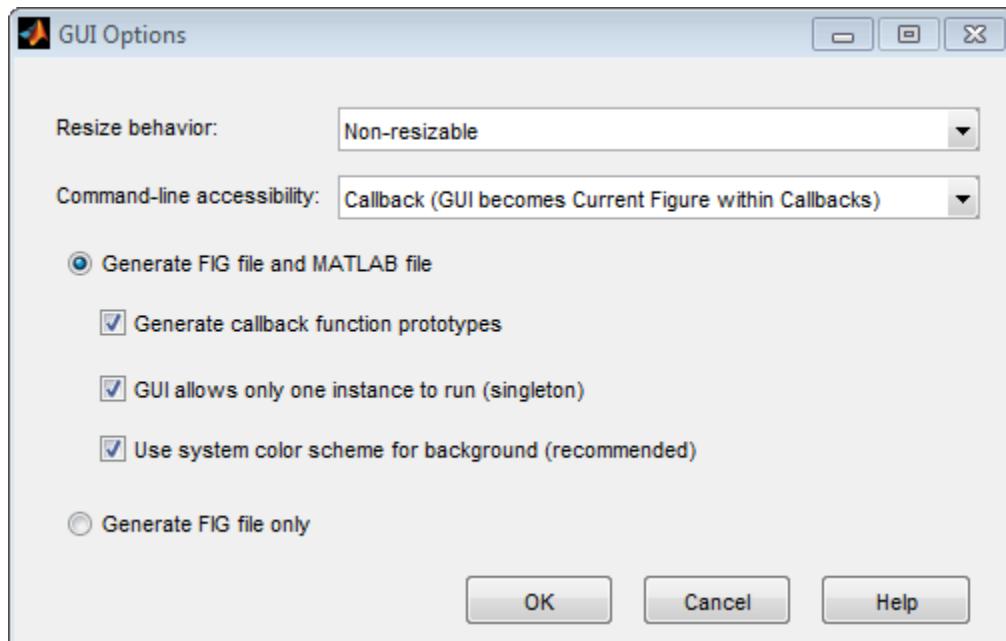
GUIDE Options

In this section...

- “The GUI Options Dialog Box” on page 5-8
- “Resize Behavior” on page 5-9
- “Command-Line Accessibility” on page 5-9
- “Generate FIG-File and MATLAB File” on page 5-10
- “Generate FIG-File Only” on page 5-12

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 cannot 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 <code>HandleVisibility</code> and <code>IntegerHandle</code> 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 on page 6-78.

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 character vectors, 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

See Also

Related Examples

- “GUIDE Preferences” on page 5-2

Lay Out a UI Using GUIDE

- “GUIDE Templates” on page 6-2
- “Set the UI Window Size in GUIDE” on page 6-11
- “Add Components to the GUIDE Layout Area” on page 6-13
- “Align GUIDE UI Components” on page 6-66
- “Customize Tabbing Behavior in a GUIDE UI” on page 6-75
- “Create Menus for GUIDE Apps” on page 6-78
- “Create Toolbars for GUIDE UIs” on page 6-95
- “Design Cross-Platform UIs in GUIDE” on page 6-107

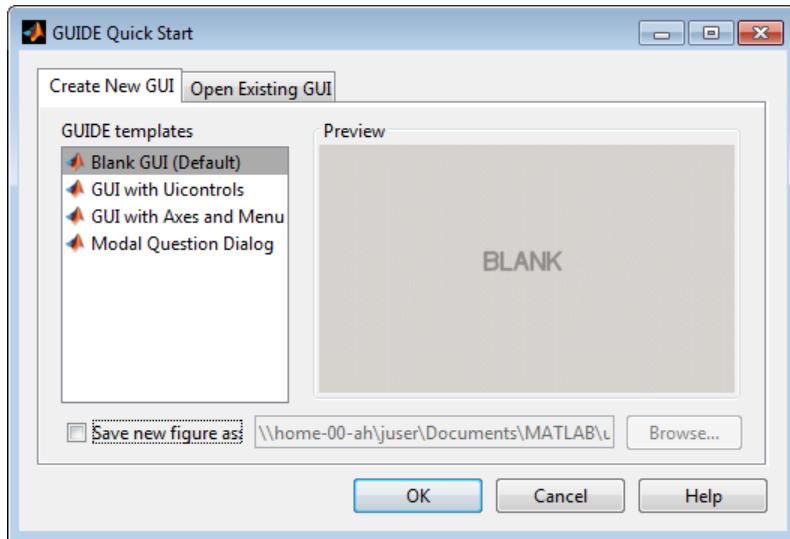
GUIDE Templates

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 apps. To access the templates in GUIDE, select **File > New**. 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:

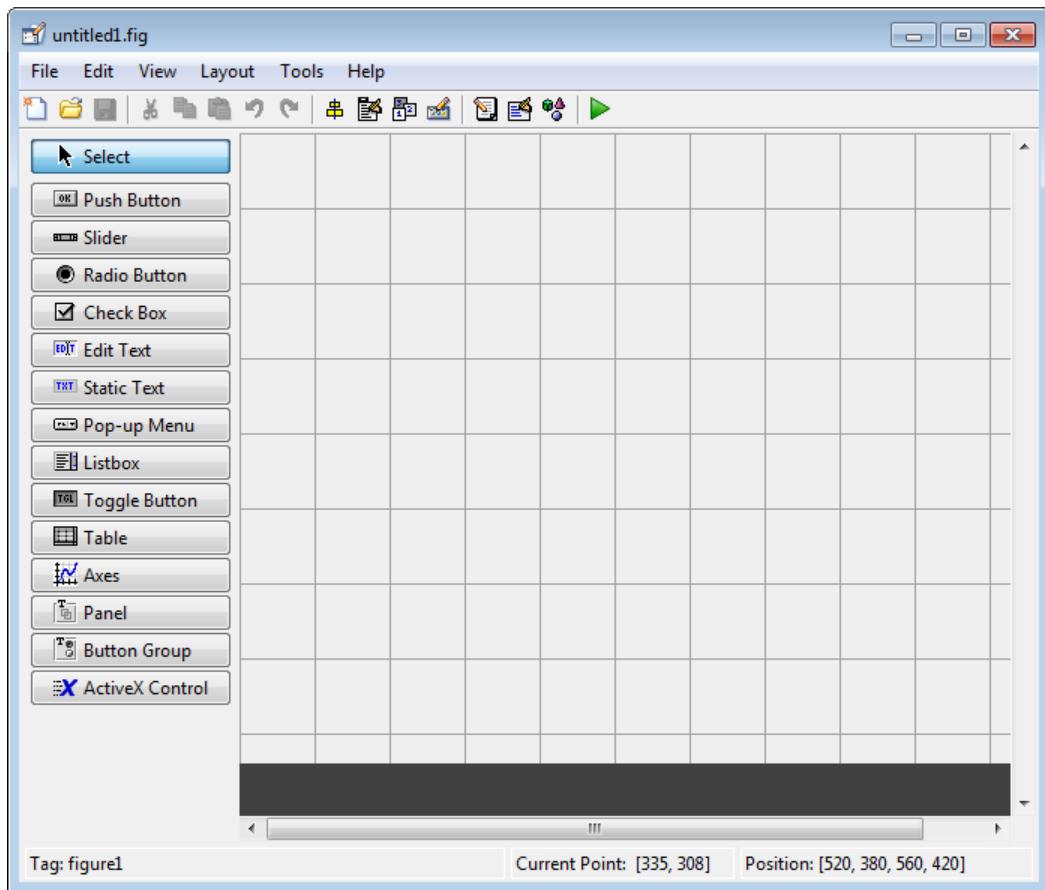
- “Blank GUI” on page 6-3
- “GUI with Uicontrols” on page 6-4
- “GUI with Axes and Menu” on page 6-6
- “Modal Question Dialog” on page 6-8

“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 Apps” on page 6-78 and “Create Toolbars for GUIDE UIs” on page 6-95 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

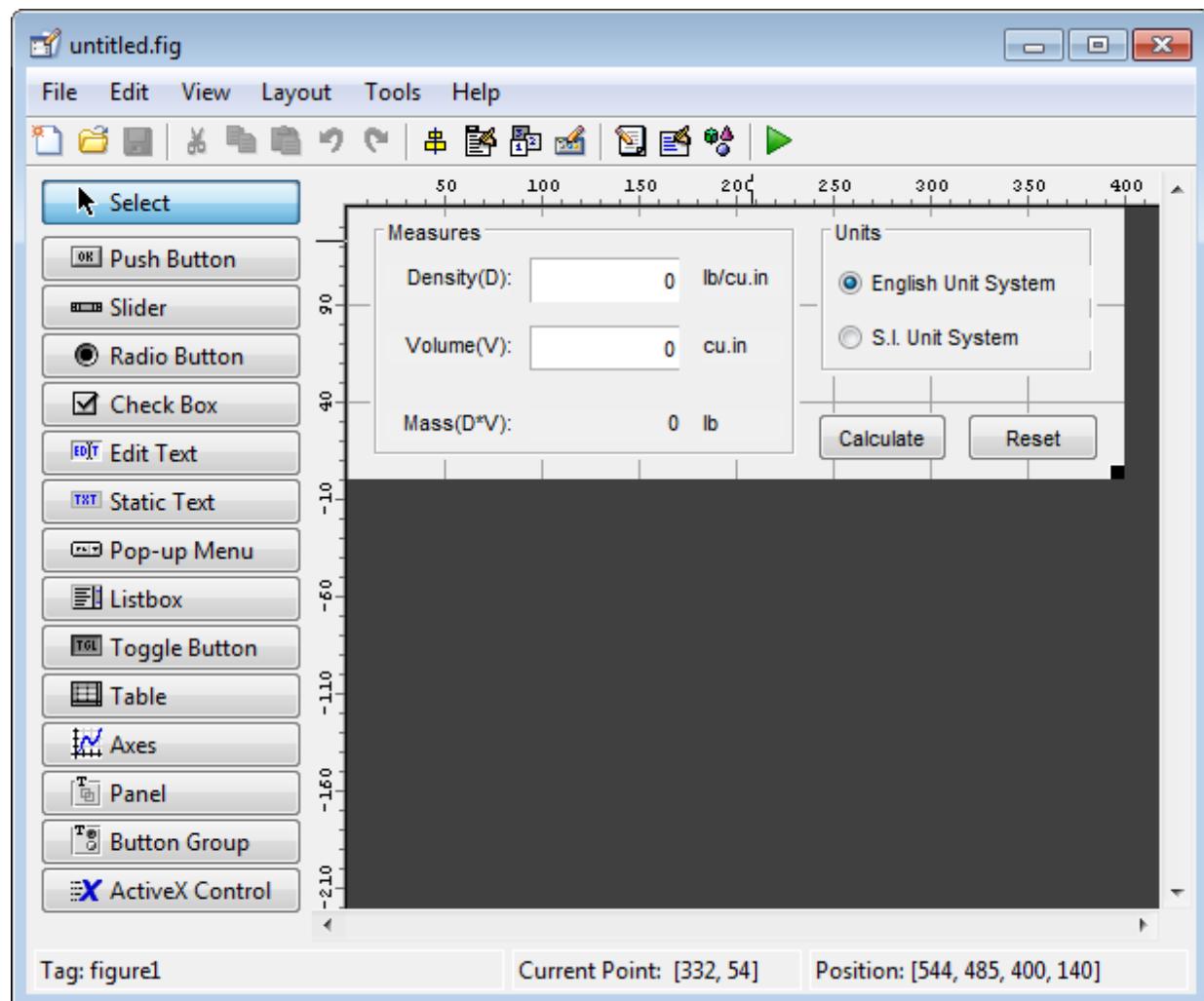
The following figure shows an example of this template.



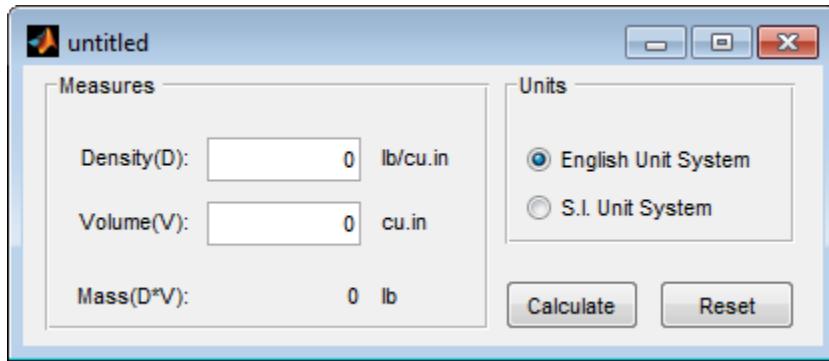
Select this template when the other templates are not suitable for the UI you want to create.

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.

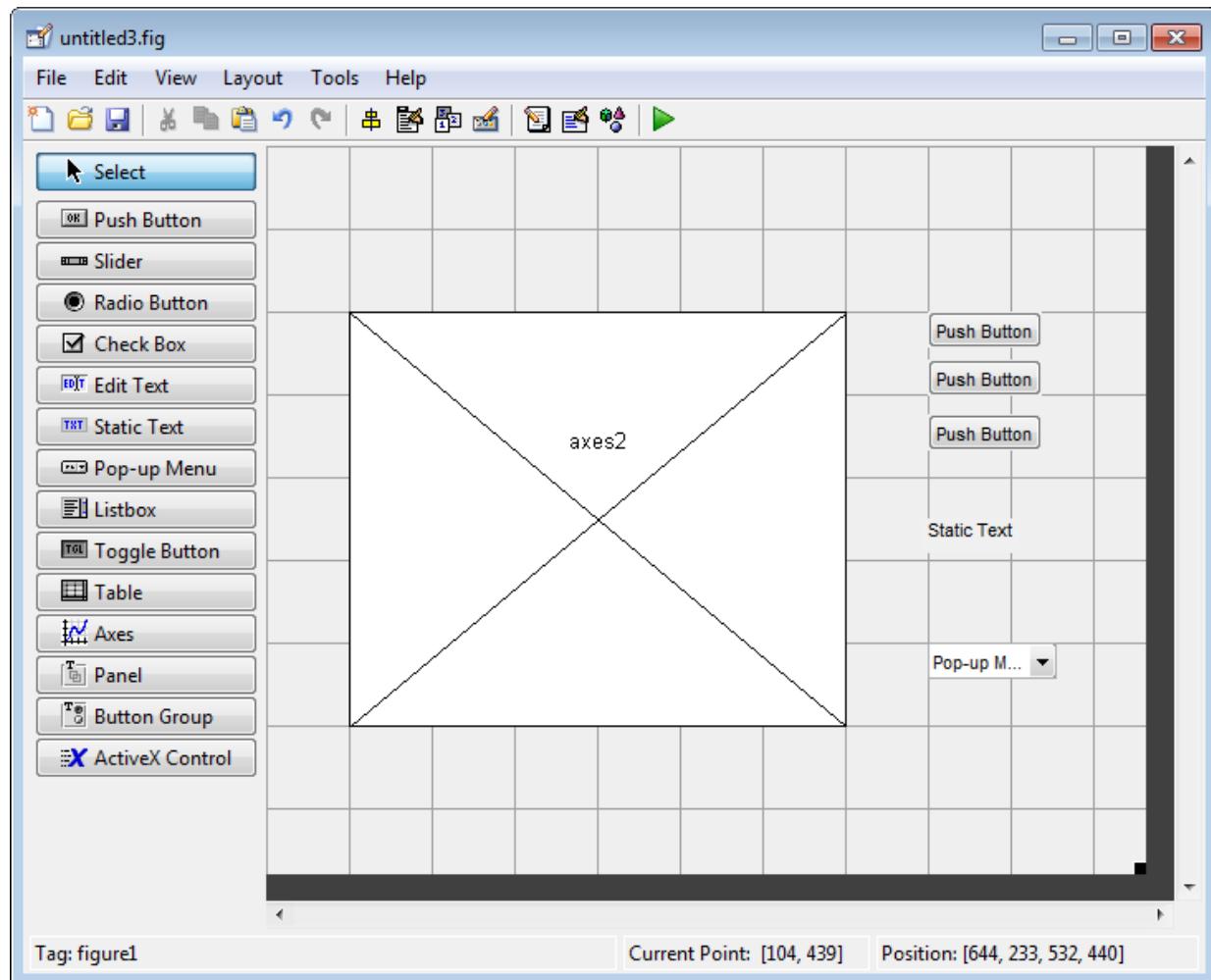


When you enter values for the density and volume of an object, and click the **Calculate** button, the program calculates the mass of the object and displays the result next to **Mass(D*V)**.

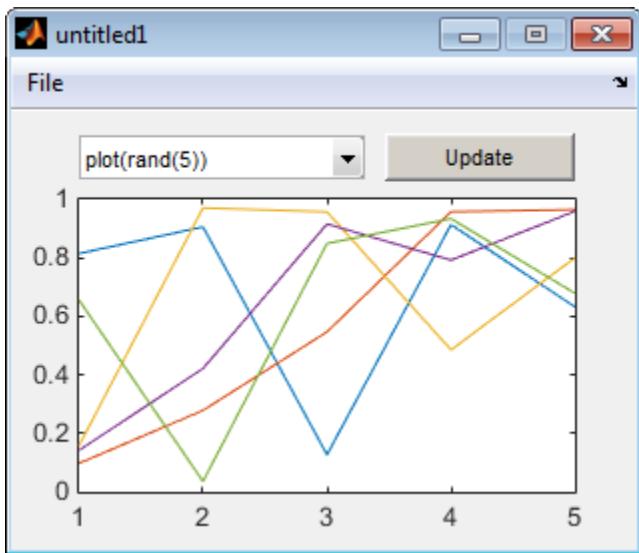
To view the code for these user interface controls, open the template in the Layout Editor and click the **Editor** button  on the toolbar.

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.

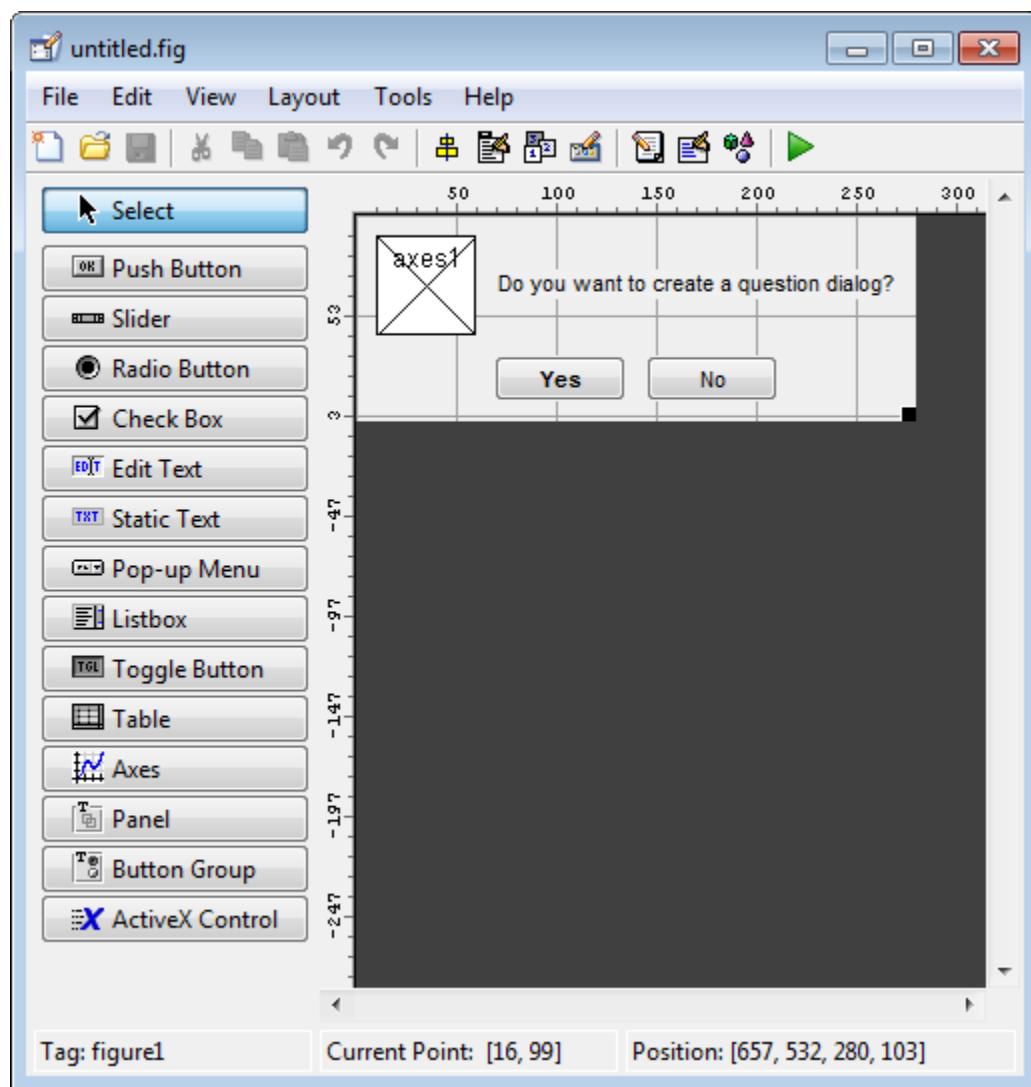
The UI also has a **File** menu with three items:

- **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.

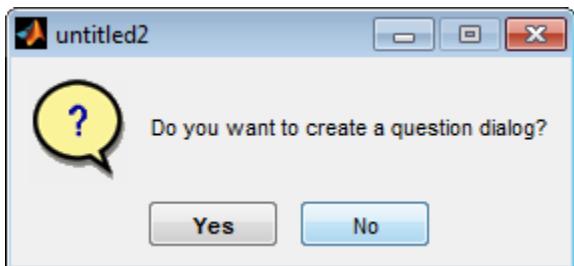
To view the code for these menu choices, open the template in the Layout Editor and click the **Editor** button  on the toolbar.

Modal Question Dialog

The following figure shows an example of this template.



When you click the **Run** button, the following dialog displays.



The dialog box returns Yes or No, depending on which button you click.

Select this template if you want the dialog box to return the user's selection, or if you want to create a *modal* dialog box.

Modal dialog boxes 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.

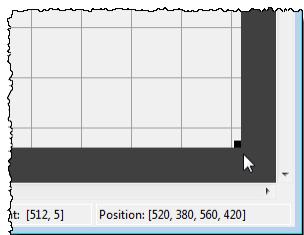
See Also

Related Examples

- “Ways to Build Apps” on page 1-2
- “Create a Simple App Using GUIDE” on page 2-2
- “Add Components to the GUIDE Layout Area” on page 6-13

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 6-108 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  and set the **Units**.

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

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  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**

See Also

Related Examples

- “Ways to Build Apps” on page 1-2
- “Create a Simple App Using GUIDE” on page 2-2
- “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 Apps” on page 6-78 for information about adding menus to a UI. See “Create Toolbars for GUIDE UIs” on page 6-95 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.

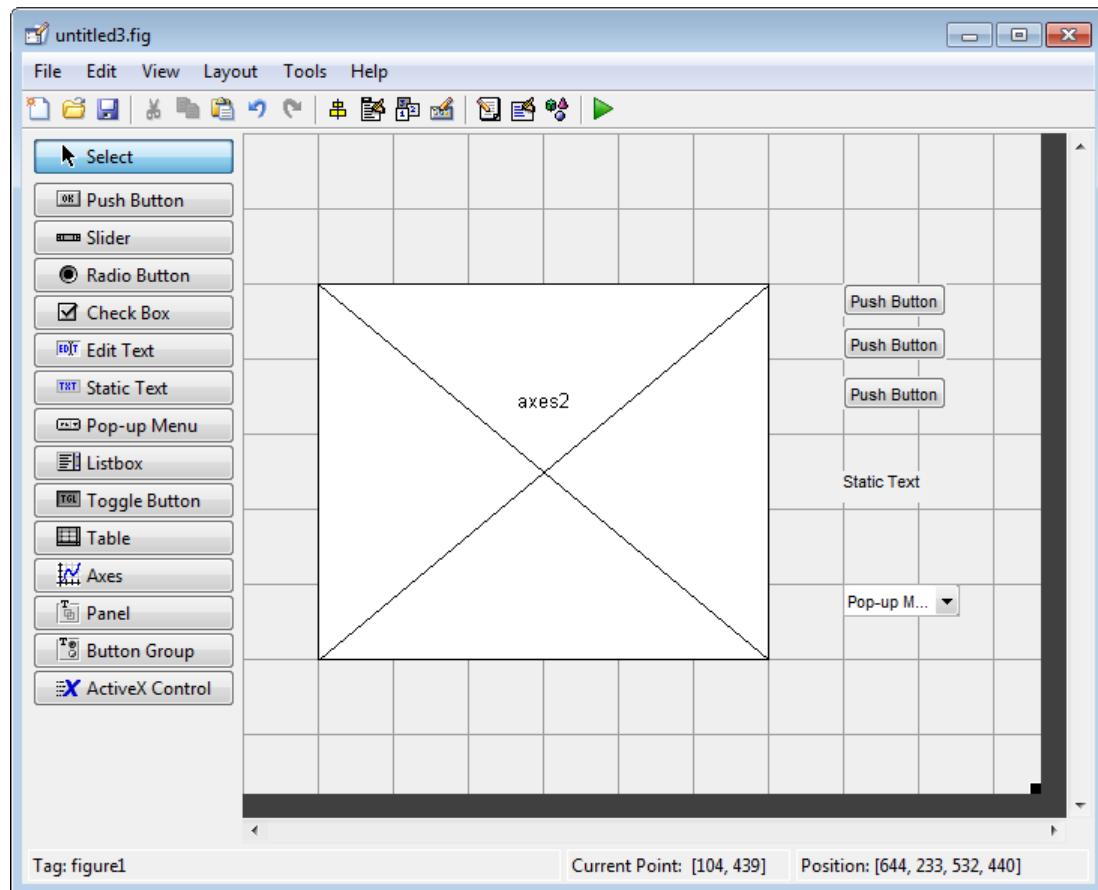
The components listed in the following table have additional considerations; read more about them in the sections described there.

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 Apps” on page 6-78
Toolbars	See “Create Toolbars for GUIDE UIs” on page 6-95
ActiveX controls	See “ActiveX Component” on page 6-60.

See “Grid and Rulers” on page 6-72 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.

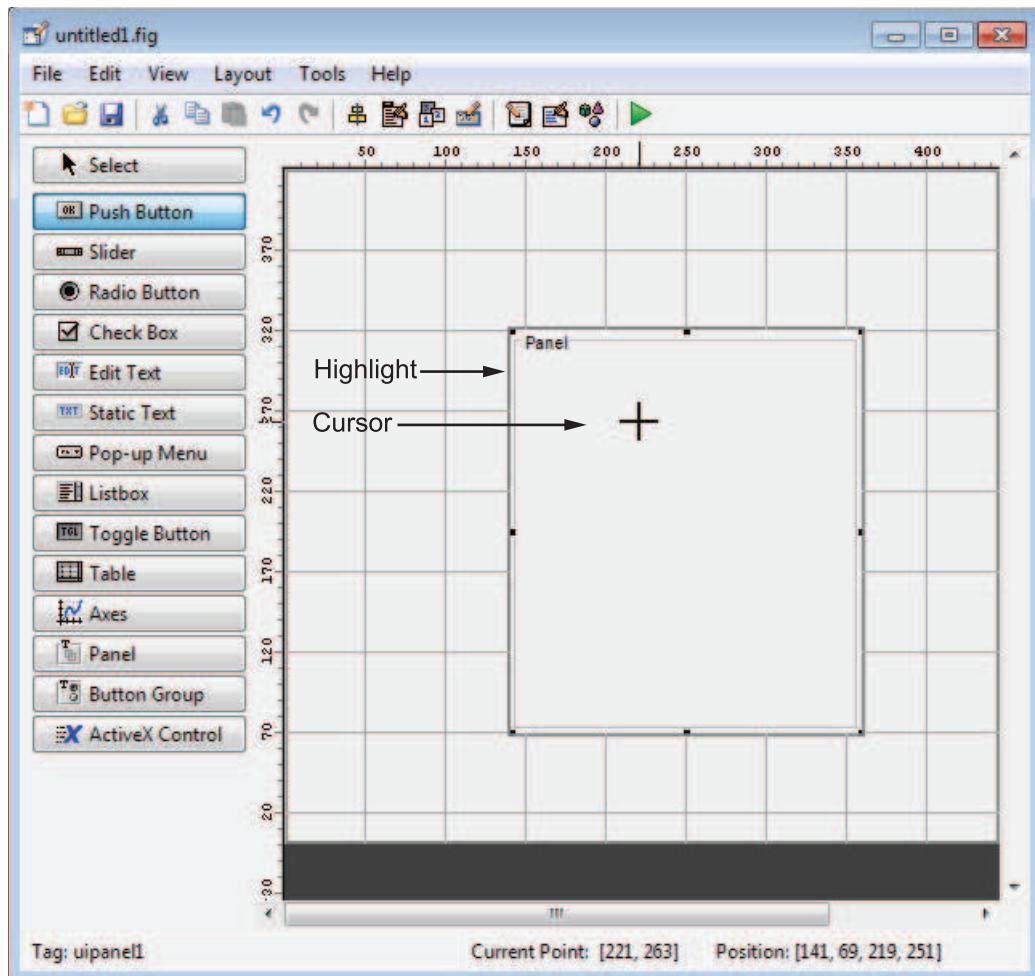
Here is how to interpret the coordinates in the status bar and rulers:

- 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 and in the running app.

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.

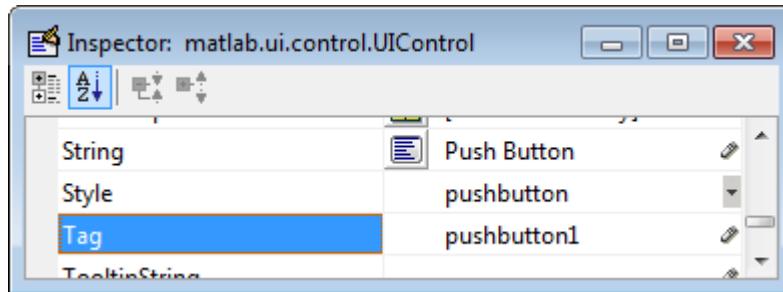
Assign an Identifier to Each Component

Use the Tag property to assign a unique and meaningful identifier to your components.

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 name or abbreviation that reflects the role of the component in the UI.

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

- 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 name you want to use as the identifier. In the following figure, Tag is set to pushbutton1.



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” on page 6-20
- “Push Button” on page 6-21
- “Slider” on page 6-23
- “Radio Button” on page 6-25
- “Check Box” on page 6-27
- “Edit Text” on page 6-28
- “Static Text” on page 6-30
- “Pop-Up Menu” on page 6-32
- “List Box” on page 6-34
- “Toggle Button” on page 6-37

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.

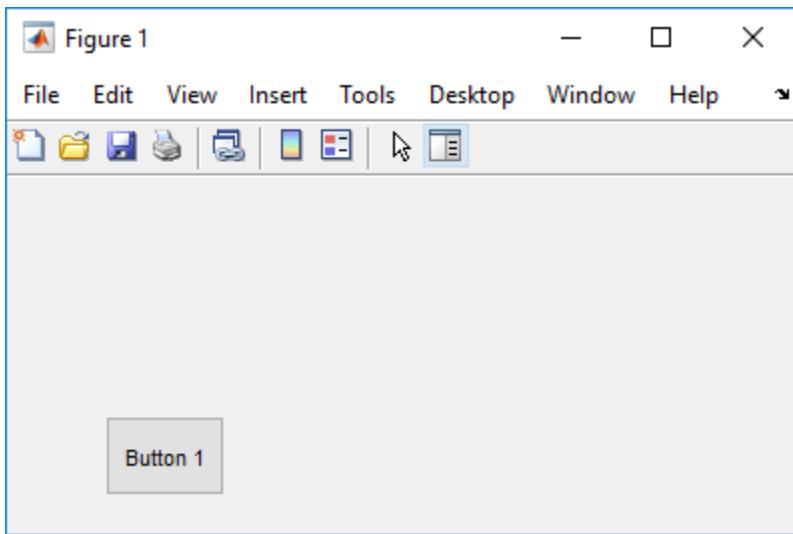
Property	Value	Description
Enable	on, inactive, off. Default is on.	Determines whether the control is available to the user
Max	Scalar. Default is 1.	Maximum value. Interpretation depends on the type of component.
Min	Scalar. Default is 0.	Minimum value. Interpretation depends on the type of component.

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	Character vector (for example, 'button1'). Can also be a character array or a cell array of character vectors.	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.

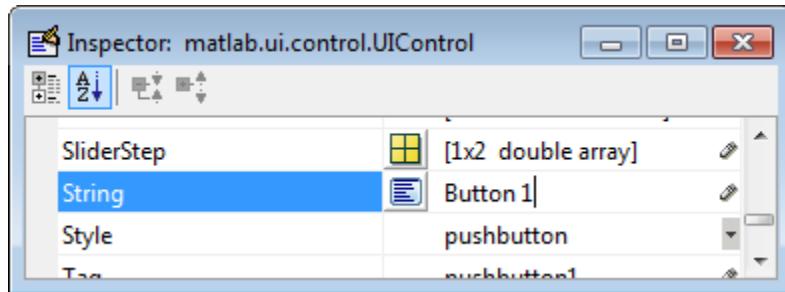
For a complete list of properties and for more information about the properties listed in the table, see [Uicontrol](#).

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. The words **remove**, **default**, and **factory** (case sensitive) are reserved. To use one of these as a label, prepend a backslash character (\). 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** property value, MATLAB truncates the value 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 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.

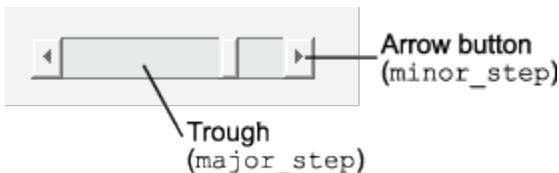
Slider

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 use this behavior to indicate how much of the document is currently visible by changing the value of `major_step`.



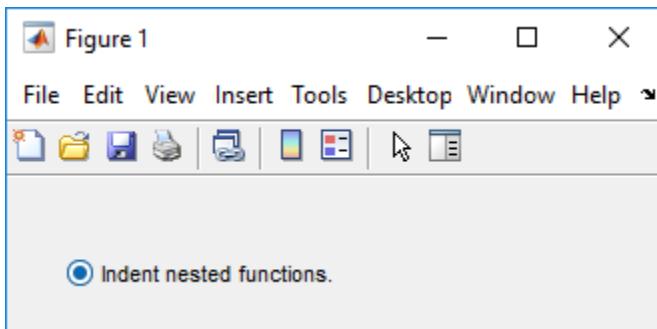
- If you want to set the location or size of the component to an exact value, then modify its **Position** property.

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.

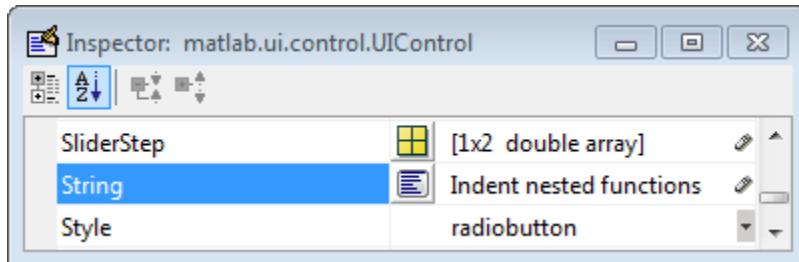
Note On Mac platforms, the height of a horizontal slider is constrained. If the height you set in the position vector exceeds this constraint, the displayed height of the slider is the maximum allowed. The height element of the position vector is not changed.

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. The words **remove**, **default**, and **factory** (case sensitive) are reserved. To use one of these as a label, prepend a backslash character (\). 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** property value, MATLAB software truncates the value with an ellipsis.

Indent ne...

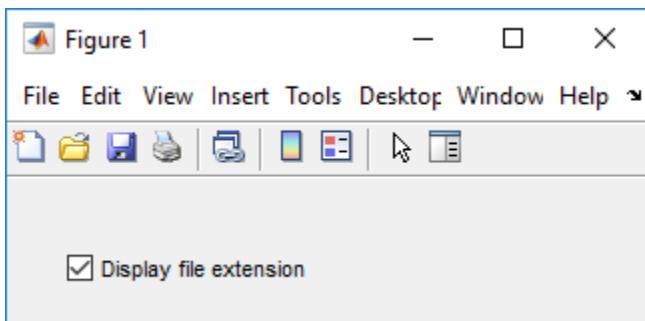
- Create the radio button with the button selected by setting its **Value** property to the value of its **Max** property (default is 1). Set **Value** to **Min** (default is 0) to leave the radio button unselected. Correspondingly, when the user selects the radio button, the software sets **Value** to **Max**, and to **Min** when the user deselects it.
- 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 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);
```

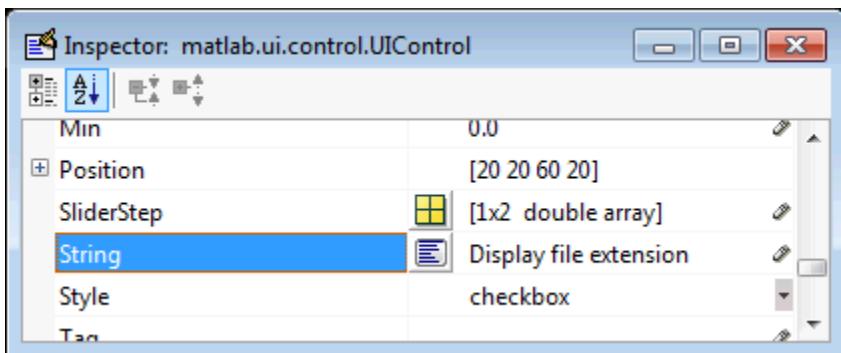
Note To manage exclusive selection of radio buttons and toggle buttons, put them in a button group. See “Button Group” on page 6-43 for more information.

Check Box

To create a check box with label **Display file extension** that is initially checked, as shown in this figure:

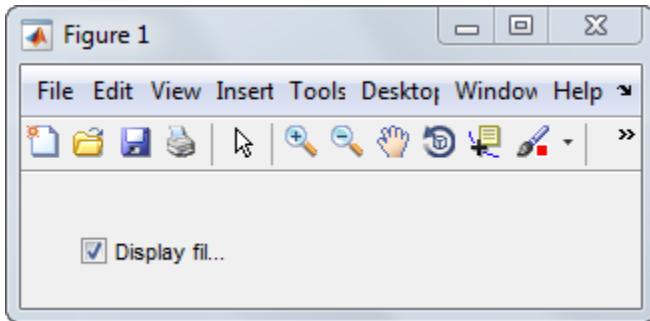


- Specify the check box label by setting the **String** property to the desired label, in this case, **Display file extension**.



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

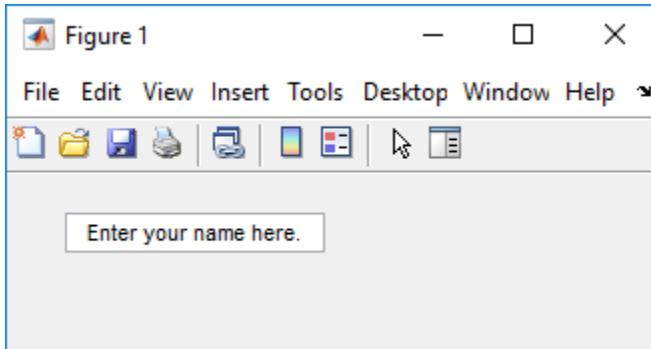
The check box accommodates only a single line of text. If you specify a component width that is too small to accommodate the specified **String** property value, MATLAB software truncates the value with an ellipsis.



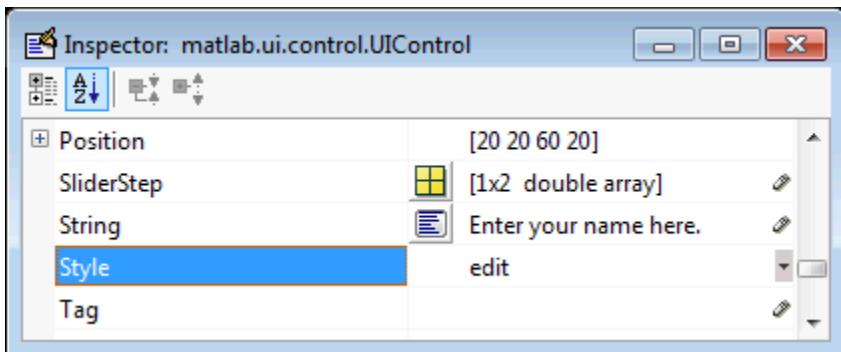
- Create the check box with the box checked by setting the `Value` property to the value of the `Max` property (default is 1). Set `Value` to `Min` (default is 0) to leave the box unchecked. Correspondingly, when the user clicks the check box, the software sets `Value` to `Max` when the user checks the 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:

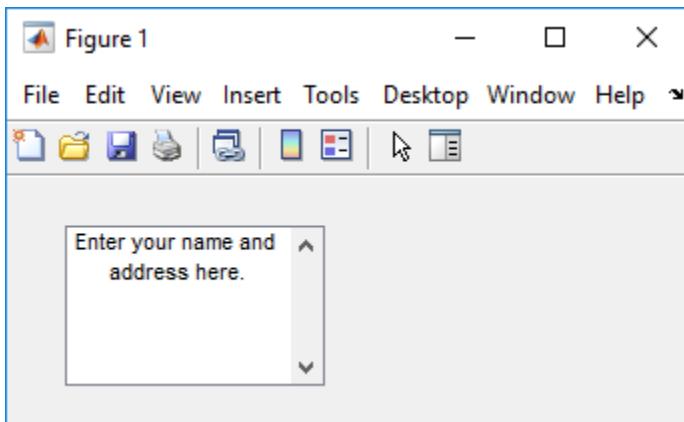


- Specify the text to be displayed when the edit text component is created by setting the `String` property to the desired value, in this case, `Enter your name here`.



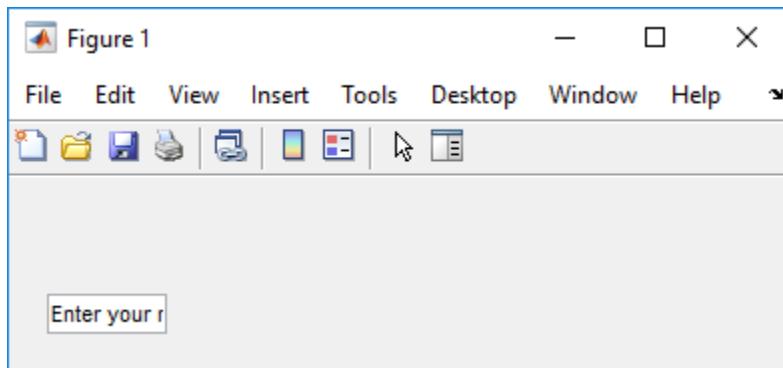
To display the & character in a label, use two & characters. The words **remove**, **default**, and **factory** (case sensitive) are reserved. To use one of these as a label, prepend a backslash character (\). For example, \remove yields **remove**.

- 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 displayed text 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 allows only a single line of input. If you specify a component width that is too small to accommodate the specified text, MATLAB displays only part of that text. The user can use the arrow keys to move the cursor through the text. 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 of the text.

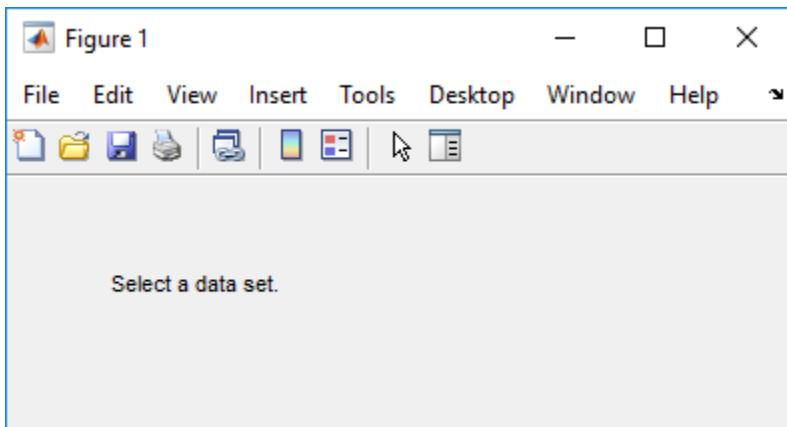


- If you want to set the position or size of the component to an exact value, then modify its **Position** property.
- You specify the text font to display in the edit box by typing the name of a font residing on your system into the **FontName** entry in the Property Inspector. On Microsoft® Windows® platforms, the default is **MS Sans Serif**; on Macintosh and UNIX® platforms, the default is **Helvetica**.

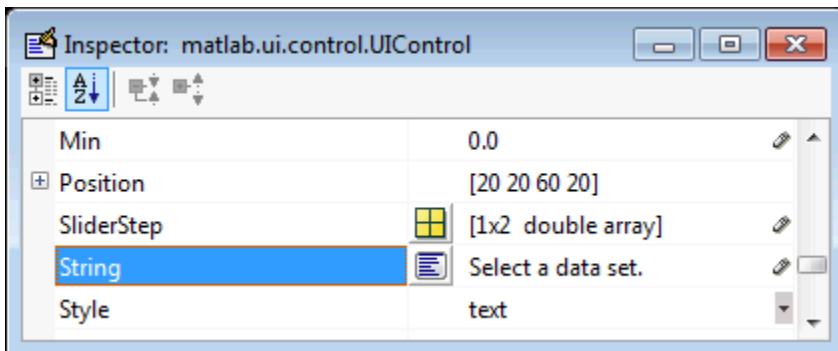
Tip To find out what fonts are available, type `uisetfont` at the MATLAB prompt; a dialog displays containing a list box from which you can select and preview available fonts. When you select a font, its name and other characteristics are returned in a structure, from which you can copy the **FontName** and paste it into the Property Inspector. Not all fonts listed may be available on other 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. The words **remove**, **default**, and **factory** (case sensitive) are reserved. To use one of these as a label, prepend a backslash character (\). For example, \remove yields **remove**.

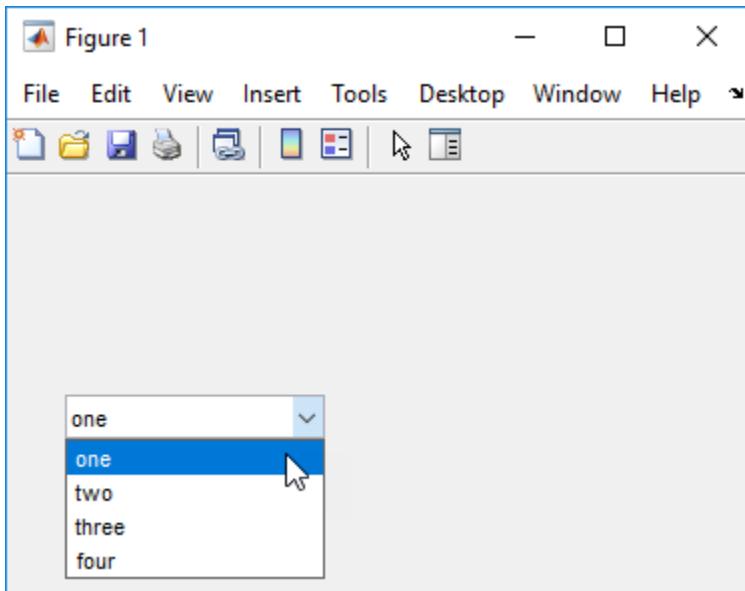
If your component is not wide enough to accommodate the specified value, MATLAB wraps the displayed text.



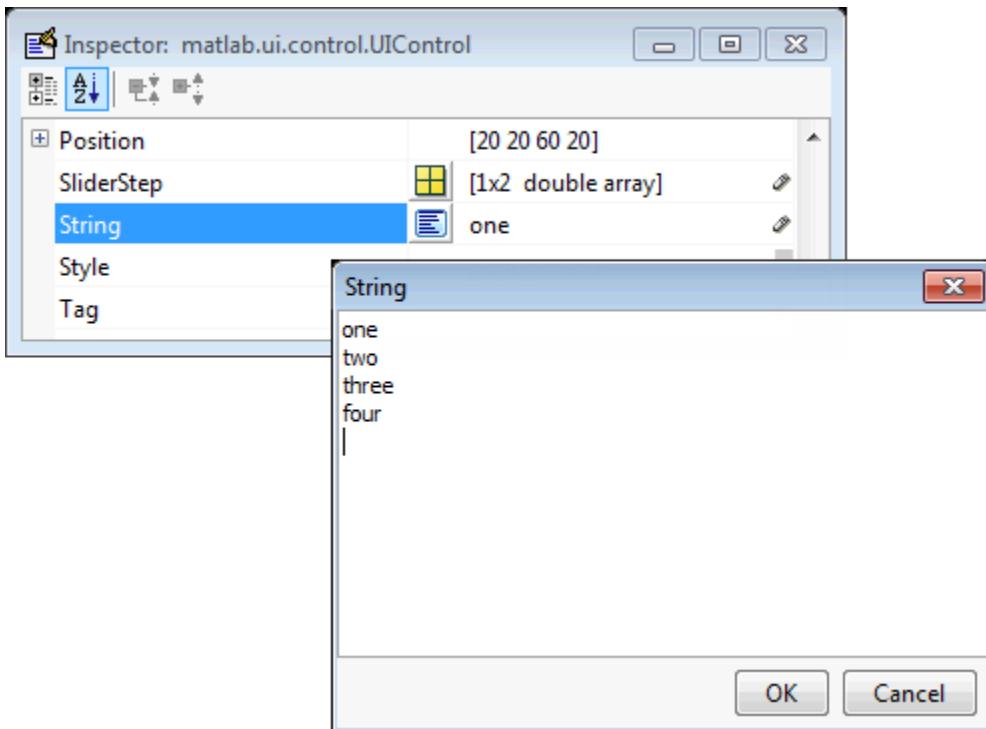
- If you want to set the position or size of the component to an exact value, then modify its **Position** property.
- You can specify a text font, including its **FontName**, **FontWeight**, **FontAngle**, **FontSize**, and **FontUnits** properties. For details, see the previous topic, “Edit Text” on page 6-28, and for a programmatic approach, the section “How to Set Font Characteristics” on page 9-22.

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:



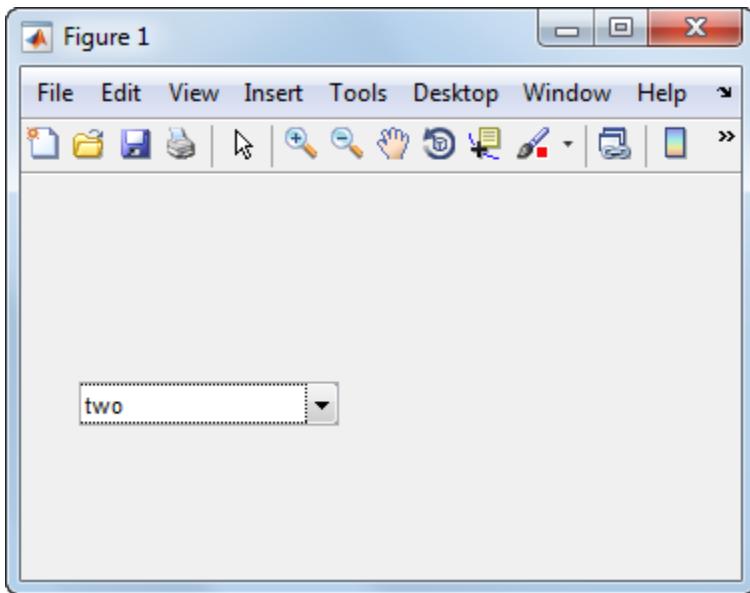
- Specify the pop-up menu items to be displayed by setting the **String** property to the desired items. Click the  button to the right of the property name to open the Property Inspector editor.



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

If the width of the component is too small to accommodate one or more of the menu items, MATLAB truncates those items 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:

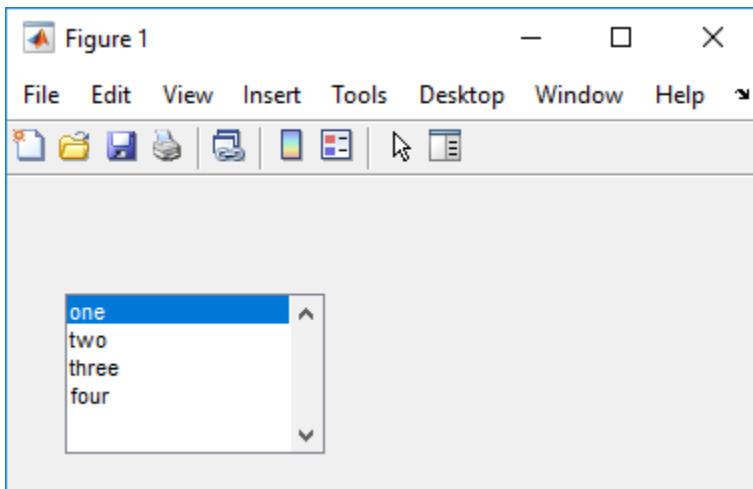


- If you want to set the position and size of the component to exact values, then modify its **Position** property. The height of a pop-up menu is determined by the font size. The height you set in the position vector is ignored.

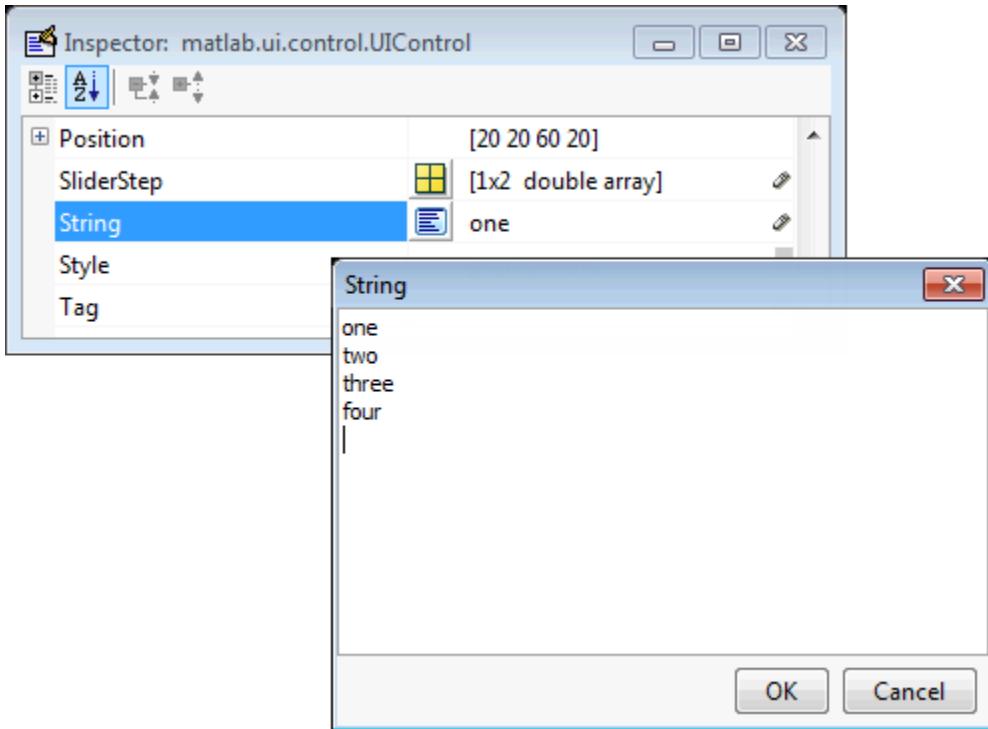
Note The pop-up menu does not provide for a label. Use a “Static Text” on page 6-30 component to label the pop-up menu.

List Box

To create a list box with items **one**, **two**, **three**, and **four**, as shown in this figure:



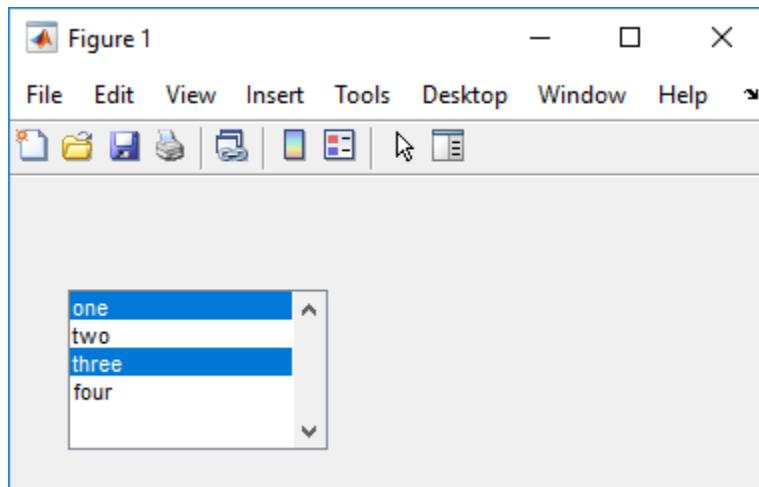
- Specify the list of items to be displayed by setting the `String` property to the desired list. Use the Property Inspector editor to enter the list. You can open the editor by clicking the  button to the right of the property name.



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

If the width of the component is too small to accommodate one or more of the specified list items, MATLAB software truncates those items 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., $\text{Max} - \text{Min} > 1$, 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.

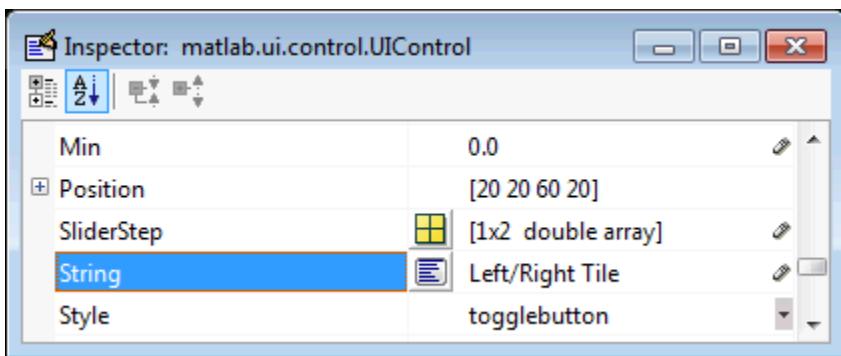
Note The list box does not provide for a label. Use a “Static Text” on page 6-30 component to label the list box.

Toggle Button

To create a toggle button with label **Left/Right Tile**, as shown in this figure:

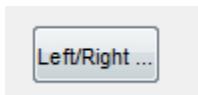


- Specify the toggle button label by setting its `String` property to the desired label, in this case, `Left/Right Tile`.

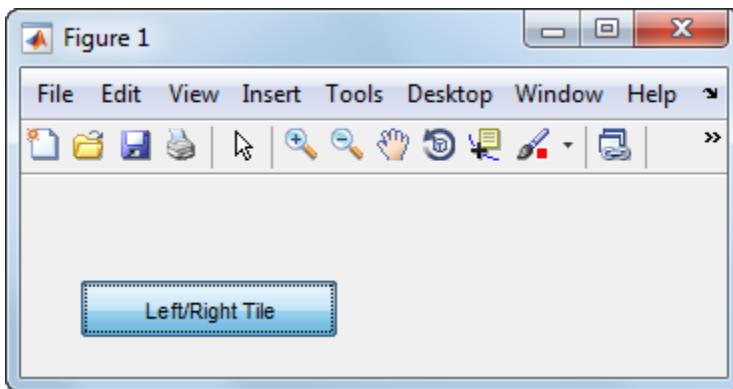


To display the & character in a label, use two & characters. The words `remove`, `default`, and `factory` (case sensitive) are reserved. To use one of these as a label, prepend a backslash character (\). For example, \remove yields **remove**.

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` value, MATLAB truncates the text with an ellipsis.



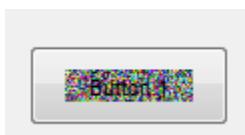
- Create the toggle button with the button selected (depressed) by setting its **Value** property to the value of its **Max** property (default is 1). Set **Value** to **Min** (default is 0) to leave the toggle button unselected (raised). Correspondingly, when the user selects the toggle button, MATLAB 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 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.

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:

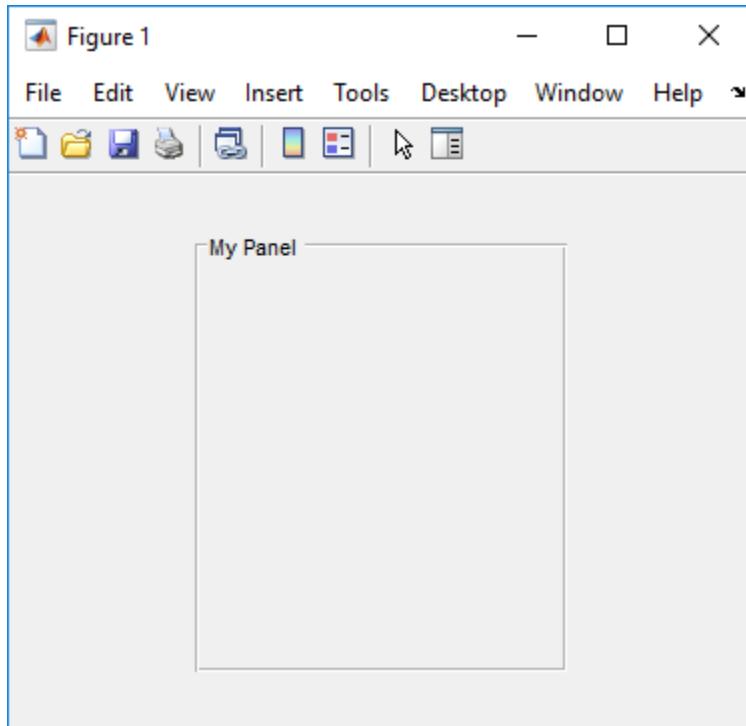
Property	Values	Description
Position	4-element vector: [distance from left, distance from bottom, width, height].	Size of the component and its location relative to its parent.
Title	Character vector (for example, 'Start').	Component label.
TitlePosition	lefttop, centertop, righttop, leftbottom, centerbottom, rightbottom. Default is lefttop.	Location of title in relation to the panel or button group.

Property	Values	Description
Units	characters, centimeters, inches, normalized, pixels, points. Default is characters.	Units of measurement used to interpret the Position property vector

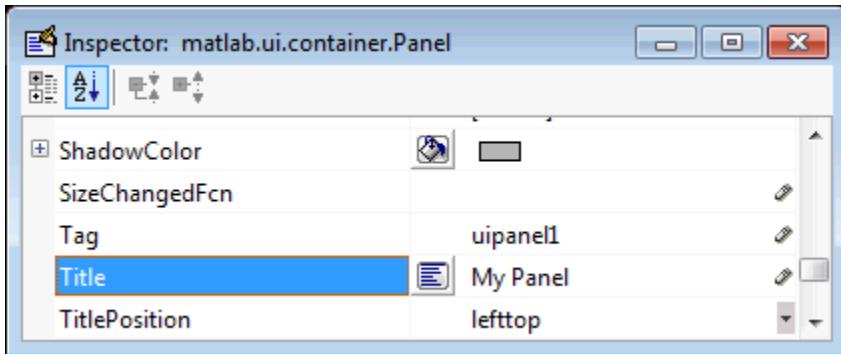
For a complete list of properties and for more information about the properties listed in the table, see the Uipanel and Uibuttongroup.

Panel

To create a panel with title **My Panel** as shown in the following figure:

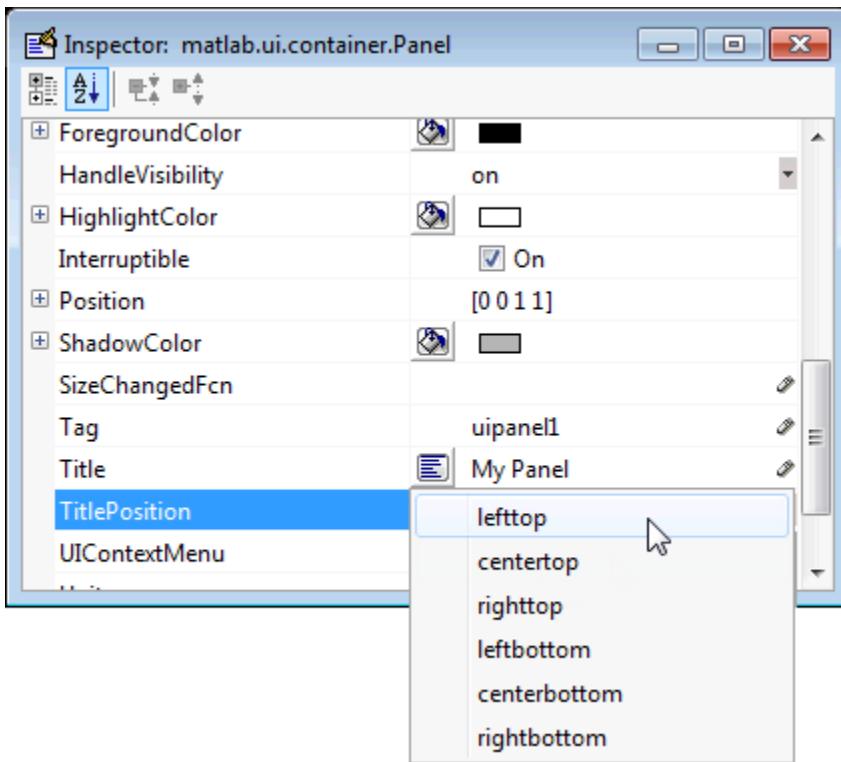


- Specify the panel title by setting the Title property to the desired value, in this case My Panel.



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

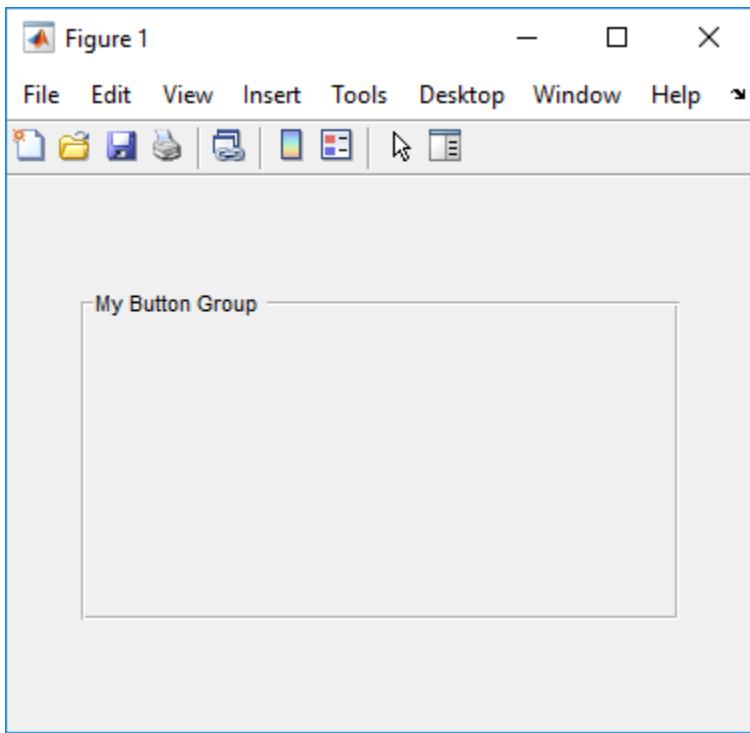
- Specify the location of the panel title by selecting one of the available TitlePosition property values from the pop-up menu, in this case lefttop. You can position the title at the left, middle, or right of the top or bottom of the panel.



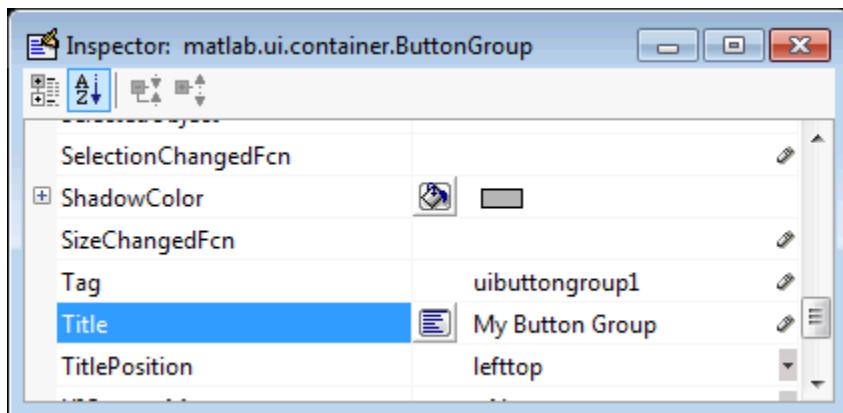
- If you want to set the position or size of the panel to an exact value, then modify its **Position** property.

Button Group

To create a button group with title **My Button Group** as shown in the following figure:

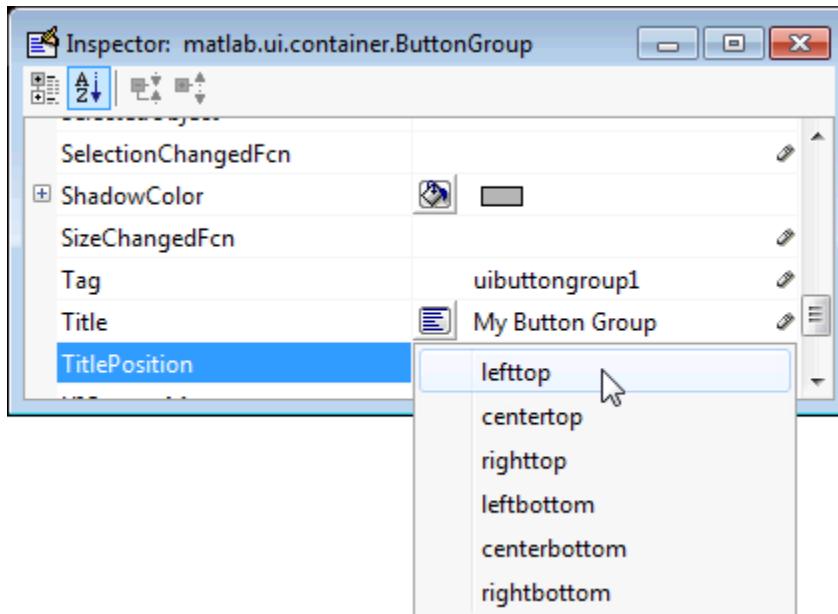


- Specify the button group title by setting the Title property to the desired value, in this case My Button Group.



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

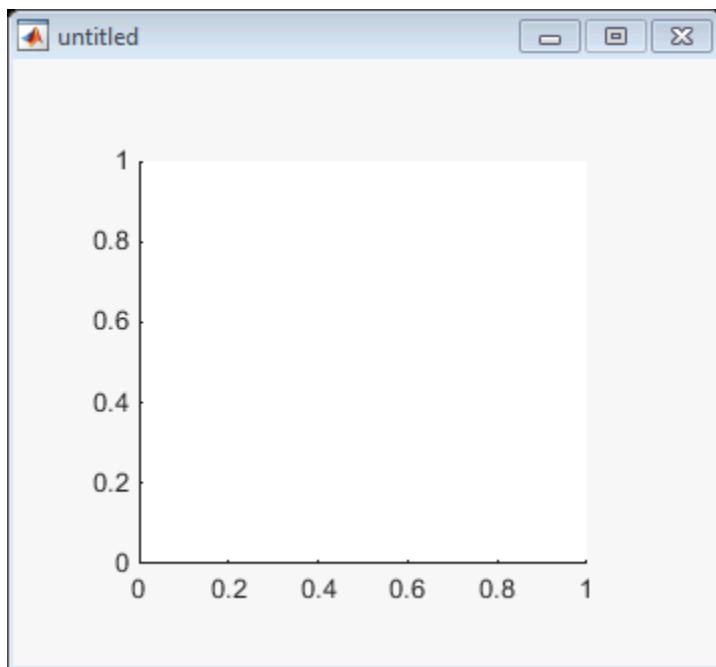
For a complete list of properties and for more information about the properties listed in the table, see [Axes](#).

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, such as resetting axis limits and removing axes context menus and callbacks. See “Create Axes” on page 6-47 and “Axes” on page 9-20 for information about setting the `NextPlot` property.

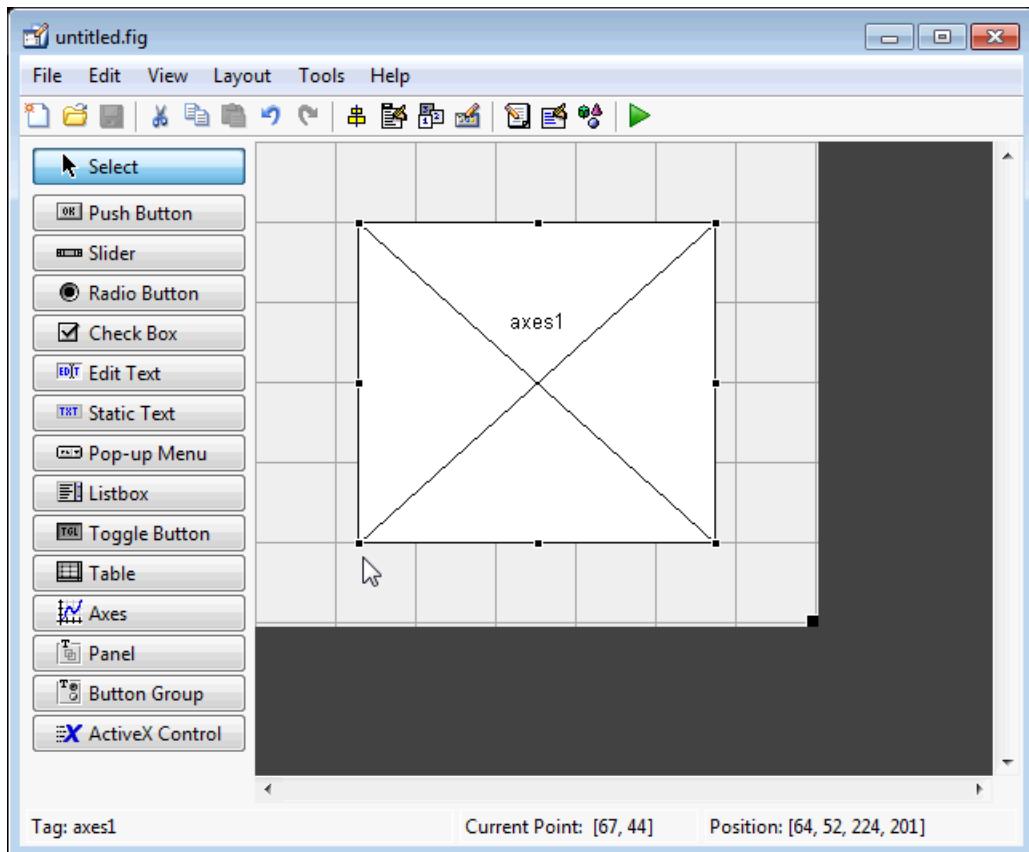
Create Axes

Here is an axes in a GUIDE app:



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 code file to label an axes component. For example,

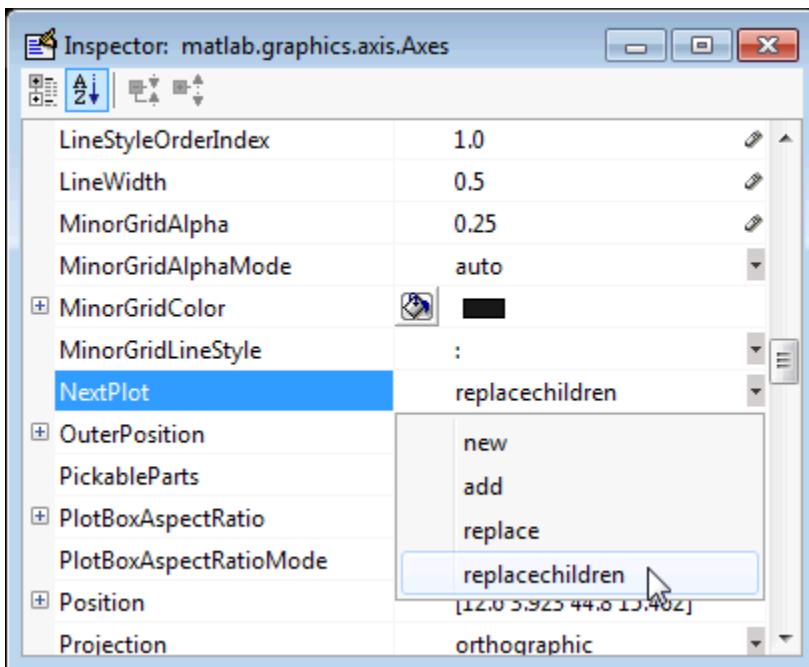
```
xlh = (axes_handle, 'Years')
```

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 character (\). 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.

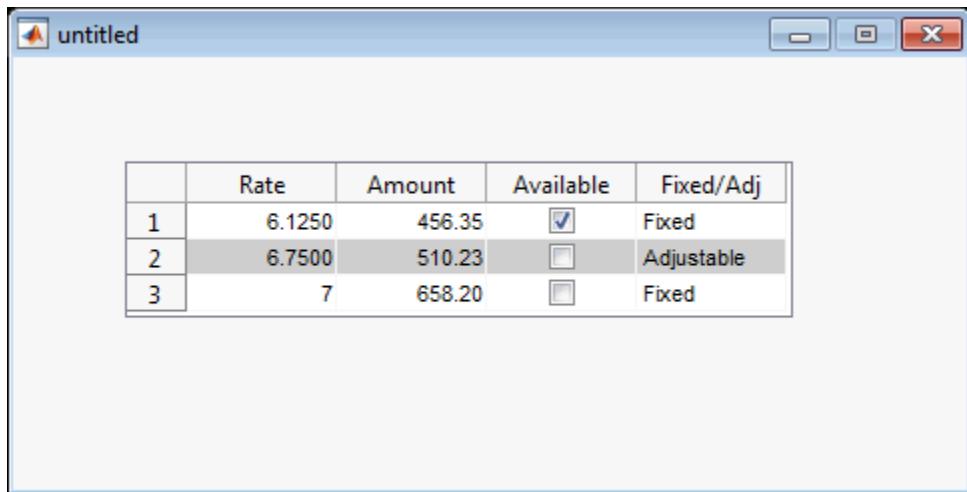
Commonly Used Properties

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:

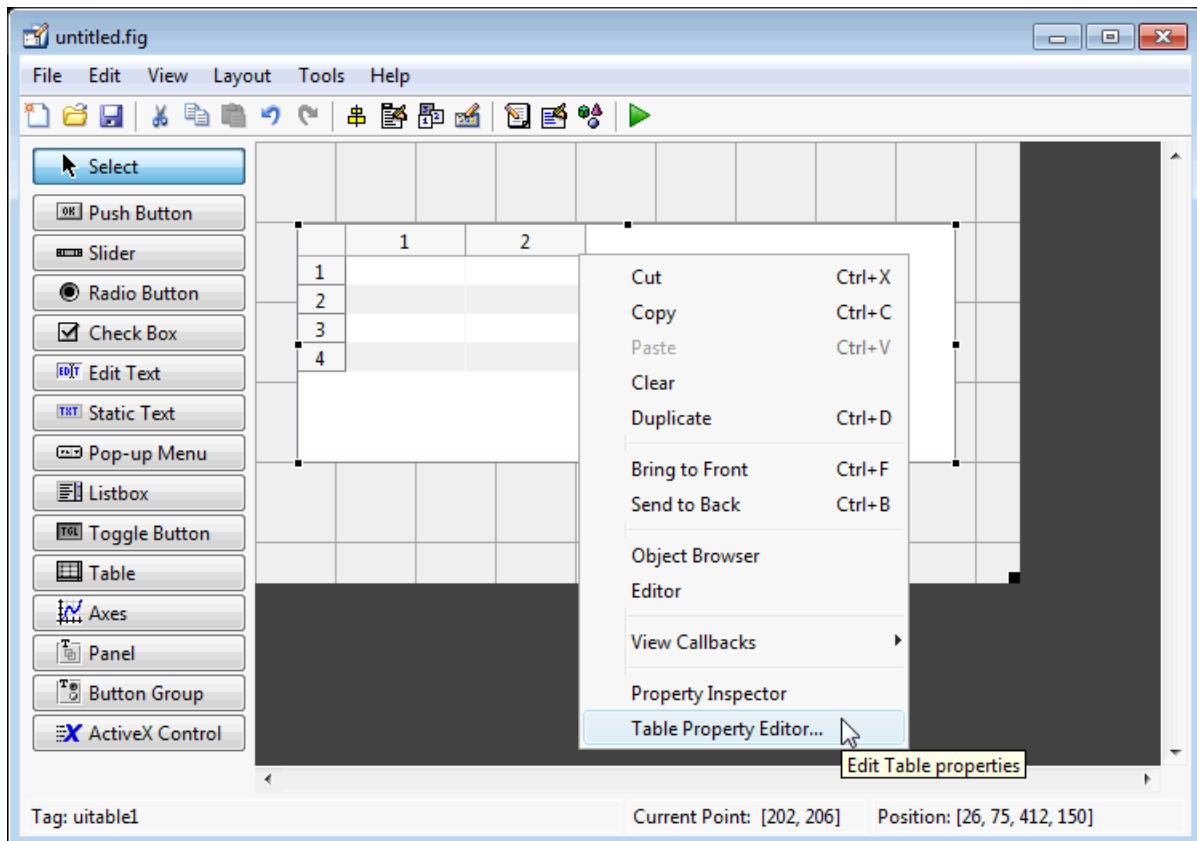
Group	Property	Values	Description
Column	ColumnName	1-by- n cell array of character vectors {'numbered'} empty matrix ([])	The header label of the column.
	ColumnFormat	Cell array of character vectors	Determines display and editability of columns
	ColumnWidth	1-by- n cell array or 'auto'	Width of each 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 character vectors	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

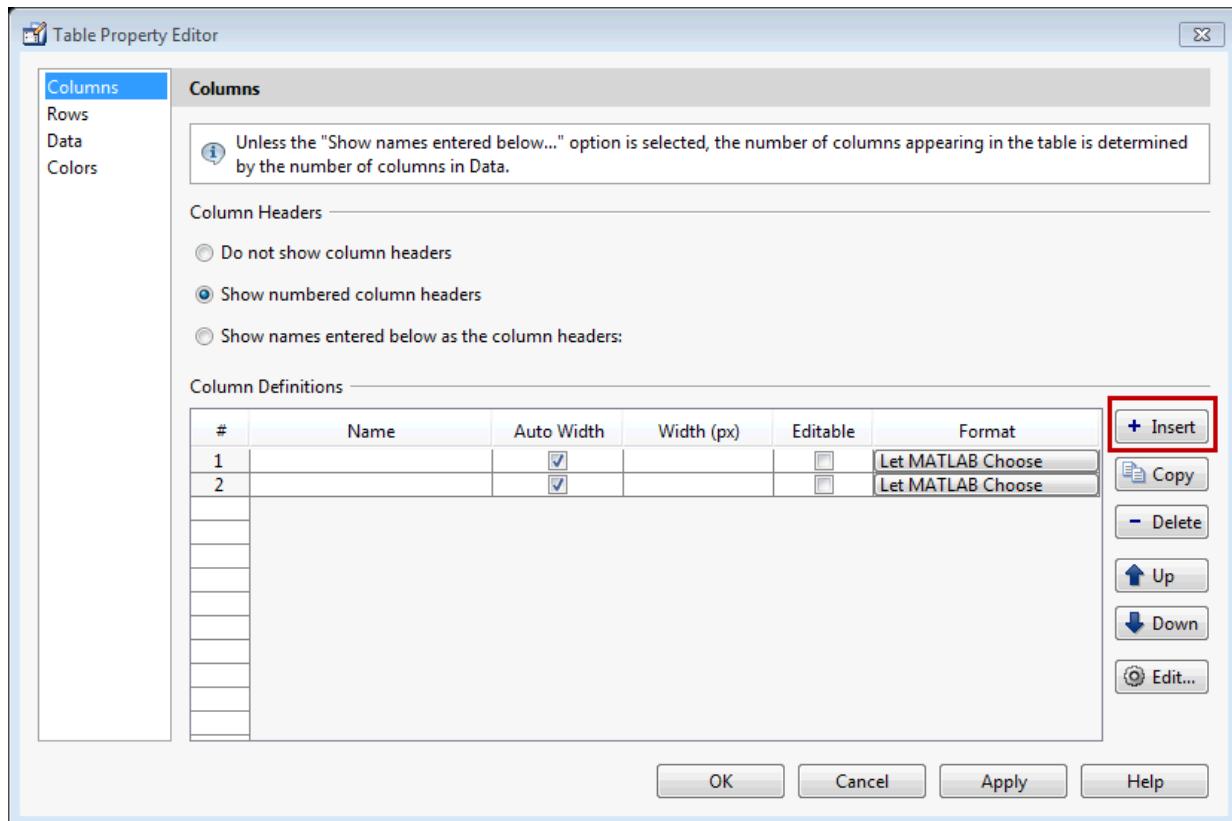
Keyboard equivalents only operate when the cursor is in the data entry area. In addition to those listed above, typing **Ctrl+T** or **Cmd+T** selects the entire field containing the cursor for editing (if the field contains text).

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.

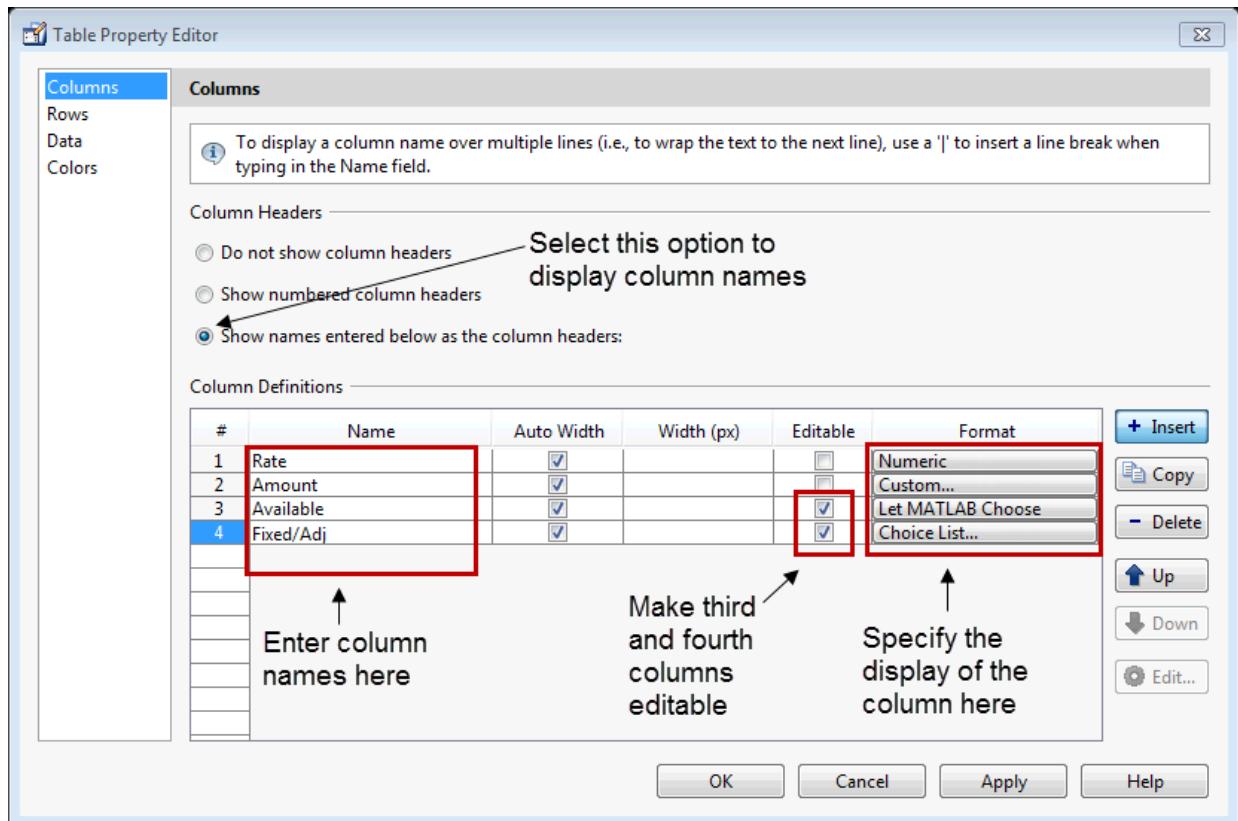
Set Column Properties

Click **Insert** to add two more columns.

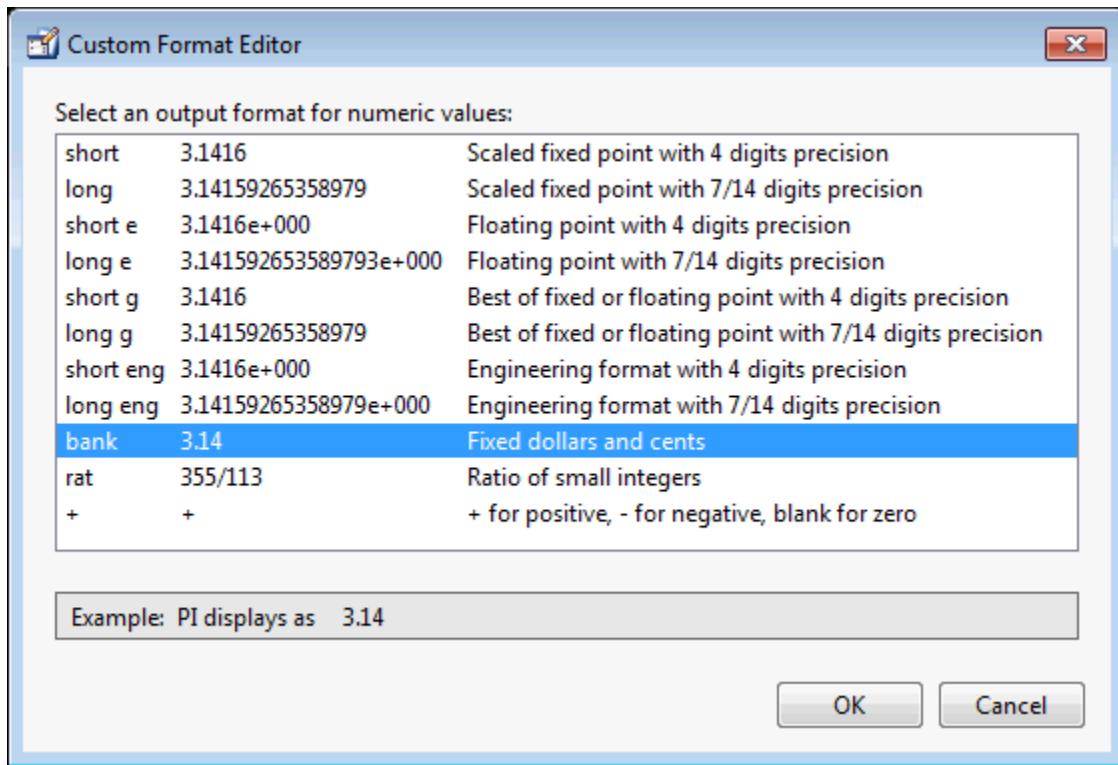
6 Lay Out a UI Using GUIDE



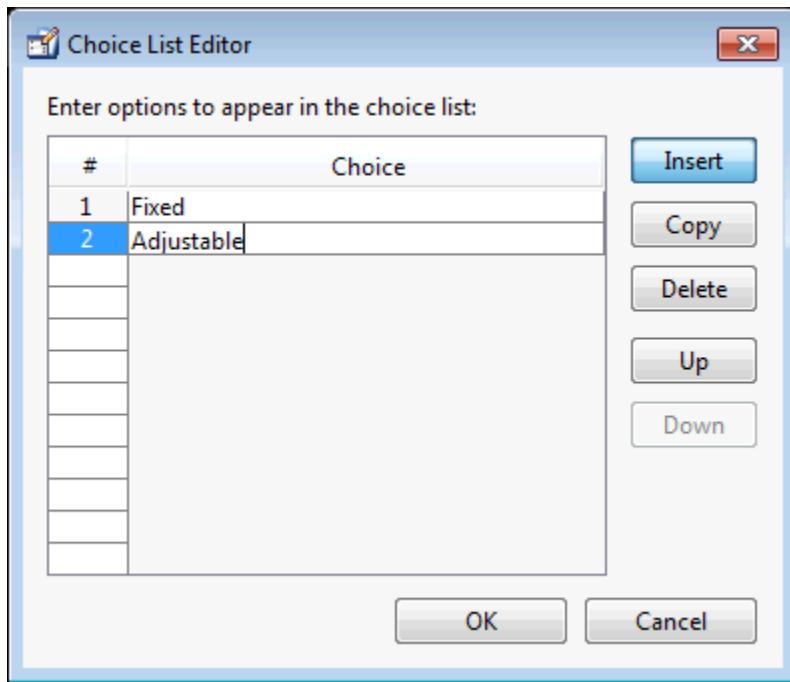
Select **Show names entered below as the column headers** and set the **ColumnName** by entering Rate, Amount, Available, and Fixed/Adj in **Name** group. for the Available and Fixed/Adj columns set the **ColumnEditable** property to on. Lastly set the **ColumnFormat** for the four columns



For the Rate column, select **Numeric**. For the Amount Column select **Custom** and in the Custom Format Editor, choose **Bank**.



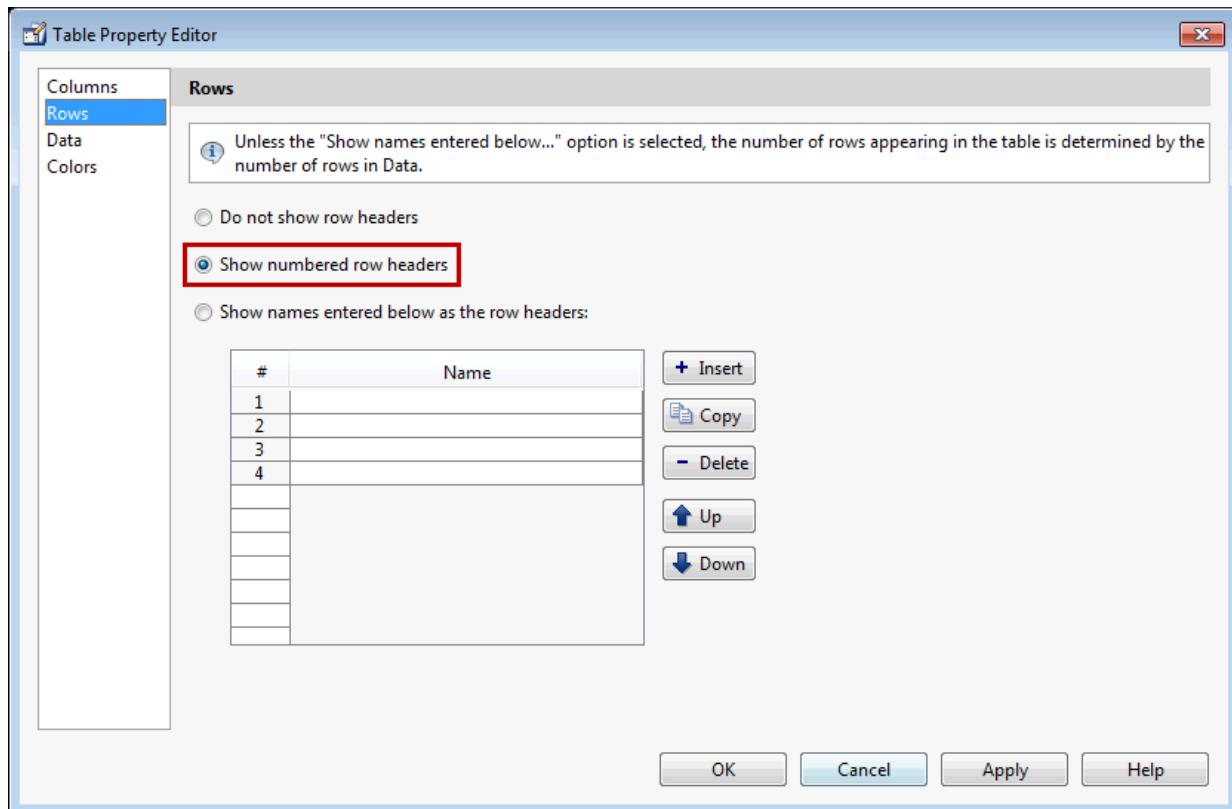
Leave the Available column at the default value. This allows MATLAB to chose based on the value of the Data property of the table. For the Fixed/Adj column select Choice List to create a pop-up menu. In the Choice List Editor, click **Insert** to add a second choice and type Fixed and Adjustable as the 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**.

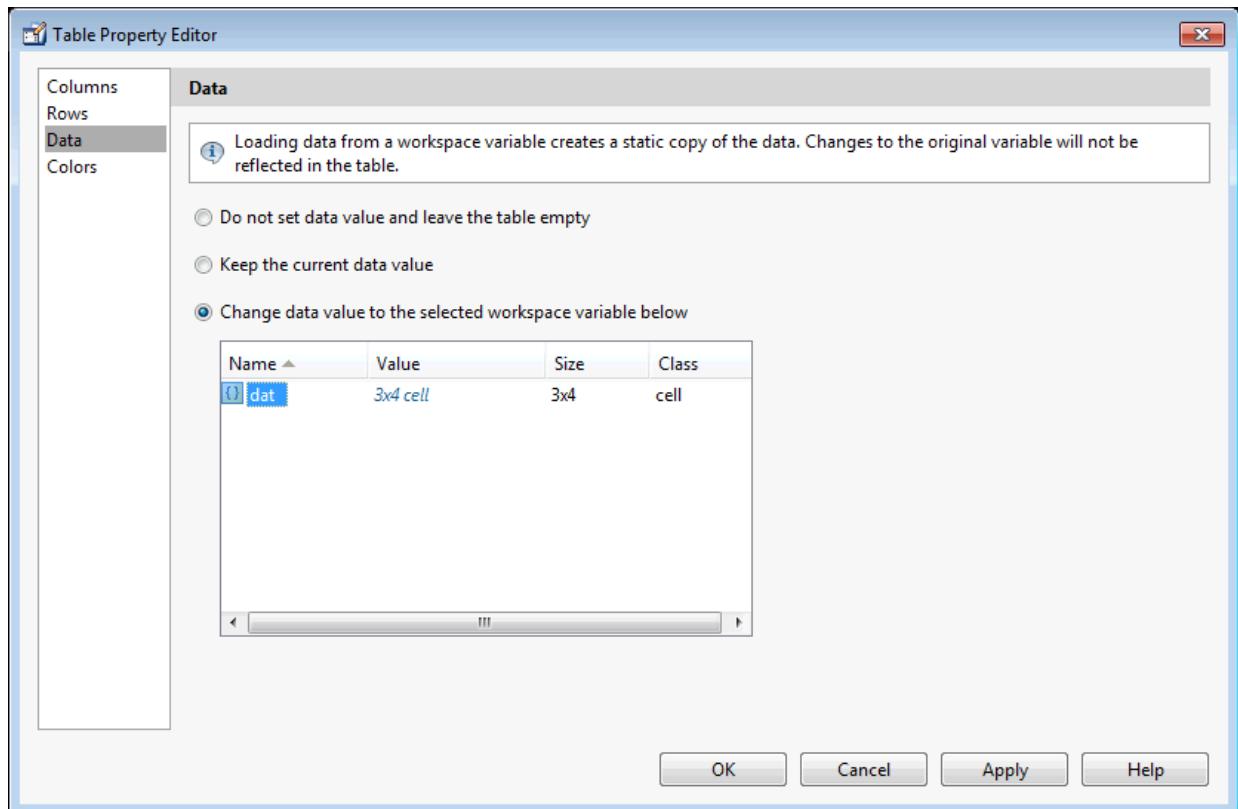


Set Data Properties

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:

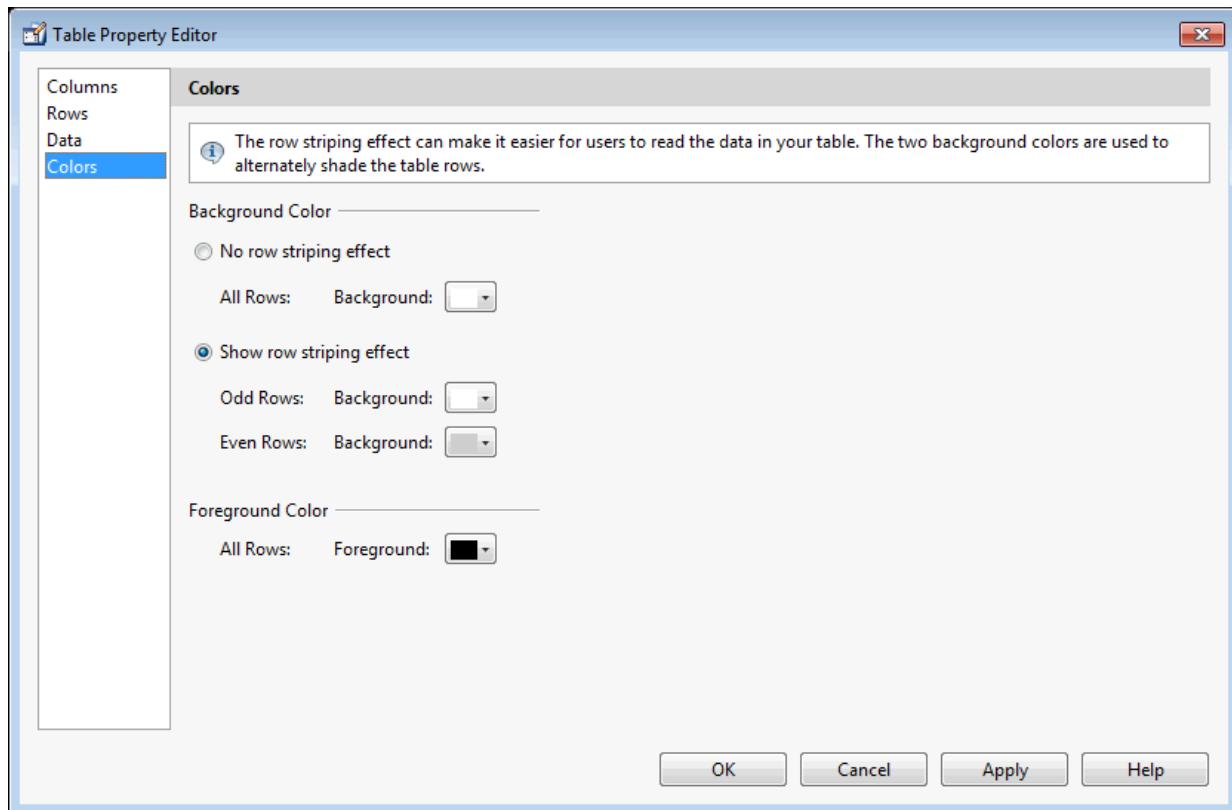
```
dat = {6.125, 456.3457, true, 'Fixed';...
6.75, 510.2342, false, 'Adjustable';...
7, 658.2, false, 'Fixed'};
```

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.

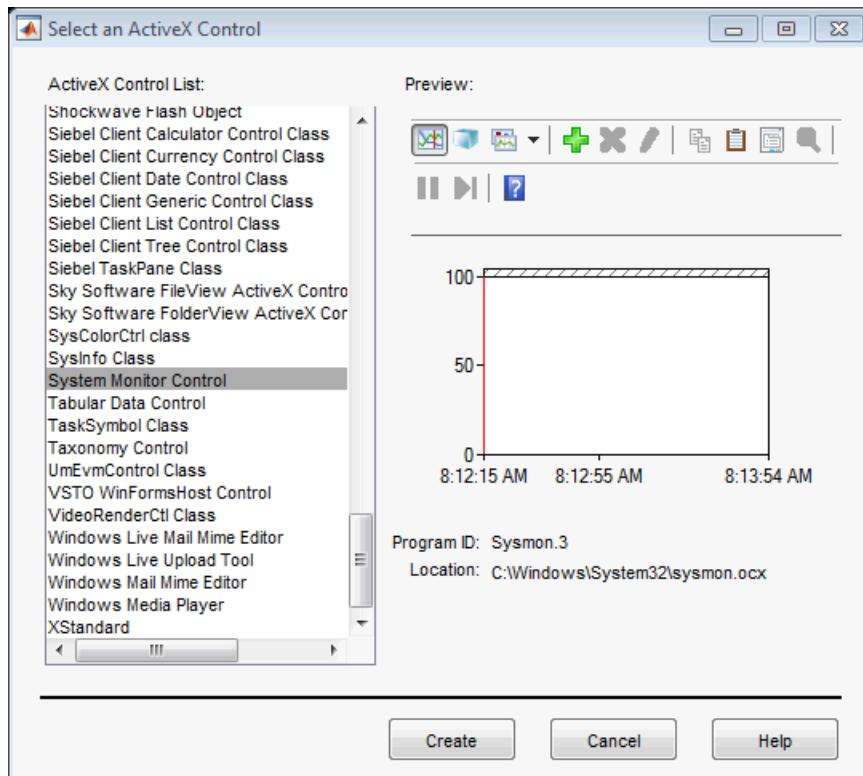


You can change other *uitable* properties to the table via the Property Inspector.

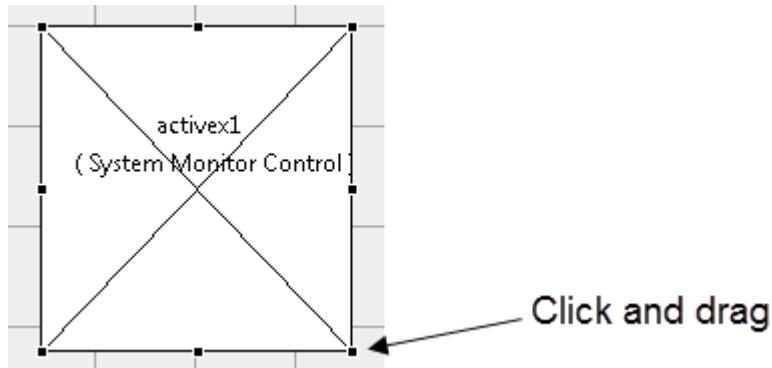
ActiveX Component

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

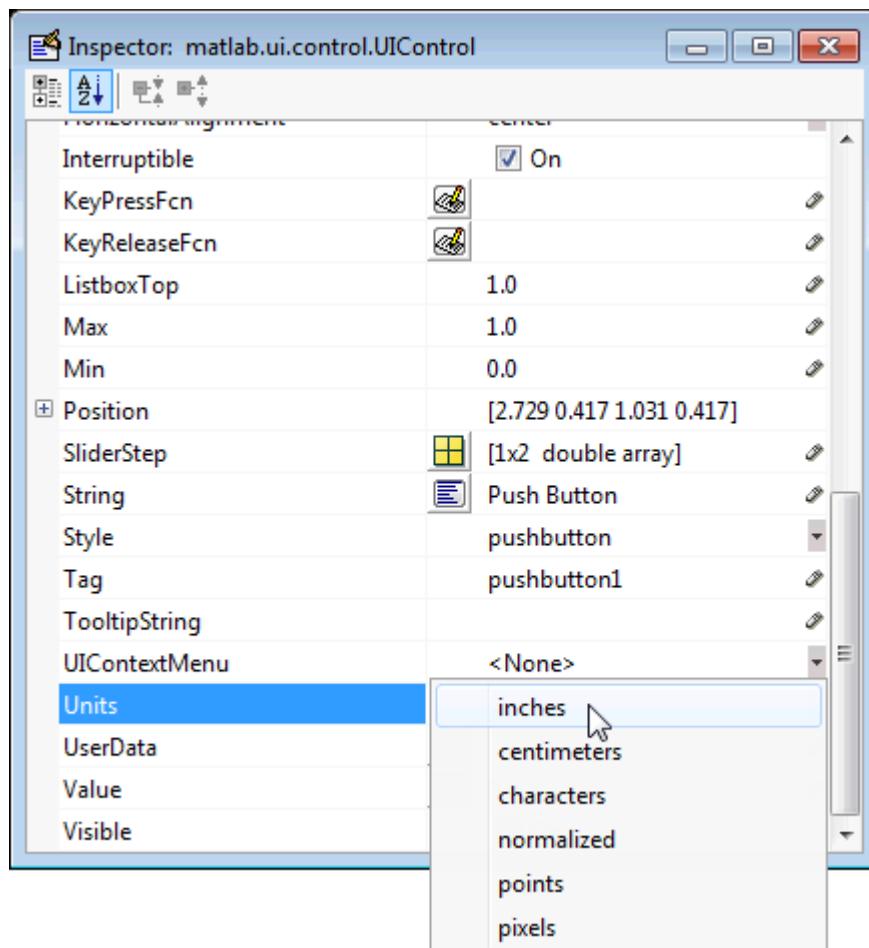
Select the component you want to resize. Click one of the corner handles and drag it until the component is the desired size.



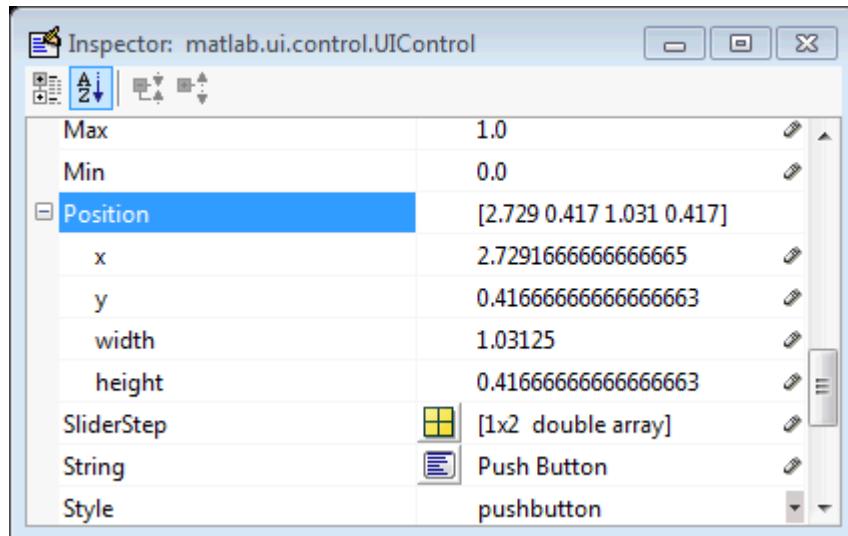
Set the Component's Position Property

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

- 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.

See Also

Related Examples

- “Ways to Build Apps” on page 1-2
- “Create a Simple App Using GUIDE” on page 2-2
- “Write Callbacks in GUIDE” on page 7-2

- “Callbacks for Specific Components” on page 7-12

Align GUIDE UI Components

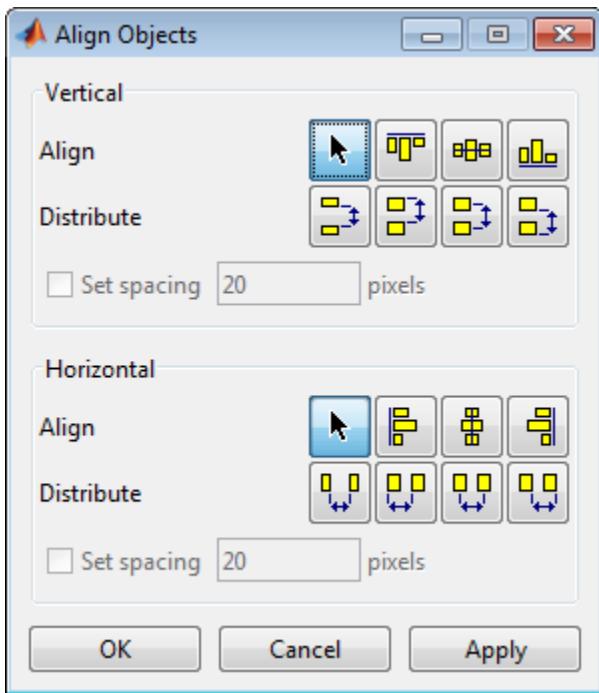
In this section...

- “Align Objects Tool” on page 6-66
- “Property Inspector” on page 6-69
- “Grid and Rulers” on page 6-72
- “Guide Lines” on page 6-73

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.



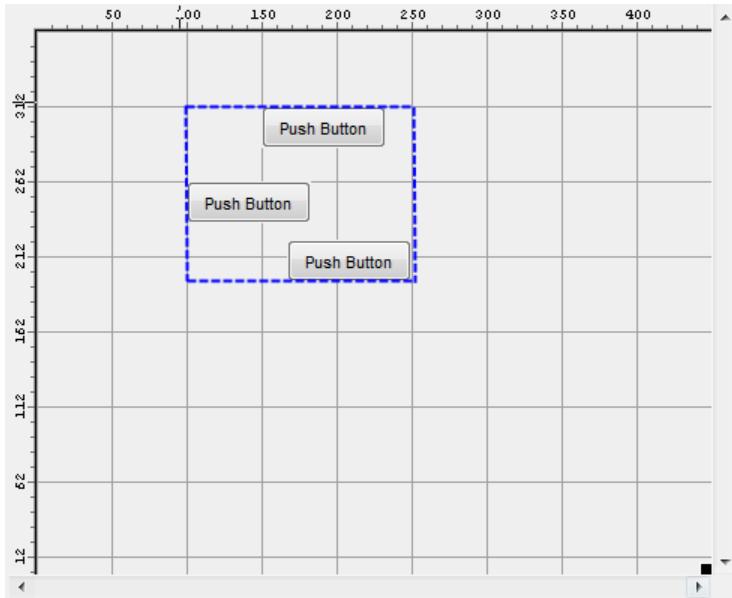
The Align Objects tool provides two types of alignment operations:

- **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

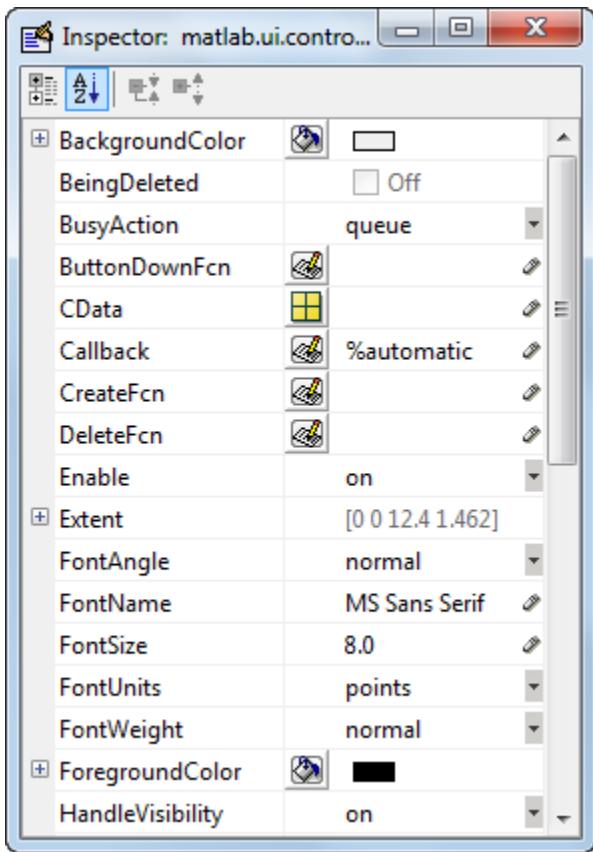
Property Inspector

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.

Inspector: matlab.ui.control.UIControl

FontSize	8.0
FontUnits	points
FontWeight	normal
ForegroundColor	[Color Chooser] [Black]
HandleVisibility	on
HorizontalAlignment	center
Interruptible	On
KeyPressFcn	
KeyReleaseFcn	
ListboxTop	1.0

Help

Uicontrol Properties

Text

- **String** — Text to display
string | cell array of char values | pipe-delimited row vector | padded column matrix
- ▼ **HorizontalAlignment** — Alignment of uicontrol text
'center' (default) | 'left' | 'right'
Alignment of the uicontrol text, specified as 'center', 'left', or 'right'. This property determines the justification of the String property text.

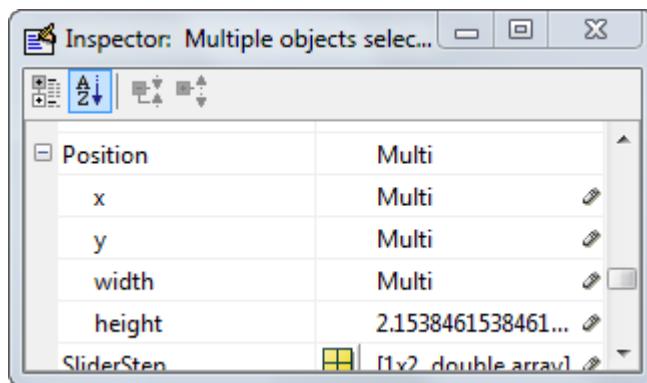
Interactive Control

- **Callback** — Callback function that executes when user interacts with uicontrol
'' (default) | function handle | cell array | string
- **ButtonDownFcn** — Button-press callback function
javascript:void(0);

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.

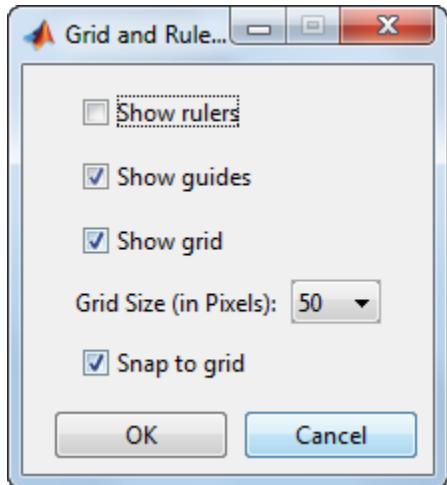


- 5 Change the value of **x** to align their left sides. Change the value of **y** to align their bottom edges. For example, setting **x** to 2.0 aligns the left sides of the components 2 inches from the left side of the window.
- 6 When the components are aligned, change the **Units** property back 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 on page 6-73
- Set the grid spacing
- Enable or disable snap-to-grid

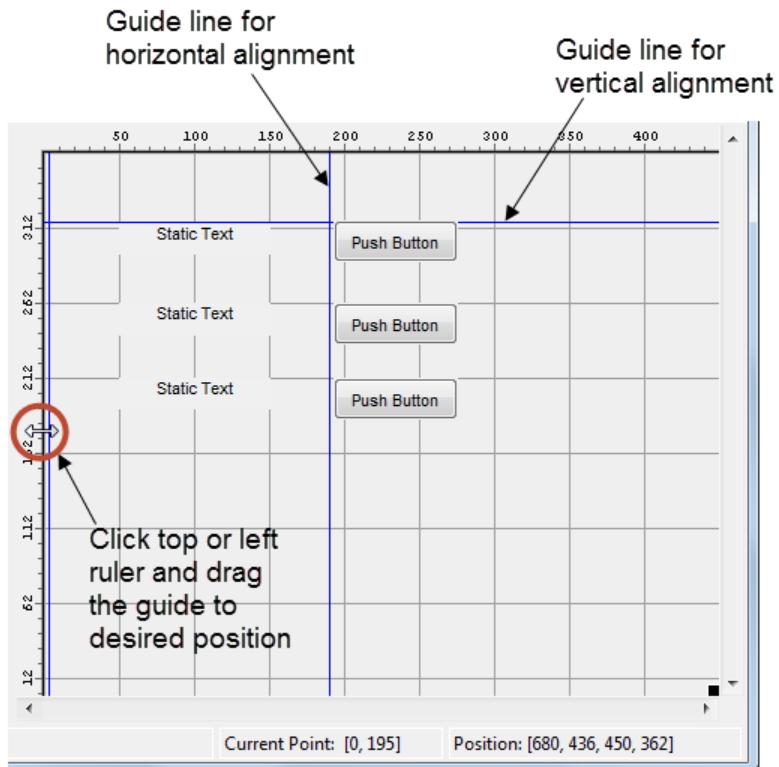
Guide Lines

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.



See Also

Related Examples

- “GUIDE Options” on page 5-8

Customize Tabbing Behavior in a GUIDE UI

The tab order is the order in which components 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.

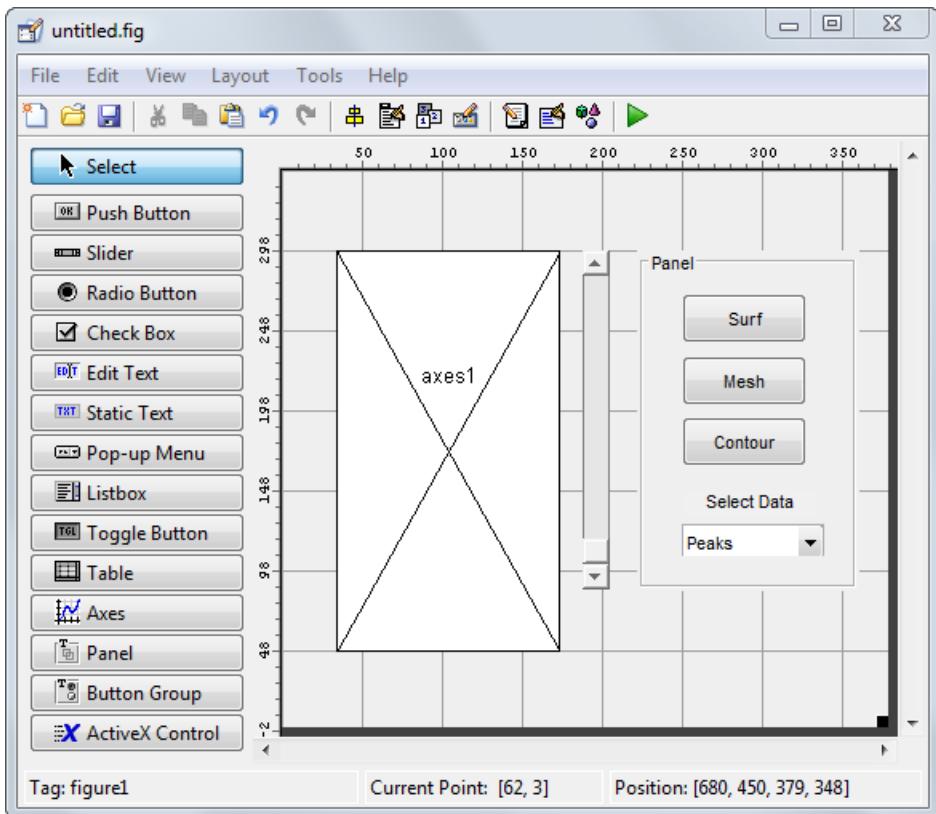
Note Axes cannot be tabbed. From GUIDE, you cannot include ActiveX components in the tab order.

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.

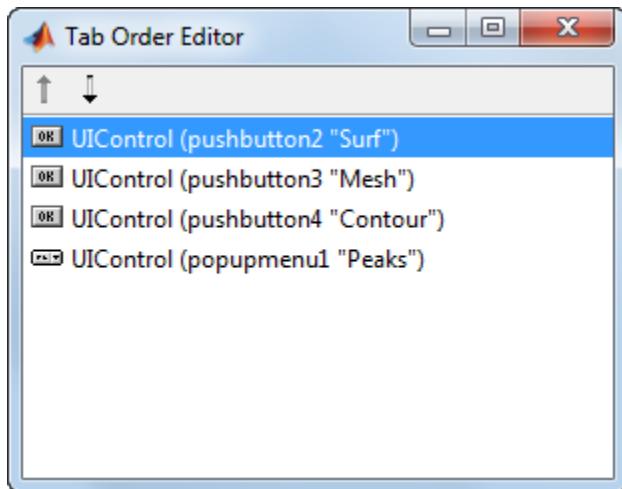
Note Tab order also affects the stacking order of components. If components overlap, those that appear lower in the tabbing order, are drawn on top of those that appear higher in the order.

The 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.

6 Lay Out a UI Using GUIDE



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.

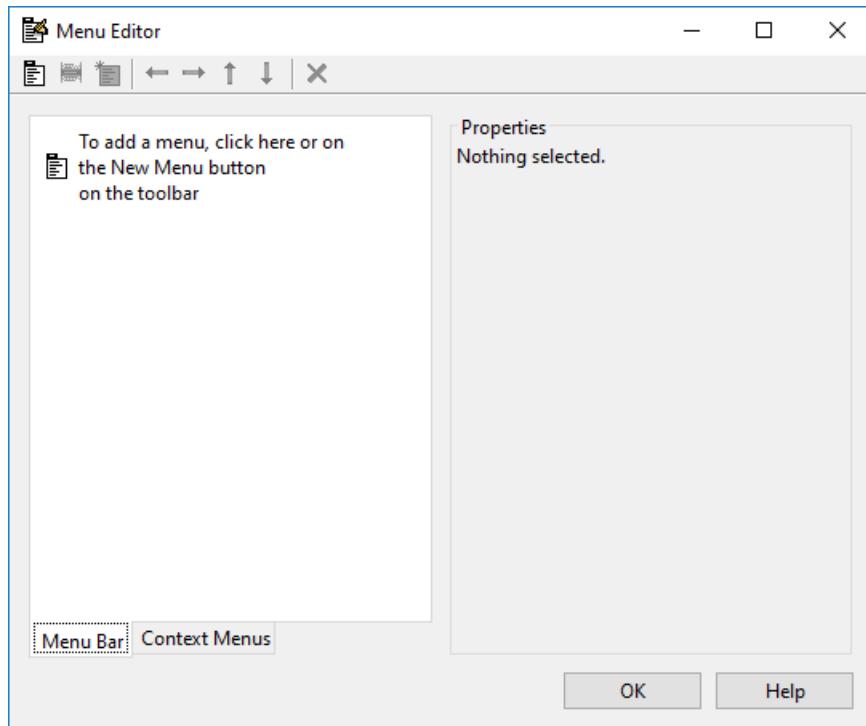
Create Menus for GUIDE Apps

In this section...

["Menus for the Menu Bar" on page 6-78](#)

["Context Menus" on page 6-88](#)

You can use GUIDE to create menu bars (containing 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 6-79](#)

- “Add Standard Menus to the Menu Bar” on page 6-80
- “Create a Menu” on page 6-80
- “Add Items to a Menu” on page 6-82
- “Additional Drop-Down Menus” on page 6-85
- “Cascading Menus” on page 6-85

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.

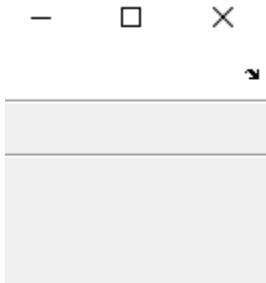


Figure windows with a standard menu bar also have a **Desktop** menu from which the user can dock and undock them.

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 6-95 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.

Add Standard Menus to the Menu Bar

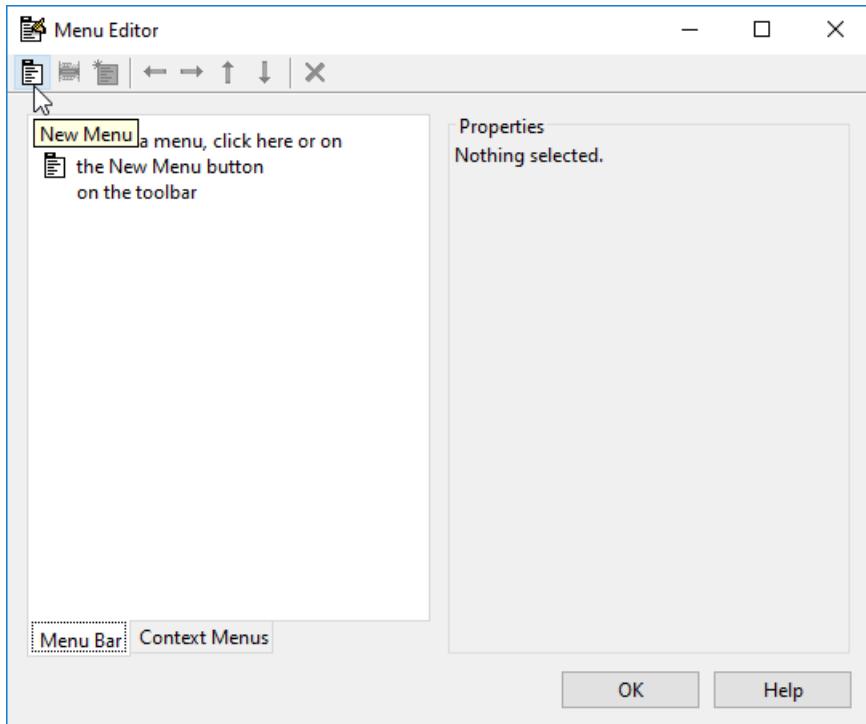
The figure `MenuBar` property controls whether your UI displays the MATLAB standard menus on the menu bar. GUIDE initially sets the value of `MenuBar` to `none`. If you want your UI to display the MATLAB standard menus, use the Property Inspector to set `MenuBar` to `figure`.

- 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'.

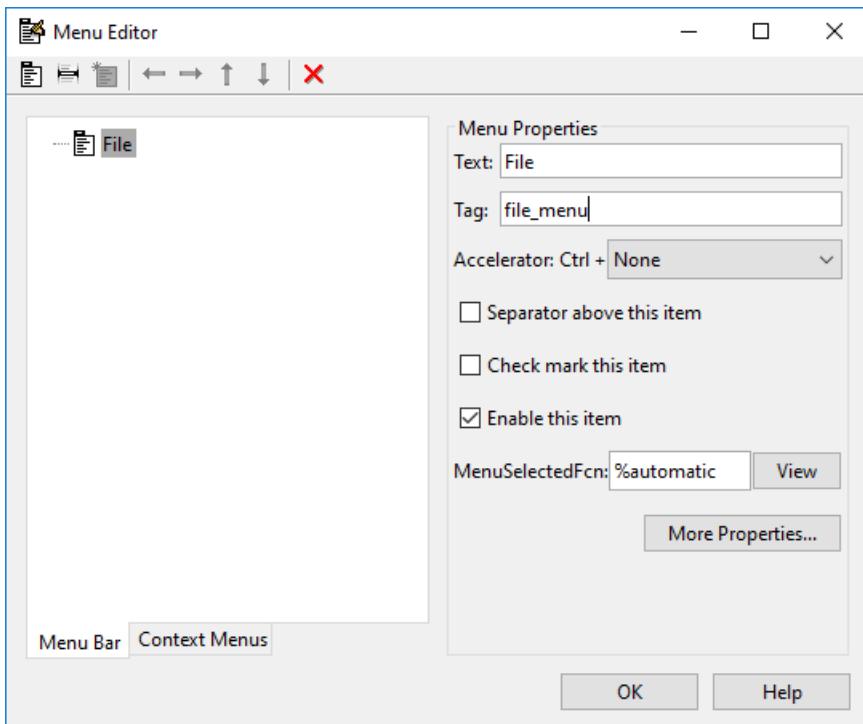
Create a Menu

- 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 **Text** and **Tag** fields for the menu. For example, set **Text** to **File** and set **Tag** to **file_menu**. Click outside the field for the change to take effect.

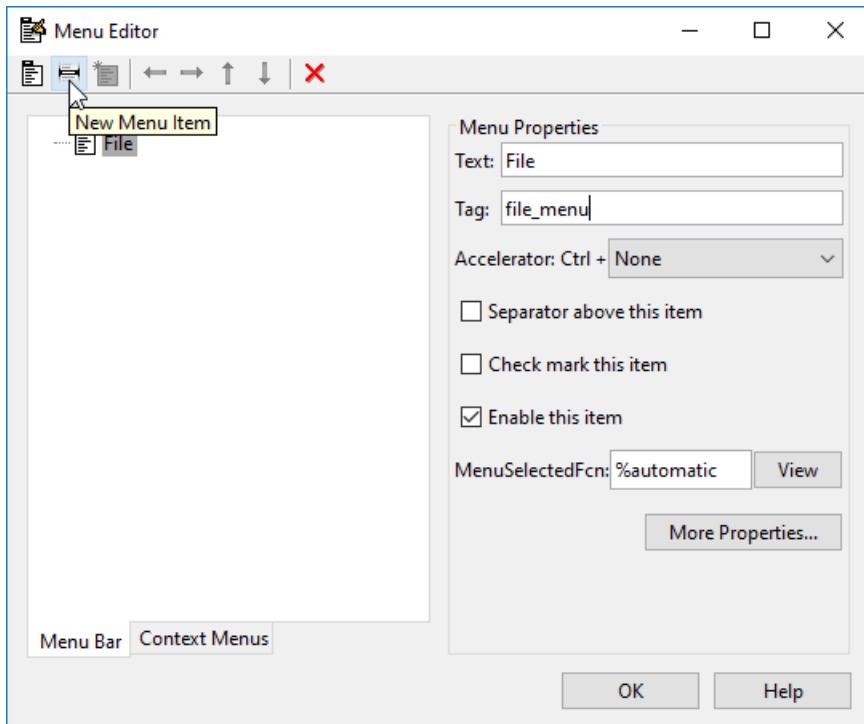
Text is a text label for the menu item. To display the & character in a label, use two & characters. The words **remove**, **default**, and **factory** (case sensitive) are reserved. To use one of these as labels, prepend a backslash character (\). For example, \remove yields **remove**.

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

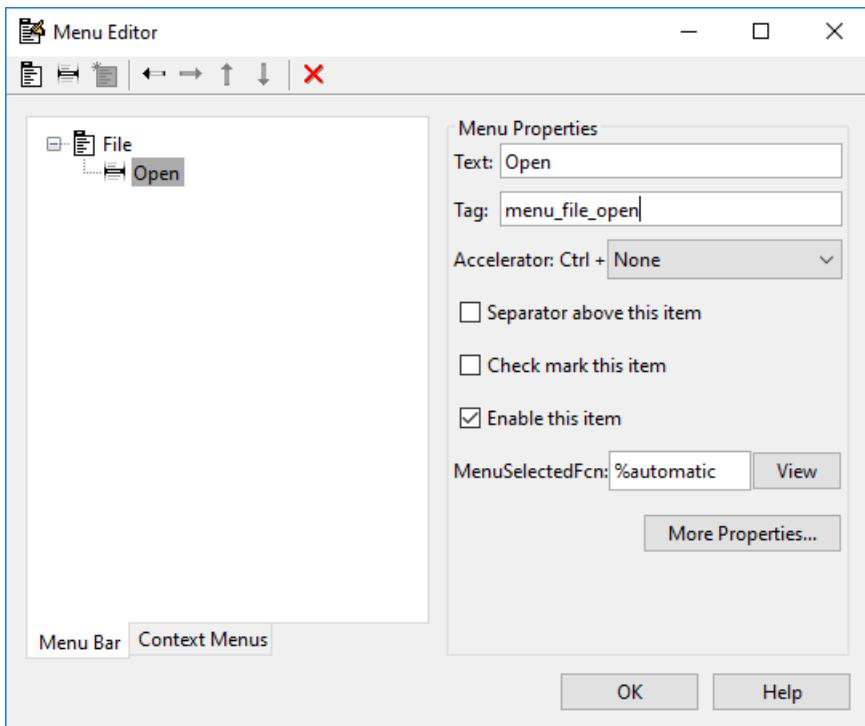
Add Items to a Menu

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 **Text** and **Tag** fields for the new menu item. For example, set **Text** 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 6-90.
- 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 the **Callback** function that executes when the users selects 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 7-22 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.

Note See “Menu Item” on page 7-22 and “How to Update a Menu Item Check” on page 7-24 for programming information and basic examples.

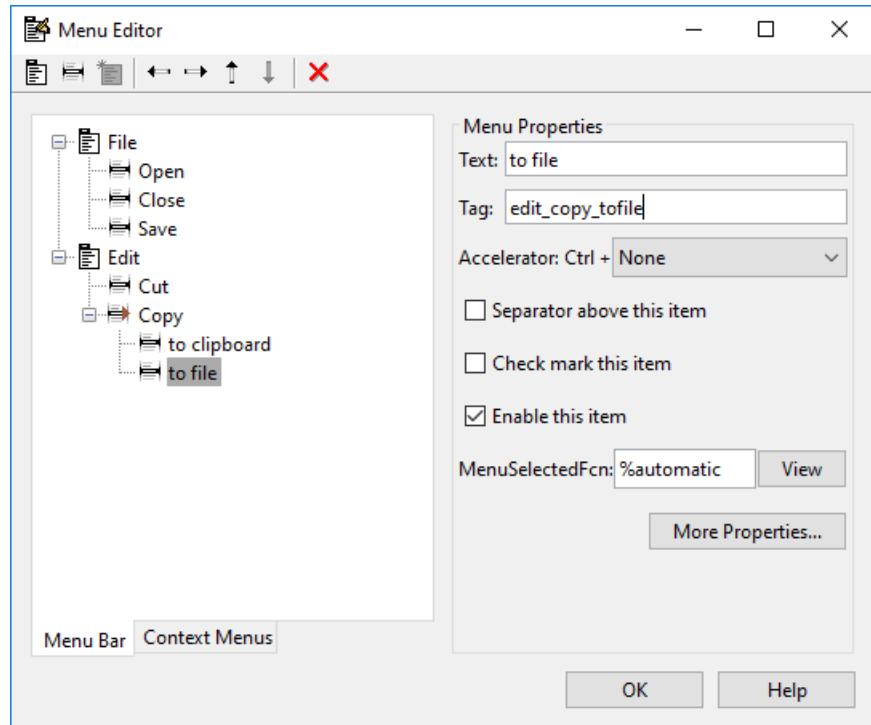
Additional Drop-Down Menus

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

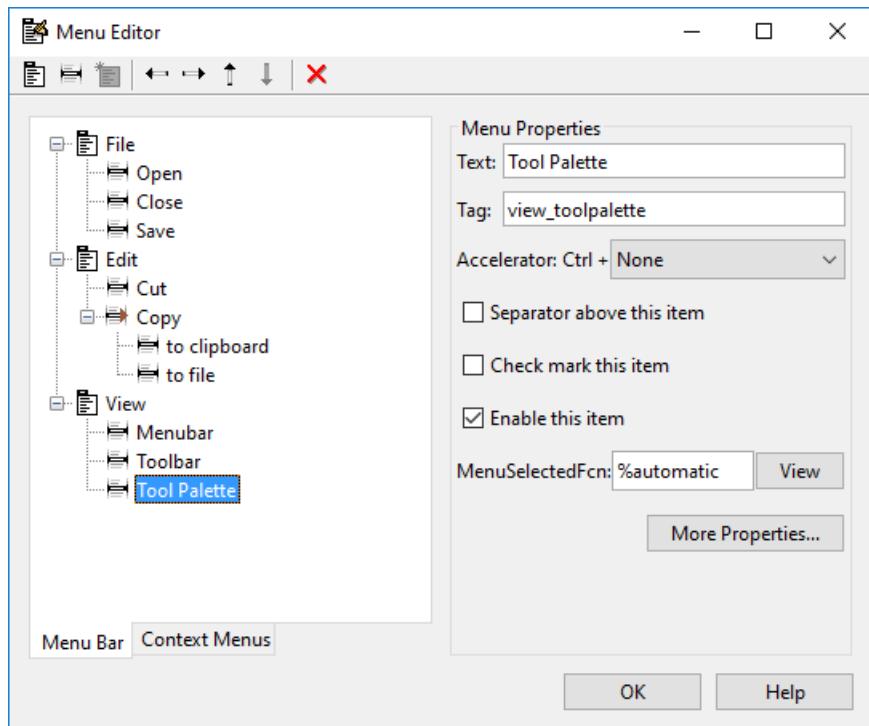
To create a cascading menu, select the menu item that will be the title for the cascading menu, then click the **New Menu Item** button. In the example below, **Edit** is a cascading menu.

6 Lay Out a UI Using GUIDE

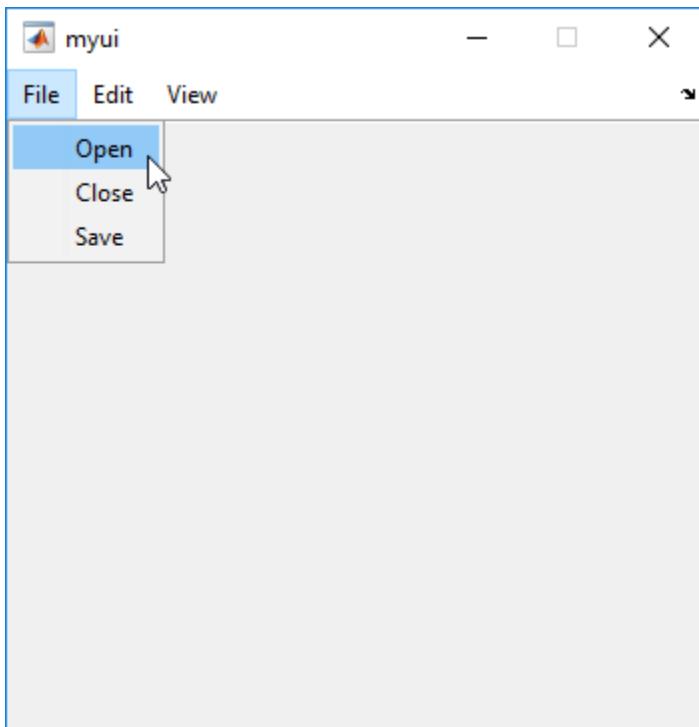


Note See “Menu Item” on page 7-22 for information about programming menu items.

The following Menu Editor illustration shows three menus defined for the figure menu bar.



When you run the app, 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:

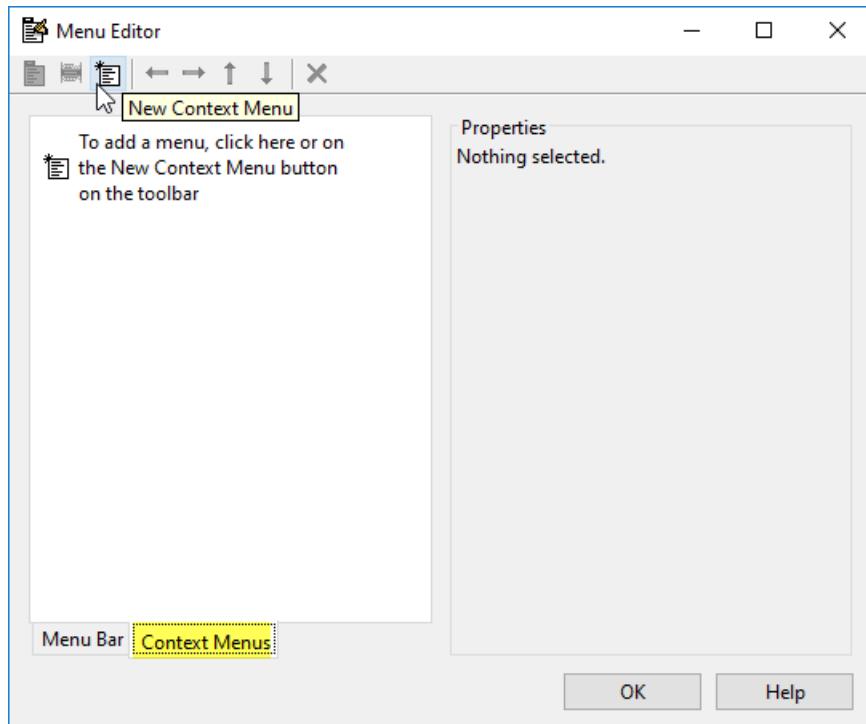
- 1 “Create the Parent Menu” on page 6-89
- 2 “Add Items to the Context Menu” on page 6-90
- 3 “Associate the Context Menu with an Object” on page 6-93

Note See “Menus for the Menu Bar” on page 6-78 for information about defining menus in general. See “Menu Item” on page 7-22 for information about defining local callback functions for your menus.

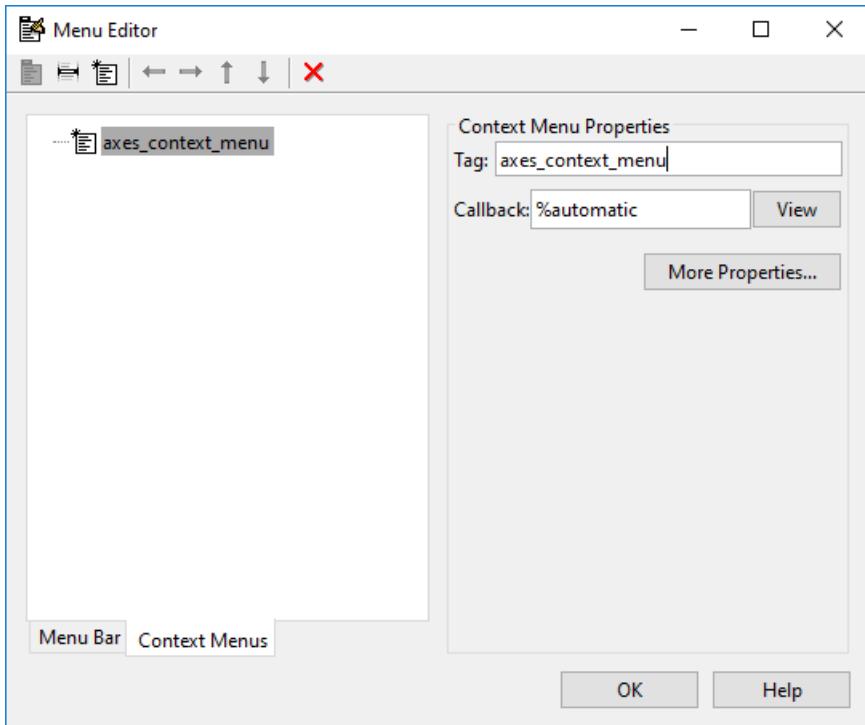
Create the Parent Menu

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.



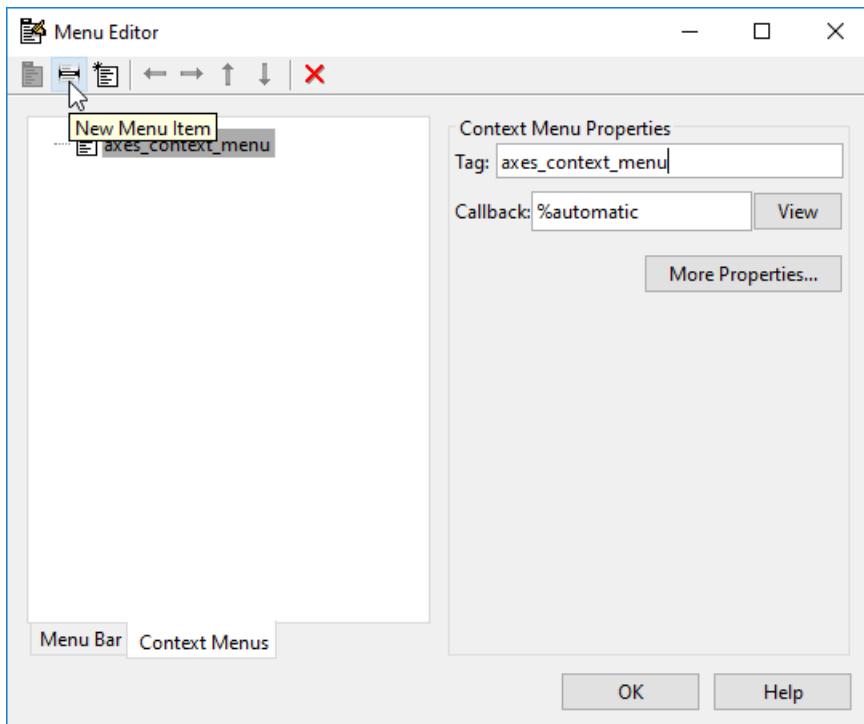
- 2 Select the menu, and in the **Tag** field type the context menu tag (axes_context_menu in this example).



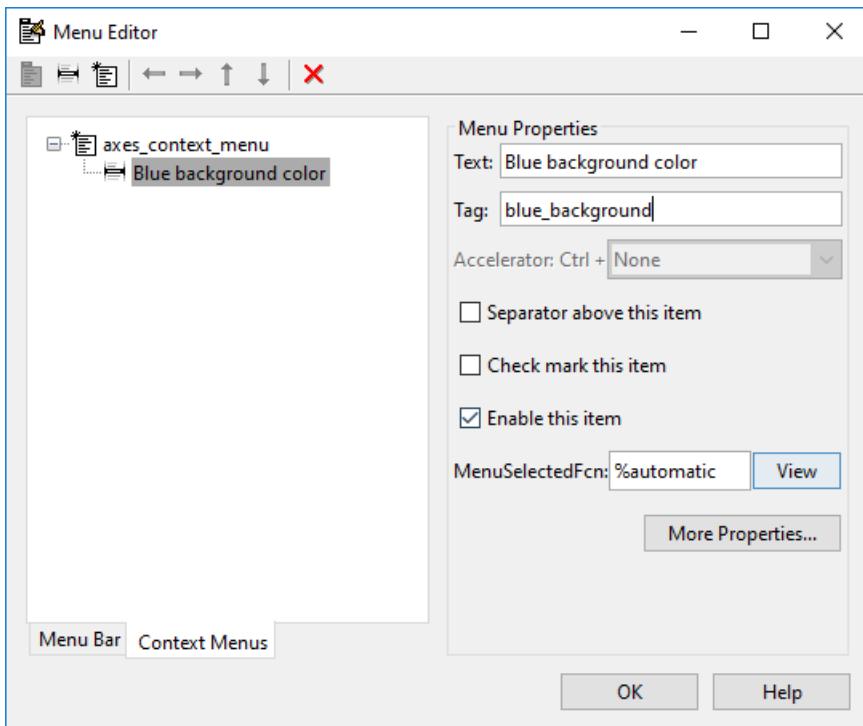
Add Items to the Context Menu

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 **Text** and **Tag** fields for the new menu item. For example, set **Text** 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 6-90. See “How to Update a Menu Item Check” on page 7-24 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 a command into the **Callback** field. It can be any valid MATLAB expression or command. For example, this command

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

See “Menu Item” on page 7-22 for more information about specifying this field and for programming menu items. For another example of programming context menus in GUIDE, see “GUIDE App Containing Tables and Plots” on page 8-12.

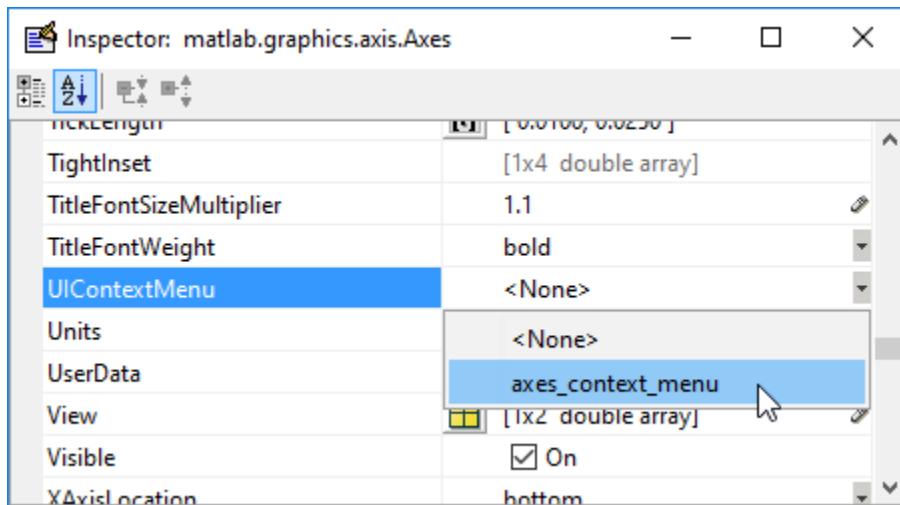
The **View** button displays the callback, if there is one, in an editor. If you have not yet saved the UI, GUIDE prompts you to save it.

- 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`.

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 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 7-22 for information on defining the syntax.

Note See “Menu Item” on page 7-22 and “How to Update a Menu Item Check” on page 7-24 for programming information and basic examples.

See Also

Related Examples

- “Write Callbacks in GUIDE” on page 7-2
- “Callbacks for Specific Components” on page 7-12
- “Create Toolbars for GUIDE UIs” on page 6-95

Create Toolbars for GUIDE UIs

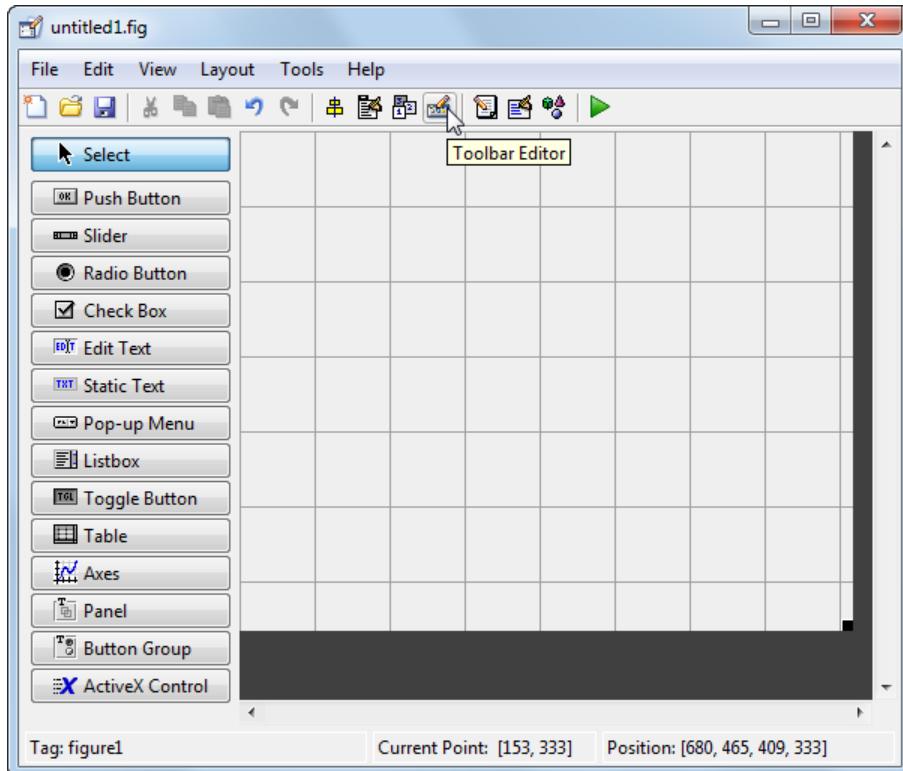
In this section...

[“Toolbar and Tools” on page 6-95](#)

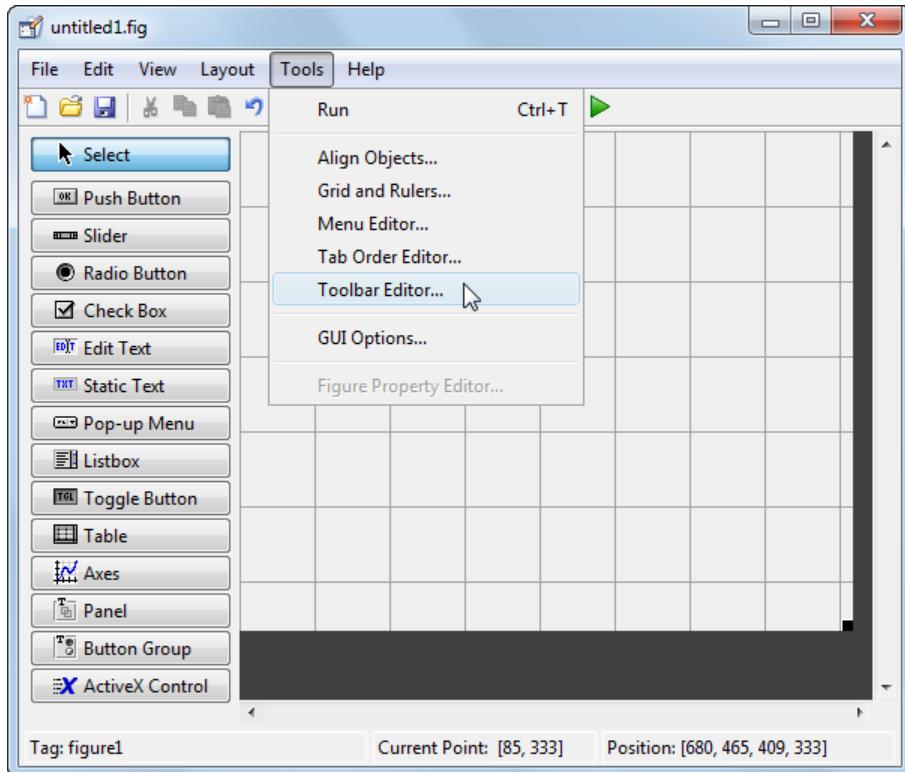
[“Editing Tool Icons” on page 6-103](#)

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 `uifToolbar`, `uipushbutton`, and `uitogglebutton` 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

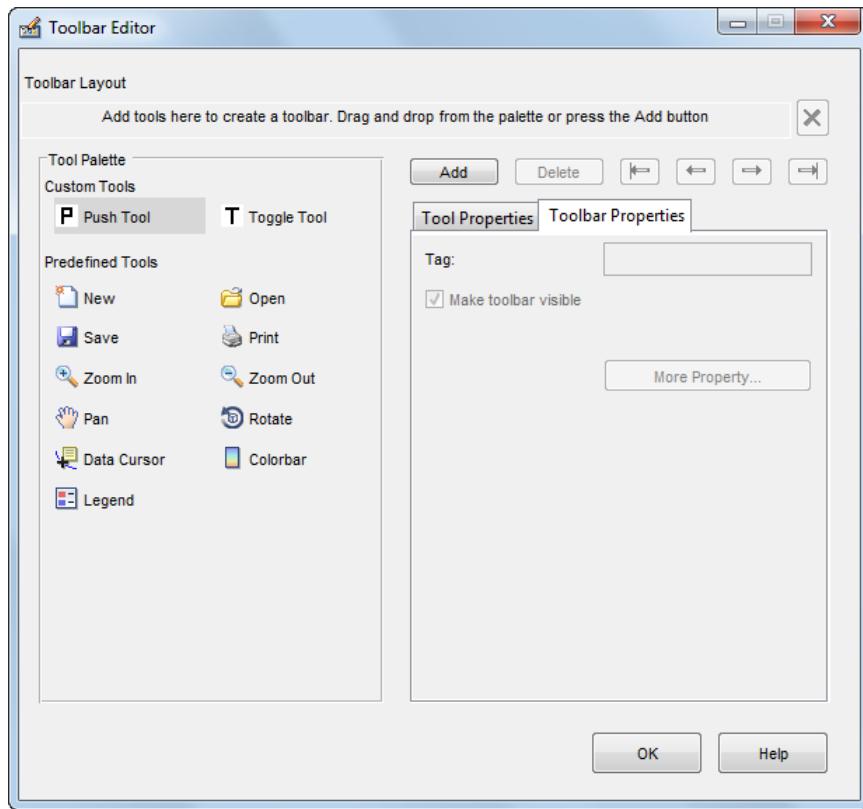
If you later want to remove the figure toolbar, set the **ToolBar** property to 'auto' and resave the UI. Doing this will not remove or hide your custom toolbar. See "Create Toolbars for Programmatic Apps" on page 9-51 for more information about making toolbars manually.

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 6-79.

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

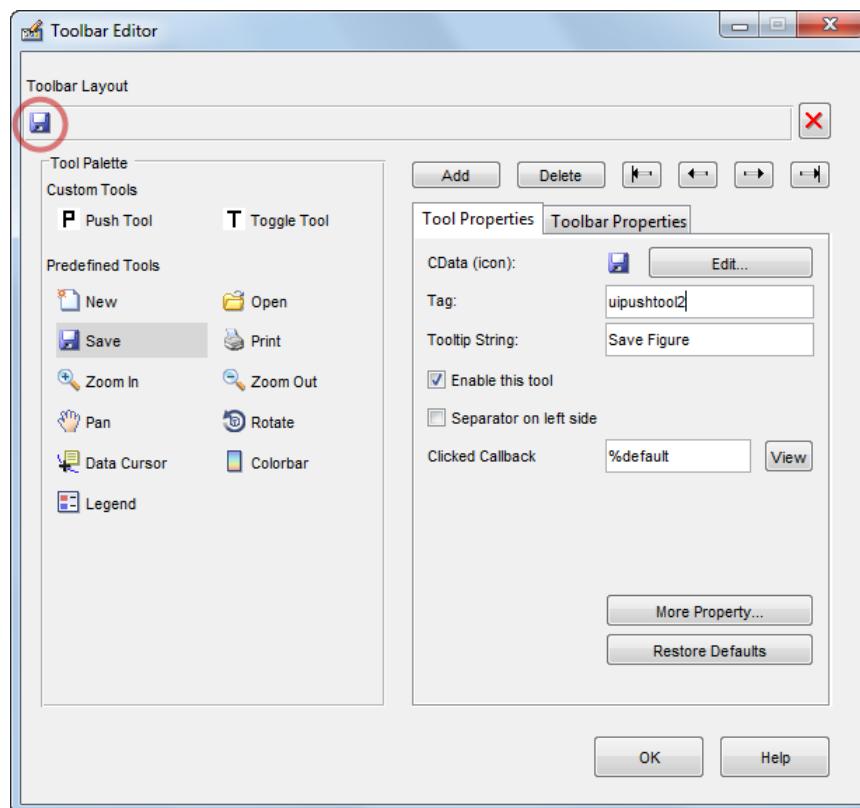
You can add a tool to a toolbar in three ways:

- 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.
- 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 6-104.

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). There 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 6-101 without removing the toolbar itself, your UI will contain an empty toolbar.

Close the Toolbar Editor

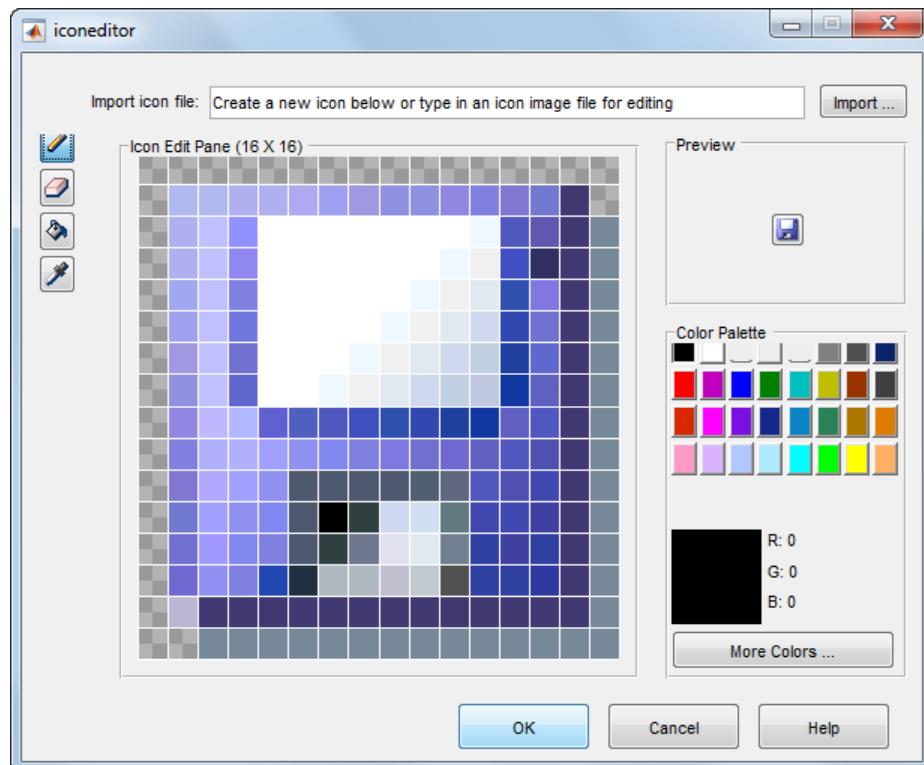
You can close the Toolbar Editor window in two ways:

- 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:

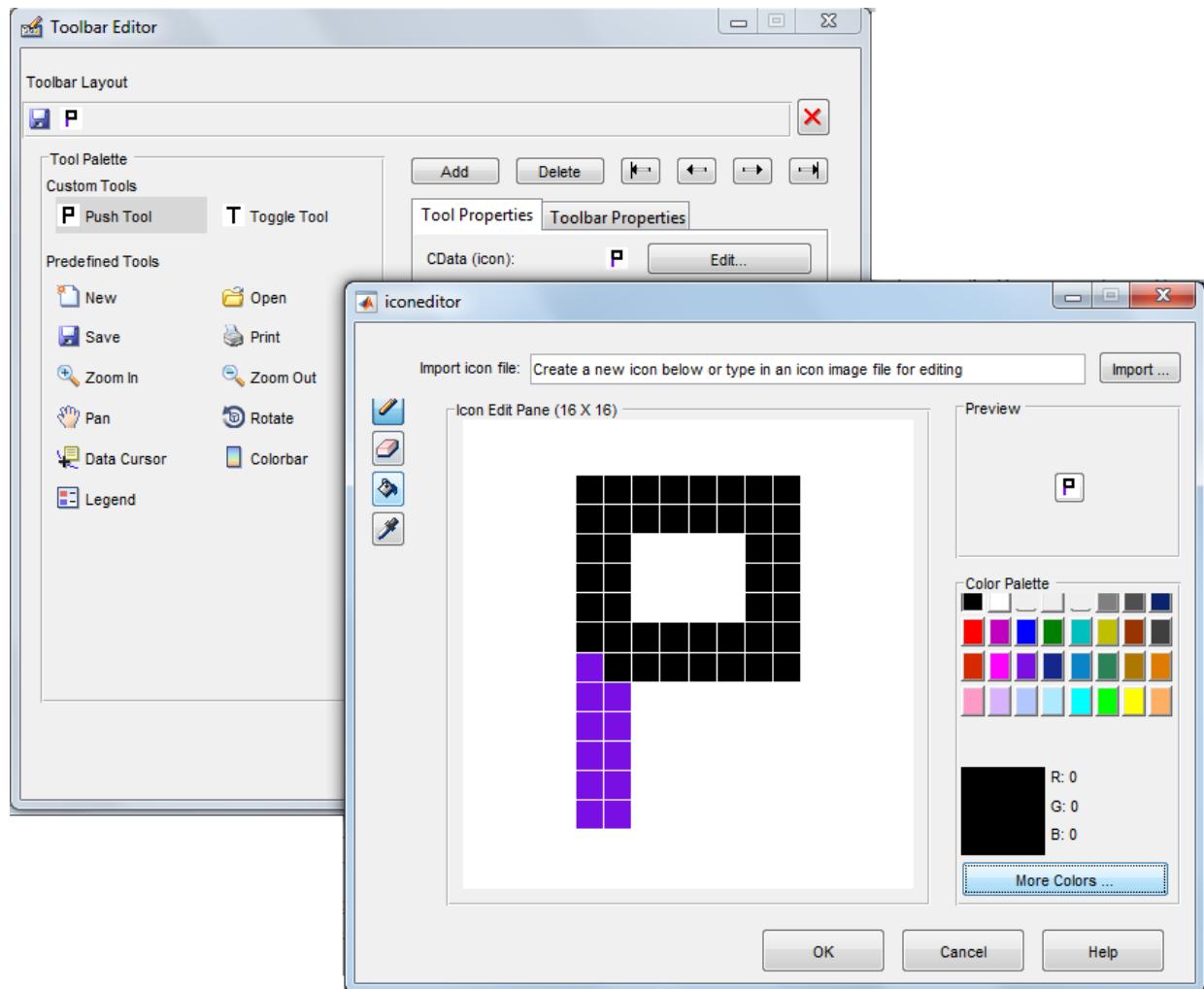
- **Icon file name** — The icon image file to be loaded for editing
- **Import** button — Opens a file dialog to select an existing icon file for editing
- Drawing tools — A group of four tools on the left side for editing icons
 - Pencil tool — Color icon pixels by clicking or dragging
 - Eraser tool — 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 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.



See Also

Related Examples

- “Write Callbacks in GUIDE” on page 7-2
- “Callbacks for Specific Components” on page 7-12
- “Create Menus for GUIDE Apps” on page 6-78

Design Cross-Platform UIs in GUIDE

In this section...

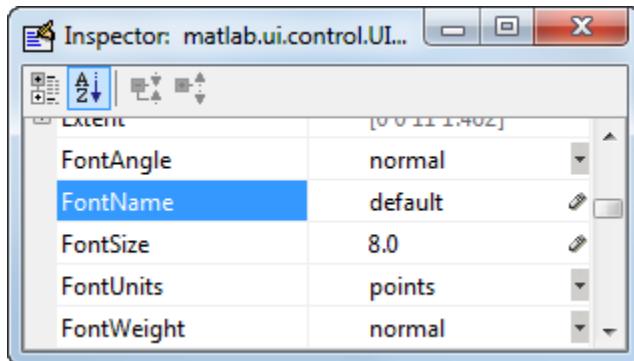
- “Default System Font” on page 6-107
- “Standard Background Color” on page 6-108
- “Cross-Platform Compatible Units” on page 6-108

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 `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 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 `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(root, '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

The default **Units** property might change if you change the resize behavior using **Tools** > **GUI Options**. This table lists the default units for each of the **Resize behavior** options.

Resize Behavior	Default Units for Figure	Default Units for Other Components
Non-resizable	characters	characters
Proportional	characters	normalized
Other (Use <code>SizeChangedFcn</code>)	characters	characters

At times it might be convenient to use other units, such as **inches** or **centimeters**. However, to preserve the look of your UI on different computers, remember to change the figure **Units** property back to the default units after completing your layout calculations.

For more information on the resize behavior options, see “GUIDE Options” on page 5-8.

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

Programming a GUIDE App

- “Write Callbacks in GUIDE” on page 7-2
- “Initialize UI Components in GUIDE Apps” on page 7-8
- “Callbacks for Specific Components” on page 7-12
- “Examples of GUIDE Apps” on page 7-29

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 character vector containing a MATLAB expression. Setting this property makes your app respond when the user interacts with the uicontrol. If the `Callback` property has no specified value, then nothing happens when the 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
<code>ButtonDownFcn</code>	End user presses a mouse button while the pointer is on the component or figure.	<code>axes</code> , <code>figure</code> , <code>uibuttongroup</code> , <code>uicontrol</code> , <code>uipanel</code> , <code>uitable</code> ,
<code>Callback</code>	End user triggers the component. For example: selecting a menu item, moving a slider, or pressing a push button.	<code>uicontextmenu</code> , <code>uicontrol</code> , <code>uimenu</code>
<code>CellEditCallback</code>	End user edits a value in a table whose cells are editable.	<code>uitable</code>
<code>CellSelectionCallback</code>	End user selects cells in a table.	<code>uitable</code>

Callback Property	User Action	Components That Use This Property
ClickedCallback	End user clicks the push tool or toggle tool with the left mouse button.	uitoggletool, uipushtool
CloseRequestFcn	The figure closes.	figure
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
SelectionChangedFcn	End user selects a different radio button or toggle button within a button group.	uibuttongroup
WindowButtonDownFcn	End user presses a mouse button while the pointer is in the figure window.	figure

Callback Property	User Action	Components That Use This Property
WindowButtonMotionFcn	End user moves the pointer within the figure window.	figure
WindowButtonUpFcn	End user releases a mouse button.	figure
WindowKeyPressFcn	End user presses a key while the pointer is on the figure or any of its child objects.	figure
WindowKeyReleaseFcn	End user releases a key while the pointer is on the figure or any of its child objects.	figure
WindowScrollWheelFcn	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 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 the name of the callback function. The last three arguments are provided by MATLAB, 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` — 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 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
<code>WindowKeyPressFcn</code>	<code>figure</code>
<code>WindowKeyReleaseFcn</code>	<code>figure</code>
<code>WindowScrollWheel</code>	<code>figure</code>
<code>KeyPressFcn</code>	<code>figure, uicontrol, uitable</code>
<code>KeyReleaseFcn</code>	<code>figure, uicontrol, uitable</code>
<code>SelectionChangedFcn</code>	<code>uibuttongroup</code>

Callback Property Name	Component
CellEditCallback	uitable
CellSelectionCallback	

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, 'pushbutton1_Callback' to the new function name.

- 3 Change all other references to the old function name to the new function name in the 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.

See Also

Related Examples

- “Callbacks for Specific Components” on page 7-12
- “Anonymous Functions”
- “Share Data Among Callbacks” on page 11-2

Initialize UI Components in GUIDE Apps

In this section...

[“Opening Function” on page 7-8](#)

[“Output Function” on page 7-10](#)

Opening Function

The opening function is the first callback in every GUIDE 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)
% uwait(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 on page 6-8 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 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;
```

See Also

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 app, copy the code from the example’s function body into your function’s body.

If you are creating an app 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 'Goodbye' in the Command Window. The next line gets the UI window using `gcf` and then closes it.

Toggle Button

This code is 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 7-21 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.

```
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.

```
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 user types characters inside the text field and presses the **Enter** key, the callback function retrieves those characters and displays them 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 an app 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 an app 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 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 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 App in GUIDE” on page 8-16 shows how to populate a list box with directory names.

Pop-Up Menu

Populate Items in the Pop-Up Menu

If you are developing an app 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.

```
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 an app programmatically (without GUIDE), then populate the pop-up menu when you create it. For example:

```
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 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 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 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 the UI window 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 app perform specific tasks when the panel resizes.

This code is an example of a panel's **SizeChangedFcn** callback in a GUIDE app. When the 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 contains 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.

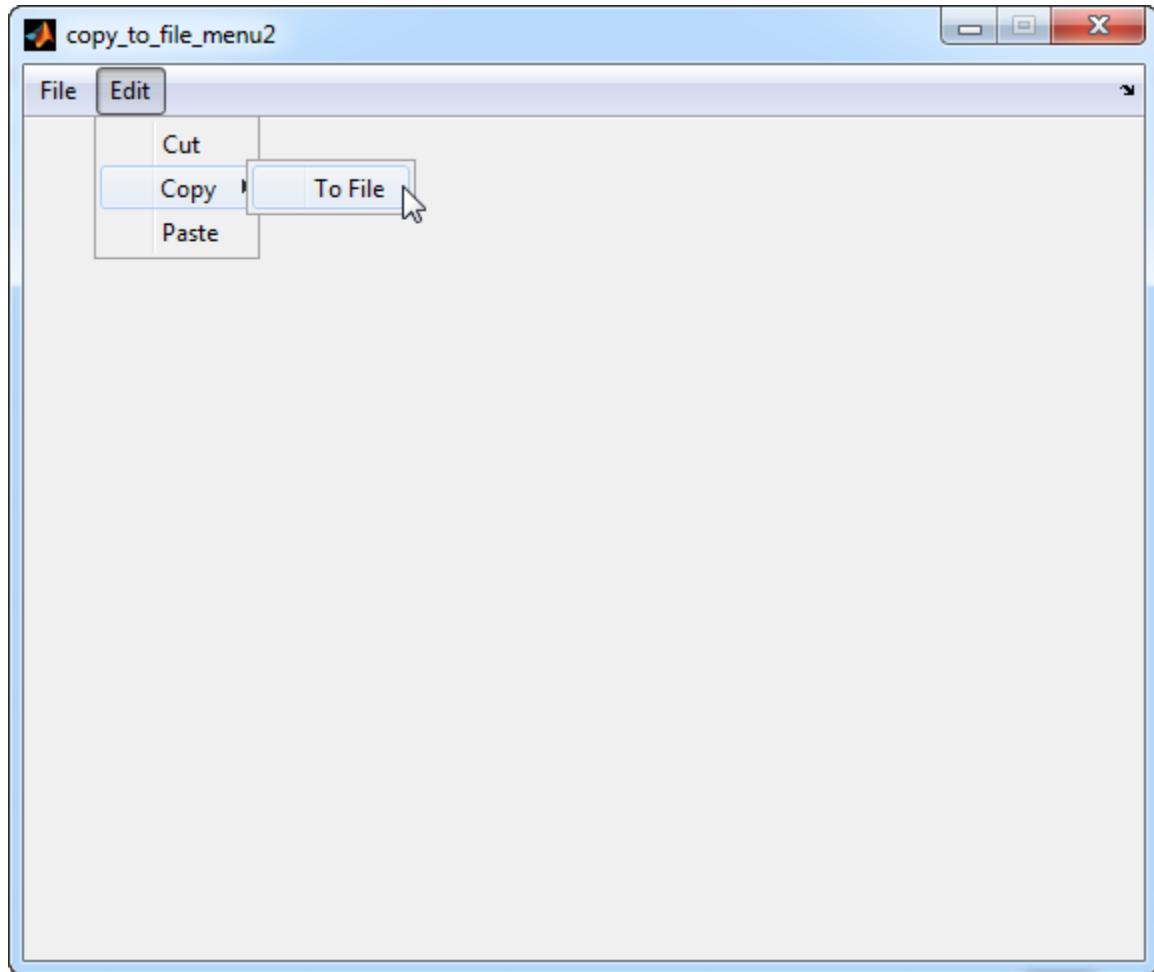
```
function uibuttongroup1_SelectionChangedFcn(hObject, eventdata, handles)
% hObject    handle to the selected object in uibuttongroup1
% eventdata  structure with the following fields
%     EventName: string 'SelectionChanged' (read only)
%     OldValue: handle of the previously selected object or 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 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.



```
% -----
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 code, 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.

```

if strcmp(get(hObject,'Checked'), 'on')
    set(hObject,'Checked','off');
else
    set(hObject,'Checked','on');
end

```

The `strcmp` function compares two character vectors and returns `true` when they match. In this case, it returns `true` when the menu item's `Checked` property matches the character vector, '`on`'.

See “Create Menus for GUIDE Apps” on page 6-78 for more information about creating menu items in GUIDE. See “Create Menus for Programmatic Apps” on page 9-38 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.

```

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.

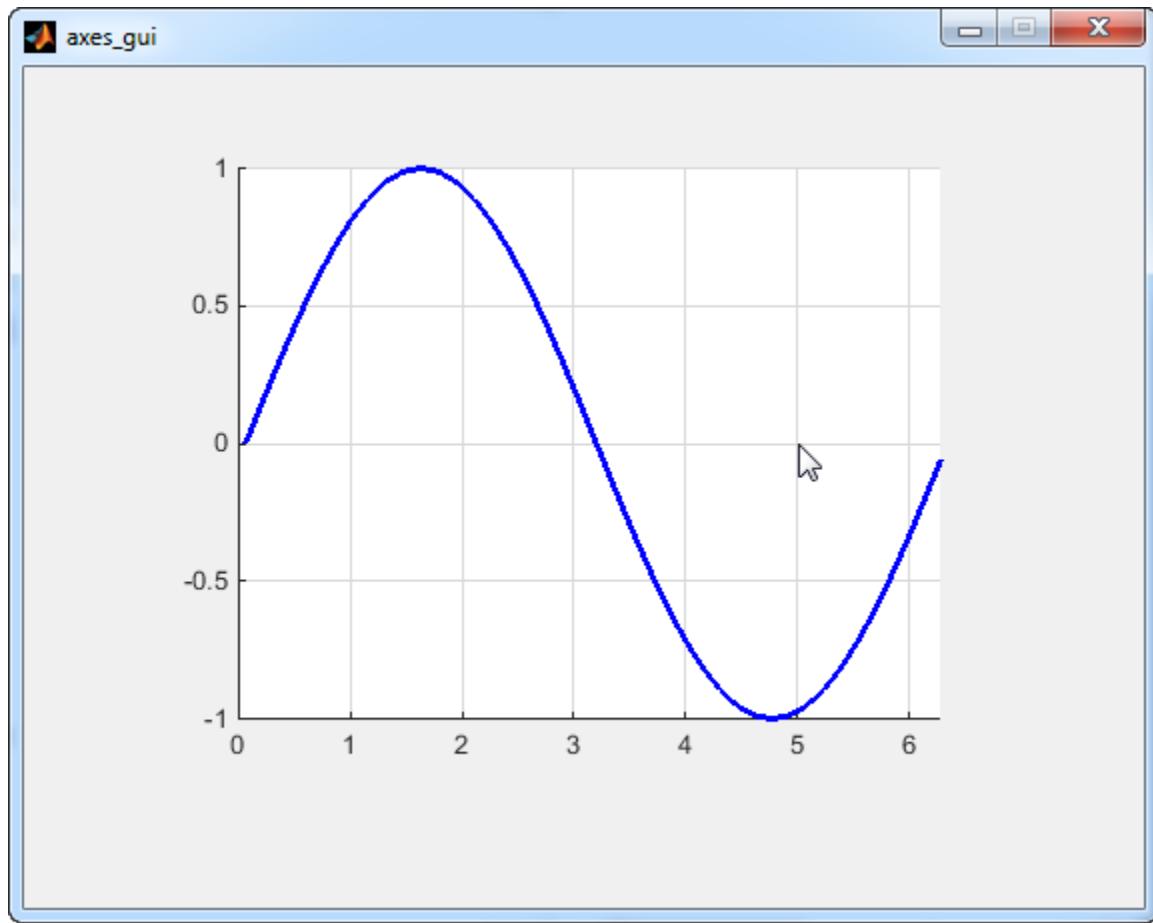
This code is an example of the table callback function, `CellEditCallback`. Associate this function with the table `CellEditCallback` property to make it execute when the end user edits a cell in the table.

```
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.



```
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`.

See Also

Related Examples

- “Write Callbacks in GUIDE” on page 7-2
- “Write Callbacks for Apps Created Programmatically” on page 10-5

Examples of GUIDE Apps

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
- “GUIDE App With Parameters for Displaying Plots” on page 8-7
- “GUIDE App Containing Tables and Plots” on page 8-12
- “Interactive List Box App in GUIDE” on page 8-16
- “Plot Workspace Variables in a GUIDE App” on page 8-21
- “Automatically Refresh Plot in a GUIDE App” on page 8-24

Examples of GUIDE UIs

- “Modal Dialog Box in GUIDE” on page 8-2
- “GUIDE App With Parameters for Displaying Plots” on page 8-7
- “GUIDE App Containing Tables and Plots” on page 8-12
- “Interactive List Box App in GUIDE” on page 8-16
- “Plot Workspace Variables in a GUIDE App” on page 8-21
- “Automatically Refresh Plot in a GUIDE App” on page 8-24

Modal Dialog Box in GUIDE

In this section...

["Create the Dialog Box" on page 8-2](#)

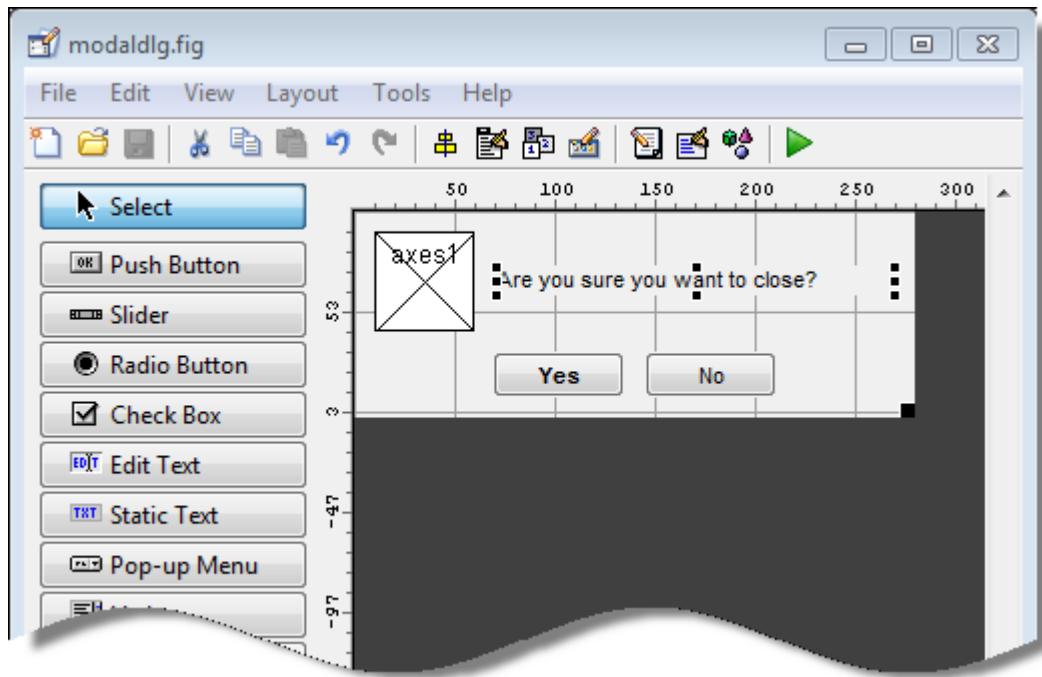
["Create the Program That Opens the Dialog Box" on page 8-3](#)

["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**.

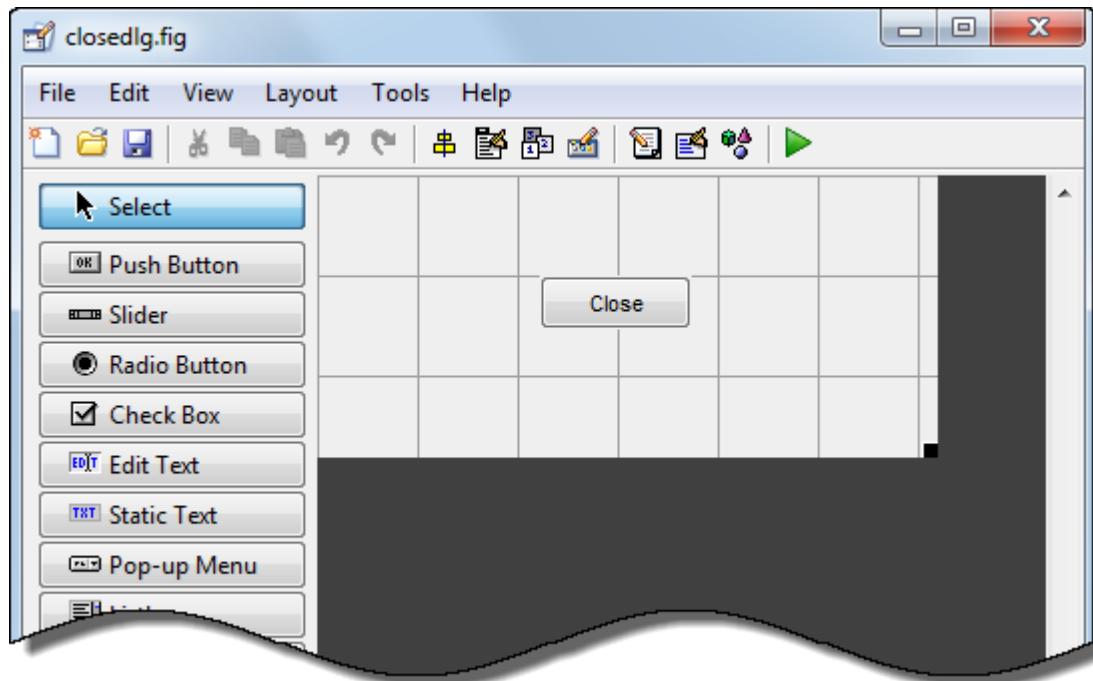


- 6 Select **File > Save As**.
- 7 In the Save As dialog box, in the **File name** field, type **modaldlg.fig**.

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');
```

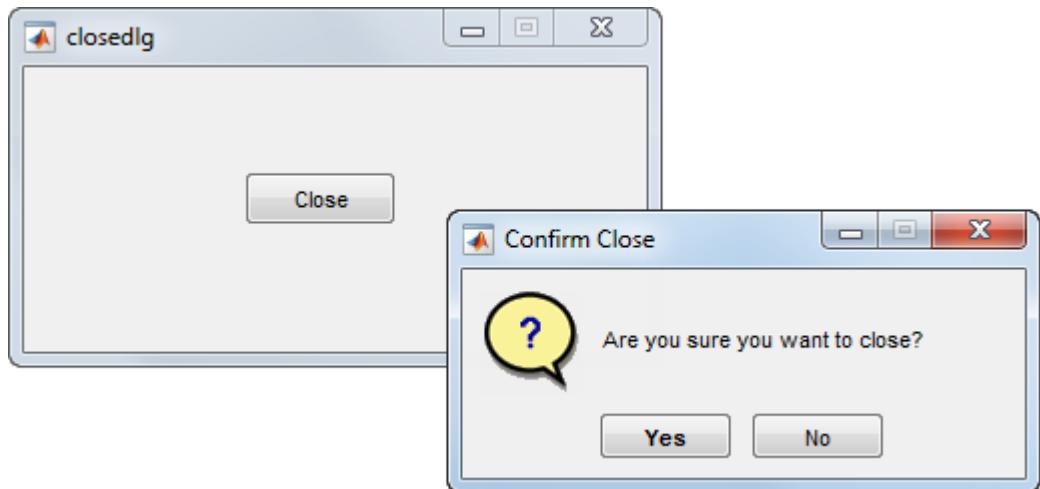
Recall that the **modaldlg** function is coded in the other program file, **modaldlg.m**. That function displays a second window: the Confirm Close dialog box. The return argument, **user_response**, is the user's selection from that dialog box.

The **switch** command decides whether to close the first window (**closedlg**) 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.

See Also

Related Examples

- “Create a Simple App Using GUIDE” on page 2-2
- “Dialog Boxes”
- “Write Callbacks in GUIDE” on page 7-2

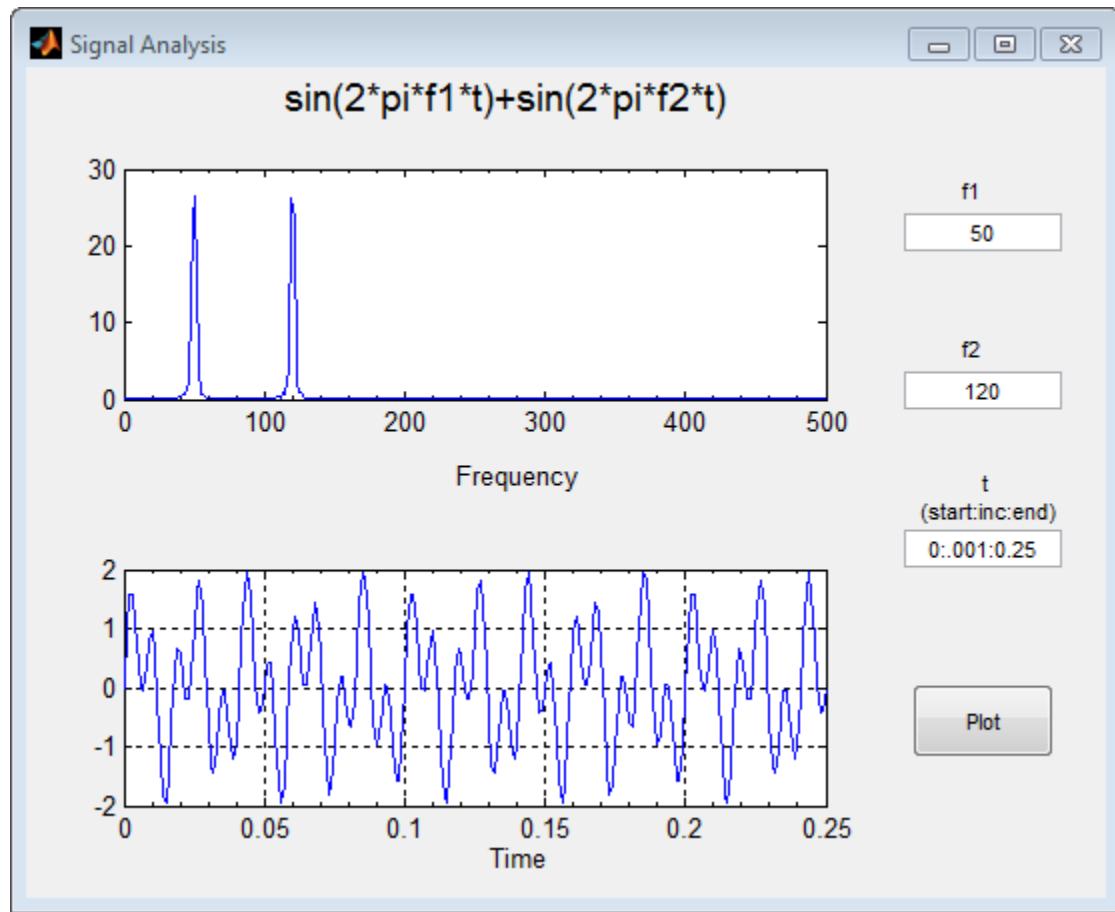
GUIDE App With Parameters for Displaying Plots

This example shows how to examine and run a prebuilt GUIDE app. The app contains three edit fields and two axes. The axes display the frequency and time domain representations of a function that is the sum of two sine waves. The top two edit fields contain the frequency for each component sine wave. The third edit field contains the time range and sampling rate for the plots.

Open and Run the Example

Open and run the app. Change the default values in the **f1** and **f2** fields to change the frequency for each component sine wave. You can also change the three numbers (separated by colons) in the **t** field. The first and last numbers specify the window of time to sample the function. The middle number specifies the sampling rate.

Press the **Plot** button to see the graph of the function in the frequency and time domains.



Examine the Code

- 1 In GUIDE, click the **Editor** button  to view the code.
- 2 Near the top of the Editor window, use the  **Go To**  button to navigate to the functions discussed below.

f1_input_Callback and f2_input_Callback

The **f1_input_Callback** function executes when the user changes the value in the **f1** edit field. The **f2_input_Callback** function responds to changes in the **f2** field, and it is almost identical to the **f1_input_Callback** function. Both functions check for valid user input. If the value in the edit field is invalid, the **Plot** button is disabled. Here is the code for the **f1_input_Callback** function.

```
f1 = str2double(get(hObject,'String'));
if isnan(f1) || ~isreal(f1)
    % 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
```

t_input_Callback

The **t_input_Callback** function executes when the user changes the value in the **t** edit field. This **try** block checks the value to make sure that it is numeric, that its length is between 2 and 1000, and that the vector is monotonically increasing.

```
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
        % Enable the Plot button with its original name
        set(handles.plot_button,'String','Plot')
```

```
    set(handles.plot_button, 'Enable', 'on')
    return
end

catch EM
    % Cannot evaluate expression user typed
    set(handles.plot_button, 'String', 'Cannot plot t');
    uicontrol(hObject);
end
```

The `catch` block changes the label on the **Plot** button to indicate that an input value was invalid. The `uicontrol` command sets the focus to the field that contains the erroneous value.

plot_button_Callback

The `plot_button_Callback` function executes when the user clicks the **Plot** button.

First, the callback gets the values in the three edit fields:

```
f1 = str2double(get(handles.f1_input, 'String'));
f2 = str2double(get(handles.f2_input, 'String'));
t = eval(get(handles.t_input, 'String'));
```

Then callback uses values of `f1`, `f2`, and `t` to sample the function in the time domain and calculate the Fourier transform. Then, the two plots are updated:

```
% 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
```

See Also

Related Examples

- “Create a Simple App Using GUIDE” on page 2-2

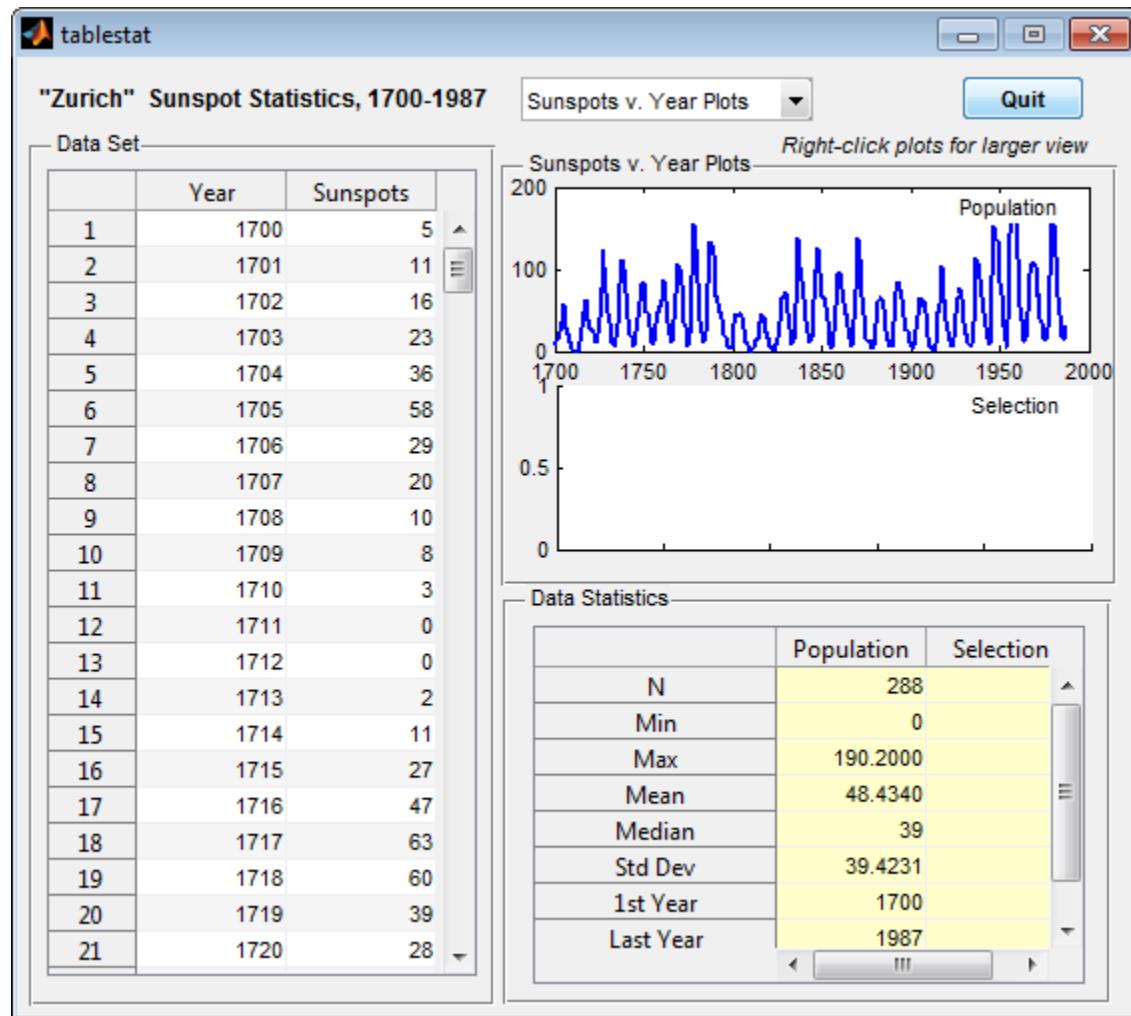
- “Write Callbacks in GUIDE” on page 7-2
- “Share Data Among Callbacks” on page 11-2

GUIDE App Containing Tables and Plots

This example shows how to examine and run a prebuilt GUIDE app. The app contains two tables, two axes, and a pop-up menu. The larger table on the left displays 288 entries of sunspot data. The top axes displays the graph of all 288 entries. When you select 11 or more items from the table on the left, the graph of the selected entries displays in the bottom axes. The table in the lower right corner displays a statistical summary of the sunspot data. The pop-up menu at the top of the window allows you to toggle between graphs in the time and frequency domains.

Open and Run the Example

Open and run the app. Select 11 or more rows in the **Data Set** table to see a plot of those points on the bottom set of axes. As you modify your selection, the numbers in the second column of the **Data Statistics** table update.



Examine the Code

- 1 In GUIDE, click the **Editor** button to view the code.
- 2 Near the top of the Editor window, use the **Go To** button to navigate to the functions discussed below.

plot_type_Callback

The `plot_type_Callback` function executes when the user changes the selection in the pop-up menu at the top of the window. The following statements get the currently selected menu item and update the label above the axes.

```
index = get(hObject, 'Value');
strlist = get(hObject, 'String');
set(handles.uipanel3, 'Title', strlist(index))
```

These commands get all 288 entries in the table and plot them in the top axes. The `refreshDisplays` function is a locally defined function.

```
table = get(handles.data_table, 'Data');
refreshDisplays(table, handles, 1);
```

These commands update the bottom plot and the statistical summary table if more than 10 entries are selected.

```
selection = handles.currSelection;
if length(selection) > 10
    refreshDisplays(table(selection,:), handles, 2)
else
    % Do nothing; insufficient observations for statistics
end
```

data_table_CellSelectionCallback

The `data_table_CellSelectionCallback` function executes when the user selects any of the cells in the larger table on the left. This command gets the currently selected entries in the table:

```
selection = eventdata.Indices(:,1);
```

These commands update the `currSelection` field of the `handles` structure so that the user's selection can be accessed from within other callbacks such as the `plot_type_Callback` function.

```
handles.currSelection = selection;
guidata(hObject, handles);
```

Finally, `refreshDisplays` updates the bottom plot and the statistical summary table.

```
refreshDisplays(table(selection,:), handles, 2);
```

See Also

Related Examples

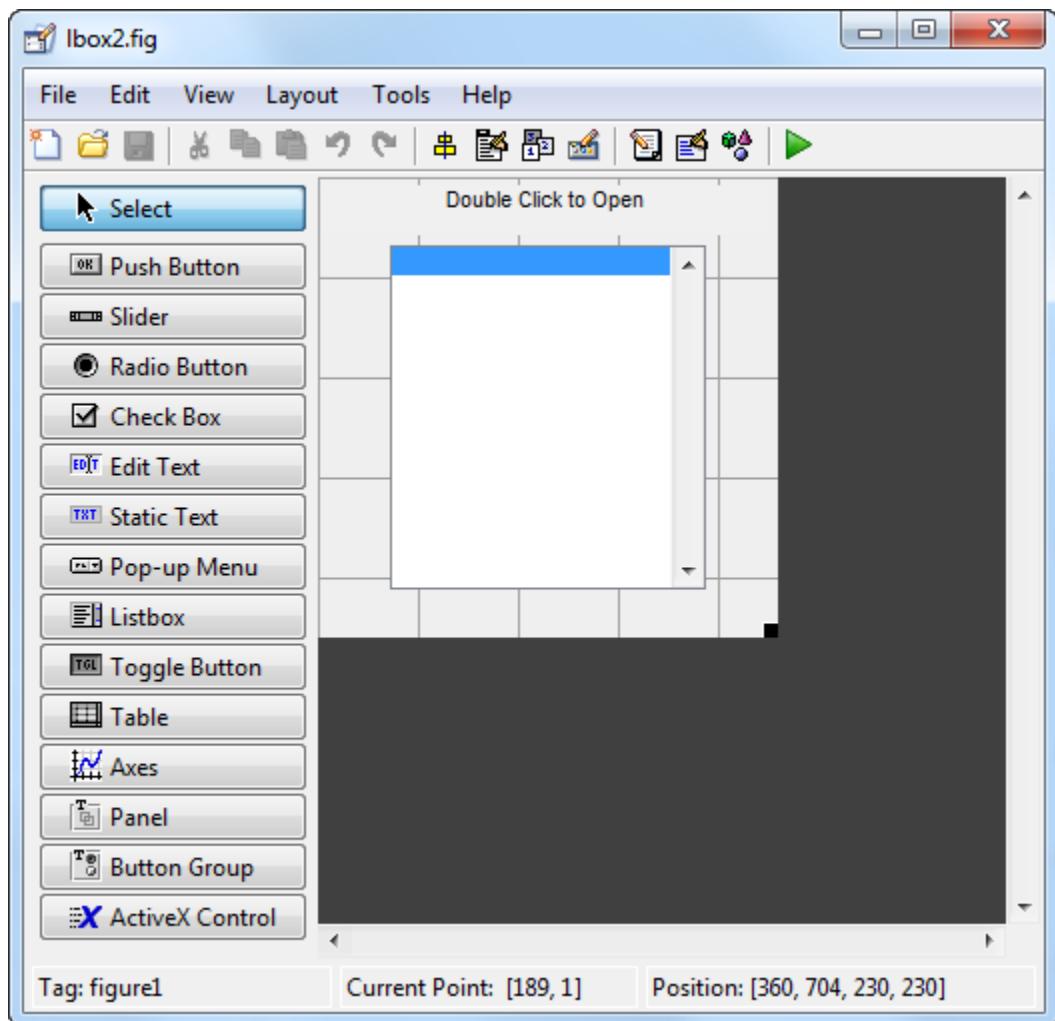
- “Create a Simple App Using GUIDE” on page 2-2
- “Write Callbacks in GUIDE” on page 7-2
- “Share Data Among Callbacks” on page 11-2

Interactive List Box App in GUIDE

This example shows how to examine and run a prebuilt GUIDE app. The app contains a list box that displays the files in a particular folder. When you double-click an item in the list, MATLAB opens the item.

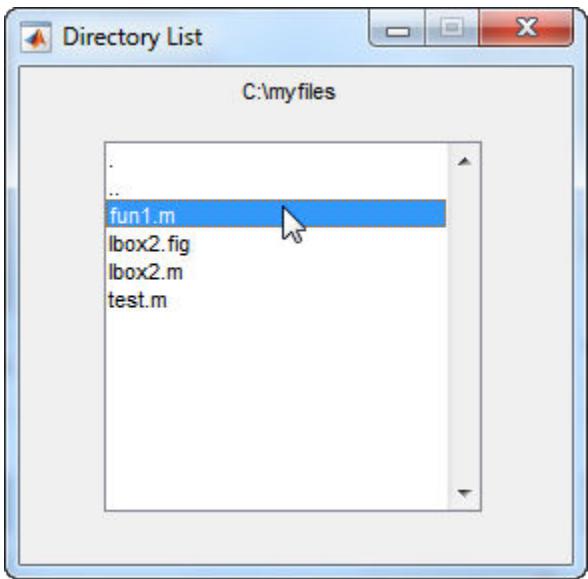
Open and Run The Example

Open the app in GUIDE, and click the **Run Figure** (green play button) to run it.



Alternatively, you can call the `lbox2` function in the Command Window with the '`dir`' name-value pair argument. The name-value pair argument allows you to list the contents of any folder. For example, this command lists the files in the `C:\` folder on a Windows® system:

```
lbox2('dir','C:\')
```



Note: Before you can call `lbox2` in the Command Window, you must save the GUIDE files in a folder on your MATLAB® path. To save the files, select **File > Save As** in GUIDE.

Examine the Layout and Callback Code

- 1 In GUIDE, click the **Editor** button  to view the code.
- 2 Near the top of the Editor window, use the  **Go To**  button to navigate to the functions discussed below.

lbox2_OpeningFcn

The callback function `lbox2_OpeningFcn` executes just before the list box appears in the UI for the first time. The following statements determine whether the user specified a path argument to the `lbox2` function.

```
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 must be a valid directory','Input Argument Error!')
        return
    end
else
    errordlg('Unrecognized input argument','Input Argument Error!');
    return;
end
end

```

If nargin==3, then the only input arguments to lbox2_OpeningFcn are hObject, eventdata, and handles. Therefore, the user did not specify a path when they called lbox2, so the list box shows the contents of the current folder. If nargin>4, then the varargin input argument contains two additional items (suggesting that the user did specify a path). Thus, subsequent if statements check to see whether the path is valid.

listbox1_callback

The callback function listbox1_callback executes when the user clicks a list box item. This statement, near the beginning of the function, returns true whenever the user double-clicks an item in the list box:

```
if strcmp(get(handles.figure1,'SelectionType'),'open')
```

If that condition is true, then listbox1_callback determines which list box item the user selected:

```

index_selected = get(handles.listbox1,'Value');
file_list = get(handles.listbox1,'String');
filename = file_list{index_selected};

```

The rest of the code in this callback function determines how to open the selected item based on whether the item is a folder, FIG file, or another type of file:

```

if handles.is_dir(handles.sorted_index(index_selected))
    cd (filename)
    load_listbox(pwd,handles)
else
    [path,name,ext] = fileparts(filename);
    switch ext
        case '.fig'
            guide (filename)
        otherwise
            try

```

```
open(filename)
catch ex
    errordlg(...
        ex.getReport('basic'), 'File Type Error', 'modal')
    end
end
```

See Also

Related Examples

- “Create a Simple App Using GUIDE” on page 2-2
- “Write Callbacks in GUIDE” on page 7-2
- “Share Data Among Callbacks” on page 11-2

Plot Workspace Variables in a GUIDE App

In this section...

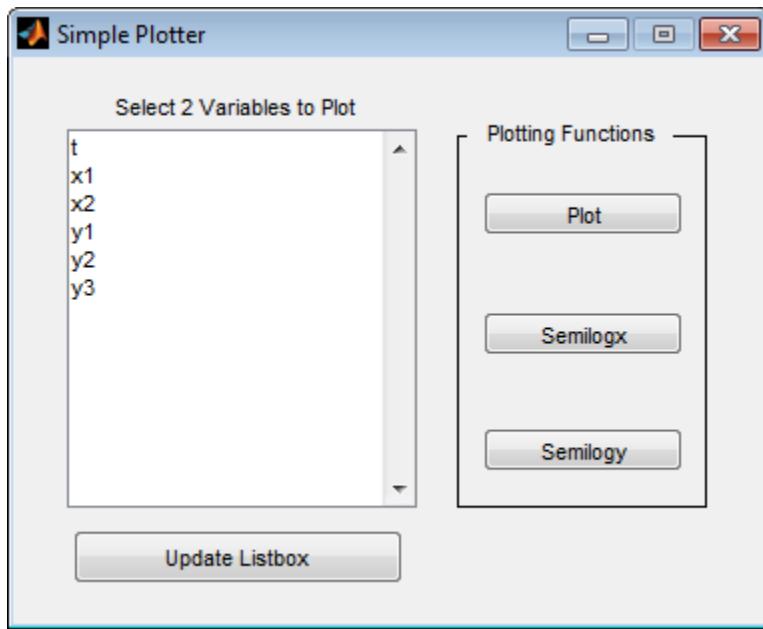
["Open and Run the App" on page 8-21](#)

["Examine the Code" on page 8-22](#)

This example shows how to examine and run a prebuilt GUIDE app. The app contains a list box that displays the variables in your MATLAB workspace. The button below the list box refreshes the list. The three buttons on the right plot the selected variables using different scales for the x and y axes.

Open and Run the App

Open and run the app. Select one variable in the list box, and then hold the **Ctrl** key to select a second variable. Then click **Plot**, **Semilogx**, or **Semilogy** to plot the variables.



Examine the Code

- 1 In GUIDE, click the **Editor** button to view the code.
- 2 Near the top of the Editor window, use the **Go To** button to navigate to the functions discussed below.

update_button_Callback

The **update_button_Callback** function executes when the user clicks the **Update Listbox** button. It contains one command that calls another local function, **update_listbox**. (That function is kept separate so it can be reused elsewhere in the app.)

The **update_listbox** function executes the **who** command in the MATLAB workspace to get the list of current variables. Then it sets the contents of the list box to that list of variables.

```
vars = evalin('base','who');  
set(handles.listbox1,'String',vars)
```

plot_button_Callback

The `plot_button_Callback` function executes when the user presses the **Plot** button. The callbacks for the **Semilogx** and **Semilogy** buttons contain most of the same code.

First, the function calls the local function `get_var_names`, which returns the two selected variables in the list.

```
[x,y] = get_var_names(handles);
```

Then it checks to make sure at least one variable is selected. If no variables are selected, the callback returns and does not plot anything.

```
if isempty(x) && isempty(y)
    return
end
```

Finally, the `plot` command executes from within the base workspace.

```
try
    evalin('base',['plot(',x,',',y,')'])
catch ex
    errordlg(ex.getReport('basic'),...
              'Error generating linear plot','modal')
end
```

The `catch` block presents an error dialog box if an error occurs.

See Also

Related Examples

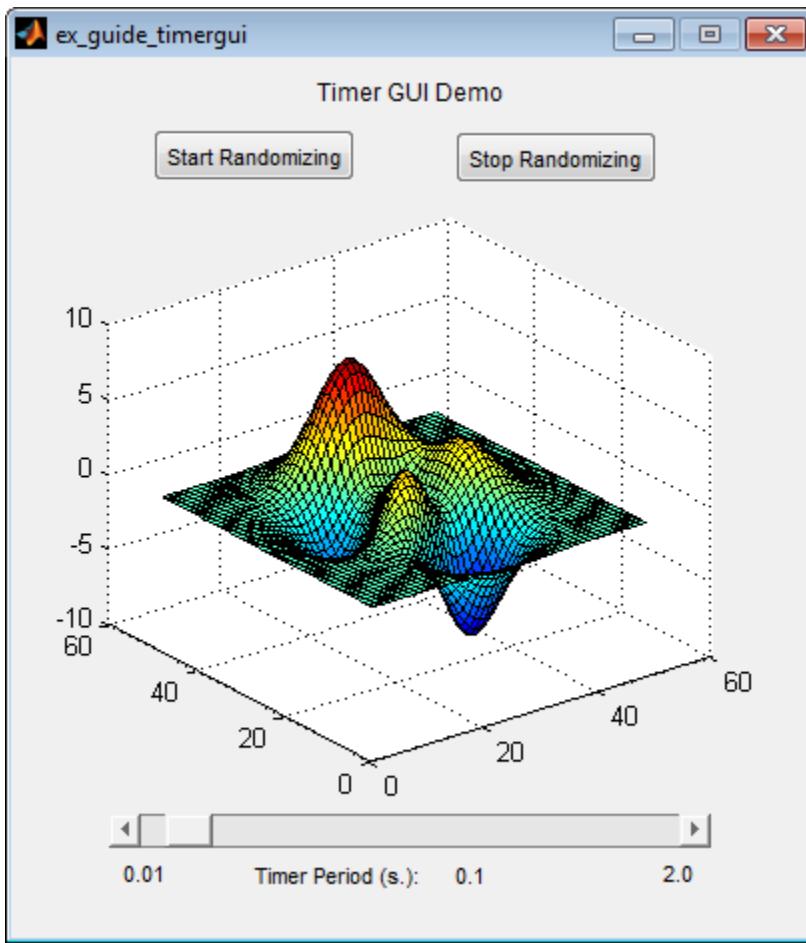
- “Create a Simple App Using GUIDE” on page 2-2
- “Write Callbacks in GUIDE” on page 7-2
- “Share Data Among Callbacks” on page 11-2

Automatically Refresh Plot in a GUIDE App

This example shows how to examine and run a prebuilt GUIDE app. The app displays a surface plot, adds random noise to the surface, and refreshes the plot at regular intervals. The app contains two buttons: one that starts adding random noise to the plot, and another that stops adding noise. The slider below the plot allows the user to set the refresh period between 0.01 and 2 seconds.

Open and Run the Example

Open and run the app. Move the slider to set the refresh interval between 0.01 and 2.0 seconds. Then click the **Start Randomizing** button to start adding random noise to the plotted function. Click the **Stop Randomizing** button to stop adding noise and refreshing the plot.



Examine the Code

- 1 In GUIDE, click the **Editor** button  to view the code.
- 2 Near the top of the Editor window, use the  **Go To**  button to navigate to the functions discussed below.

ex_guide_timergui_OpeningFcn

The `ex_guide_timergui_OpeningFcn` function executes when the app opens and starts running. This command creates the `timer` object and stores it in the `handles` structure.

```
handles.timer = timer(...  
    'ExecutionMode', 'fixedRate', ... % Run timer repeatedly.  
    'Period', 1, ... % Initial period is 1 sec.  
    'TimerFcn', {@update_display,hObject}); % Specify callback function.
```

The callback function for the timer is `update_display`, which is defined as a local function.

update_display

The `update_display` function executes when the specified `timer` period elapses. The function gets the values in the `ZData` property of the `Surface` object and adds random noise to it. Then it updates the plot.

```
handles = guidata(hObject);  
Z = get(handles.surf,'ZData');  
Z = Z + 0.1*randn(size(Z));  
set(handles.surf,'ZData',Z);
```

periodsldr_Callback

The `periodsldr_Callback` function executes when the user moves the slider. It calculates the timer period by getting the slider value and truncating it. Then it updates the label below the slider and updates the period of the `timer` object.

```
% Read the slider value  
period = get(handles.periodsldr,'Value');  
% 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.  
    set(handles.timer,'Period',period)  
end
```

startbtn_Callback

The `startbtn_Callback` function calls the `start` method of the `timer` object if the timer is not already running.

```
if strcmp(get(handles.timer, 'Running'), 'off')
    start(handles.timer);
end
```

stopbtn_Callback

The `stopbtn_Callback` function calls the `stop` method of the `timer` object if the timer is currently running.

```
if strcmp(get(handles.timer, 'Running'), 'on')
    stop(handles.timer);
end
```

figure1_CloseRequestFcn

The `figure1_CloseRequestFcn` callback executes when the user closes the app. The function stops the `timer` object if it is running, deletes the `timer` object, and then deletes the figure window.

```
if strcmp(get(handles.timer, 'Running'), 'on')
    stop(handles.timer);
end
% Destroy timer
delete(handles.timer)
% Destroy figure
delete(hObject);
```

See Also

Related Examples

- “Timer Callback Functions”
- “Write Callbacks in GUIDE” on page 7-2

Create UIs Programmatically

- “Lay Out a UI Programmatically” on page 9-25
- “Create Menus for Programmatic Apps” on page 9-38
- “Create Toolbars for Programmatic Apps” on page 9-51
- “Create a Simple App Programmatically” on page 3-2
- “Write Callbacks for Apps Created Programmatically” on page 10-5
- “Callbacks for Specific Components” on page 7-12
- “Share Data Among Callbacks” on page 11-2

Lay Out a Programmatic UI

- “Structure of Programmatic App Code Files” on page 9-2
- “Add Components to a Programmatic App” on page 9-4
- “Lay Out a UI Programmatically” on page 9-25
- “Customize Tabbing Behavior in a Programmatic App” on page 9-34
- “Create Menus for Programmatic Apps” on page 9-38
- “Create Toolbars for Programmatic Apps” on page 9-51
- “DPI-Aware Behavior in MATLAB” on page 9-58

Structure of Programmatic App 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, the code file for an app 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 App” on page 10-2 for more information.
- 3** Construction of figure and components.
- 4** Initialization tasks that require the components to exist, and output return. See “Initialize a Programmatic App” 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 an app 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 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 app and callbacks to service the components, you can manipulate components that it contains.

See Also

Related Examples

- “Create a Simple App Programmatically” on page 3-2

Add Components to a Programmatic App

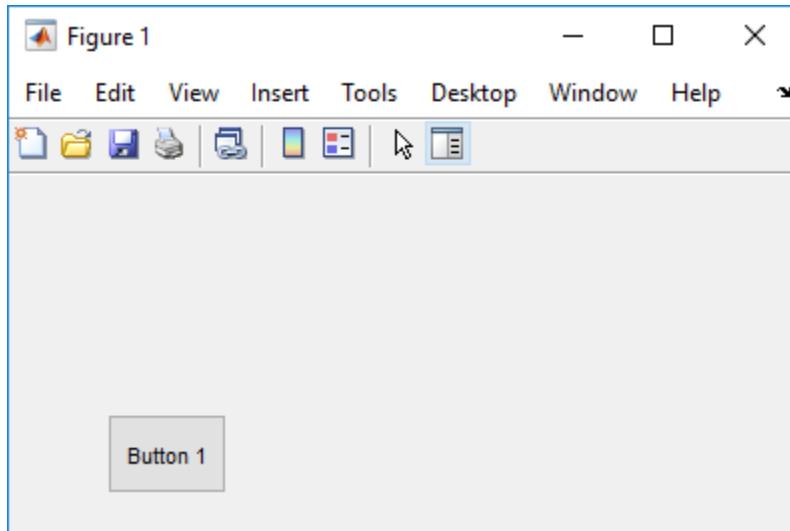
User interface controls are common UI components, such as buttons, check boxes, and sliders. Tables present data in rows and columns. Panels and button groups are containers in which you can group together related elements in your UI. ActiveX components enable you to display ActiveX controls.

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.

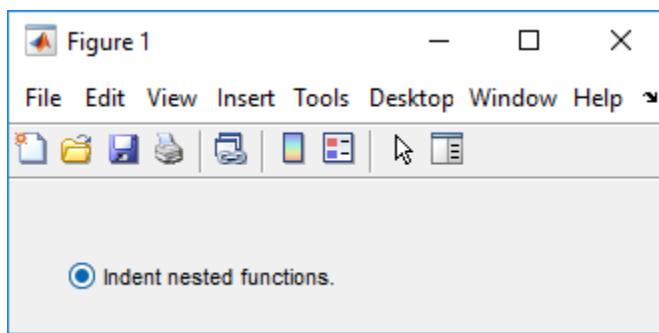
Displaying an Icon on a Push Button

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 9-18 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`.

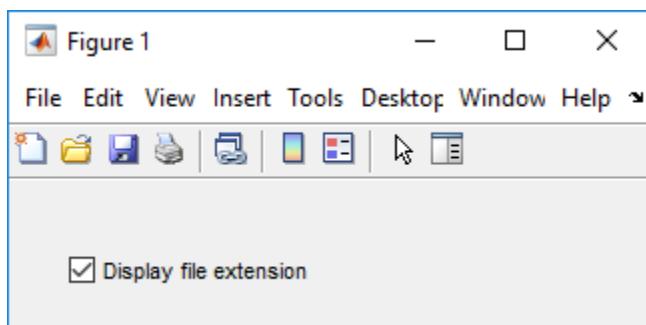
'Position',[30 20 100 30] specifies the location and size of the toggle button. In this example, the toggle button is 100 pixels wide and 30 pixels high. It is positioned 30 pixels from the left of the figure and 20 pixels from the bottom.

Note You can also display an icon on a toggle button. See “Displaying an Icon on a Push Button” on page 9-5 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

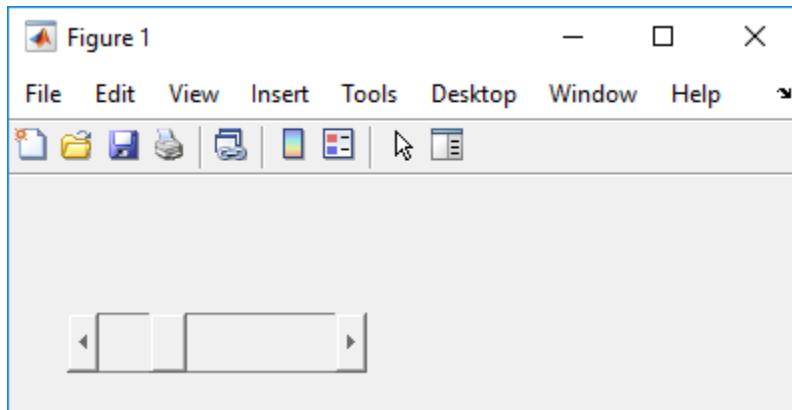
check box, MATLAB sets `Value` to `Max` when the user checks the box and to `Min` when the user unchecks it.

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.

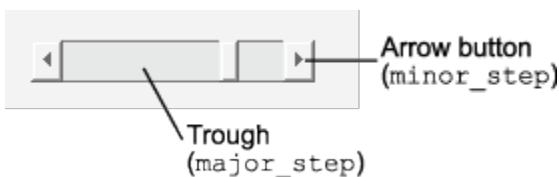
`'Min',0` and `'Max',100` specify the range of the slider to be [0, 100]. The `Min` property must be less than `Max`.

`'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 use this behavior to indicate how much of the document is currently visible by changing the value of `major_step`.



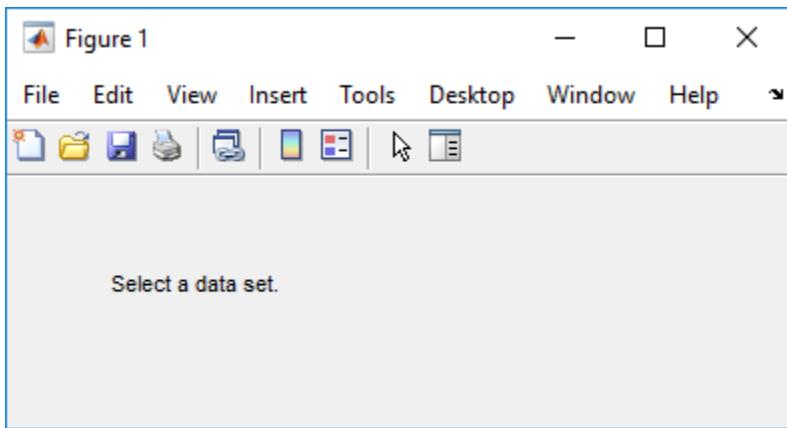
'Position',[30 20 150 30] specifies the location and size of the slider. In this example, the slider is 150 pixels wide and 30 high. It is positioned 30 pixels from the left of the figure and 20 pixels from the bottom.

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.

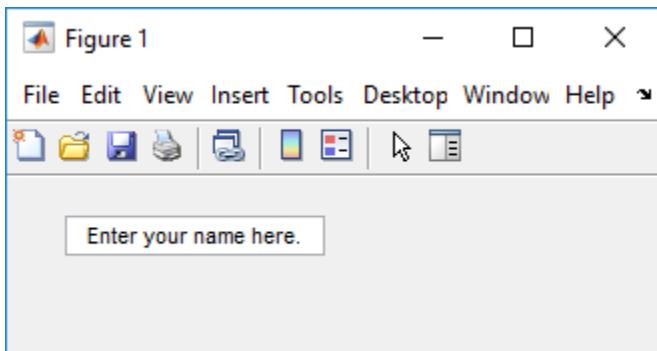
`'String'`, `'Select a set'` specifies the text that displays. If you specify a component width that is too small to accommodate all of the text, MATLAB wraps the text.

`'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.

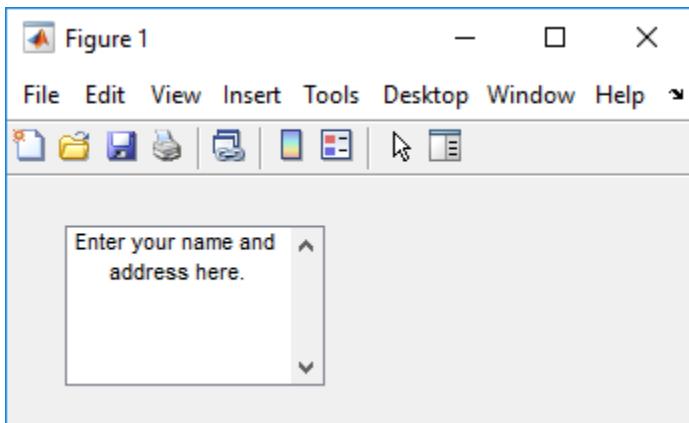
The name-value pair arguments, '`Style`', '`edit`', specify the style of the `uicontrol` to be an editable text field.

'`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]);
```



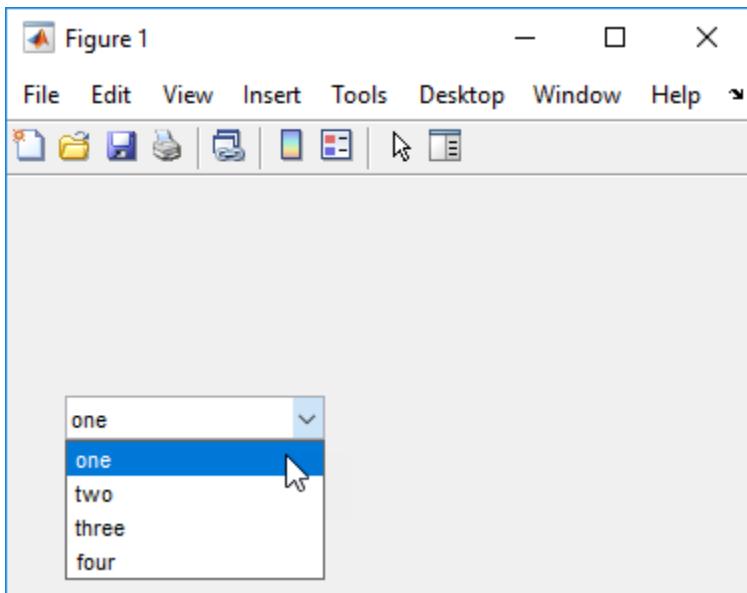
If the value of `Max - Min` is less than or equal to 1, the editable text field allows only a single line of input. If the width of the text field is too narrow for the text, MATLAB displays only part of the text. The user can use the arrow keys to move the cursor over the entire line of text.



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.

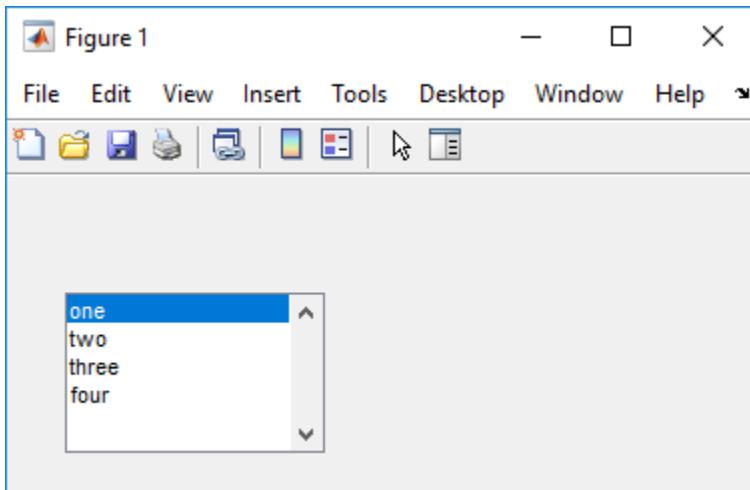
`'Position', [30 80 130 20]` specifies the location and size of the pop-up menu. In this example, the pop-up menu is 130 pixels wide. It is positioned 30 pixels from the left of the figure and 80 pixels from the bottom. The height of a pop-up menu is determined by the font size; the height you set in 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'}, ...
    'Value', 1, ...
    'Position', [30 80 130 20]);
```

```
'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.

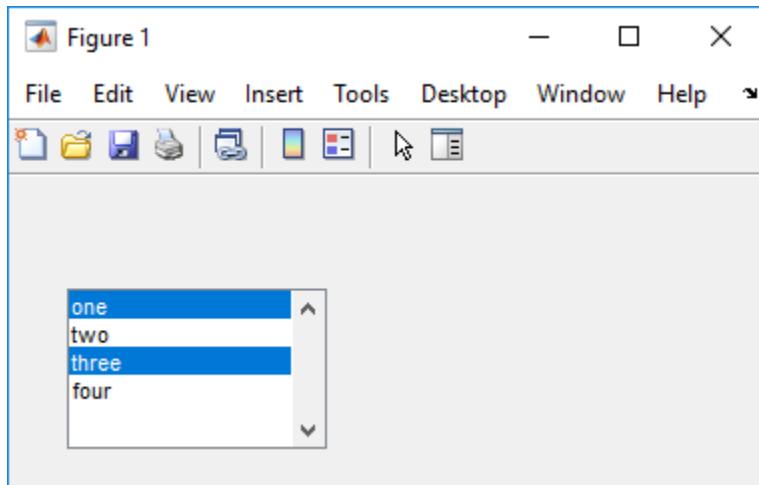
`'String',{'one','two','three','four'}` defines the list items.

`'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.

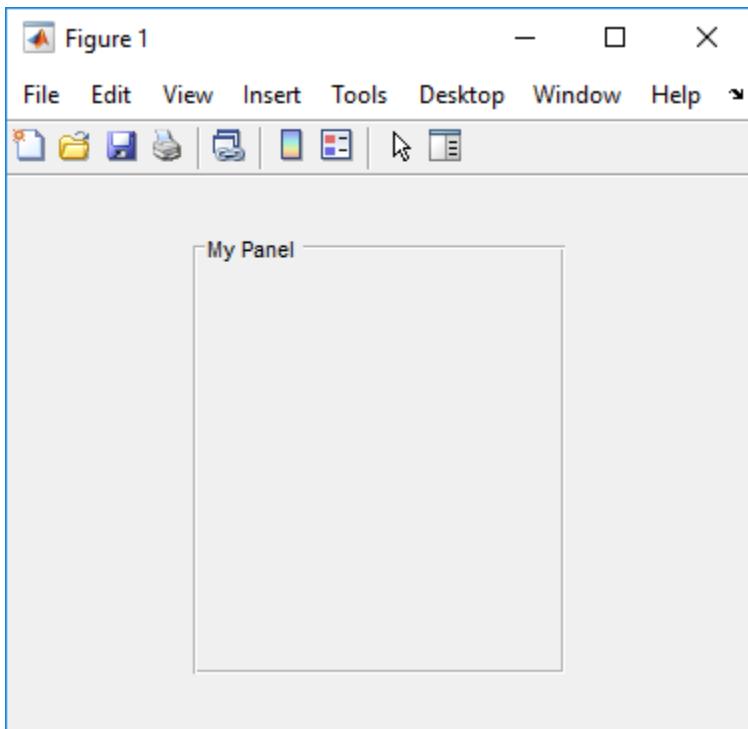
See **Uitable** for the entire list of properties.

If you are building an app 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]);
```



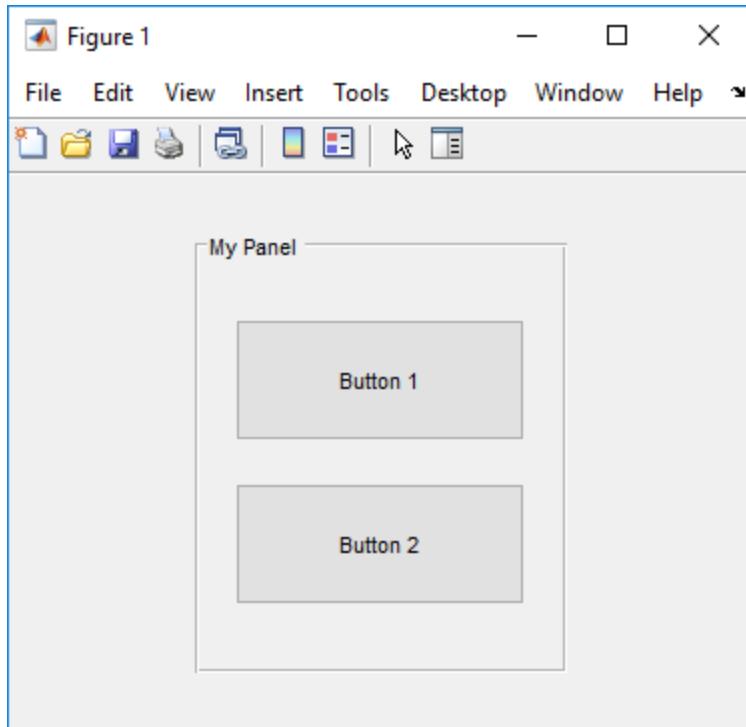
The first argument passed to `uipanel`, `f`, specifies the parent container. In this case, the parent is a figure, but you can also specify the parent to be any container, such as another panel or a button group.

`'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.

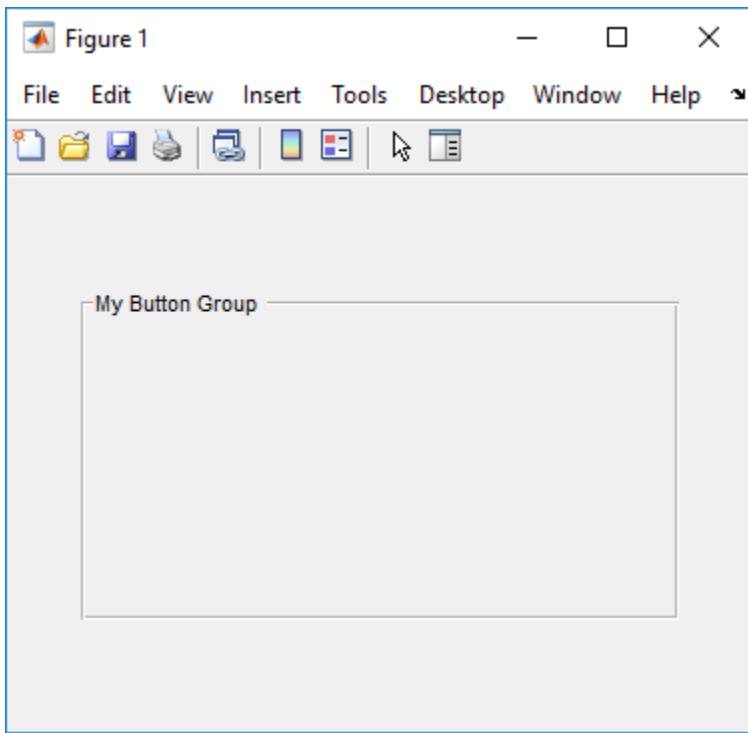
```
b1 = uicontrol(p,'Style','pushbutton','String','Button 1',...
    'Units','normalized',...
    'Position',[.1 .55 .8 .3]);
b2 = uicontrol(p,'Style','pushbutton','String','Button 2',...
    'Units','normalized',...
    'Position',[.1 .15 .8 .3]);
```



Button Groups

This code creates a button group:

```
f = figure;
bg = uibuttongroup(f,'Title','My Button Group',...
    'Position',[.1 .2 .8 .6]);
```



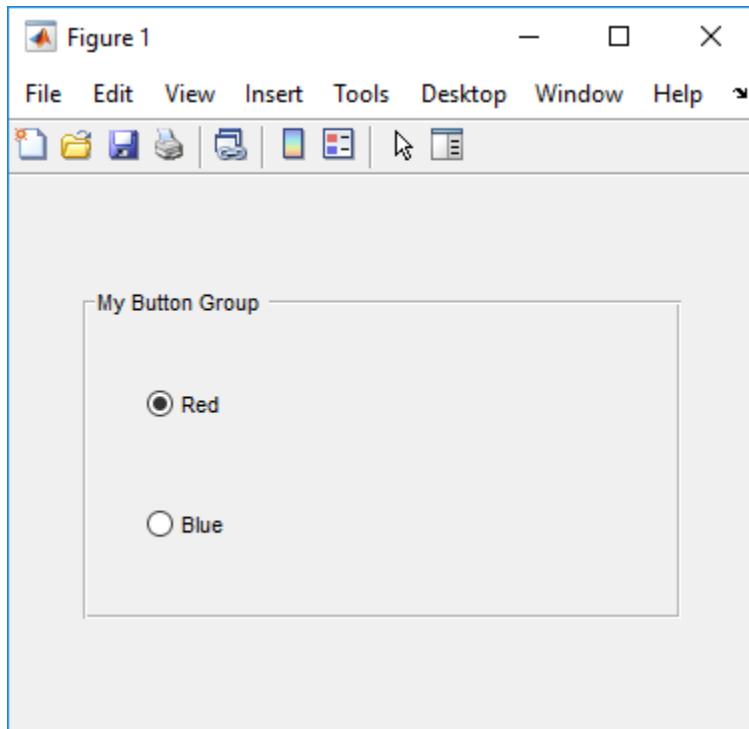
The first argument passed to `uibuttongroup`, `f`, specifies the parent container. In this case, the parent is a figure, but you can also specify the parent to be any container, such as a panel or another button group.

`'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]);
```



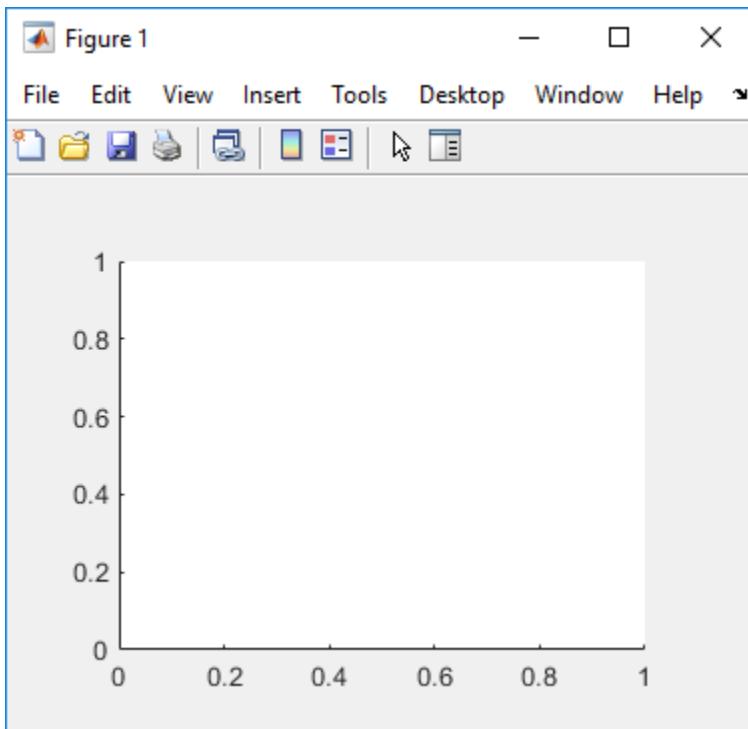
By default, the first radio button added to the `uibuttongroup` is selected. To override this default, set any other radio button's `Value` property to its `Max` property value.

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 place an axes in a UI, 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.

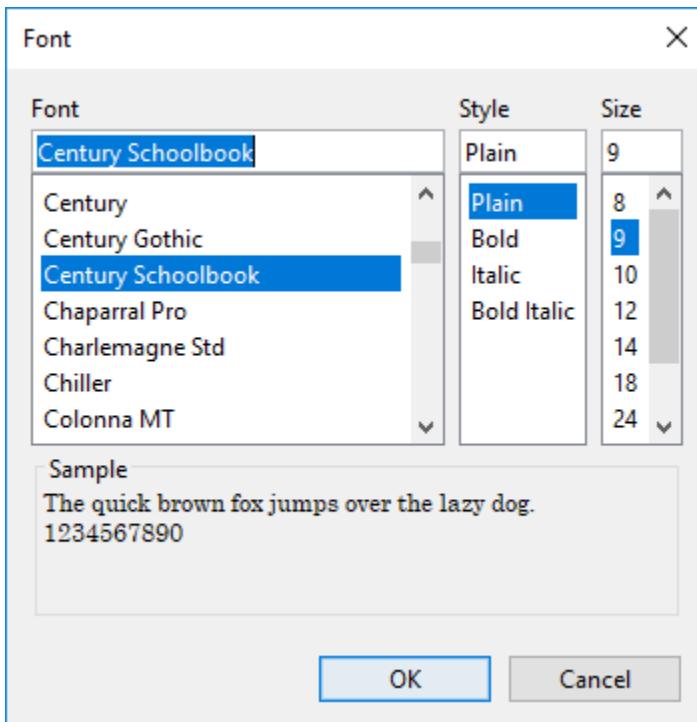
See “Creating an ActiveX Control” about adding an ActiveX control to a figure. See “Create COM Objects” for general information about ActiveX controls.

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 =  
  
struct with fields:  
  
    FontName: 'Century Schoolbook'  
    FontWeight: 'normal'  
    FontAngle: 'normal'  
    FontUnits: 'points'  
    FontSize: 9
```

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);
```

See Also

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.

Component Placement and Sizing

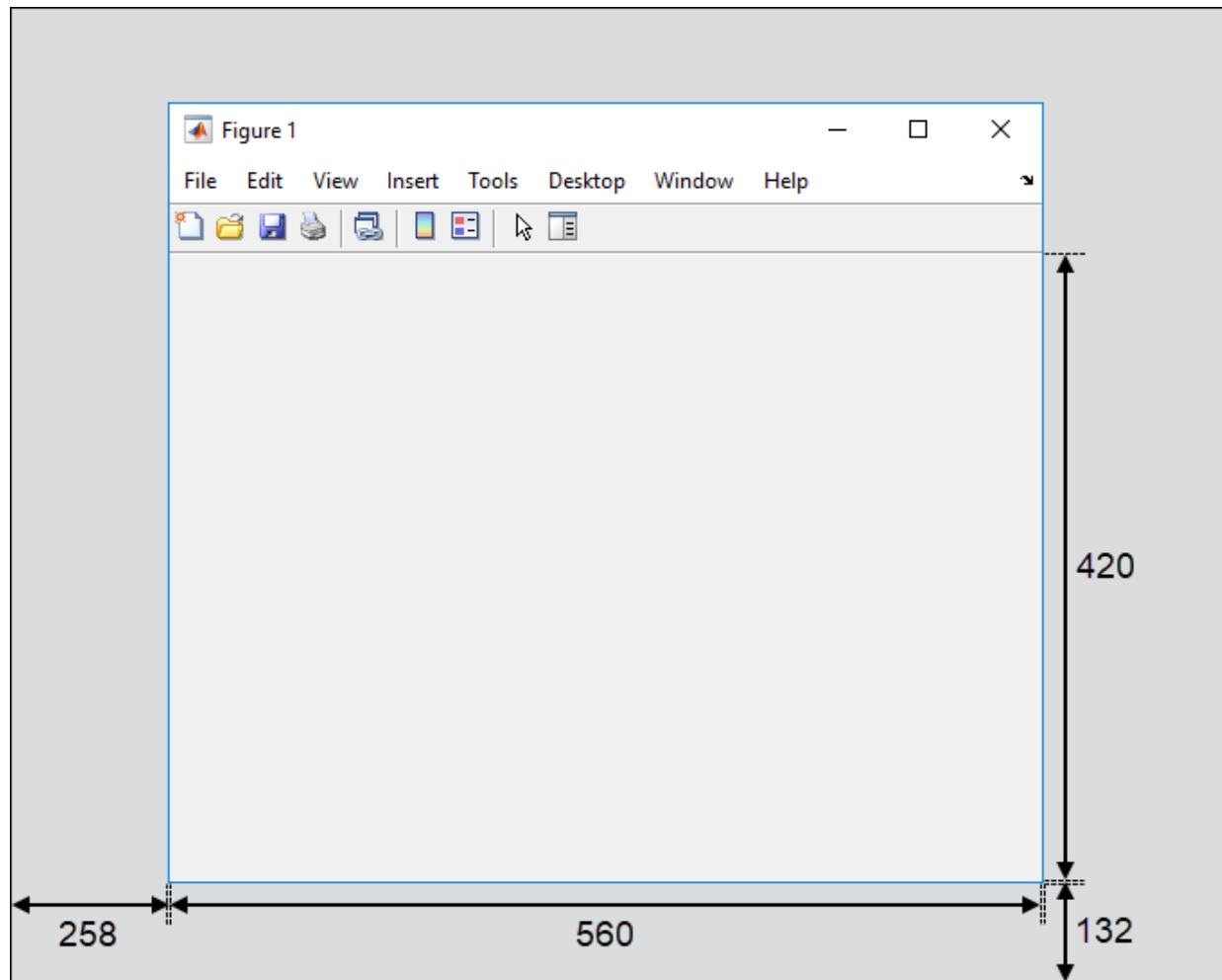
A UI layout consists of a figure and one or more components that you place inside the figure. Accurate placement and sizing of each component involves setting certain properties and understanding how the inner and outer boundaries of the figure relate to each other.

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 *excludes* the figure borders, title bar, menu bar, and tool bars. You can control the location and size of the drawable area by setting the `Position` property of the figure 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]);
```

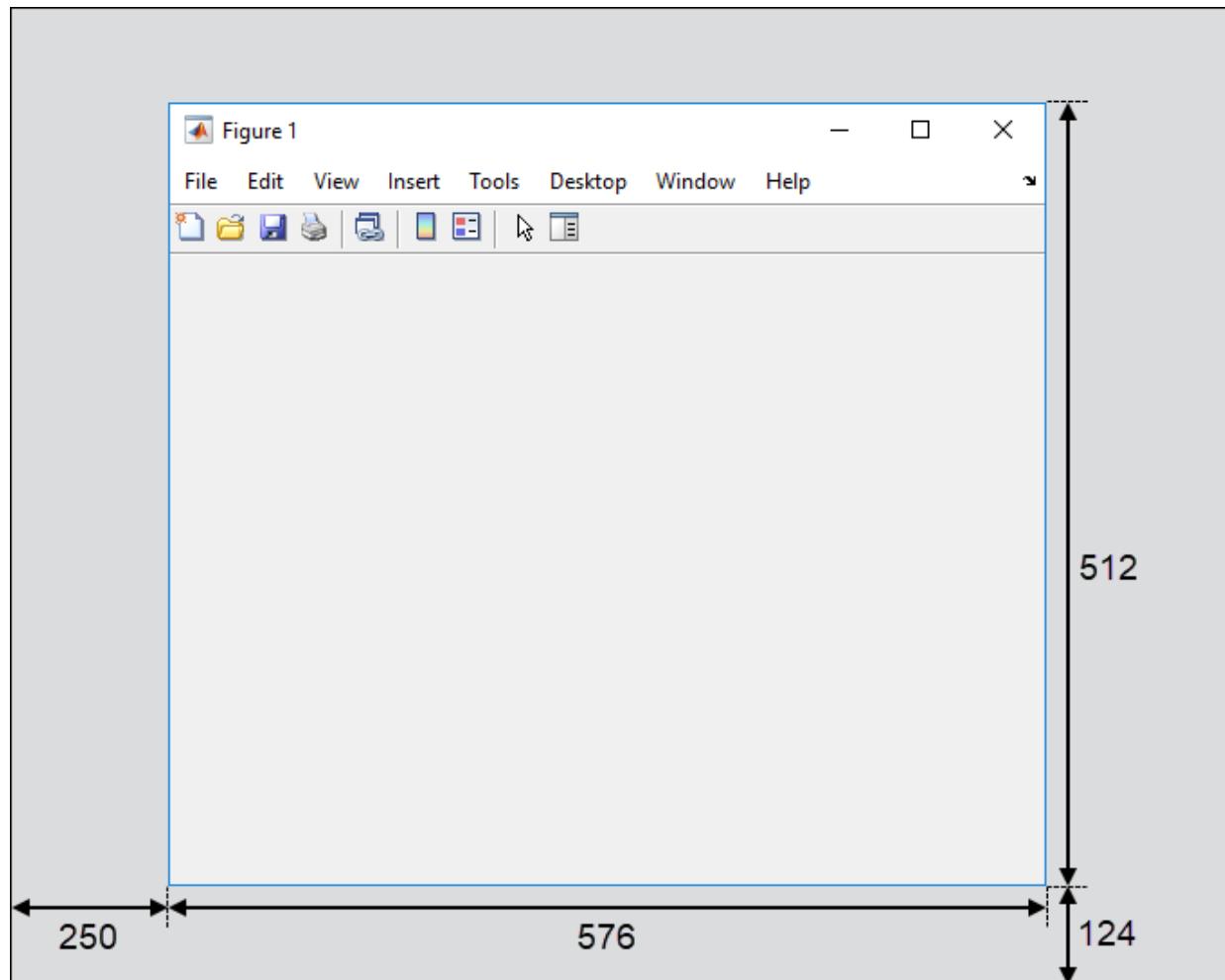


You can query or change the outer bounds of the figure by using the `OuterPosition` property. The region enclosed by the outer bounds of the figure includes the figure borders, title bar, menu bar, and tool bars. Like the `Position` property, the `OuterPosition` is a four element row vector:

```
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 app 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

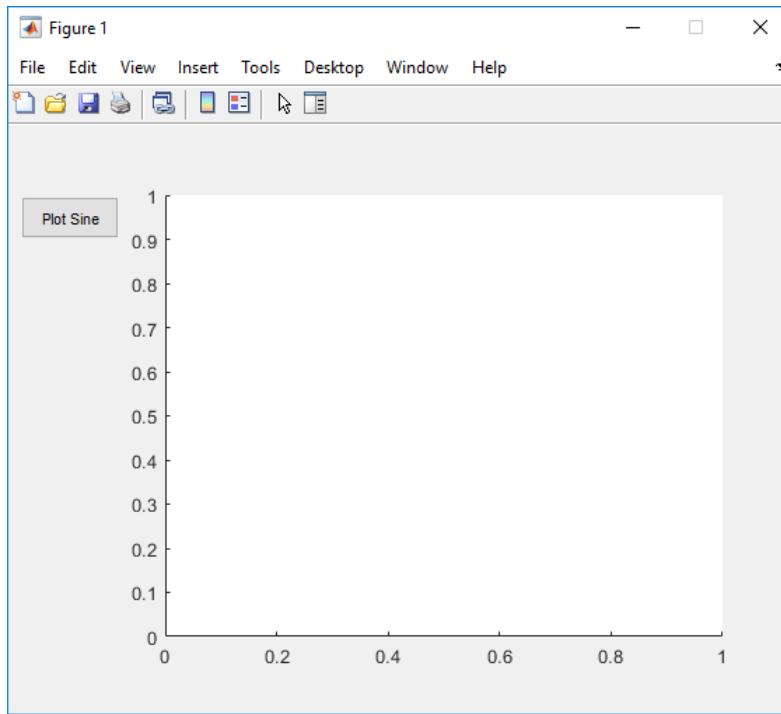
    function plotsine(hObject,event)
        theta = 0:pi/64:6*pi;
        y = sin(theta);
        plot(hs.ax,theta,y);
    end
end
```

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 app is a resizable version of the simple app defined in “Example of a Simple Layout” on page 9-29. 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,event)
        theta = 0:pi/64:6*pi;
        y = sin(theta);
    end

```

```
    plot(hs.ax,theta,y);
end

function resizeui(hObject,event)

    % 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.

See Also

Related Examples

- “DPI-Aware Behavior in MATLAB” on page 9-58

Customize Tabbing Behavior in a Programmatic App

In this section...

["How Tabbing Works" on page 9-34](#)

["Default Tab Order" on page 9-34](#)

["Change the Tab Order in the uipanel" on page 9-36](#)

How Tabbing Works

The tab order is the order in which UI components 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.

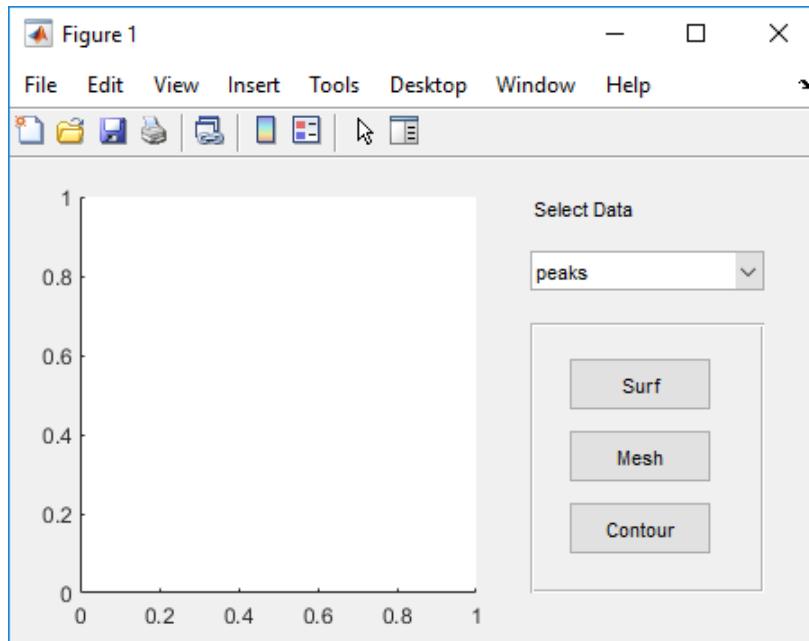
Default Tab Order

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 run the app 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 Apps

In this section...

["Add Menu Bar Menus" on page 9-38](#)

["Add Context Menus to a Programmatic App" on page 9-46](#)

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, ...)
```

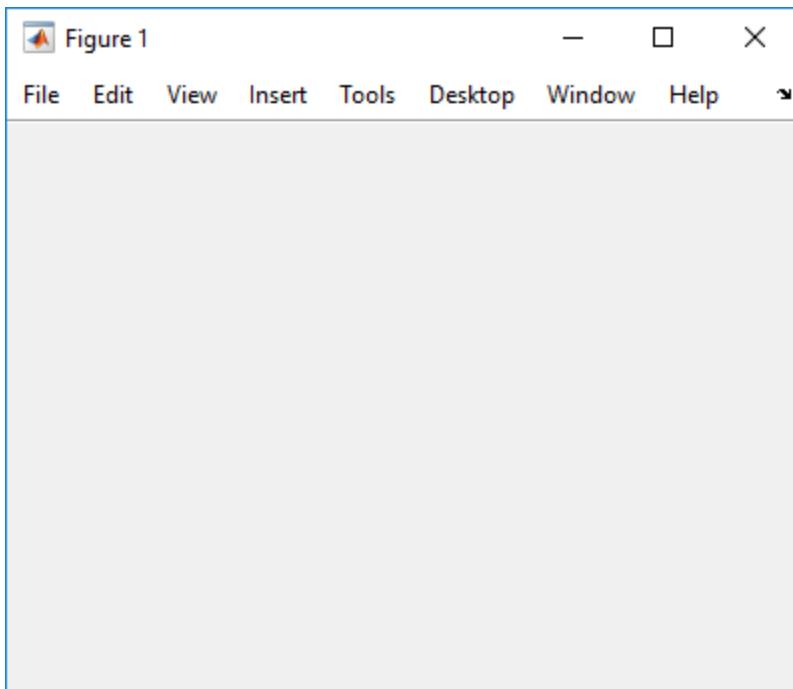
Where `mh` is the handle of the resulting menu or menu item. See the `uimenu` reference page for other valid syntaxes.

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 9-38](#)
- ["How Menus Affect Figure Docking" on page 9-39](#)
- ["Menu Bar Menu" on page 9-41](#)

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.

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 app, but if you want the user to be able to dock or undock the 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.

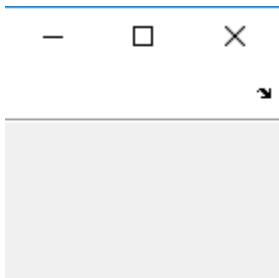


Figure windows with a standard menu bar also have a **Desktop** menu from which the user can dock and undock them.

To display the docking arrow and the **Desktop > Dock Figure** menu item, 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 the app.
- 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 page.

Menu Bar Menu

The following statements create a menu bar menu with two menu items.

```
mh = uimenu(fh,'Text','My menu');  
eh1 = uimenu(mh,'Text','Item 1');  
eh2 = uimenu(mh,'Text','Item 2','Checked','on');
```

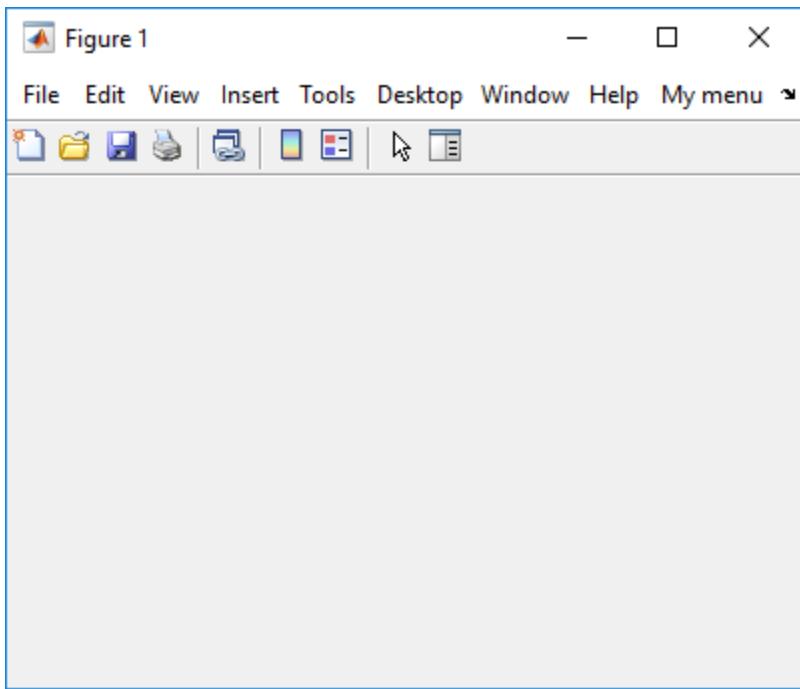
`fh` is the handle of the parent figure.

`mh` is the handle of the parent menu.

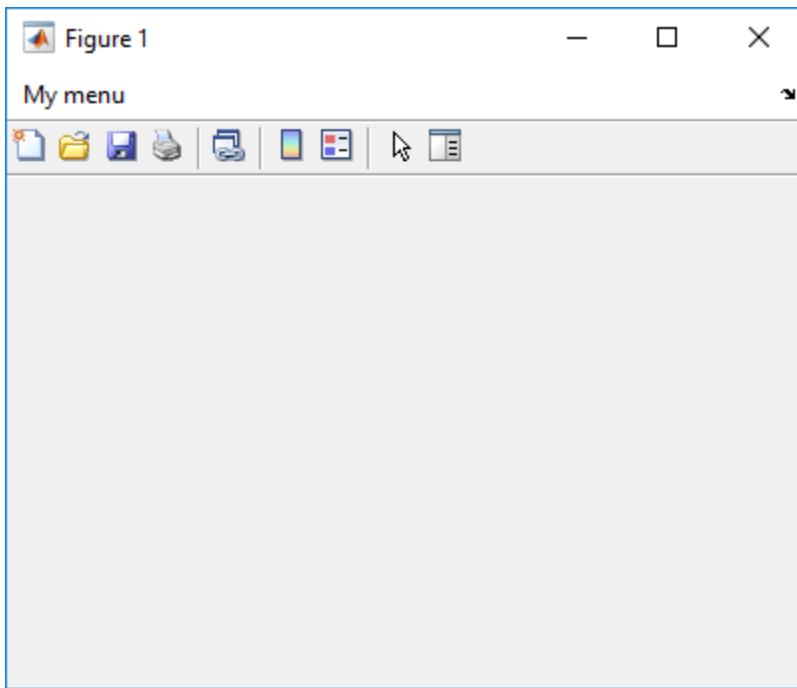
The `Text` property specifies the text that appears in the menu.

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.

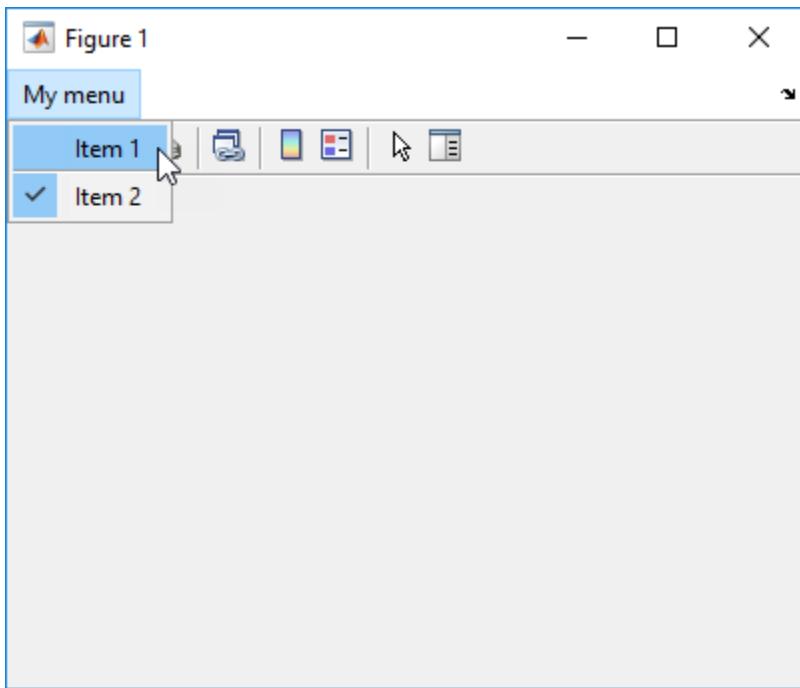


If your UI does not display the standard menu bar, MATLAB creates a menu bar if none exists and then adds the menu to it.



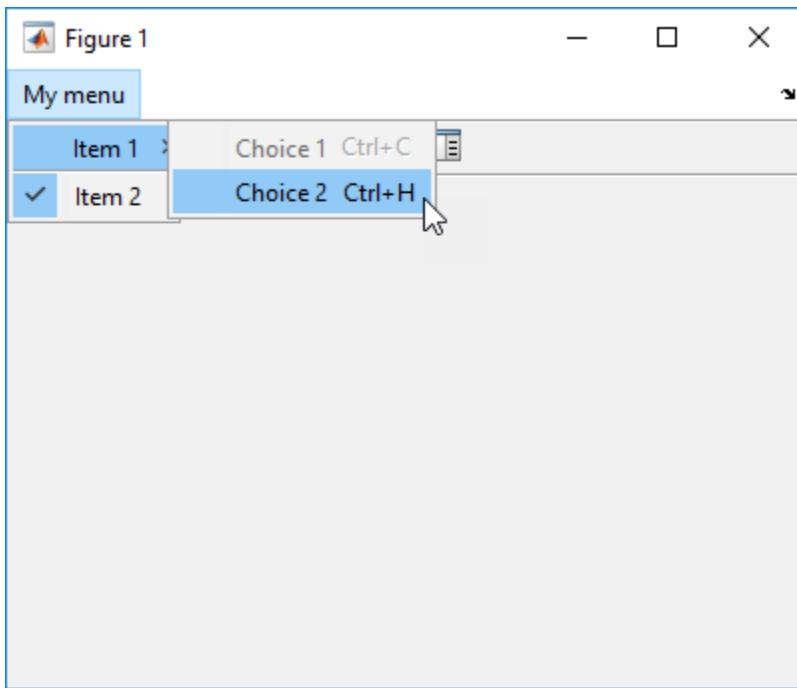
This command adds a separator line preceding the second menu item.

```
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,'Text','Choice 1','Accelerator','C',...
    'Enable','off');
seh2 = uimenu(eh1,'Text','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 submenu **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 7-22, for information about programming menu items.

Add Context Menus to a Programmatic App

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.

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 9-49 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,'Text','Item 1');
mh2 = uimenu(cmenu,'Text','Item 2');
mh3 = uimenu(cmenu,'Text','Item 3');
```

You can specify any applicable `Uimenu` when you define the context menu items. See the `uimenu` reference page and “Add Menu Bar Menus” on page 9-38 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:

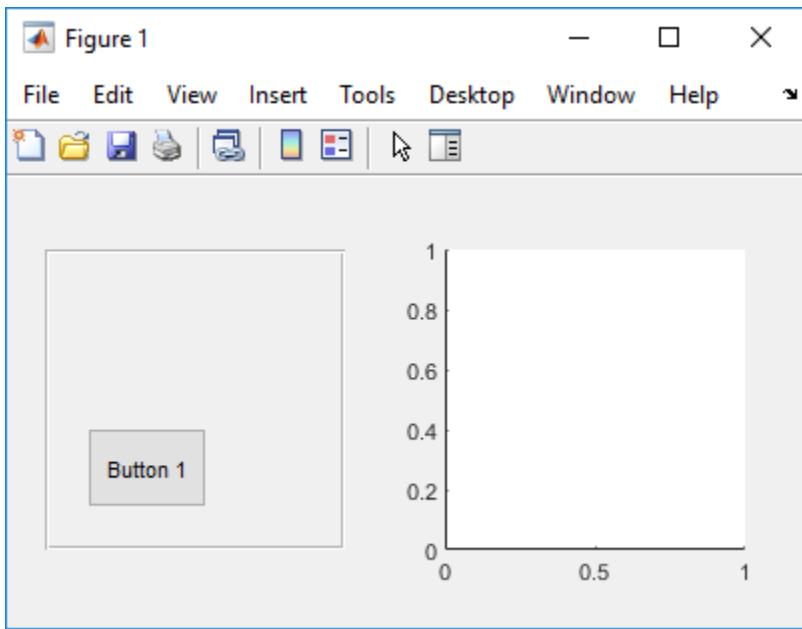
```
cmenuhandles = findall(figurehandle,'type','uicontextmenu');  
cmenuhandles.HandleVisibility = 'off';  
menuitemhandles = findall(cmenuhandles,'type','uimenu');  
menuitemhandles.HandleVisibility = 'off';
```

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.

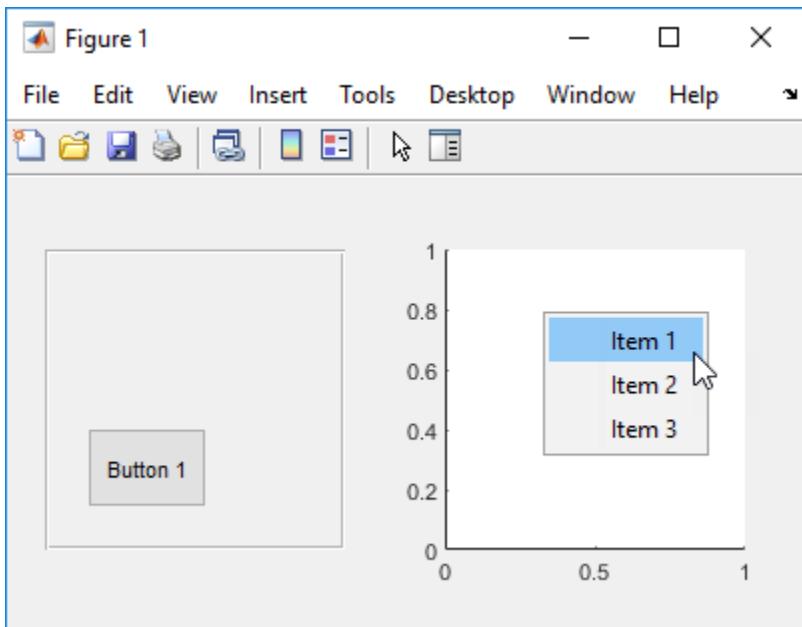
```
ph = uipanel('Parent',fh,'Units','pixels',...  
            'Position',[20 40 150 150]);  
bh1 = uicontrol(ph,'String','Button 1',...  
                'Position',[20 20 60 40]);  
ah = axes('Parent',fh,'Units','pixels',...  
            'Position',[220 40 150 150]);
```



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.

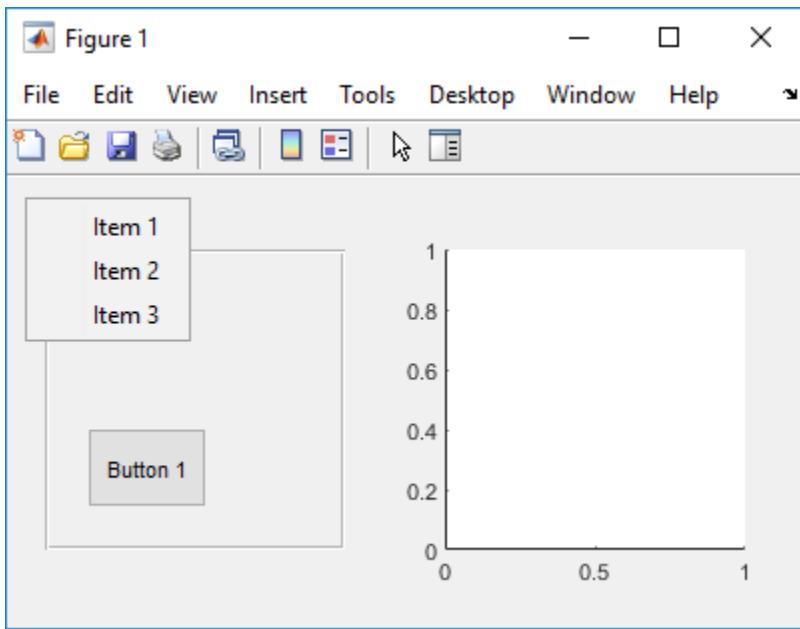


Force Display of the Context Menu

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.

See Also

[Menu](#) | [uimenu](#)

Related Examples

- “Create Toolbars for Programmatic Apps” on page 9-51

Create Toolbars for Programmatic Apps

In this section...

- “Use the `uimenu` Function” on page 9-51
- “Commonly Used Properties” on page 9-51
- “Toolbars” on page 9-52
- “Display and Modify the Standard Toolbar” on page 9-55

Use the `uimenu` Function

Use the `uimenu` function to add a custom menu 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 `uimenu`, `uipushtool`, and `uitoggletool` functions include the following:

```
tbh = uimenu(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 menu, push tool, and toggle tool. See the `uimenu`, `uipushtool`, and `uitoggletool` reference pages for other valid syntaxes.

Subsequent topics describe commonly used properties of menus and menu tools, offer a simple example, and discuss use of the MATLAB standard menu.

Commonly Used Properties

The most commonly used properties needed to describe a menu 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 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.
Tooltip	Character vector or string scalar	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`, `Uipushtool`, and `Uitoggletool`.

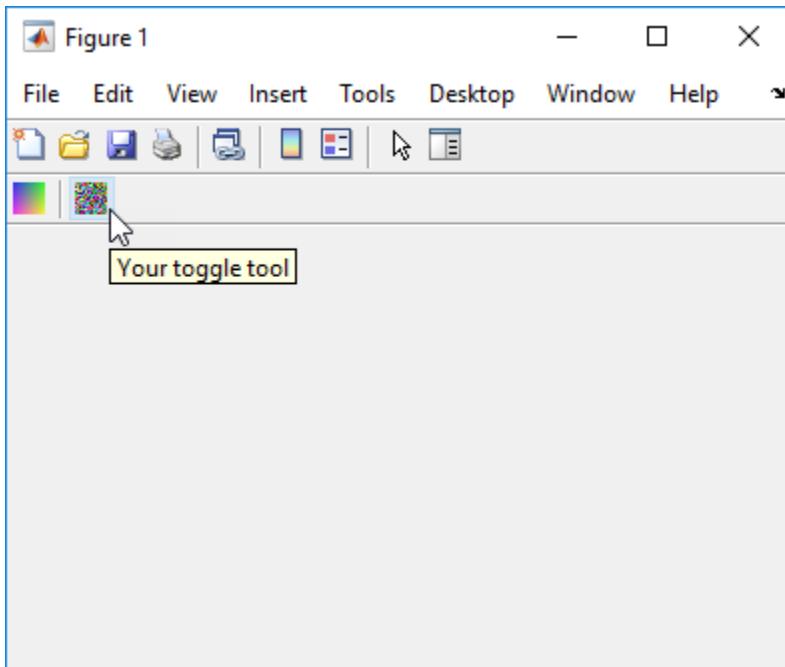
Toolbars

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,...
    'Tooltip','My push tool',...
```

```
'HandleVisibility','off');  
% Add a toggle tool to the toolbar  
img2 = rand(16,16,3);  
tth = uitoggletool(tbh,'CData',img2,'Separator','on',...
    'Tooltip','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.

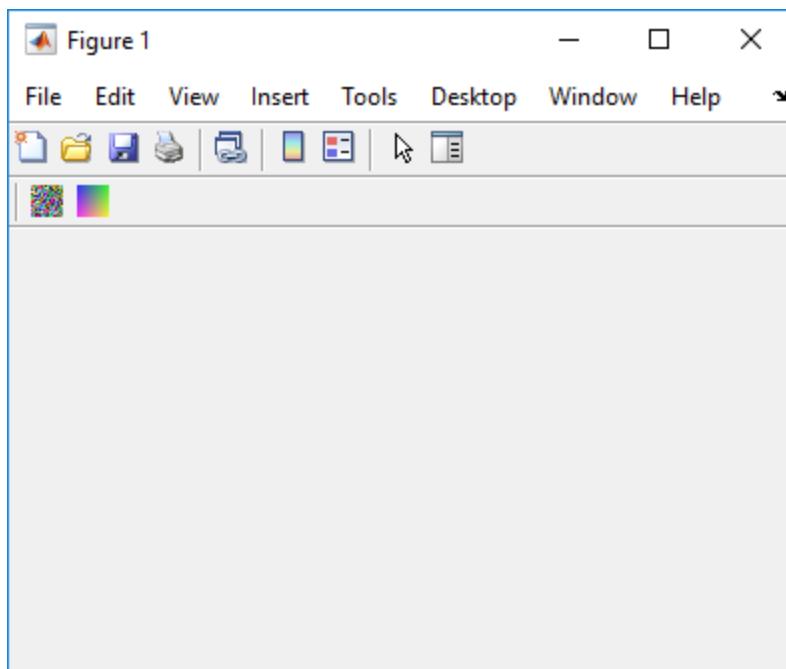
Tooltip specifies the tooltips for the push tool and the toggle tool as `My push tool` and `Your toggle tool`, respectively.

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)
```

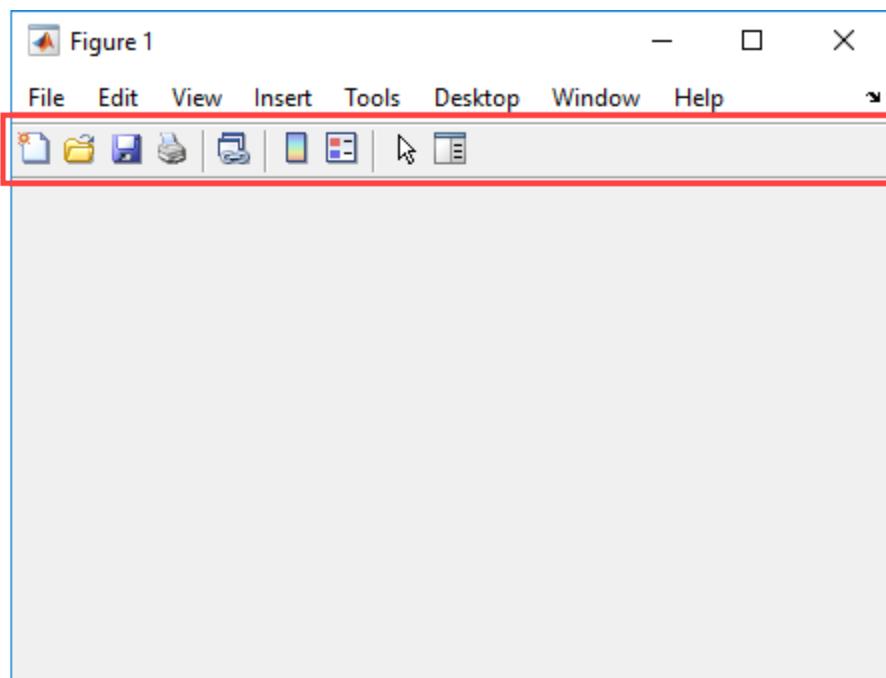
If necessary, you can use the `findall` function to determine the handles of the tools on a particular toolbar.

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.

```
fhToolBar = 'figure'; % Display the standard toolbar  
fhToolBar = 'none'; % Hide the standard toolbar
```

In these statements, fh is the handle of the figure.

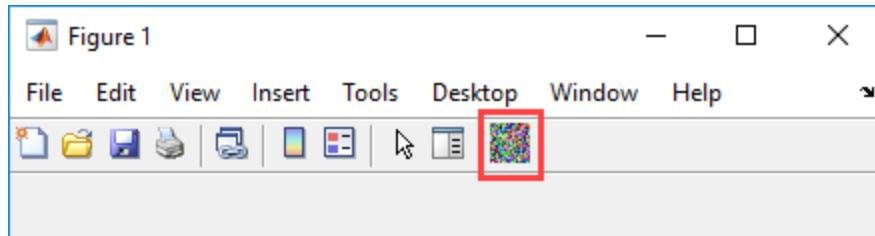
The default ToolBar value is 'auto', which uses theMenuBar property value.

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`

Related Examples

- “Create Menus for Programmatic Apps” on page 9-38

DPI-Aware Behavior in MATLAB

In this section...

[“Visual Appearance” on page 9-58](#)

[“Using Object Properties” on page 9-60](#)

[“Using print, getframe, and publish Functions” on page 9-61](#)

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

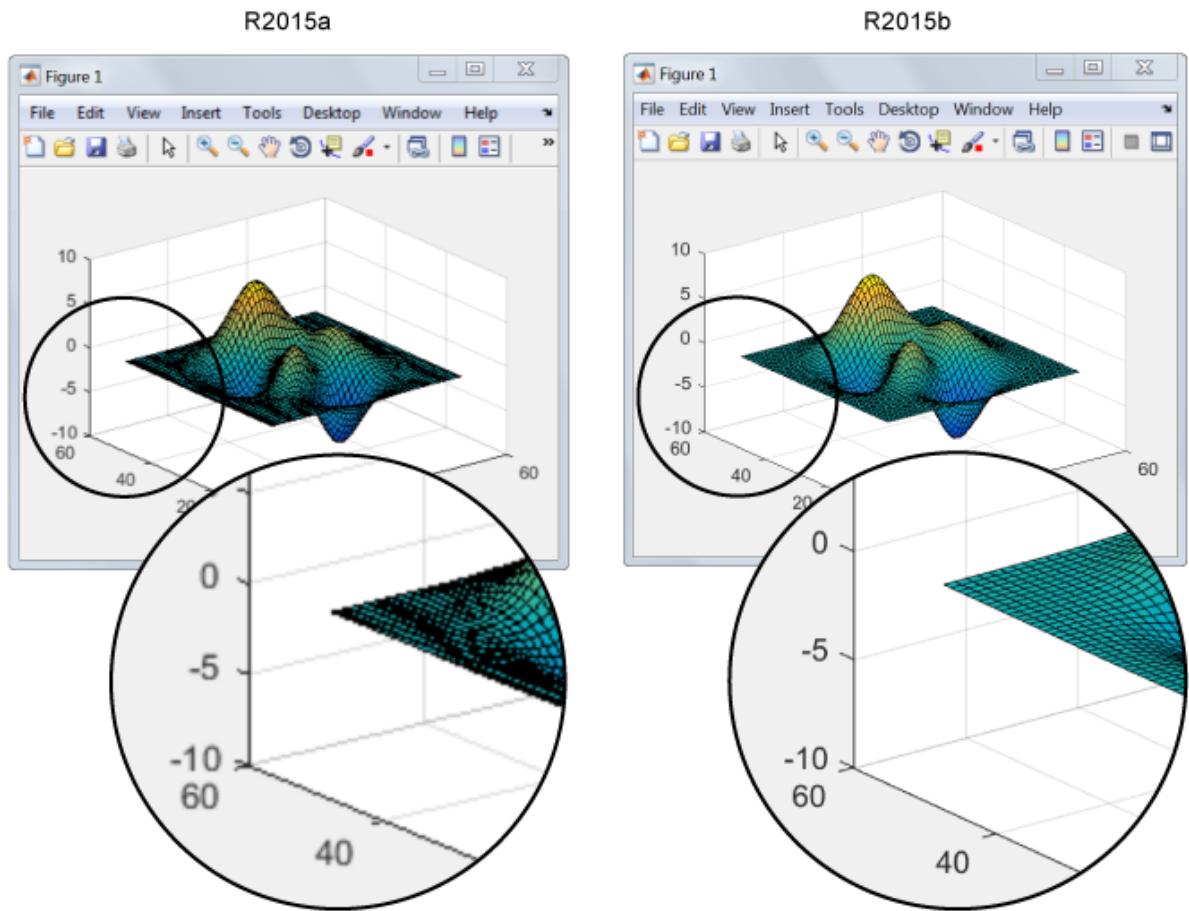
DPI-aware behavior does not apply to Linux systems.

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](#) | [Root](#)

Code a Programmatic App

- “Initialize a Programmatic App” on page 10-2
- “Write Callbacks for Apps Created Programmatically” on page 10-5

Initialize a Programmatic App

Some apps 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 an app, 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, `processUserInputs`, 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.

See Also

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](#)

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 character vector containing a MATLAB expression. Setting this property makes your app respond when the user interacts with the uicontrol. If the `Callback` property has no specified value, then nothing happens when the 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
<code>ButtonDownFcn</code>	End user presses a mouse button while the pointer is on the component or figure.	<code>axes</code> , <code>figure</code> , <code>uibuttongroup</code> , <code>uicontrol</code> , <code>uipanel</code> , <code>uitable</code> ,
<code>Callback</code>	End user triggers the component. For example: selecting a menu item, moving a slider, or pressing a push button.	<code>uicontextmenu</code> , <code>uicontrol</code> , <code>uimenu</code>
<code>CellEditCallback</code>	End user edits a value in a table whose cells are editable.	<code>uitable</code>
<code>CellSelectionCallback</code>	End user selects cells in a table.	<code>uitable</code>
<code>ClickedCallback</code>	End user clicks the push tool or toggle tool with the left mouse button.	<code>uitoggletool</code> , <code>uipushtool</code>

Callback Property	User Action	Components That Use This Property
CloseRequestFcn	The figure closes.	figure
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
SelectionChangedFcn	End user selects a different radio button or toggle button within a button group.	uibuttongroup
WindowButtonDownFcn	End user presses a mouse button while the pointer is in the figure window.	figure
WindowButtonMotionFcn	End user moves the pointer within the figure window.	figure

Callback Property	User Action	Components That Use This Property
WindowButtonUpFcn	End user releases a mouse button.	figure
WindowKeyPressFcn	End user presses a key while the pointer is on the figure or any of its child objects.	figure
WindowKeyReleaseFcn	End user releases a key while the pointer is on the figure or any of its child objects.	figure
WindowScrollWheelFcn	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 the following 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 any input arguments you want to use in the function.
- “Specify an Anonymous Function” on page 10-9.
- “Specify a Character Vector Containing MATLAB Commands (Not Recommended)” on page 10-9

Specify a Function Handle

Function handles provide a way to represent a function as a variable. The function must be a local or nested function in the same file as the app code, or you can write it in a separate file that is on the MATLAB path.

To create the function handle, specify the @ operator before the name of the function. For example, this `uicontrol` command specifies the `Callback` property to be a handle to the function `pushbutton_callback`:

```
b = uicontrol('Style','pushbutton','Callback',@pushbutton_callback);
```

Here is the function definition for pushbutton_callback:

```
function pushbutton_callback(src,event)
    display('Button pressed');
end
```

Notice that the function handle does not explicitly refer to any input arguments, but the function declaration includes two input arguments. These two input arguments are required for all callbacks you specify as a function handle. MATLAB passes these arguments automatically when the callback executes. The first argument is the UI component that triggered the callback. The second argument provides event data to the callback function. If there is no event data available to the callback function, then MATLAB passes the second input argument as an empty array. The following table lists the callbacks and components that use event data.

Callback Property Name	Component
WindowKeyPressFcn	figure
WindowKeyReleaseFcn	figure
WindowScrollWheel	figure
KeyPressFcn	figure, uicontrol, uitable
KeyReleaseFcn	figure, uicontrol, uitable
SelectionChangedFcn	uibuttongroup
CellEditCallback	uitable
CellSelectionCallback	uitable

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 user to trigger the callback. This behavior helps you to find problems in your code before the user encounters them.

Specify a Cell Array

Use a cell array to specify a callback function that accepts additional 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 additional input arguments you want to use, separated by commas. The function you specify must define the same two input

arguments as described in “Specify a Function Handle” on page 10-7. However, you can define additional inputs in your function declaration after the first two arguments.

This `uicontrol` command creates a push button and specifies the `Callback` property to be a cell array. In this case, the name of the function is `pushbutton_callback`, and the value of the additional input argument is 5.

```
b = uicontrol('Style','pushbutton','Callback',{@pushbutton_callback,5});
```

Here is the function definition for `pushbutton_callback`:

```
function pushbutton_callback(src,event,x)
    display(x);
end
```

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 user to trigger the callback. This behavior helps you to find problems in your code before the user encounters them.

Specify an Anonymous Function

Specify an anonymous function when you want a UI component to execute a function that does not support the two arguments that are required for function handles and cell arrays. For example, this `uicontrol` command creates a push button and specifies the `Callback` property to be an anonymous function. In this case, the name of function is `myfun`, and its function declaration defines only one input argument, `x`.

```
b = uicontrol('Style','pushbutton','Callback',{@(src,event)myfun(x)});
```

Specify a Character Vector Containing MATLAB Commands (Not Recommended)

You can specify a character vector when you want to execute a few simple commands, but the callback can become difficult to manage if it contains more than a few commands. The character vector 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 character vector, `'plot(rand(20,3))'`, is a valid command, and MATLAB evaluates it when the user clicks the button. If the character vector includes a variable, for example,

```
'plot(x)'
```

The variable `x` must exist in the base workspace when the 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 user triggers the callback.

Unlike callbacks that are specified as function handles or cell arrays, MATLAB does *not* check character vectors for syntax errors or missing dependencies. If there is a problem with the MATLAB expression, it remains undetected until the user triggers the callback.

See Also

Related Examples

- “Callbacks for Specific Components” on page 7-12
- “Share Data Among Callbacks” on page 11-2
- “Interrupt Callback Execution” on page 12-2
- “Anonymous Functions”

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 11-8
- “Create Nested Callback Functions (Programmatic Apps)” on page 11-12
- “Store Data Using the `guidata` Function” on page 11-13
- “GUIDE Example: Share Slider Data Using `guidata`” on page 11-16
- “GUIDE Example: Share Data Between Two Apps” on page 11-16
- “GUIDE Example: Share Data Among Three Apps” on page 11-17

Overview of Data Sharing Techniques

Many apps 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 app that need to access it. The table below describes several different methods for sharing data within your app.

Method	Description	Requirements and Trade-Offs
“Store Data in <code>UserData</code> or Other Object Properties” on page 11-3	Get or set property values directly through the component object. All UI components have a <code>UserData</code> property that can store any MATLAB data.	<ul style="list-style-type: none">• Requires access to the component to set or retrieve the properties.• <code>UserData</code> holds only one variable at a time, but you can store multiple values as a <code>struct</code> array or cell array.
“Store Data as Application Data” on page 11-8	Associate data with a specific component using the <code>setappdata</code> function. You can access it later using the <code>getappdata</code> function.	<ul style="list-style-type: none">• Requires access to the component to set or retrieve the application data.• Can share multiple variables.

Method	Description	Requirements and Trade-Offs
"Create Nested Callback Functions (Programmatic Apps)" on page 11-12	Nest your callback functions inside your main function. This gives your callback functions access to all the variables in the main function.	<ul style="list-style-type: none"> Requires callback functions to be coded in the same file as the main function. Not recommended for GUIDE apps. Can share multiple variables.
"Store Data Using the guidata Function" on page 11-13	Share data with the figure window using the <code>guidata</code> function.	<ul style="list-style-type: none"> Stores or retrieves the data through any UI component. Stores only one variable at a time, but you can store multiple values as a <code>struct</code> 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 Apps Created Programmatically

Use dot notation, `component.propertyname`, to get or set property values programmatically. Dot notation works in R2014b and later releases. This code gets and sets 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 app 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()
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 Apps

To set up a GUIDE app 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 friendly name that describes 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 name 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 Apps Created Programmatically

This app 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 Apps

To set up a GUIDE app 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)
% uwait(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 Toolbar.

Create Nested Callback Functions (Programmatic Apps)

You can nest callback functions inside the main function of a programmatic app. 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.

```
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 apps.

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 get 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 Apps Created Programmatically

To use `guidata` in a programmatic app, store the data with some initial values in the main function. Then you can retrieve and modify the data in any callback function.

The following code is a simple example of a programmatic app 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.

```
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 Apps

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.

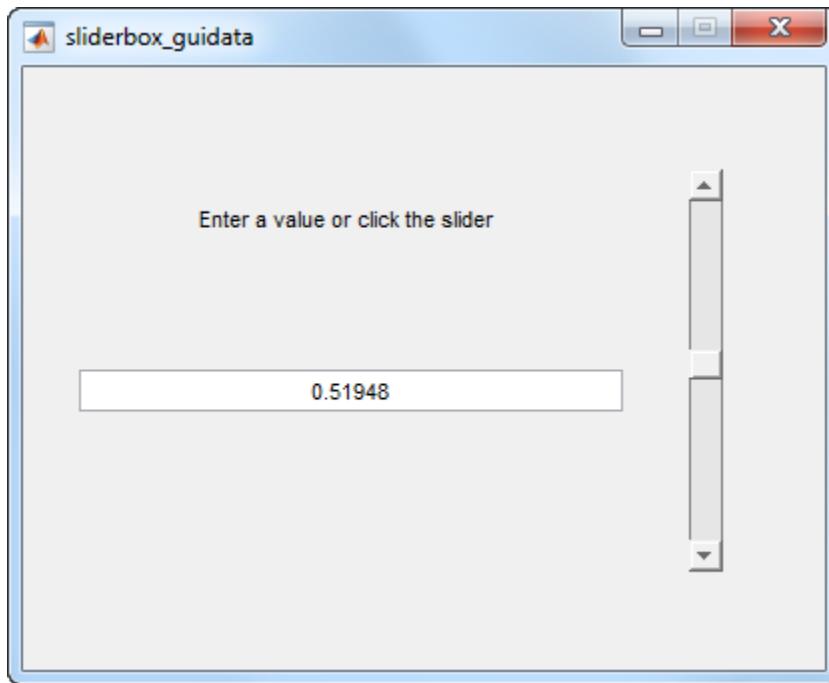
```
function slider1_Callback(hObject, eventdata,handles)
% hObject    handle to slider1 (see GCBO)
% eventdata reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)

% 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
```

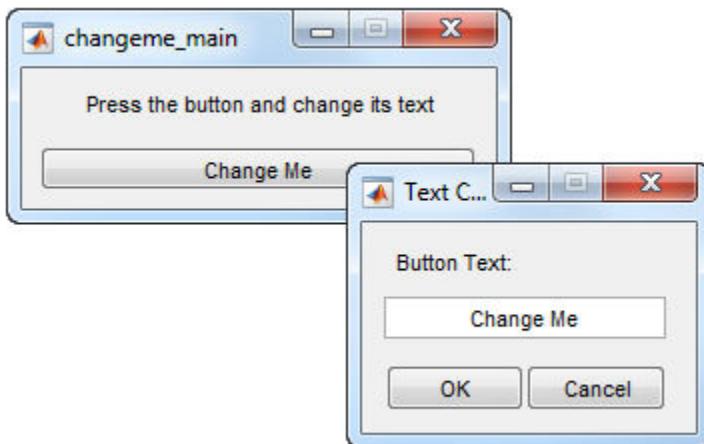
GUIDE Example: Share Slider Data Using guidata

Here is a prebuilt GUIDE app that uses the `guidata` function to share data between a slider and a text field. When you move the slider, the number displayed in the text field changes to show the new slider position.



GUIDE Example: Share Data Between Two Apps

Here is a prebuilt GUIDE app that uses application data and the `guidata` function to share data between two dialog boxes. When you enter text in the second dialog box and click **OK**, the button label changes in the first dialog box.



In `changeme_main.m`, the `buttonChangeMe_Callback` function executes this command to display the second dialog box:

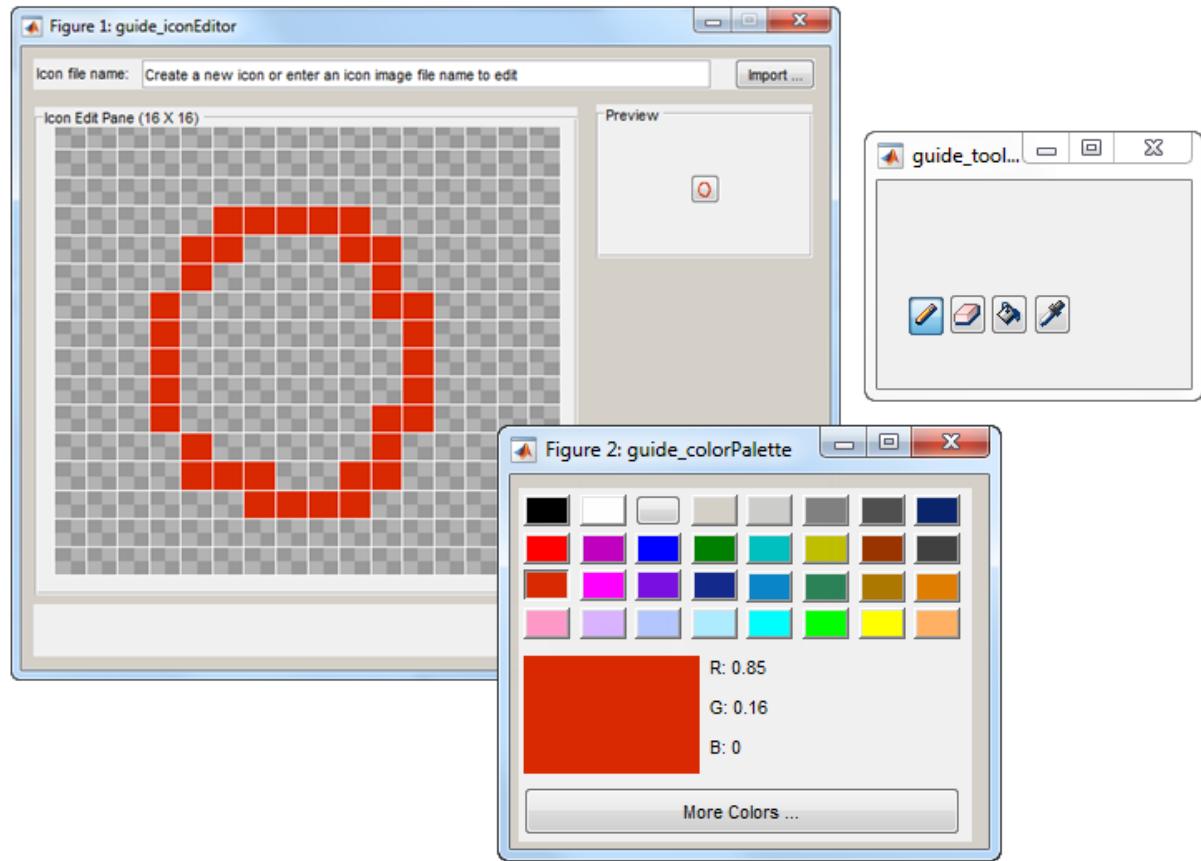
```
changeme_dialog('changeme_main', handles.figure)
```

The `handles.figure` input argument is the `Figure` object for the **changeme_main** dialog box.

The `changeme_dialog` function retrieves the `handles` structure from the `Figure` object. Thus, the entire set of components in the **changeme_main** dialog box is available to the second dialog box.

GUIDE Example: Share Data Among Three Apps

Here is a prebuilt GUIDE app that uses `guidata` and `Userdata` to share data among three app windows. The large window is an icon editor that accepts information from the tool palette and color palette windows.



In `guide_inconeditor.m`, the function `guide_iconeditor_OpeningFcn` contains this command:

```
colorPalette = guide_colorpalette('iconEditor', hObject)
```

The arguments are:

- '`iconEditor`' specifies that a callback in the **guide_iconeditor** window triggered the execution of the function.
- `(hObject`) is the Figure object for the **guide_iconeditor** window.

- `colorPalette` is the `Figure` object for the **guide_colorPalette** window.

Similarly, `guide_iconeditor_OpeningFcn` calls the `guide_toolpalette` function with similar input and output arguments.

Passing the `Figure` object between these functions allows the **guide_iconEditor** window to access the `handles` structure of the other two windows. Likewise, the other two windows can access the `handles` structure for the **guide_iconEditor** window.

See Also

Related Examples

- “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`, `pause`, or `waitbar`.

- 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](#) page.

Example

This example shows how to control callback interruption using the `Interruptible` and `BusyAction` properties and a wait bar.

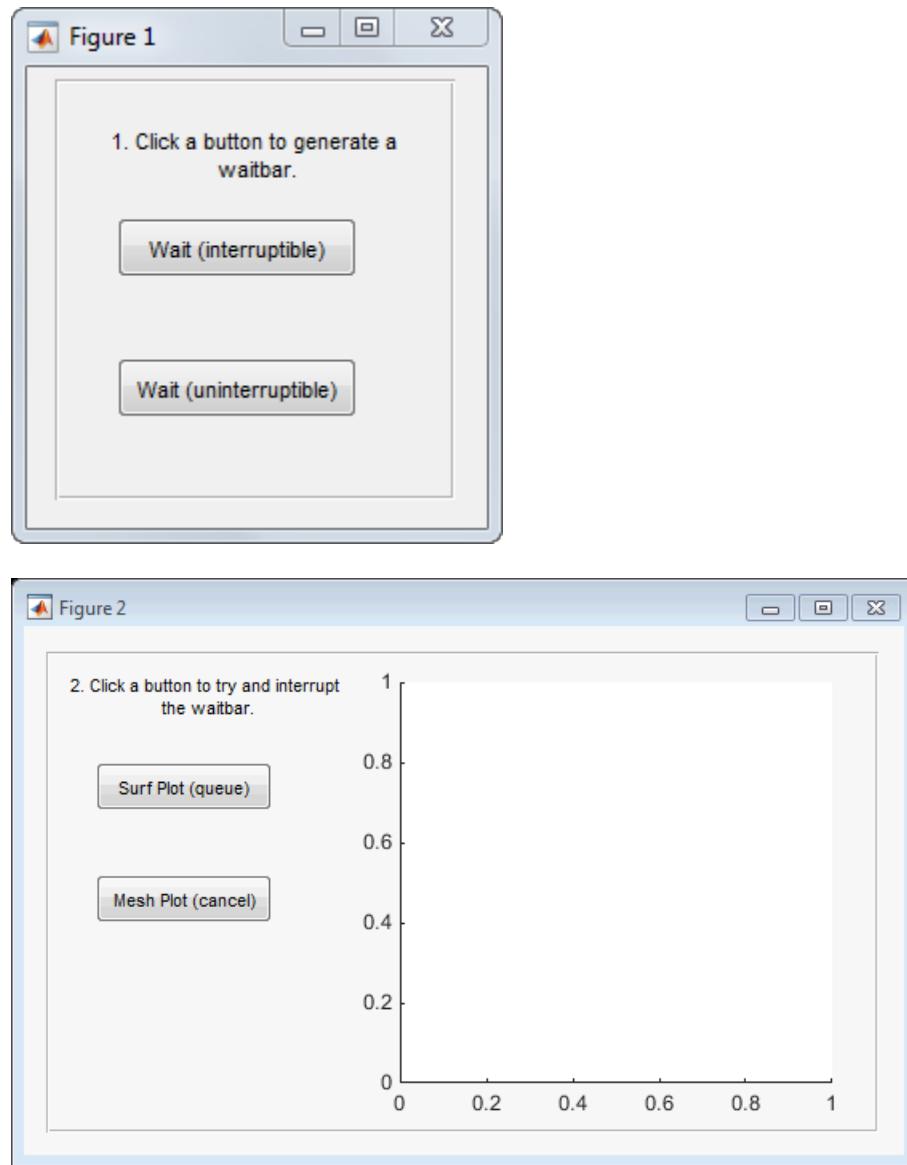
Copy the Source File

- 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)',...
    'Tooltip','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)',...
    'Tooltip','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',...
    'Tooltip','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',...
    'Tooltip','BusyAction = cancel',...
    'Callback',@mesh_cancel);
```

See Also

`drawnow` | `timer` | `uiwait` | `waitFor`

Related Examples

- “Write Callbacks for Apps Created Programmatically” on page 10-5
- “Automatically Refresh Plot in a GUIDE App” on page 8-24
- “Schedule Command Execution Using Timer”
- “Finding Code Bottlenecks”

Examples of Programmatic Apps

Programmatic App that Displays a Table

This example shows how to create a table in an app using the `uitable` function. It also shows how to modify the appearance of the table and how to restrict editing of the table in the running app.

Create a Table Containing Simple Numeric Data

The `uitable` function creates an empty table. You can populate the table by setting the `Data` property. For example, you can create a table containing magic square values.

```
f = figure('Position', [100 100 752 250]);
t = uitable('Parent', f, 'Position', [25 50 700 200], 'Data', magic(10))
```

	1	2	3	4	5	6	7	8	9	
1	92	99	1	8	15	67	74	51		^
2	98	80	7	14	16	73	55	57		
3	4	81	88	20	22	54	56	63		
4	85	87	19	21	3	60	62	69		
5	86	93	25	2	9	61	68	75		
6	17	24	76	83	90	42	49	26		
7	23	5	82	89	91	48	30	32		
8	79	6	13	95	97	29	31	38		
9	10	12	94	96	78	35	37	44		>

```
t =
Table with properties:

    Data: [10x10 double]
    ColumnWidth: 'auto'
    ColumnEditable: []
    CellEditCallback: ''
    Position: [25 50 700 200]
    Units: 'pixels'
```

Show all properties

Create a Table Containing Mixed Data Types

Display mixed data types by setting the `Data` property to a cell array.

```
load patients LastName Age Weight Height SelfAssessedHealthStatus  
PatientData = [LastName num2cell([Age Weight Height]) SelfAssessedHealthStatus];  
  
t.Data = PatientData;
```

	1	2	3	4	5
1	Smith	38	176	71	Excellent
2	Johnson	43	163	69	Fair
3	Williams	38	131	64	Good
4	Jones	40	133	67	Fair
5	Brown	49	119	64	Good
6	Davis	46	142	68	Good
7	Miller	33	142	64	Good
8	Wilson	40	180	68	Good
9	Moore	28	183	68	Excellent
10	Taylor	31	132	66	Excellent

Customize the Display

You can customize the display of a table in several ways. Use the `ColumnName` property to add column headings. To create multiline headings, use the pipe (|) symbol.

```
t.ColumnName = {'LastName', 'Age', 'Weight', 'Height', 'Self Assessed|Health Status'};
```

13 Examples of Programmatic Apps

	LastName	Age	Weight	Height	Self Assessed Health Status
1	Smith	38	176	71	Excellent
2	Johnson	43	163	69	Fair
3	Williams	38	131	64	Good
4	Jones	40	133	67	Fair
5	Brown	49	119	64	Good
6	Davis	46	142	68	Good
7	Miller	33	142	64	Good
8	Wilson	40	180	68	Good
9	Moore	28	183	68	Excellent

To adjust the widths of the columns, specify the `ColumnWidth` property. The `ColumnWidth` property is a 1-by-N cell array, where N is the number of columns in the table. Set a specific column width, or let MATLAB® set the width based on the contents.

```
t.ColumnWidth = {100, 'auto', 'auto', 'auto', 150};
```

	LastName	Age	Weight	Height	Self Assessed Health Status
1	Smith	38	176	71	Excellent
2	Johnson	43	163	69	Fair
3	Williams	38	131	64	Good
4	Jones	40	133	67	Fair
5	Brown	49	119	64	Good
6	Davis	46	142	68	Good
7	Miller	33	142	64	Good
8	Wilson	40	180	68	Good
9	Moore	28	183	68	Excellent

To remove the row names, set the `RowName` property to an empty array ([]).

```
t.RowName = [];
```

Last Name	Age	Weight	Height	Self Assessed Health Status
Smith	38	176	71	Excellent
Johnson	43	163	69	Fair
Williams	38	131	64	Good
Jones	40	133	67	Fair
Brown	49	119	64	Good
Davis	46	142	68	Good
Miller	33	142	64	Good
Wilson	40	180	68	Good
Moore	28	183	68	Excellent

Resize the table and remove any extra space using the **Position** property.

```
t.Position = [15 25 495 200];
```

Last Name	Age	Weight	Height	Self Assessed Health Status
Smith	38	176	71	Excellent
Johnson	43	163	69	Fair
Williams	38	131	64	Good
Jones	40	133	67	Fair
Brown	49	119	64	Good
Davis	46	142	68	Good
Miller	33	142	64	Good
Wilson	40	180	68	Good
Moore	28	183	68	Excellent

By default, tables use row striping. To turn off row striping, set the **RowStriping** property to 'off'. To control the colors of the stripes, set two different colors for the **BackgroundColor** property. Use the **ForegroundColor** property to control the color of the text.

```
t.BackgroundColor = [.4 .4 .4; .4 .4 .8];
t.ForegroundColor = [1 1 1];
```

LastName	Age	Weight	Height	Self Assessed Health Status
Smith	38	176	71	Excellent
Johnson	43	163	69	Fair
Williams	38	131	64	Good
Jones	40	133	67	Fair
Brown	49	119	64	Good
Davis	46	142	68	Good
Miller	33	142	64	Good
Wilson	40	180	68	Good
Moore	28	183	68	Excellent

Restrict Editing of Cell Values

To restrict the user's ability to edit data in the table, set the `ColumnEditable` property. By default, data cannot be edited in the app. Setting the `ColumnEditable` property to `true` for a column allows the user to edit data in that column.

```
t.ColumnEditable = [false true true true true];
```

LastName	Age	Weight	Height	Self Assessed Health Status
Smith	38	176	71	Excellent
Johnson	43	163	69	Fair
Williams	38	131	64	Good
Jones	40	133	67	Fair
Brown	49	119	64	Good
Davis	46	142	68	Good
Miller	33	142	64	Good
Wilson	40	180	68	Good
Moore	28	183	68	Excellent

Change Column Format

The `ColumnFormat` property controls how data is displayed and edited. To specify choices in a drop-down list, specify a cell array of character vectors as the column format. In this example, the *Self Assessed Health Status* column has a drop-down list containing four options: `Excellent`, `Fair`, `Good`, and `Poor`.

```
t.ColumnFormat = {[[] [] [] []] {'Excellent', 'Fair', 'Good', 'Poor'}};
```

Last Name	Age	Weight	Height	Self Assessed Health Status
Smith	38	176	71	Excellent
Johnson	43	163	69	Fair
Williams	38	131	64	Good
Jones	40	133	67	Fair
Brown	49	119	64	Good
Davis	46	142	68	Good
Miller	33	142	64	Good
Wilson	40	180	68	Good
Moore	28	183	68	Excellent

Create a Callback

The Table object has two commonly used callbacks. The CellSelectionCallback executes when the user selects a different cell. The CellEditCallback executes when the user changes a value in a cell.

```
t.CellEditCallback = @ageCheckCB;
```

For example, if you want the *Age* column to contain values between 0 and 120, set the CellEditCallback to a function such as this one:

```
function ageCheckCB(src, eventdata)
if (eventdata.Indices(2) == 2 && ...
    (eventdata.NewData < 0 || eventdata.NewData > 120)) % check if column 2
    tableData = src.Data;
    tableData{eventdata.Indices(1), eventdata.Indices(2)} = eventdata.PreviousData;
    src.Data = tableData; % set the data back to its original value
    warning('Age must be between 0 and 120.') % warn the user
end
end
```

If the user enters a value that is outside the acceptable range, the callback function returns a warning and sets the cell value back to the previous value.

Get All Table Properties

To see all the properties of the table, use the `get` command.

```
get(t)

    BackgroundColor: [2x3 double]
    BeingDeleted: 'off'
    BusyAction: 'queue'
    ButtonDownFcn: ''
    CellEditCallback: @ageCheckCB
    CellSelectionCallback: ''
        Children: [0x0 handle]
    ColumnEditable: [0 1 1 1 1]
    ColumnFormat: {[[] [] [] []] {1x4 cell}}
    ColumnName: {5x1 cell}
    ColumnWidth: {[100] 'auto' 'auto' 'auto' [150]}
    CreateFcn: ''
        Data: {100x5 cell}
    DeleteFcn: ''
        Enable: 'on'
        Extent: [0 0 479 1842]
    FontAngle: 'normal'
    FontName: 'MS Sans Serif'
    FontSize: 8
    FontUnits: 'points'
    FontWeight: 'normal'
    ForegroundColor: [1 1 1]
    HandleVisibility: 'on'
        InnerPosition: [15 25 495 200]
    Interruptible: 'on'
        KeyPressFcn: ''
    KeyReleaseFcn: ''
        Layout: [0x0 matlab.ui.layout.LayoutOptions]
    OuterPosition: [15 25 495 200]
        Parent: [1x1 Figure]
        Position: [15 25 495 200]
    RearrangeableColumns: 'off'
        RowName: ''
    RowStriping: 'on'
        Tag: ''
    Tooltip: ''
        Type: 'uitable'
    UIContextMenu: [0x0 GraphicsPlaceholder]
        Units: 'pixels'
```

```
UserData: []
Visible: 'on'
```

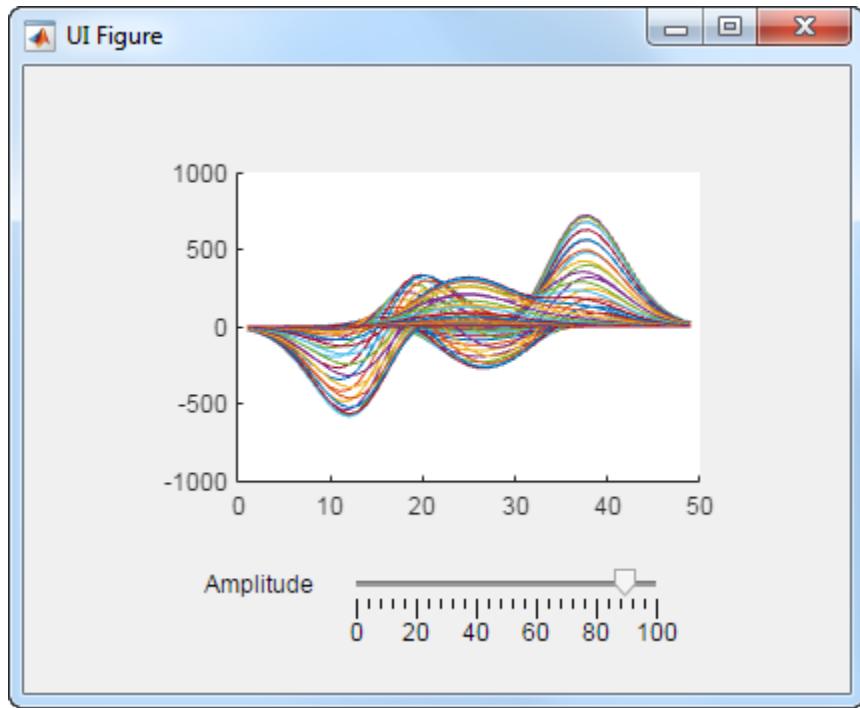

App Designer

App Designer Basics

- “Create and Run a Simple App Using App Designer” on page 14-2
- “App Designer Versus GUIDE” on page 14-6
- “Displaying Graphics in App Designer” on page 14-9
- “App Designer Preferences” on page 14-14

Create and Run a Simple App Using App Designer

App Designer provides a tutorial that guides you through the process of creating a simple app containing a plot and a slider. The slider controls the amplitude of the plotted function. You can create this app by running the tutorial, or you can follow the tutorial steps listed below.



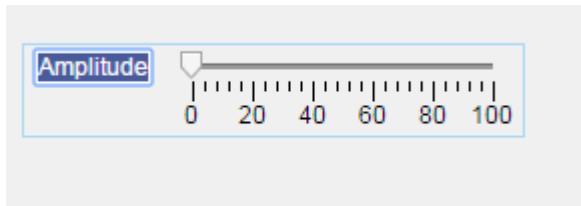
Run the Tutorial

To run the tutorial in App Designer, select **Open > Interactive Tutorial** on the **Designer** tab in the App Designer toolbar.

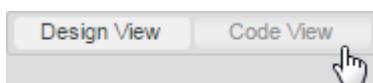
Tutorial Steps for Creating the App

Perform the following steps in App Designer.

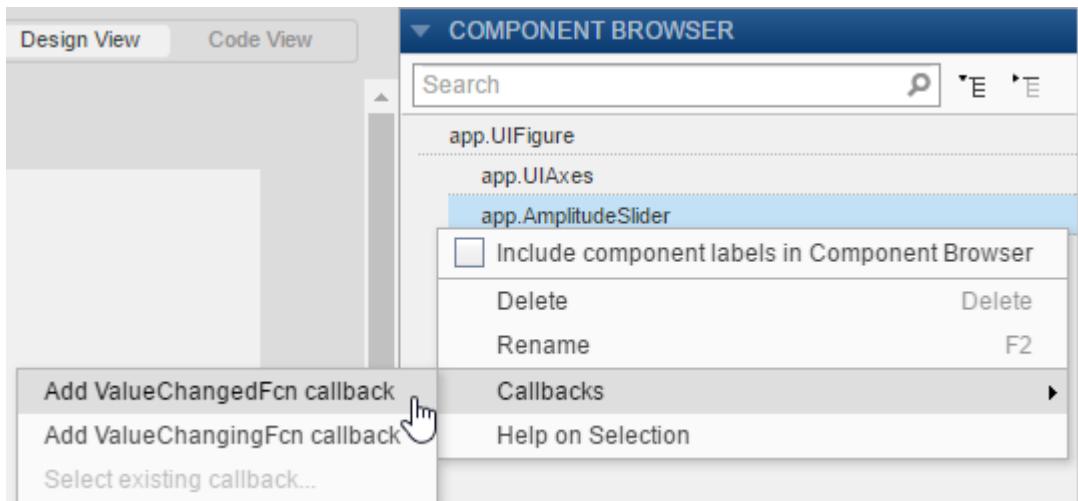
- 1 Drag an **Axes** component from the **Component Library** onto the canvas.
- 2 Drag a **Slider** component from the **Component Library** onto the canvas. Place it below the axes, as in the preceding image.
- 3 Replace the slider label text. Double-click the label and replace the word **Slider** with **Amplitude**.



- 4 Above the canvas, click **Code View** to edit the code. (Notice that you can switch back to edit your layout by clicking **Design View**.)



- 5 In the code view, add a callback function that executes MATLAB commands whenever the user moves the slider. Right-click `app.AmplitudeSlider` in the **Component Browser**. Then select **Callbacks > Add ValueChangedFcn callback** in the context menu. App Designer creates a callback function and places the cursor in the body of that function.



- 6 Plot the `peaks` function in the axes. Add this command to the second line of the `AmplitudeSliderValueChanged` callback:

```
plot(app.UIAxes,value*peaks)
```

Notice that the `plot` command specifies the target axes (`app.UIAxes`) as the first argument. The target axes is always required when you call the `plot` command in App Designer.

- 7 Change the limits of the y-axis by setting the `YLim` property of the `UIAxes` object. Add this command to the third line of the `AmplitudeSliderValueChanged` callback:

```
app.UIAxes.YLim = [-1000 1000];
```

Notice that the command uses dot notation to access the `YLim` property. Always use the pattern **app.Component.Property** to access property values.

- 8 Click **Run** to save and run the app. After saving your changes, your app is available for running again in App Designer, or by typing its name (without the `.mlapp` extension) at the MATLAB command prompt. When you run the app from the command prompt, the file must be in the current folder or on the MATLAB path.

See Also

Related Examples

- “Managing Code in App Designer Code View” on page 17-2
- “Write Callbacks in App Designer” on page 17-18
- “Displaying Graphics in App Designer” on page 14-9

App Designer Versus GUIDE

App Designer is a rich development environment that provides layout and code views, a fully integrated version of the MATLAB Editor, and a large set of interactive components. You can package an app directly from the App Designer toolbar, or you can create a standalone desktop or web app (requires MATLAB Compiler™). It is the recommended environment for building most apps.

If you have existing apps that you developed in GUIDE, consider migrating them to App Designer.

Migrate GUIDE Apps to App Designer

Migrating your app allows you to take advantage of the new components and features in App Designer that GUIDE does not offer. Keep in mind that some components in GUIDE are not available in App Designer, and that you must update your app code to make it compatible with App Designer. See the table below for details.

For assistance in migrating your apps, use the GUIDE to App Designer Migration Tool for MATLAB. For more information, see <https://www.mathworks.com/app-designer/add-ons.html>.

Differences Between App Designer and GUIDE

The main differences between App Designer and GUIDE are the code structure, callback syntax, and the way that you access UI components and share data. This table summarizes the differences.

Difference	GUIDE	App Designer	More Information
Using Figures and Graphics	<p>GUIDE calls the <code>figure</code> function to create the app window.</p> <p>GUIDE calls the <code>axes</code> function to create the axes for displaying plots.</p> <p>All MATLAB graphics functions are supported.</p>	<p>App Designer calls the <code>uifigure</code> function to create the app window.</p> <p>App Designer calls the <code>uiaxes</code> function to create the axes for displaying plots.</p> <p>Most MATLAB graphics functions are supported.</p>	"Displaying Graphics in App Designer" on page 14-9
Using Components	GUIDE creates most components with the <code>uicontrol</code> function.	<p>App Designer has a new set of components that are created with separate functions.</p> <p>App Designer has several components that GUIDE does not support, such as <code>Tree</code>, <code>Gauge</code>, and <code>Switch</code> components.</p>	"App Designer Components" on page 15-2
Accessing Component Properties	<p>GUIDE uses <code>set</code> and <code>get</code> to access component properties. For example:</p> <pre>name = get(HFig, 'Name')</pre>	<p>App Designer uses dot notation to access component properties. For example:</p> <pre>name = app.UIFigure.Name</pre>	"Write Callbacks in App Designer" on page 17-18
Managing App Code	The code is structured as a main function and local functions. All code is editable.	The code is defined as a MATLAB class. Only callbacks, helper functions, and custom properties are editable.	"Managing Code in App Designer Code View" on page 17-2

Difference	GUIDE	App Designer	More Information
Writing Callbacks	Required input arguments are handles, hObject, and eventdata.	Required input arguments are app, and event.	"Write Callbacks in App Designer" on page 17-18
Sharing Data	Use the UserData property, or the guidata, setappdata, or getappdata functions.	Use custom properties.	"Share Data Within App Designer Apps" on page 17-28

See Also

Related Examples

- "Create and Run a Simple App Using App Designer" on page 14-2
- "Displaying Graphics in App Designer" on page 14-9
- "Ways to Build Apps" on page 1-2

Displaying Graphics in App Designer

In this section...

- “Calling Graphics Functions” on page 14-9
- “Displaying Plots Using Other Types of Axes” on page 14-10
- “Unsupported Functionality” on page 14-11

Displaying graphics in App Designer requires a different workflow than you typically use at the MATLAB command line. Once you understand this workflow and a few special cases, you will know how to call the functions you need for displaying almost any type of plot.

Calling Graphics Functions

Many of the graphics functions in MATLAB (and MATLAB toolboxes) have an argument for specifying the target axes or parent object. This argument is optional in most contexts, but when you call these functions in App Designer, you must specify that argument. Otherwise, MATLAB uses `gcf` or `gca` to get the target object for the operation. However, `gcf` cannot return an App Designer figure, and `gca` cannot return any axes within an App Designer figure. Thus, omitting the argument might produce unexpected results.

This code shows how to specify the target axes when plotting two lines. The first argument passed to `plot` and `hold` is `app.UIAxes`, which is the default name for the App Designer axes.

```
plot(app.UIAxes,[1 2 3 4],'-r');  
hold(app.UIAxes);  
plot(app.UIAxes,[10 9 4 7], '--b');
```

Some functions (such as `imshow` and `triplot`) use a name-value pair argument to specify the target object. For example, this code shows how to call the `imshow` function in App Designer.

```
imshow('peppers.png','Parent',app.UIAxes);
```

Whether you specify the target object as the first argument or a name-value pair argument depends on the function. See the documentation for the specific function you want to use to determine the appropriate arguments.

Displaying Plots Using Other Types of Axes

You can create most 2-D and 3-D plots using the App Designer axes (a `uiaxes` object). Starting in R2018b, you can create additional plots, such as those listed in the following table. Most of these plots require a different type of parent object and additional lines of code in your app. All of them use normalized units by default.

Functions	Coding Details
<code>polarplot</code> <code>polarhistogram</code> <code>polarscatter</code> <code>compass</code>	Create the polar axes by calling the <code>polaraxes</code> function. Specify the parent container as the first input argument (for example, <code>app.UIFigure</code>). Then call the plotting function with the polar axes as the first argument. For example: <pre>theta = 0:0.01:2*pi; rho = sin(2*theta).*cos(2*theta); pax = polaraxes(app.UIFigure); polarplot(pax,theta,rho)</pre>
<code>subplot</code>	Follow these steps: <ol style="list-style-type: none">1 Set the <code>AutoResizeChildren</code> property to '<code>off</code>'. Subplots do not support automatic resize behavior. You can set this property in the App Designer Component Properties pane or in your code.2 Specify the parent container using the '<code>Parent</code>' name-value pair argument when you call <code>subplot</code>. Also, specify an output argument to store the axes.3 Call the plotting function with the axes as the first input argument. For example: <pre>app.UIFigure.AutoScaleChildren = 'off'; ax1 = subplot(1,2,1,'Parent',app.UIFigure); ax2 = subplot(1,2,2,'Parent',app.UIFigure); plot(ax1,[1 2 3 4]) plot(ax2,[10 9 4 7])</pre>

Functions	Coding Details
<p><code>pareto</code> <code>plotmatrix</code></p>	<p>Follow these steps:</p> <ol style="list-style-type: none"> 1 Set the <code>AutoResizeChildren</code> property to '<code>off</code>'. These plots do not support automatic resize behavior. You can set this property in the App Designer Component Properties pane or in your code. 2 Create the axes by calling the <code>axes</code> function. Specify the parent container as the first input argument (for example, <code>app.UIFigure</code>). 3 Call the <code>pareto</code> or <code>plotmatrix</code> function with the axes as the first input argument. <p>For example:</p> <pre>app.UIFigure.AutoScaleChildren = 'off'; ax = axes(app.UIFigure); pareto(ax,[10 20 40 40])</pre>
<p><code>geobubble</code> <code>heatmap</code> <code>scatterhistogram</code> <code>stackedplot</code> <code>wordcloud</code></p>	<p>Specify the parent container when calling these functions (for example, <code>app.UIFigure</code>).</p> <p>For example:</p> <pre>h = heatmap(app.UIFigure,rand(10));</pre>
<p><code>geoplot</code> <code>geoscatter</code> <code>geodensityplot</code></p>	<p>Create the geographic axes by calling the <code>geoaxes</code> function. Specify the parent container as the first input argument (for example, <code>app.UIFigure</code>). Then call the plotting function with the axes as the first argument. For example:</p> <pre>latSeattle = 47 + 37/60; lonSeattle = -(122 + 20/60); gx = geoaxes(app.UIFigure); geoplot(gx,latSeattle,lonSeattle)</pre>

Unsupported Functionality

As of R2018b, some graphics functionality is not supported in App Designer. This table lists the functionality that is relevant to most app building workflows.

Category	Not Supported
Animation	<ul style="list-style-type: none"><code>movie</code>, <code>getframe</code>.
Retrieving and Saving Data	<ul style="list-style-type: none">For example, <code>hgexport</code>, <code>hgload</code>, <code>hgsave</code>, and <code>saveas</code>.
Utilities	<ul style="list-style-type: none"><code>copyobj</code>, <code>findfigs</code>, <code>gca</code>, <code>gcf</code>, <code>gco</code>, <code>clf</code>, <code>print</code>, <code>ginput</code>, <code>gtext</code>.
Functions not Recommended	<ul style="list-style-type: none">For example, <code>ezplot</code> and <code>hist</code>.
Axes Toolbar	<ul style="list-style-type: none">The axes toolbar is not supported for any plots in App Designer. However, you can enable some of the interactive features by calling the <code>pan</code>, <code>zoom</code>, or <code>rotate3d</code> functions. Specify the axes as the first argument, and specify '<code>on</code>', '<code>off</code>', or a scale factor (for <code>zoom</code>) as the second argument.
Axes Interactions	<ul style="list-style-type: none">Pan, zoom, and rotate (for 3-D) are supported for plots you create using <code>axes</code> and <code>uiaxes</code>. Data tips are not supported.Some interactions might not work in <code>geobubble</code>, <code>heatmap</code>, <code>scatterhistogram</code>, <code>stackedplot</code>, or plots you create using <code>geoaxes</code>.
Axes in Grid Layout Managers	<ul style="list-style-type: none"><code>axes</code>, <code>polaraxes</code>, <code>geoaxes</code>, <code>geobubble</code>, <code>heatmap</code>, <code>scatterhistogram</code>, <code>stackedplot</code>, and <code>wordcloud</code> do not support grid layout managers. To work around this limitation, place the axes or chart into a panel. Then place the panel into the grid.<code>uiaxes</code> does not have this limitation.
Axes in Scrollable Containers	<ul style="list-style-type: none"><code>axes</code>, <code>polaraxes</code>, <code>geoaxes</code>, <code>geobubble</code>, <code>heatmap</code>, <code>scatterhistogram</code>, <code>stackedplot</code>, and <code>wordcloud</code> do not support scrollable containers. To work around this limitation, place the axes or chart into a panel with the <code>Scrollable</code> property set to '<code>off</code>'. Then place the panel into the scrollable container.<code>uiaxes</code> does not have this limitation.
Components	<ul style="list-style-type: none"><code>uicontrol</code>, <code>uitoolbar</code>, and <code>uicontextmenu</code> are not supported in App Designer. However, App Designer supports a new set of components, including tabs, trees, switches, and gauges. For a full list of supported components, see "Designing Apps in App Designer".

Category	Not Supported
Properties	<ul style="list-style-type: none">Some component properties are not supported in App Designer. For a list of supported properties for a particular component, see its property page on “Designing Apps in App Designer”.

See Also

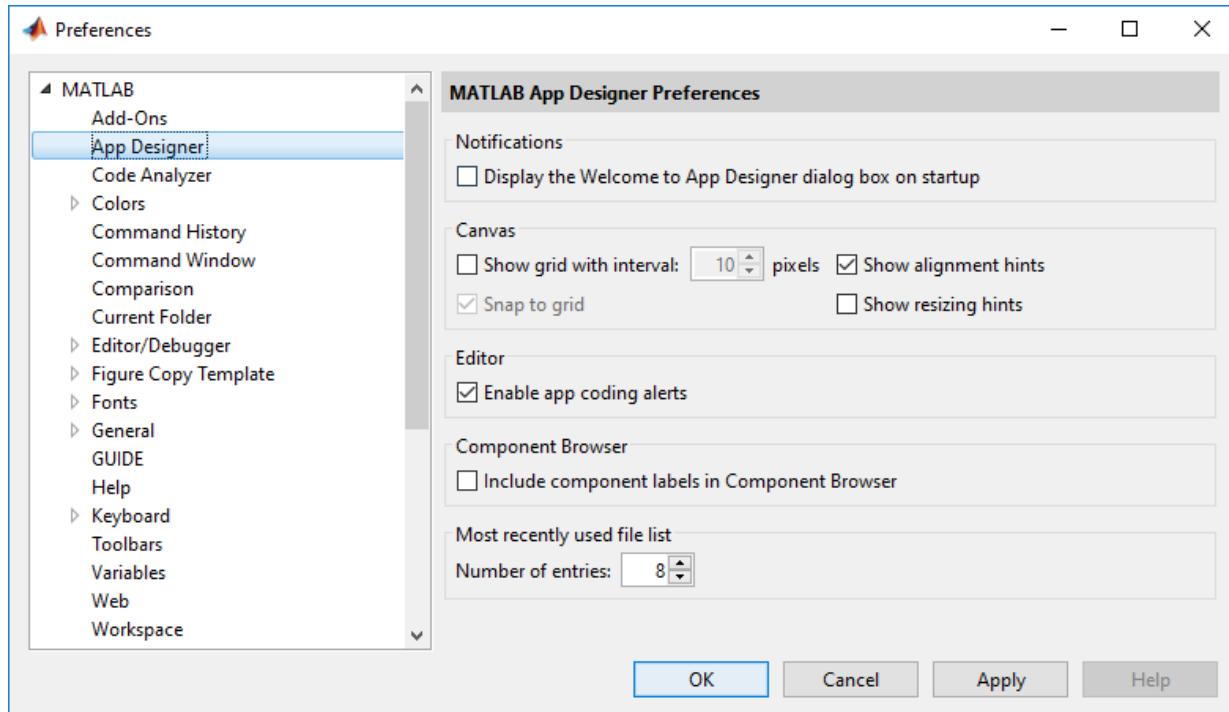
[UI Figure](#) | [UIAxes](#)

More About

- Graphics Support in App Designer (R2017b - R2018a)
- Graphics Support in App Designer (R2016a - R2017a)
- “App Designer Versus GUIDE” on page 14-6
- “Polar Plotting App in App Designer” on page 18-12

App Designer Preferences

You can set App Designer preferences in the MATLAB Preferences dialog box. To open the dialog box, click  **Preferences** in the MATLAB Toolstrip. Then, select App Designer in the left pane.



This table describes each option in the right pane.

Option	Description
Display the Welcome to App Designer dialog box on startup	When selected, a dialog box displays every time you start App Designer. The dialog box contains links to introductory information and a brief tutorial.

Option	Description
Show grid with interval	When selected, App Designer overlays a grid onto the canvas as an alignment aide. You can change the grid spacing to a specific number of pixels. The default spacing is 10.
Snap to grid	When selected, the upper left corner of a component always snaps to the intersection of two grid lines whenever you resize or move the component on the canvas.
Show alignment hints	When selected, App Designer displays alignment hints as you resize or move a component on the canvas.
Show resizing hints	When selected, App Designer displays the size of a component as you resize it on the canvas.
Enable app coding alerts	When selected, App Designer flags coding problems in the editor as you write code.
Include component labels in Component Browser	When selected, labels included with components (such as edit fields) appear as separate items in the Component Browser . When this item is not selected, those labels do not appear in the Component Browser .
Number of entries (most recently used file list)	This number specifies how many of the most recently accessed apps appear under the Recent Files section of the Open menu in the Designer tab.

See Also

Related Examples

- “Lay Out Apps in App Designer” on page 16-2

Component Choices and Customizations

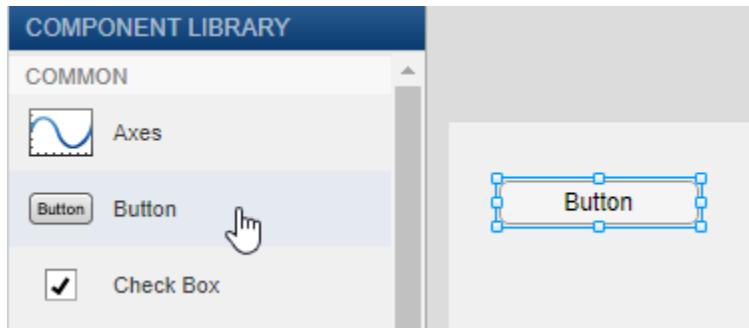
- “App Designer Components” on page 15-2
- “Create Menus for App Designer Apps” on page 15-10
- “Table Array Data Types in App Designer Apps” on page 15-17
- “Add UI Components to App Designer Programmatically” on page 15-24

App Designer Components

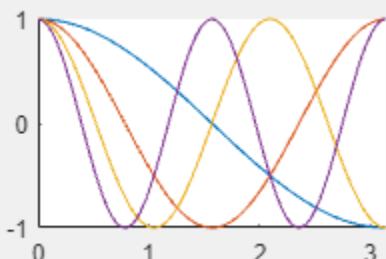
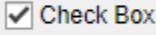
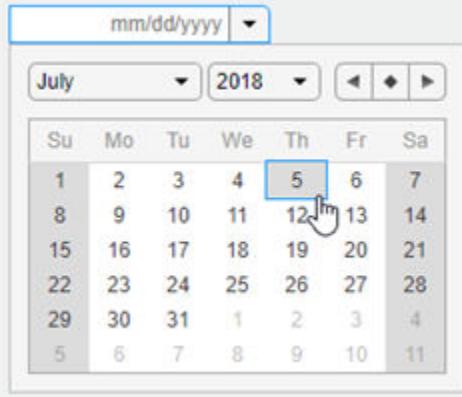
App Designer provides a large set of components for designing modern, full-featured applications. The tables below list the components that are available in the **Component Library**:

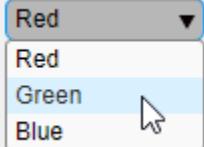
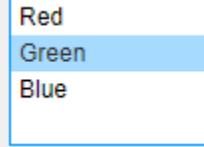
- “Common Components” on page 15-3 — Include axes for creating plots, and several components that respond to interactions, such as buttons, sliders, drop-down lists, and trees.
- “Containers and Figure Tools” on page 15-6 — Include panels and tabs for grouping components, as well as menu bars.
- “Instrumentation” on page 15-7 — Include gauges and lamps for visualizing status, as well as knobs and switches for selecting input parameters.

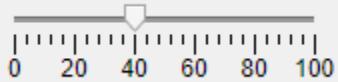
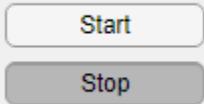
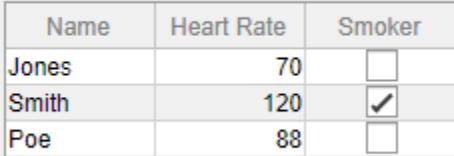
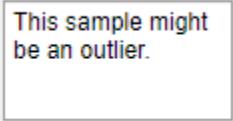
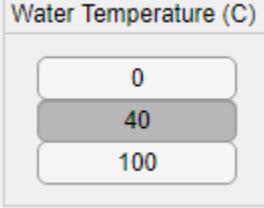
To add a component to your app, drag it onto the canvas from the **Component Library**. Then use the **Component Properties** pane to modify characteristics of the component, such as the color, font, or text.



Common Components

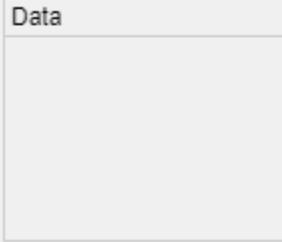
Component	Example	More Information
Axes		UIAxes
Button		Button
Check Box		CheckBox
Date Picker		DatePicker Properties

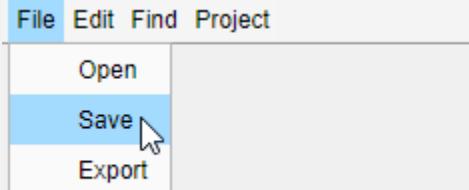
Component	Example	More Information
Drop Down		DropDown
Edit Field (Numeric)	Sample Size <input type="text" value="12"/>	NumericEditField
Edit Field (Text)	Name <input type="text" value="Cleve"/>	EditField
Label	Select an Option	Label
List Box		ListBox
Radio Button Group	<p>Select a Color</p> <p><input type="radio"/> Red</p> <p><input checked="" type="radio"/> Green</p> <p><input type="radio"/> Blue</p>	ButtonGroup RadioButton

Component	Example	More Information
Slider		Slider
Spinner		Spinner
State Button		StateButton
Table		Table
Text Area		TextArea
Toggle Button Group		ButtonGroup ToggleButton

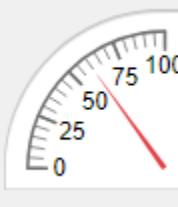
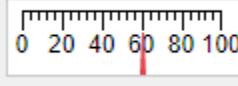
Component	Example	More Information
Tree		Tree TreeNode

Containers and Figure Tools

Component	Example	More Information
Panel		Panel
Tab Group		TabGroup Tab

Component	Example	More Information
Menu Bar		Menu

Instrumentation

Component	Example	More Information
Gauge		Gauge
90 Degree Gauge		NinetyDegreeGauge
Linear Gauge		LinearGauge

Component	Example	More Information
Semicircular Gauge		SemicircularGauge
Knob		Knob
Discrete Knob		DiscreteKnob
Lamp		Lamp
Switch		Switch

Component	Example	More Information
Rocker Switch		RockerSwitch
Toggle Switch		ToggleSwitch

See Also

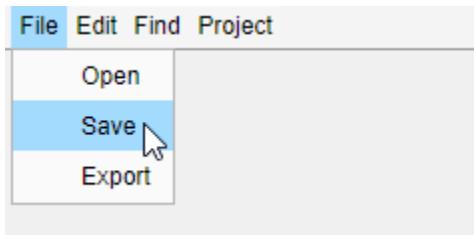
Related Examples

- “Create and Run a Simple App Using App Designer” on page 14-2
- “Add UI Components to App Designer Programmatically” on page 15-24

Create Menus for App Designer Apps

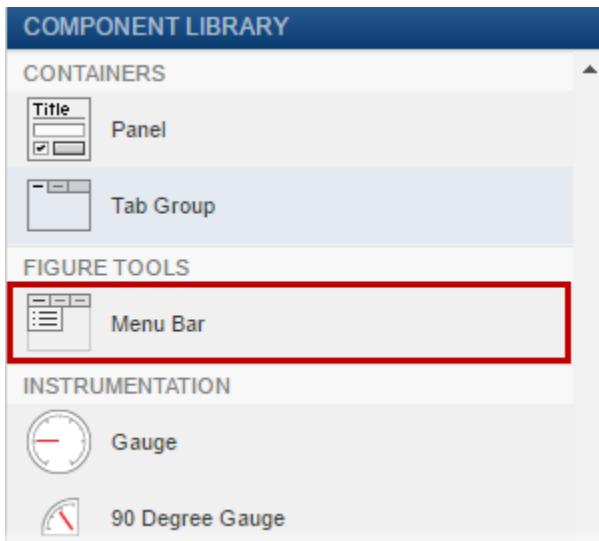
Note For information on creating menus in GUIDE, see “Create Menus for GUIDE Apps” on page 6-78.

A common way to organize tasks in your app is to arrange them in menus at the top of the window. Most apps group similar tasks into categories, where the top-level menus display the title of each category. For example, many apps list the **Open**, **Close**, and **Print** tasks under the top-level menu called **File**.

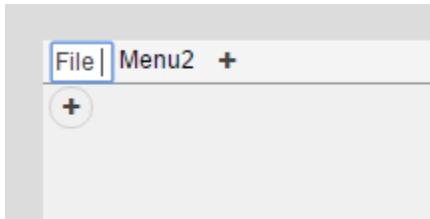


Create and Arrange Menus

Add a menu bar by dragging a **Menu Bar** from the **Figure Tools** section of the **Component Library** onto the canvas. A menu bar that contains two menus snaps into place at the top of the canvas.



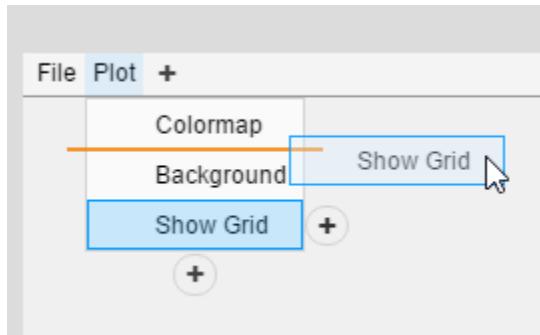
Change the menu text by typing directly on the canvas. Commit your changes and advance to the next menu item by pressing the **Tab** key.



Add menu items by clicking one of the + buttons below or to the right of the existing items. Alternatively, you can press the down and right arrow keys.

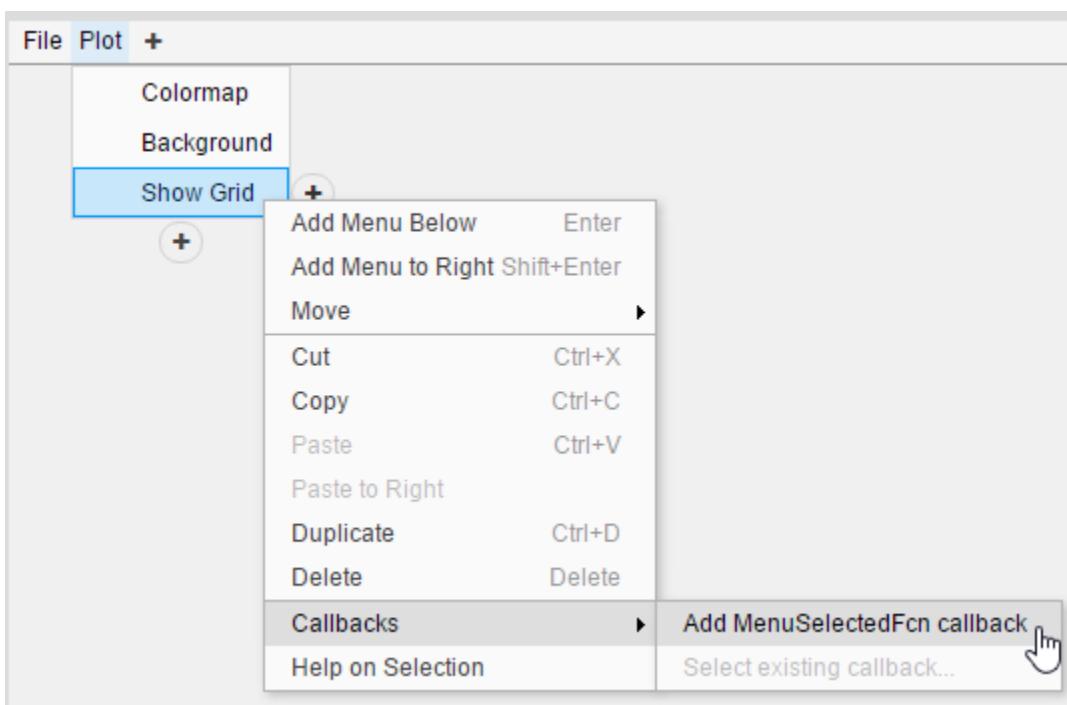


Reorder the menu items by selecting and dragging them to different locations within the menu.



Add Callbacks to Menu Items

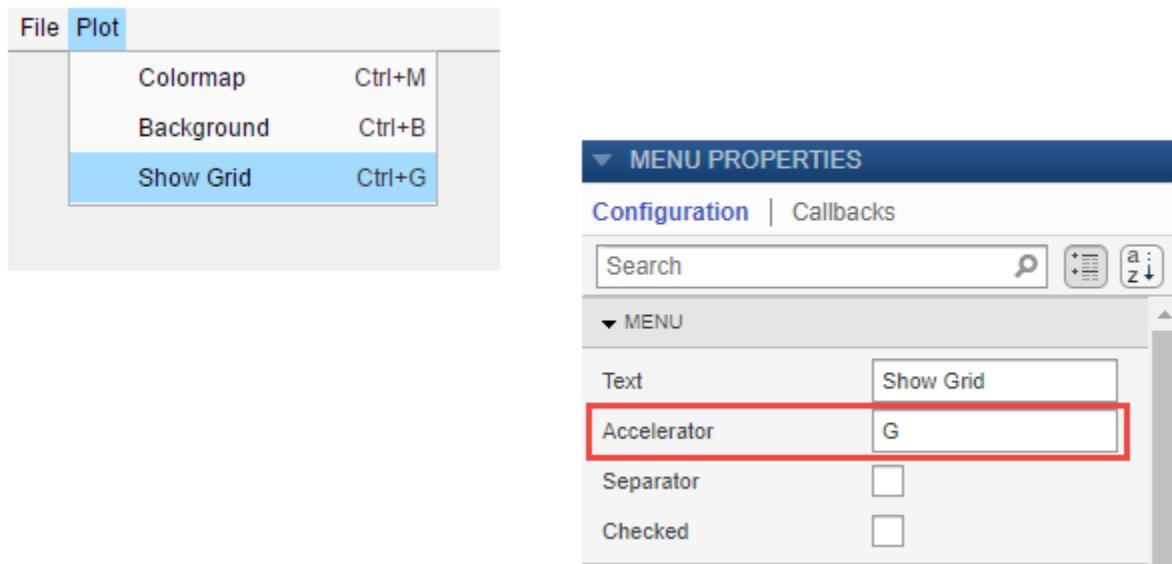
To execute a command when the user selects a menu item, add a callback by right-clicking the menu item in the canvas and selecting **Callbacks > Add MenuSelectedFcn callback**.



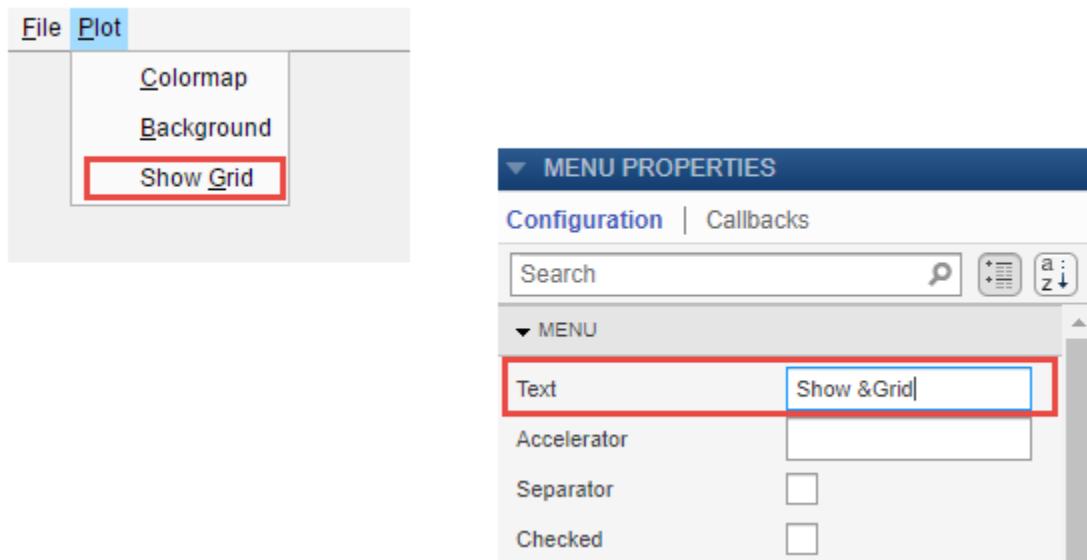
A common practice is to share callbacks between menu items and other UI components in the app. This practice allows your users accomplish tasks in different ways, depending on how they like to work. For more information, see “Use One Callback for Multiple App Designer Components” on page 17-35.

Create Keyboard Shortcuts

Add menu shortcuts that execute the `MenuSelectedFcn` when the user presses a specific key sequence. Accelerators execute the callback when the user holds down the **Ctrl** key and presses another key that you define in the **Menu Properties** pane. The accelerator key is not case-sensitive, and it always displays as a capital letter next to the menu item in the app. For more information, see the `Accelerator` property description.

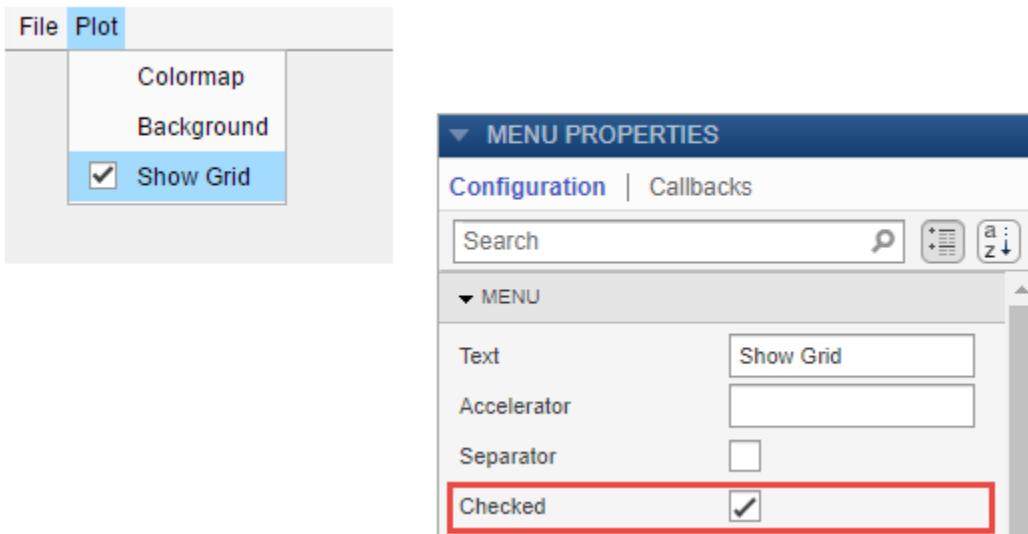


Mnemonics allow users to navigate through menu items by holding down the **Alt** key and pressing the key of the underlined character shown in the menu text. To specify a mnemonic character, go to the **Menu Properties** pane and insert an ampersand (&) before one of the characters in the menu text. To make this behavior work, all menu items must have accelerators. For more information, see the **Text** property description.



Use Check Marks to Indicate Status

If you create a menu item that changes the state of some aspect of your app, you can indicate that state using a check mark. You can control whether the check mark displays by default by selecting **Checked** in the **Menu Properties** pane. You can also control whether the check mark appears by setting the **Checked** property in the callback function. Only leaf menu items can display check marks.



Here is an example of a callback function that toggles the state of an axes grid. It also changes the state of the menu check mark.

```
function ShowGridMenuItemSelected(app, event)
    grid(app.UIAxes);
    if strcmp(app.ShowGridMenuItem.Checked, 'on')
        app.ShowGridMenuItem.Checked = 'off';
    else
        app.ShowGridMenuItem.Checked = 'on';
    end
end
```

See Also

Menu Properties

More About

- “Use One Callback for Multiple App Designer Components” on page 17-35
- “App Designer Keyboard Shortcuts” on page 19-2

Table Array Data Types in App Designer Apps

Note Only App Designer apps and figures created with the `uifigure` function support table arrays. For information on displaying table data in traditional figures, see “Programmatic App that Displays a Table” on page 13-2.

Table arrays are useful for storing tabular data as MATLAB variables. For example, you can call the `readtable` function to create a table array from a spreadsheet.

Table UI components, by contrast, are user interface components that display tabular data in apps. Starting in R2018a, the types of data you can display in a Table UI component include table arrays.

When you display table array data in apps, you can take advantage of the interactive features for certain data types. And unlike other types of arrays that Table UI components support, table array data does not display according to the `ColumnFormat` property of the Table UI component.

Logical Data

In a Table UI component, logical values display as check boxes. `true` values are checked, whereas `false` values are unchecked. When the `ColumnEditable` property of the Table UI component is `true`, the user can select and clear the check boxes in the app.

```
f = uifigure;
tdata = table([true; true; false]);
uit = uitable(f,'Data',tdata);
```

	Var1
1	<input checked="" type="checkbox"/>
2	<input checked="" type="checkbox"/>
3	<input type="checkbox"/>

Categorical Data

categorical values can appear as drop-down lists or as text. The categories appear in drop-down lists when the `ColumnEditable` property of the Table UI component is `true`. Otherwise, the categories display as text without a drop-down list.

```
f = uifigure;
cnames = categorical({'Blue';'Red'},{'Blue','Red'});
w = [400; 700];
tdata = table(cnames,w,'VariableNames',{'Color','Wavelength'});
uit = uitable(f,'Data',tdata,'ColumnEditable',true);
```

	Color	Wavelength
1	Blue	400
2	Red	700
	Blue	
	Red	

If the `categorical` array is not protected, users can add new categories in the running app by typing in the cell.

Datetime Data

datetime values display according to the `Format` property of the corresponding table variable (a `datetime` array).

```
f = uifigure;
dates = datetime([2016,01,17; 2017,01,20],'Format','MM/dd/yyyy');
m = [10; 9];
tdata = table(dates,m,'VariableNames',{'Date','Measurement'});
uit = uitable(f,'Data',tdata);
```

	Date	Measurement
1	01/17/2016	10
2	01/20/2017	9

To change the format, use dot notation to set the `Format` property of the table variable. Then, replace the data in the Table UI component.

```
tdata.Date.Format = 'dd/MM/yyyy';
uit.Data = tdata;
```

	Date	Measurement
1	17/01/2016	10
2	20/01/2017	9

When the `ColumnEditable` property of the Table UI component is `true`, users can change date values in the app. When the column is editable, the app expects input values that conform to the `Format` property of the `datetime` array. If the user enters an invalid date, the value displayed in the table is `NaT`.

Duration Data

duration values display according to the `Format` property of the corresponding table variable (a duration array).

```
f = uifigure;
mtime = duration([0;0],[1;1],[20;30]);
dist = [10.51; 10.92];
tdata = table(mtime,dist,'VariableNames',{'Time','Distance'});
uit = uitable(f,'Data',tdata);
```

	Time	Distance
1	00:01:20	10.5100
2	00:01:30	10.9200

To change the format, use dot notation to set the `Format` property of the table variable.

```
tdata.Time.Format = 's';
uit.Data = tdata;
```

	Time	Distance
1	80 sec	10.5100
2	90 sec	10.9200

Cells containing duration values are not editable in the running app, even when `ColumnEditable` of the `Table` UI component is `true`.

Nonscalar Data

Nonscalar values display in the app the same way as they display in the Command Window. For example, this table array contains 3-D arrays and `struct` arrays.

```
f = uifigure;
arr = {rand(3,3,3); rand(3,3,3)};
s = {struct; struct};
tdata = table(arr,s,'VariableNames',{'Array','Structure'});
uit = uitable(f,'Data',tdata);
```

	Array	Structure
1	3×3×3 double	1×1 struct
2	3×3×3 double	1×1 struct

A multicolumn table array variable displays as a combined column in the app, just as it does in the Command Window. For example, the `RGB` variable in this table array is a 3-by-3 array.

```
n = [1;2;3];
rgbs = [128 122 16; 0 66 155; 255 0 0];
tdata = table(n,rgbs,'VariableNames',{'ROI','RGB'})

tdata =
3×2 table
    ROI          RGB
    --          -----
    1      128      122      16
```

2	0	66	155
3	255	0	0

The **Table** UI component provides a similar presentation. Selecting an item in the RGB column selects all the subcolumns in that row. The values in the subcolumns are not editable in the running app, even when **ColumnEditable** property of the Table UI component is **true**.

```
f = uifigure;
uit = uitable(f,'Data',tdata);
```

	ROI	RGB		
		1	128	122
	1	1	128	122
	2	2	0	66
	3	3	255	0

Missing Data Values

Missing values display as indicators according to the data type:

- Missing strings display as `<missing>`.
- Undefined categorical values display as `<undefined>`.
- Invalid or undefined numbers or duration values display as `NaN`.
- Invalid or undefined datetime values display as `NaT`.

If the **ColumnEditable** property of the Table UI component is **true**, then the user can correct the values in the running app.

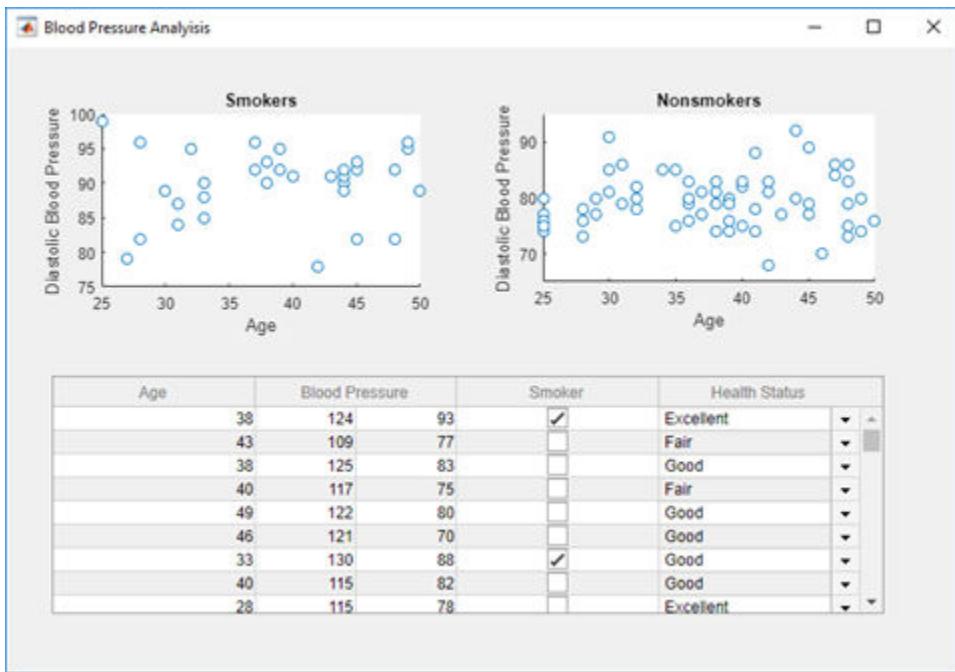
```
f = uifigure;
sz = categorical([1; 3; 4; 2],1:3,{'Large','Medium','Small'});
num = [NaN; 10; 12; 15];
tdata = table(sz,num,'VariableNames',{'Size','Number'});
uit = uitable(f,'Data',tdata,'ColumnEditable',true);
```

	Size	Number
1	Large	<input type="text" value="NaN"/>
2	Small	<input type="text" value="10"/>
3	<undefined>	<input type="text" value="12"/>
4	Medium	<input type="text" value="15"/>

Example: App that Displays a Table Array

This app shows how to display a `Table` UI component in an app that uses table array data. The table array contains numeric, logical, categorical, and multicolumn variables.

The `StartupFcn` callback loads a spreadsheet into a table array. Then a subset of the data displays and is plotted in the app. Both plots update when the user edits a value in the app.



See Also

[Table \(App Designer\)](#) | [uitable](#)

Related Examples

- “Write Callbacks in App Designer” on page 17-18
- “Create Helper Functions in App Designer” on page 17-24

Add UI Components to App Designer Programmatically

Most UI components are available in the App Designer **Component Library** for you to drag and drop onto the canvas. Occasionally, you might need to add components programmatically in Code View. Here are a few common situations:

- Creating components that are not available in the **Component Library**. For example, an app that displays a dialog box must call the appropriate function to display the dialog box.
- Creating components dynamically according to run-time conditions.

When you add UI components programmatically, you must call the appropriate function to create the component, assign a callback to the component, and then write the callback as a helper function.

Create the Component and Assign the Callback

Call the function that creates the component from within an existing callback (for a list of component functions, see “Designing Apps in App Designer”). The `StartupFcn` callback is a good place to create components because that callback runs when the app starts up. In other cases, you might create components within a different callback function. For example, if you want to display a dialog box when the user presses a button, call the dialog box function from within the button's callback function.

When you call a function to create a component, specify the figure or one of its child containers as the parent object. For example, this command creates a button and specifies the figure as the parent object. In this case, the figure has the default name that App Designer assigns (`app.UIFigure`).

```
b = uibutton(app.UIFigure);
```

Next, specify the component's callback property as a function handle of the form `@app.callbackname`. For example, this command sets the `ButtonPushedFcn` property of button `b` to a callback function named `mybuttonpress`.

```
b.ButtonPushedFcn = @app.mybuttonpress;
```

Write the Callback

Write the callback function for the component as a private helper function. The function must have `app`, `src`, and `event` as the first three arguments. Here is an example of a callback written as a private helper function.

```
methods (Access = private)

    function mybuttonpress(app,src,event)
        disp('Have a nice day!');
    end

end
```

To write a callback that accepts additional input arguments, specify the additional arguments after the first three. For example, this callback has accepts two additional inputs, `x` and `y`:

```
methods (Access = private)

    function addxy(app,src,event,x,y)
        disp(x + y);
    end

end
```

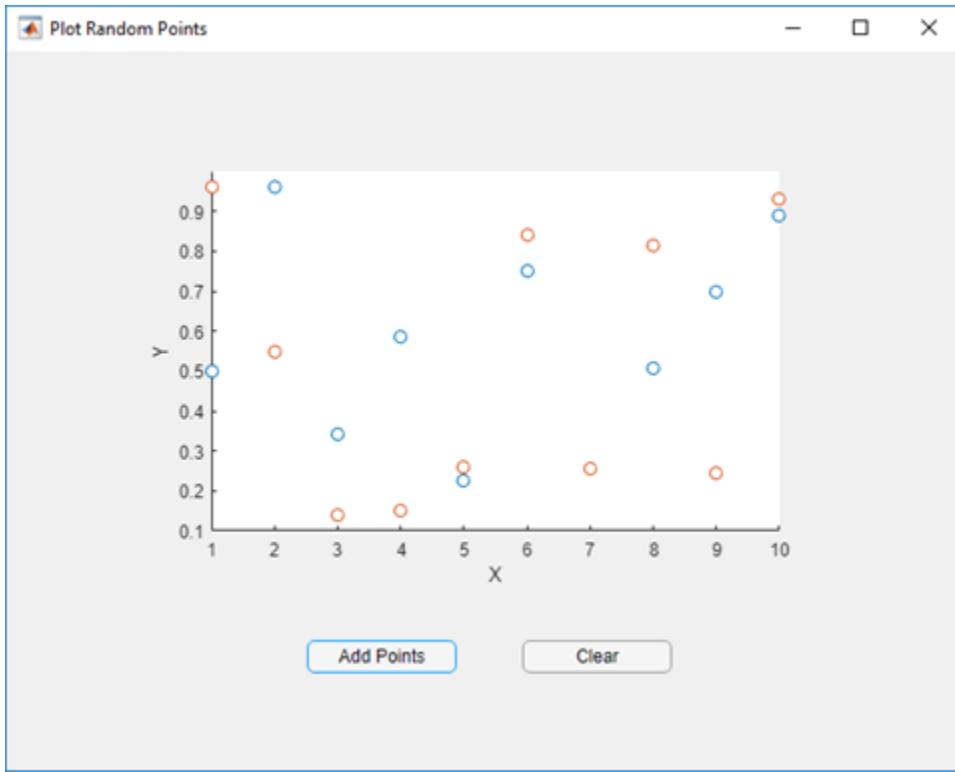
To assign this callback to a component, specify the component's callback property as cell array. The first element in the cell array must be the function handle. Subsequent elements must be the additional input values. For example:

```
b.ButtonPushedFcn = {@app.addxy,10,20};
```

Example: Confirmation Dialog Box with a Close Function

This app shows how to display a confirmation dialog box that executes a callback when the dialog box closes.

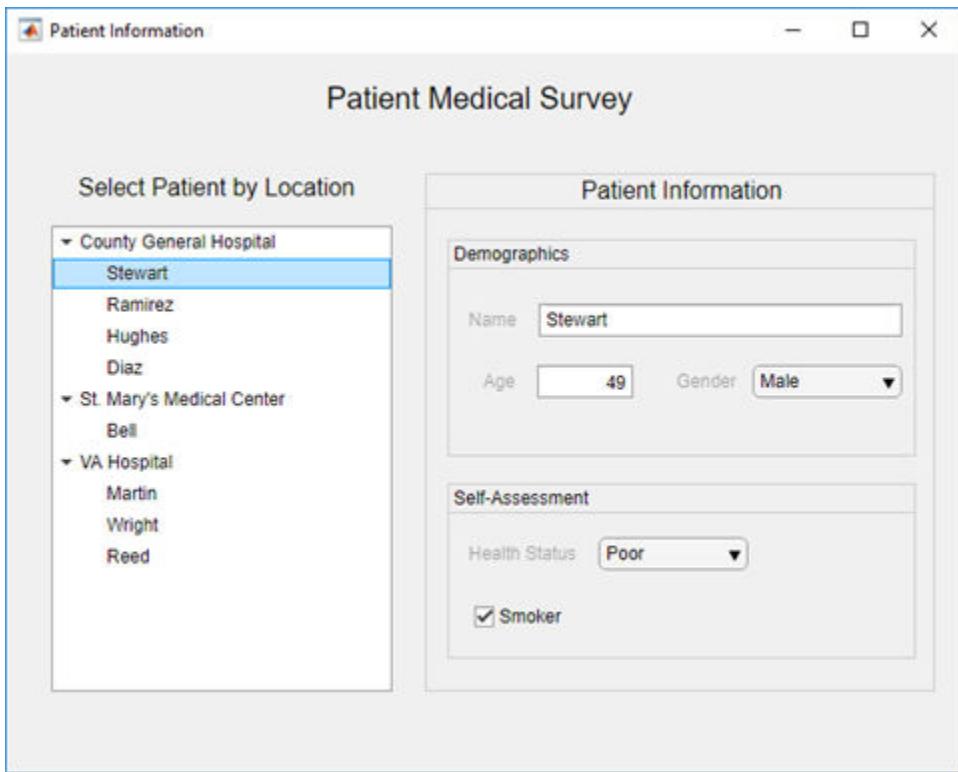
When the user clicks the window's close button (**X**), a dialog box displays to confirm that the user wants to close the app. When the user dismisses the dialog box, the `CloseFcn` callback executes.



Example: App that Populates Tree Nodes Based on a Data File

This app shows how to dynamically add tree nodes at run time. The three hospital nodes exist in the tree before the app runs. However at run time, the app adds several child nodes under each hospital name. The number of child nodes, and the labels on the child nodes are determined by the contents of the `patients.xls` spreadsheet.

When the user clicks a patient name in the tree, the **Patient Information** panel displays data such as age, gender, and health status. The app stores changes to the data in a table array.



See Also

More About

- “Write Callbacks in App Designer” on page 17-18
- “Create Helper Functions in App Designer” on page 17-24

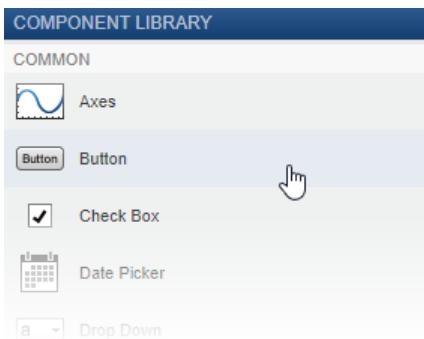
App Layout

- “Lay Out Apps in App Designer” on page 16-2
- “Managing Resizable Apps in App Designer” on page 16-9
- “Using Grid Layout Managers” on page 16-12

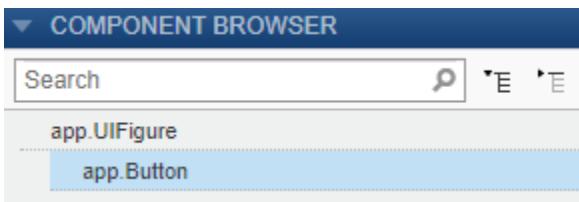
Lay Out Apps in App Designer

Design View in App Designer provides a rich set of layout tools for designing modern, professional-looking applications. It also provides an extensive library of UI components, so you can create a variety of interactive features. Any changes you make in **Design View** are automatically reflected in **Code View**. Thus, you can configure many aspects of your app without writing any code.

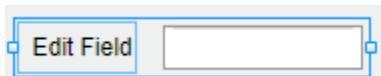
To add a component to your app, drag it from the **Component Library** onto the canvas.



The name of the component appears in the **Component Browser** after you add it to the canvas. You can select components in either the canvas or the **Component Browser**. The selection occurs in both places simultaneously.



Some components, such as edit fields and sliders, are grouped with a label when you drag them onto the canvas. These labels do not appear in the **Component Browser** by default, but you can add them to the list by right-clicking anywhere in the **Component Browser** and selecting **Include component labels in Component Browser**. If you do not want the component to have a label, you can exclude it by pressing and holding the **Ctrl** key as you drag the component onto to the canvas.

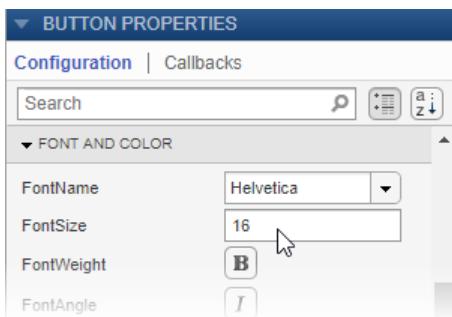


If a component has a label, and you change the label text, the name of the component in the **Component Browser** changes to match that text. You can customize the name of the component by double-clicking it and typing a new name.

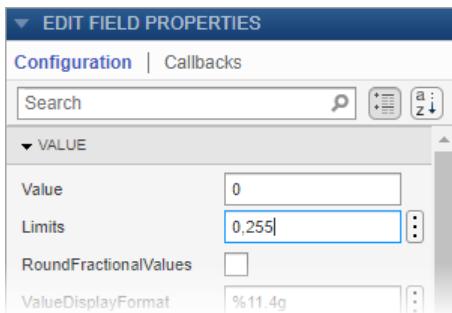


Customizing Components

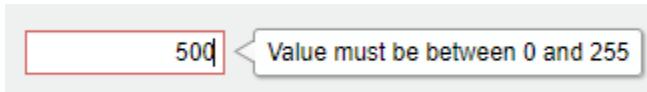
You can customize the appearance of a component by selecting it and then editing its properties in the **Component Properties** pane. For example, you can change the font size of a button in this pane.



Some properties control the behavior of the component. For example, you can change the range of values that a numeric edit field accepts by changing the **Limits** property.



When the app runs, the edit field accepts values only within that range.



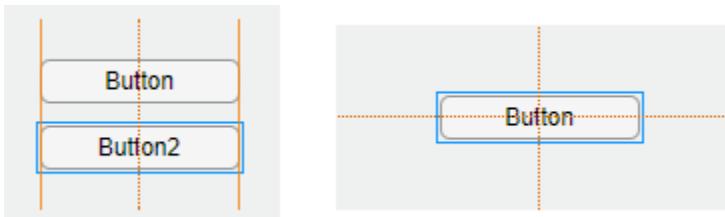
You can edit some properties directly in the canvas by double-clicking the component. For example, you can edit a button label by double-clicking it and typing the desired text. To add multiple lines of text, hold down the **Shift** key and press **Enter**.



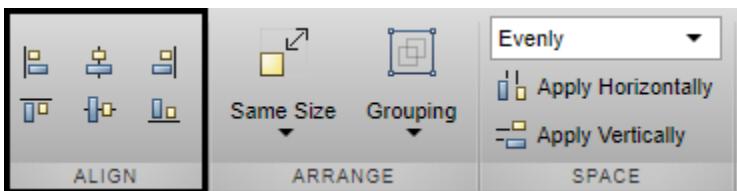
Aligning and Spacing Components

In **Design View**, you can arrange and resize components by dragging them on the canvas, or you can use the tools available in the **Canvas** tab of the toolbar.

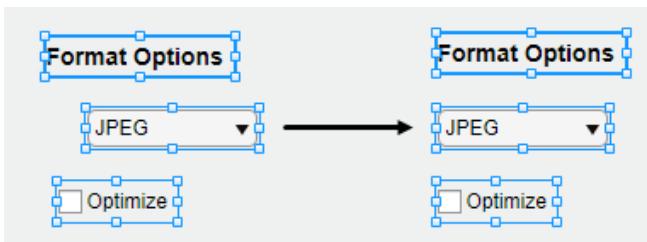
App Designer provides alignment hints to help you align components as you drag them in the canvas. Orange dotted lines passing through the centers of multiple components indicate that their centers are aligned. Orange solid lines at the edges indicate that the edges are aligned. Perpendicular lines indicate that a component is centered in its parent container.



As an alternative to dragging components on the canvas, you can align components using the tools in the **Align** section of the toolbar.



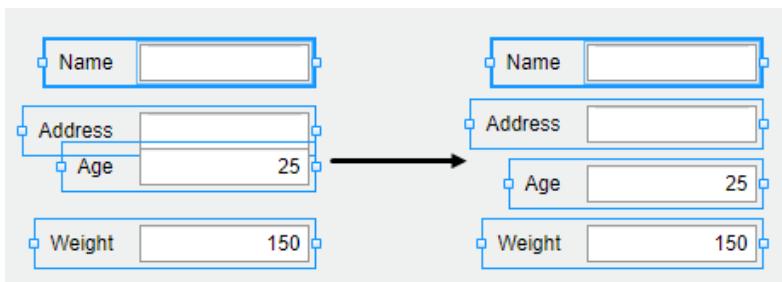
When you use an alignment tool, the selected components align to an anchor component. The anchor component is the last component selected, and it has a thicker selection border than the other components. To select a different anchor, hold down the **Ctrl** or **Shift** key and click the desired component twice (once to deselect the component, and a second time to select it again). For example, in the following image, the **Format Options** label is the anchor. Clicking the **Align left** button aligns the left edges of the dropdown and check box to the left edge of the label.



You can control the spacing among neighboring components using the tools in the **Space** section of the toolbar. Select a group of three or more components, and then select an option from the drop-down list in the **Space** section of the toolbar. The **Evenly** option distributes the space evenly within the space occupied by the components. The **20** option spaces the components 20 pixels apart. If you want to customize the number of pixels between the components, type a number into the drop-down list.



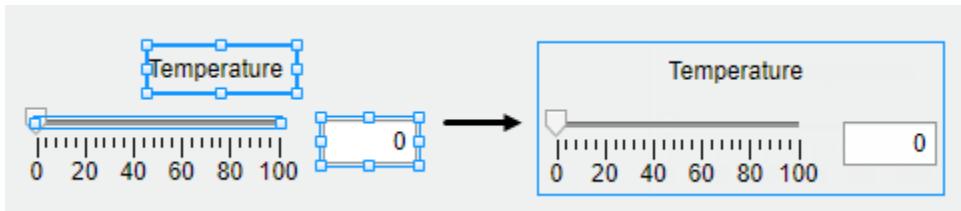
Next, click **Apply Horizontally** or **Apply Vertically** . For example, select **Evenly** and then click **Apply Vertically** to distribute the space among a vertical stack of components.



Grouping Components

You can group two or more components together to modify them as a single unit. For example, you might group a set of components after finalizing their relative positions, so that you can move them without changing that relationship.

To group a set of components, select them in the canvas, right-click, and then select **Grouping > Group** in the **Arrange** section of the toolstrip.

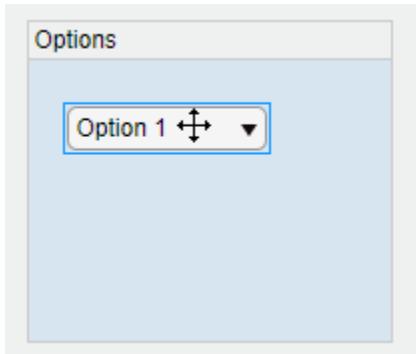


The **Grouping** tool also provides functionality for these common tasks:

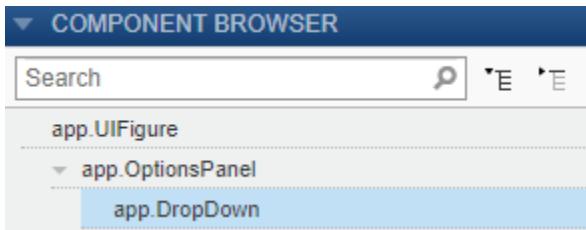
- Ungroup all components in a group — Select the group. Then select **Grouping > Ungroup**.
- Add a component to a group — Select the component and the group. Then select **Grouping > Add to Group**.
- Remove a component from a group — Select the component. Then select **Grouping > Remove from Group**.

Arranging Components in Containers

When you drag a component into a container such as a panel, the container turns blue to indicate that the component is a child of the container. This process of placing components into containers is called parenting.



The **Component Browser** shows the parent-child relationship by indenting the name of the child component under the parent container.



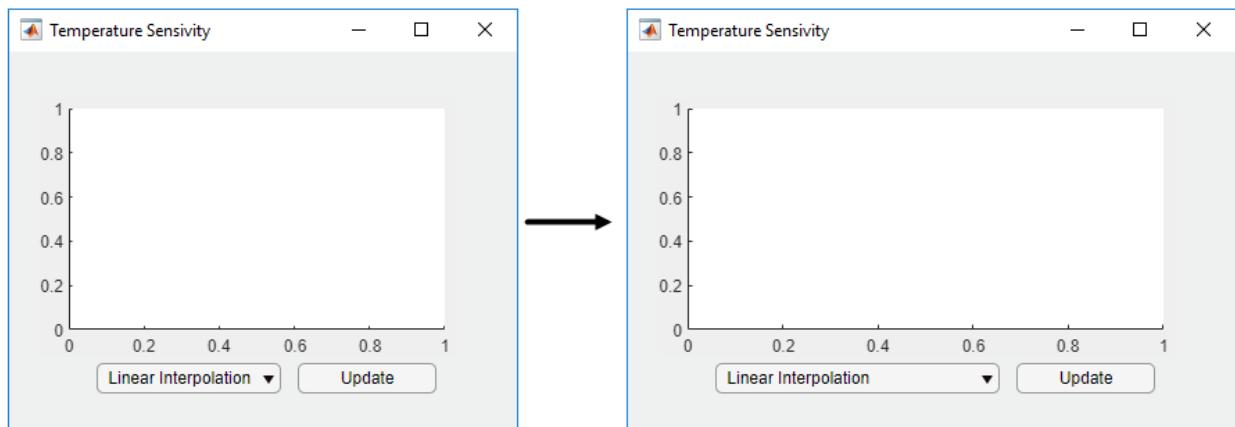
See Also

More About

- “App Designer Components” on page 15-2
- “App Designer Keyboard Shortcuts” on page 19-2
- “Managing Resizable Apps in App Designer” on page 16-9

Managing Resizable Apps in App Designer

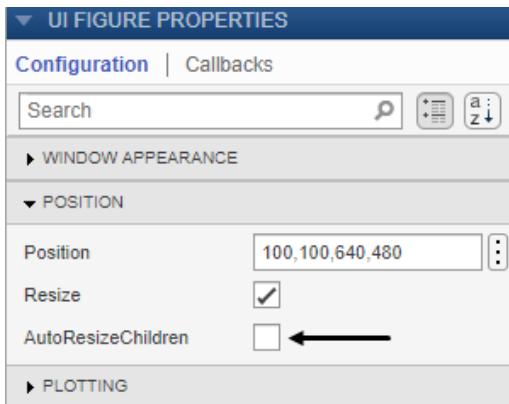
Apps you create in App Designer are resizable by default. The components reposition and resize automatically as the user changes the size of the window at run time. This automatic resize behavior is controlled by the **AutoResizeChildren** property. By default, this property is enabled for the figure and all child containers such as panels and tabs.



When the **AutoResizeChildren** property is enabled for a container, MATLAB manages the size and position of only the immediate children in the container. Components in nested containers are managed by the **AutoResizeChildren** property of their immediate parent.

You can disable the automatic resize behavior by disabling the property. To disable the property in App Designer, select the container and clear the **AutoResizeChildren** check box in the **Component Properties** pane.

Whenever you select or clear this check box, App Designer sets the property to the same value on all its child containers. To set the **AutoResizeChildren** property of a child container to a different value, set the value for the child container after setting the value for the parent.



You can also set the property programmatically by setting the value to 'on' or 'off'. When you set the property programmatically, the value does not change for the child containers.

```
app.UIFigure.AutoScaleChildren = 'off';
```

If the automatic resize behavior is not the behavior that you want, disable the `AutoSizeChildren` property and write a `SizeChangedFcn` callback for the container. In this callback, you write code to adjust the `Position` property of the child components. The callback executes when the size of the container changes.

For example, a `SizeChangedFcn` might contain code that keeps the width of an edit field at one quarter of the width of the figure.

```
figwidth = app.UIFigure.Position(3);  
app.EditField.Position(3) = .25 * figwidth;
```

Note Starting in R2017a, you must disable the `AutoSizeChildren` property to allow the `SizeChangedFcn` callback to execute. For more information, see "App Designer: Disable automatic resize behavior when writing `SizeChangedFcn` callbacks".

To completely disable resizing, set the `Resize` property of the figure to 'off'.

See Also

UI Figure

More About

- “Lay Out Apps in App Designer” on page 16-2
- “Write Callbacks in App Designer” on page 17-18

Using Grid Layout Managers

Note Grid layout managers are only for apps created using the `uifigure` function.

When you design an app using a grid layout manager, you place components in the rows and columns of an invisible grid. Using a grid layout is straightforward, but it is important to understand the relationship between the grid, its parent container, and the components that the grid manages.

When you create a grid, it always spans the entire app window or container that you place it in. You must configure its rows and columns so that they divide the space of the parent container appropriately.

To configure the rows and columns, specify the `RowHeight` and `ColumnWidth` properties of the grid. Specify the value of each property as a cell array. The length of the cell array controls the number of rows or columns. For example, to create a grid that has three rows, specify the `RowHeight` property as a 1-by-3 cell array. The values in each cell array control the size of each row or column.

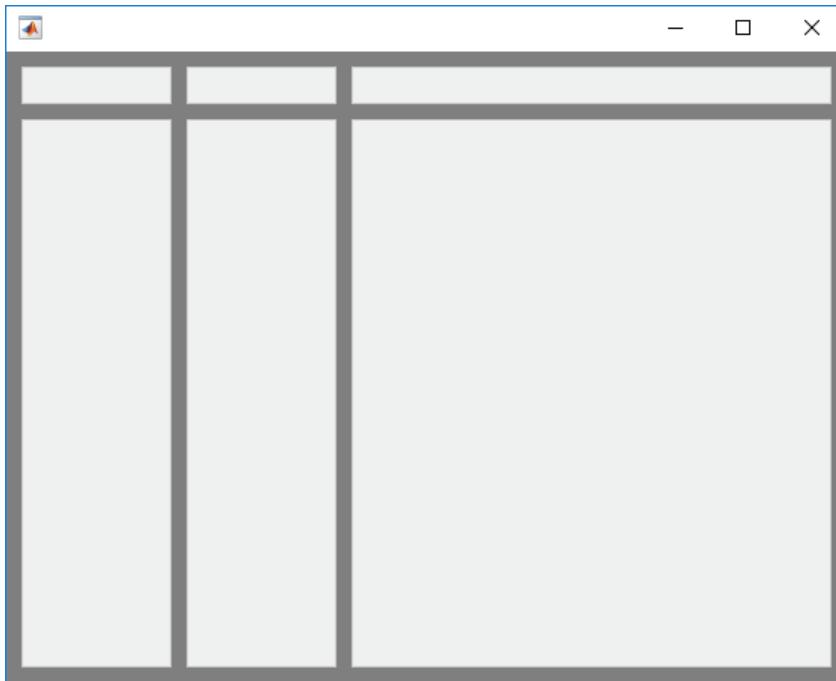
There are two types of sizes:

- Fixed size in pixels — Specify a number. The size of the row or column is fixed at the number of pixels you specify. When the parent container resizes, the size does not change.
- Variable size — Specify a number paired with an '`x`' character (for example, '`1x`'). When the parent container resizes, the row or column grows or shrinks. Variable-size rows and columns fill the remaining space that the fixed rows or columns do not use. The number you pair with the '`x`' character is a weight for dividing up the remaining space among all the variable-size rows or columns.

For example, this code creates a 2-by-3 grid. The first row is fixed at 25 pixels high, while the second row has a variable height. The first two columns are 100 pixels wide, and the last column has a variable width.

```
f = uifigure;
g = uigridlayout(f);
g.RowHeight = {25, '1x'};
g.ColumnWidth = {100, 100, '1x'};
```

The grid is invisible, but this picture includes lines to illustrate how the space is distributed.



To place a component in a specific row and column of the grid, you must:

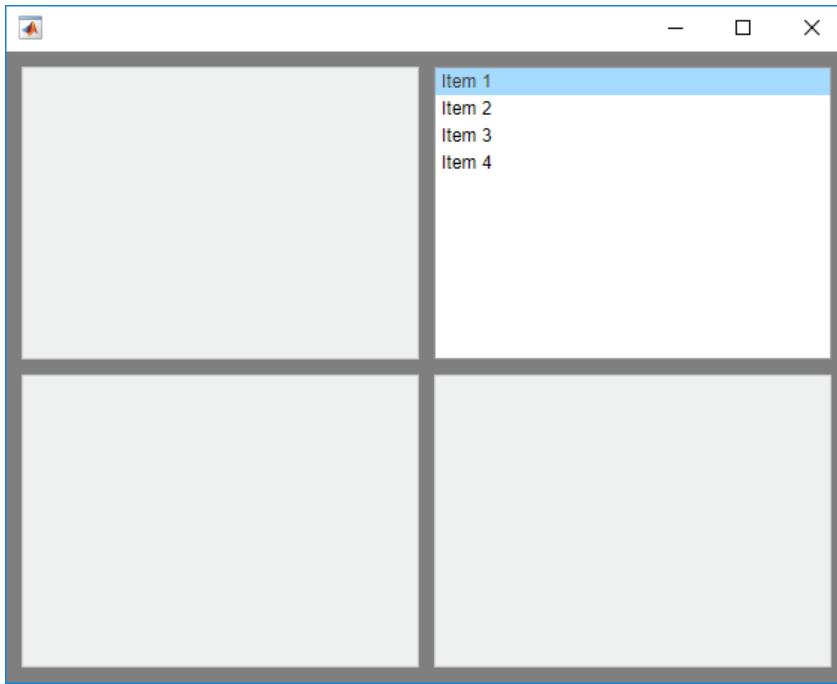
- Specify the grid as the parent of the component.
- Specify the target row and column by setting the `Layout` property of the component.

For example, this code creates a grid that has the default size (two '`1x`' rows and two '`1x`' columns). Then it places a list box in the first row and second column of that grid.

```
g = uigridlayout;
list = uilistbox(g);
list.Layout.Row = 1;
list.Layout.Column = 2;
```

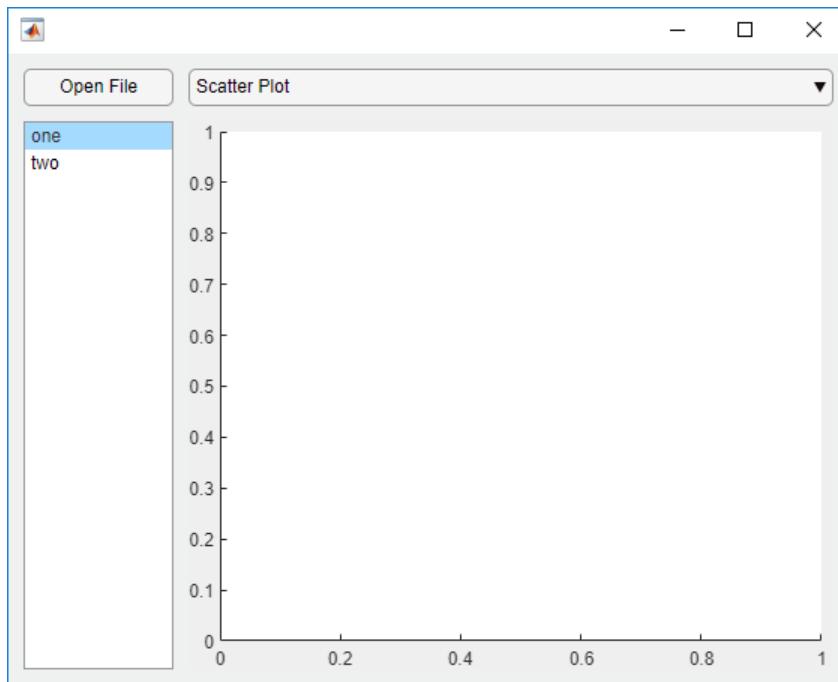
Again, the grid lines in this picture do not appear in the figure.

16 App Layout



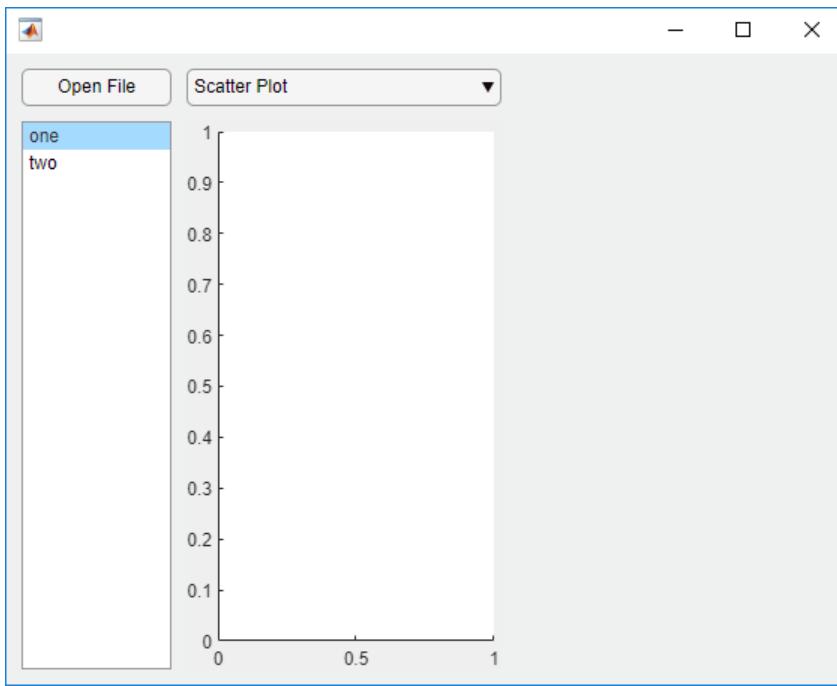
If you add components and do not specify the `Layout` property, the grid places the components in default locations. The components fill the grid from left to right and top to bottom initially. For example, this code creates a 2-by-2 grid containing four components in the default order.

```
f = uifigure;
g = uigridlayout(f);
g.RowHeight = {25, '1x'};
g.ColumnWidth = {100, '1x'};
b = uibutton(g, 'Text', 'Open File');
dd = uidropdown(g, 'Items', {'Scatter Plot', 'Line Plot'});
list = uilistbox(g, 'Items', {'one', 'two'});
ax = uiaxes(g);
```



If you reconfigure the grid after adding components to it, the grid does not redistribute the components. For example, if you add a third column in the preceding example, the grid does not shift the list box back to the third column of the first row.

```
g.ColumnWidth = {100, '1x', '1x'};
```



Some changes you make in the layout can change the size of the grid. For example, adding a component to a fully populated grid adds a row to accommodate the new component.

To view the list of component objects in the grid, query the `Children` property of the grid. Changing the order in the list does not change the layout in the grid.

Example: Hide Rows Based on Run-Time Conditions

This example shows how to hide components within a row of a grid based on the user's selection in a drop-down menu. The code performs these high-level tasks:

- Creates `grid1`, a 1-by-2 grid in the figure that manages a panel and an axes component.
- Creates `grid2`, a 3-by-2 grid inside the panel. This grid manages the layout of a drop-down menu, two spinners, and their labels.

- Creates a callback function called `findMethodSelected` for the drop-down menu. When the value of the drop-down menu changes to 'Quartiles', the callback hides the components in second row of `grid2` by setting `grid2.RowHeight{2}` to 0.

Create a program file called `showhide.m`. Then paste this code into the file and run it.

```
function showhide

f = uifigure('Name','Statistical Analysis');

% Create grid1 in the figure
grid1 = uigridlayout(f);
grid1.RowHeight = {'1x'};
grid1.ColumnWidth= {220,'1x'};

% Add a panel and axes
p = uipanel(grid1);
ax = uiaxes(grid1);

% Create grid2 in the panel
grid2 = uigridlayout(p);
grid2.RowHeight = {22, 22, 22};
grid2.ColumnWidth = {80,'1x'};

% Add method label and drop-down
findMethodLabel = uilabel(grid2,'Text','Find Method:');
findMethod = uidropdown(grid2,'Items',{'Moving median','Quartiles'});
findMethod.ValueChangedFcn = @findMethodSelected;

% Add window size label and spinner
winSizeLabel = uilabel(grid2,'Text','Window Size:');
winSize = uislider(grid2,'Value',0);

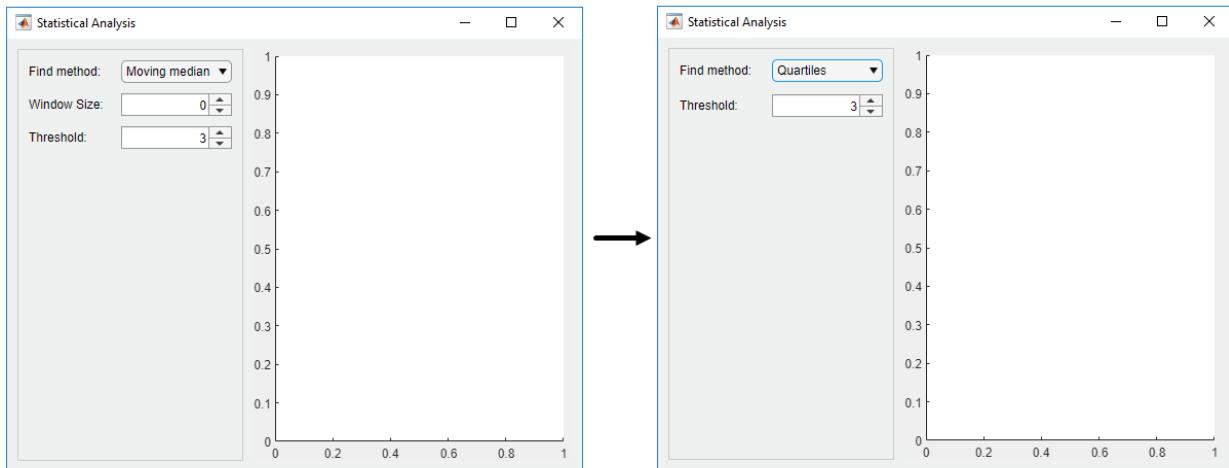
% Add threshold label and spinner
thresLabel = uilabel(grid2,'Text','Threshold:');
thres = uislider(grid2,'Value',3);

function findMethodSelected(src,~)
method = src.Value;

switch method
case 'Quartiles'
    % Collapse the second row (hides winSize spinner)
    grid2.RowHeight{2} = 0;
```

```
case 'Moving median'  
    % Expand the second row  
    grid2.RowHeight{2} = 22;  
end  
end  
end
```

When you set the **Find Method** to **Quartiles** in the app, the **Window Size** label and the spinner next to it become hidden.



See Also

[GridLayout Properties | uigridlayout](#)

App Programming

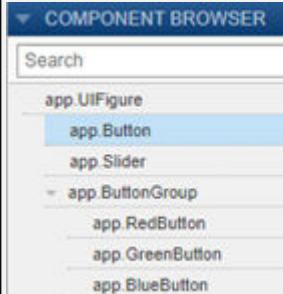
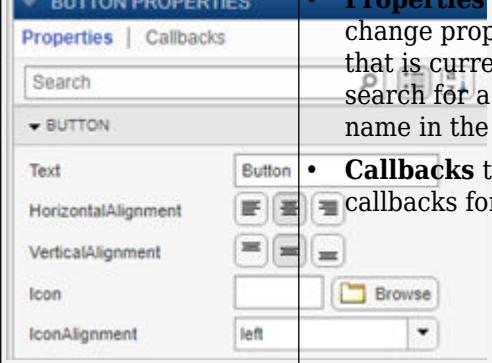
- “Managing Code in App Designer Code View” on page 17-2
- “Startup Tasks and Input Arguments in App Designer” on page 17-8
- “Creating Multiwindow Apps in App Designer” on page 17-12
- “Write Callbacks in App Designer” on page 17-18
- “Create Helper Functions in App Designer” on page 17-24
- “Share Data Within App Designer Apps” on page 17-28
- “Compatibility Between Different Releases of App Designer” on page 17-32
- “Use One Callback for Multiple App Designer Components” on page 17-35

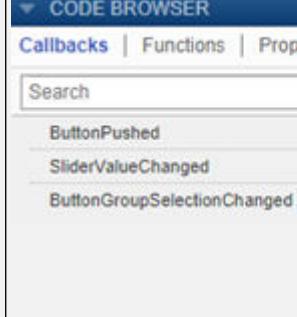
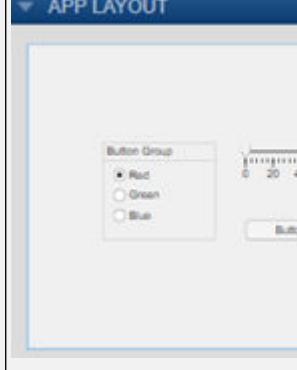
Managing Code in App Designer Code View

Code View provides most of the same programming features that the MATLAB Editor provides. It also provides a rich set of features that help you to navigate your code and avoid many tedious tasks. For example, you can search for a callback by typing part of its name in a search bar. Clicking a search result scrolls the editor to the definition of that callback. And if you change the name of a callback, App Designer automatically updates all references to it in your code.

Managing Components, Functions, and Properties

Code View has four panes to help you manage different aspects of your code. This table describes each of them.

Pane Name	Pane Appearance	Pane Features
Component Browser		<ul style="list-style-type: none"> Context menu — Right-click a component in the list to display a context menu that has options for deleting or renaming the component, adding a callback, or displaying help. Select the Include Component Labels in Component Browser option to display grouped component labels. Search bar — Quickly locate a component by typing part of its name in the search bar.
Component Properties		<ul style="list-style-type: none"> Properties tab — Use this tab to view or change property values for the component that is currently selected. You can also search for a property by typing part of the name in the search bar at the top of this tab. Callbacks tab — Use this tab to manage the callbacks for the component that is selected.

Pane Name	Pane Appearance	Pane Features
Code Browser	CODE BROWSER Callbacks Functions Properties 	<ul style="list-style-type: none"> • Callbacks, Functions, and Properties tabs — Use these tabs to add, delete, or rename any of the callbacks, helper functions, or custom properties in your app. Clicking an item in the Callbacks or Functions tab scrolls the editor to the corresponding section in your code. • Search bar — Quickly locate a callback, helper function, or property by typing part of its name in the search bar.
App Layout	APP LAYOUT 	<ul style="list-style-type: none"> • App thumbnail — Use the thumbnail image to locate components in large, complex apps that have many components. Selecting a component in the thumbnail selects the component in the Component Browser.

Identifying Editable Sections of Code

In the editor, some sections of code are editable and some are not. Gray sections of code are not editable. Those sections are generated and managed by App Designer. However, white sections are editable, and they correspond to:

- The body of functions you define (e.g., callbacks and helper functions)
- Custom property definitions

```
12
13
14     properties (Access = private)
15         X = 5 % Average value
16     end
17
18
19     methods (Access = private)
20
21         % Button pushed function: Button
22         function ButtonPushed(app, event)
23             disp('Hello World');
24
25         end
```

Programming Your App

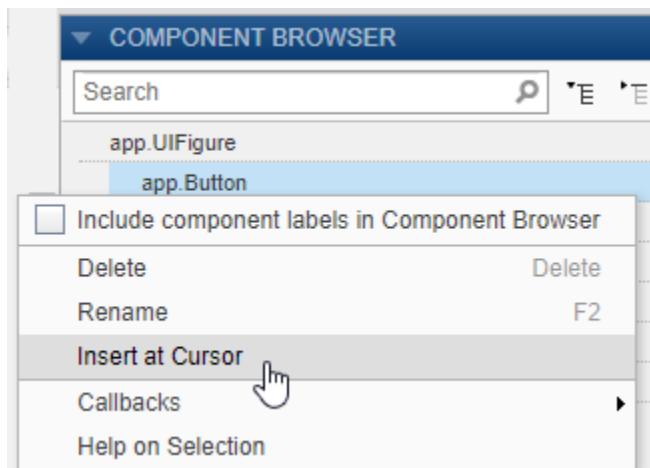
App Designer defines your app as a MATLAB class. You do not need to understand classes or object-oriented programming to create an app because App Designer manages those aspects of the code. However, programming in App Designer requires a different workflow than working strictly with functions. You can review a summary of this workflow at any time by clicking the **Show Tips**  button in the **Resources** tab of the toolbar.

Managing UI Components

When you add a UI component to your app, App Designer assigns a default name to the component. Use that name (including the `app` prefix) to refer to the component in your code. You can change the name of a component by double-clicking the name in the **Component Browser** and typing a new name. App Designer automatically updates all references to that component when you change its name.



To use the name of a component in your code, you can save some time by copying the name from the **Component Browser**. Right-click the component name and select **Insert at Cursor**. Alternatively, you can drag the component name from the list into your code.



To delete a component, select its name in the **Component Browser** and press the **Delete** key.

Managing Callbacks

To make a component respond to user interactions, add a callback. Right-click the component in the **Component Browser** and select **Callbacks > Add (callback property) callback**.

If you delete a component from your app, App Designer deletes the associated callback only if the callback has not been edited and is not shared with other components.

To delete a callback manually, select the callback name in the **Callbacks** tab of the **Code Browser** and press the **Delete** key.

For more information about callbacks, see “Write Callbacks in App Designer” on page 17-18.

Sharing Data Within Your App

To store data, and share it among different callbacks, create a custom property. For example, you might want your app to read a data file and allow different callbacks in your app to access that data.

To create a property, expand the **Property**  drop-down in the **Editor** tab, and select **Private Property** or **Public Property**. App Designer creates a template property definition and places your cursor next to that definition. Change the name of the property as desired.

```
properties (Access = public)
    X % Average cost
end
```

To reference the property in your code, use dot notation of the form `app.Propertyname`. For example, `app.X` references the property named X.

For more information about creating and using custom properties, see “Share Data Within App Designer Apps” on page 17-28.

Single-Sourcing Code that Runs in Multiple Places

If you want to execute a block of code in multiple parts of your app, create a helper function. For example, you might want to update a plot after the user changes a number in an edit field or selects an item in a drop-down list. Creating a helper function allows you to single-source the common commands and avoid having to maintain redundant sets of code.

To add a helper function, expand the **Function**  drop-down in the **Editor** tab, and select **Private Function** or **Public Function**. App Designer creates a template function and places your cursor in the body of that function.

To delete a helper function, select the function name in the **Functions** tab of the **Code Browser** and press the **Delete** key.

For more information about writing helper functions, see “Create Helper Functions in App Designer” on page 17-24.

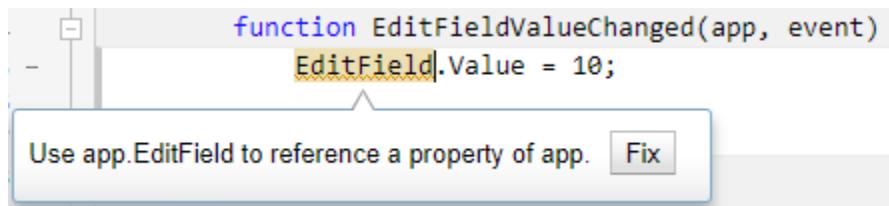
Creating Input Arguments

To add input arguments to your app, click **App Input Arguments**  in the **Editor** tab. Input arguments are commonly used for creating apps that have multiple windows. For

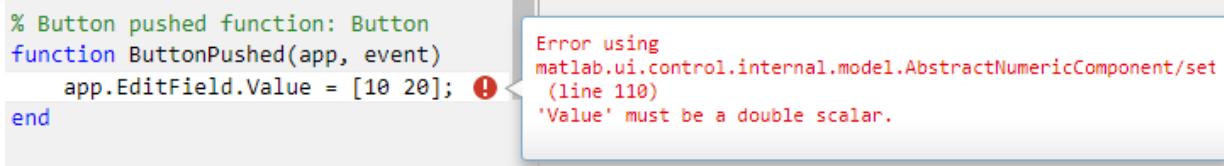
more information, see “Startup Tasks and Input Arguments in App Designer” on page 17-8.

Fixing Coding Problems and Run-Time Errors

Like the MATLAB Editor, the **Code View** editor provides Code Analyzer messages to help you discover errors in your code.



If you run your app directly from App Designer (by clicking **Run**), App Designer highlights the source of errors in your code, should any errors occur at run time. To hide the error message, click the error indicator (the red circle). To make the error indicator disappear, fix your code and save your changes.



See Also

Related Examples

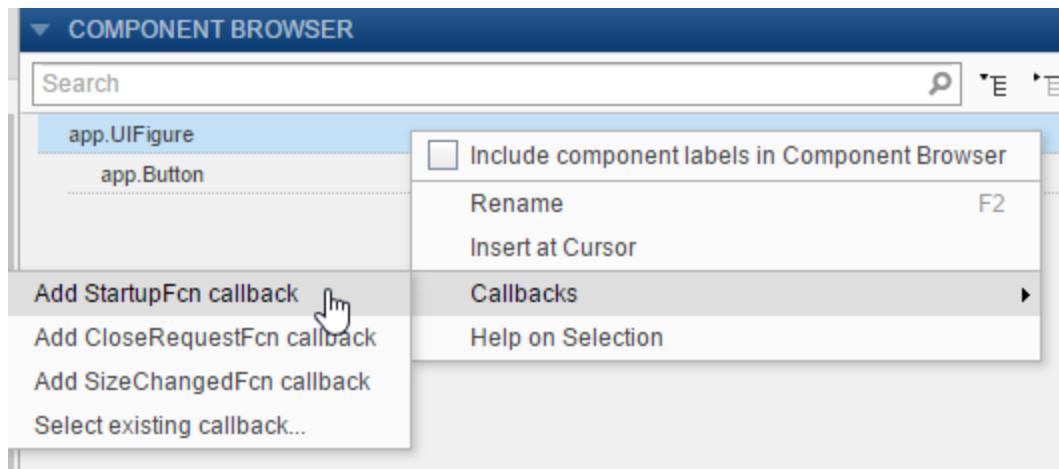
- “Write Callbacks in App Designer” on page 17-18
- “Share Data Within App Designer Apps” on page 17-28
- “Create Helper Functions in App Designer” on page 17-24
- “Startup Tasks and Input Arguments in App Designer” on page 17-8

Startup Tasks and Input Arguments in App Designer

App Designer allows you to create a special function that executes when the app starts up, but before the user interacts with the UI. This function is called the `StartupFcn` callback, and it is useful for setting default values, initializing variables, or executing commands that affect initial state of the app. For example, you might use the `StartupFcn` callback to display a default plot or a show a list of default values in a table.

Create a `StartupFcn` Callback

To create a `StartupFcn` callback, right-click the `UIFigure` component in the **Component Browser**, and select **Callbacks > Add StartupFcn callback**.



App designer creates the function and places the cursor in the body of the function. Add commands to this function as you would do for any callback function. Then save and run your app.

```
methods (Access = private)

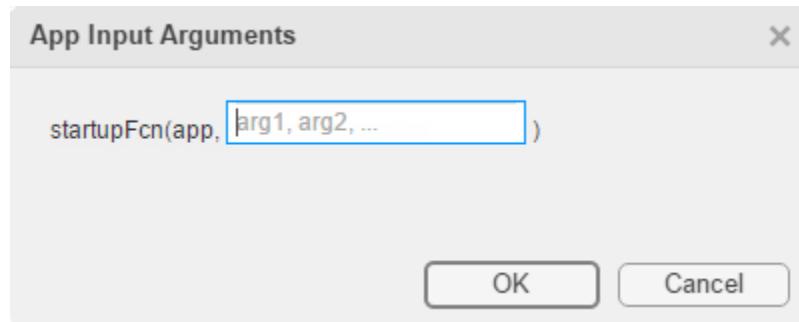
    % Code that executes after component creation
    function startupFcn(app)
        %
    end
end
```

See “Data Analysis App in App Designer” on page 18-4 for an example of an app that has a `StartupFcn` callback.

Define Input App Arguments

The `StartupFcn` callback is also the function where you can define input arguments for your app. Input arguments are useful for letting the user (or another app) specify initial values when the app starts up.

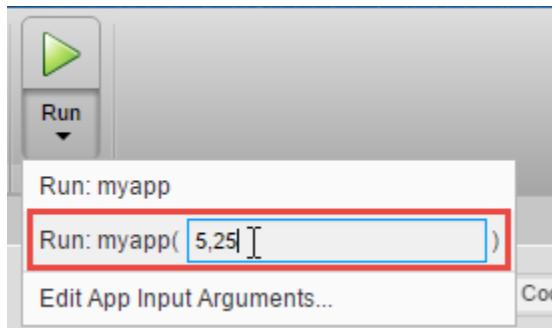
To add input arguments to an app, open the app in App Designer and click **Code View**. Then click **App Input Arguments**  in the **Editor** tab.



The **App Input Arguments** dialog box allows you to add or remove input arguments in the function signature of the `StartupFcn` callback. The `app` argument is always first, so you cannot change that part of the signature. Enter a comma-separated list of variable names for your input arguments. You can also enter `varargin` to make any of the arguments optional. Then click **OK**.

After you click **OK**, App Designer creates a **StartupFcn** callback that has the function signature you defined in the dialog box. If your app already has a **StartupFcn** callback, then the function signature is updated to include the new input arguments.

After you have created the input arguments and coded the **StartupFcn**, you can test the app. Expand the drop-down list from the **Run** button in the toolbar. In the second menu item, specify comma-separated values for each input argument. The app runs after you enter the values and press **Enter**.



Note MATLAB might return an error if you click the **Run** button without entering input arguments in the drop-down list. The error occurs because the app has required input arguments that you did not specify.

After successfully running the app with a set of input arguments, the **Run** button icon contains a blue circle.



The blue circle indicates that your last set of input values are available for re-running your app without having to type them again. Up to seven sets of input values are available to choose from. Click the top half of the **Run** button to re-run the app with the last set of values. Or, click the bottom half of the **Run** button and select one of the previous sets of values.

The **Run** button also allows you to change the list of arguments in the function signature. Select **Edit App Input Arguments...** from the drop-down list in the bottom half of the **Run** button.



Alternatively, you can open the same **App Input Arguments** dialog box by clicking **App Input Arguments** in the toolbar, or by right-clicking the **StartupFcn** callback in the **Code Browser**.

See “Creating Multiwindow Apps in App Designer” on page 17-12 for an example of an app that uses input arguments.

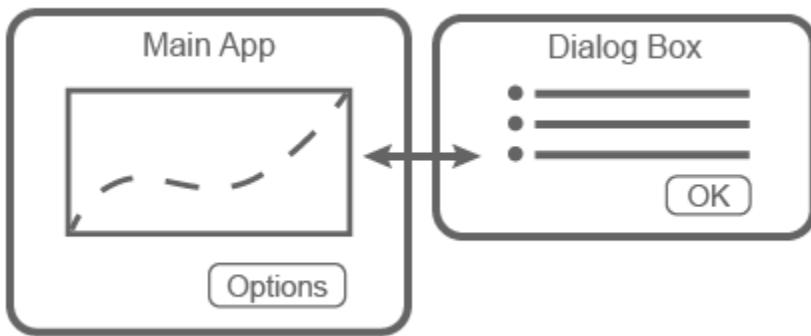
See Also

Related Examples

- “Write Callbacks in App Designer” on page 17-18
- “Creating Multiwindow Apps in App Designer” on page 17-12

Creating Multiwindow Apps in App Designer

A multiwindow app consists of two or more apps that share data. The way that you share data between the apps depends on the design. One common design involves two apps: a main app and a dialog box. Typically, the main app has a button that opens the dialog box. When the user closes the dialog box, the dialog box sends the user's selections to the main window, which performs calculations and updates the UI.



These apps share information in different ways at different times:

- When the dialog box opens, the main app passes information to the dialog box by calling the dialog box app with input arguments.
- When the user clicks the **OK** button in the dialog box, the dialog box returns information to the main app by calling a public function in the main app with input arguments.

Overview of the Process

To create the app described in the preceding section, you must create two separate apps (a main app and a dialog box app). Then perform these high-level tasks. Each task involves multiple steps.

- “Send Information to the Dialog Box” on page 17-13 — Write a `StartupFcn` callback in the dialog box app that accepts input arguments. One of the input arguments must be the main app object. Then, in the main app, call the dialog box app with the input arguments.

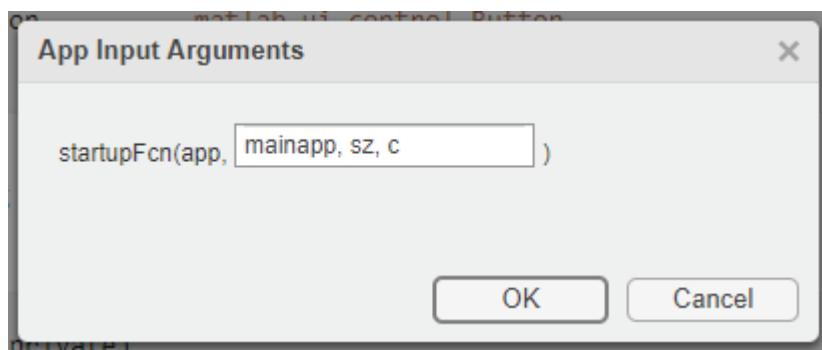
- “Return Information to the Main App” on page 17-14 — Write a public function in the main app that updates the UI based on the user’s selections in the dialog box. Because it is a public function, the dialog box can call it and pass values to it.
- “Manage Windows When They Close” on page 17-15 — Write `CloseRequest` callbacks in both apps that perform maintenance tasks when the windows close.

To see an implementation of all the steps in this process, see Plotting App That Opens a Dialog Box on page 17-16.

Send Information to the Dialog Box

Perform these steps to pass values from the main app to the dialog box app.

- In the dialog box app, define input arguments for the `StartupFcn` callback, and then add code to the callback. Open the dialog box app into **Code View**. In the **Editor** tab, click **App Input Arguments** . In the **App Input Arguments** dialog box, enter a comma-separated list of variable names for your input arguments. Designate one of the inputs as a variable that stores the main app object. Then click **OK**.



Add code to the `StartupFcn` callback to store the value of `mainapp`.

```
function StartupFcn(app,mainapp,sz,c)
  % Store main app object
  app.CallingApp = mainapp;

  % Process sz and c inputs
  ...
end
```

For a fully coded example of a `StartupFcn` callback, see Plotting App That Opens a Dialog Box on page 17-16.

- 2 Call the dialog box app from within a callback in the main app. Open the main app into **Code View** and add a callback function for the **Options** button. This callback disables the **Options** button to prevent users from opening multiple dialog boxes. Next, it gets the values to pass to the dialog box, and then it calls the dialog box app with input arguments and an output argument. The output argument is the dialog box app object.

```
function OptionsButtonPushed(app,event)
    % Disable Plot Options button while dialog is open
    app.OptionsButton.Enable = 'off';

    % Get szvalue and cvalue
    % ....

    % Call dialog box with input values
    app.DialogApp = DialogAppExample(app,szvalue,cvalue);
end
```

- 3 Define a property in the main app to store the dialog box app. Keeping the main app open, create a private property called `DialogApp`. Select **Property > Private Property** in the **Editor** tab. Then, change the property name in the `properties` block to `DialogApp`.

```
properties (Access = private)
    DialogApp % Dialog box app
end
```

Return Information to the Main App

Perform these steps to return the user's selections to the main app.

- 1 Create a public function in the main app that updates the UI. Open the main app into **Code View** and select **Function > Public Function** in the **Editor** tab.

Change the default function name to the desired name, and add input arguments for each option you want to pass from the dialog box to the main app. The `app` argument must be first, so specify the additional arguments after that argument. Then add code to the function that processes the inputs and updates the main app.

```
function updateplot(app,sz,c)
    % Process sz and c
```

```
end ...
```

For a fully coded example of a public function, see Plotting App That Opens a Dialog Box on page 17-16.

- 2 Create a property in the dialog box app to store the main app. Open the dialog box app into **Code View**, and create a private property called **CallingApp**. Select **Property > Private Property** in the **Editor** tab. Then change the property name in the properties block to **CallingApp**.

```
properties (Access = private)
    CallingApp % Main app object
end
```

- 3 Call the public function from within a callback in the dialog box app. Keeping the dialog box app open, add a callback function for the **OK** button.

In this callback, pass the **CallingApp** property and the user's selections to the public function. Then call the **delete** function to close the dialog box.

```
function ButtonPushed(app,event)
    % Call main app's public function
    updateplot(app.CallingApp,app.EditField.Value,app.DropDown.Value);

    % Delete the dialog box
    delete(app)
end
```

Manage Windows When They Close

Both apps must perform certain tasks when the user closes them. Before the dialog box closes, it must re-enable the **Options** button in the main app. Before the main app closes, it must ensure that the dialog box app also closes.

- 1 Open the dialog box app into **Code View**, right-click the **app.UIFigure** object in the **Component Browser**, and select **Callbacks > Add CloseRequestFcn callback**. Then add code that re-enables the button in the main app and closes the dialog box app.

```
function DialogAppCloseRequest(app,event)
    % Enable the Plot Options button in main app
    app.CallingApp.OptionsButton.Enable = 'on';
    % Close the dialog box
    delete(app)
```

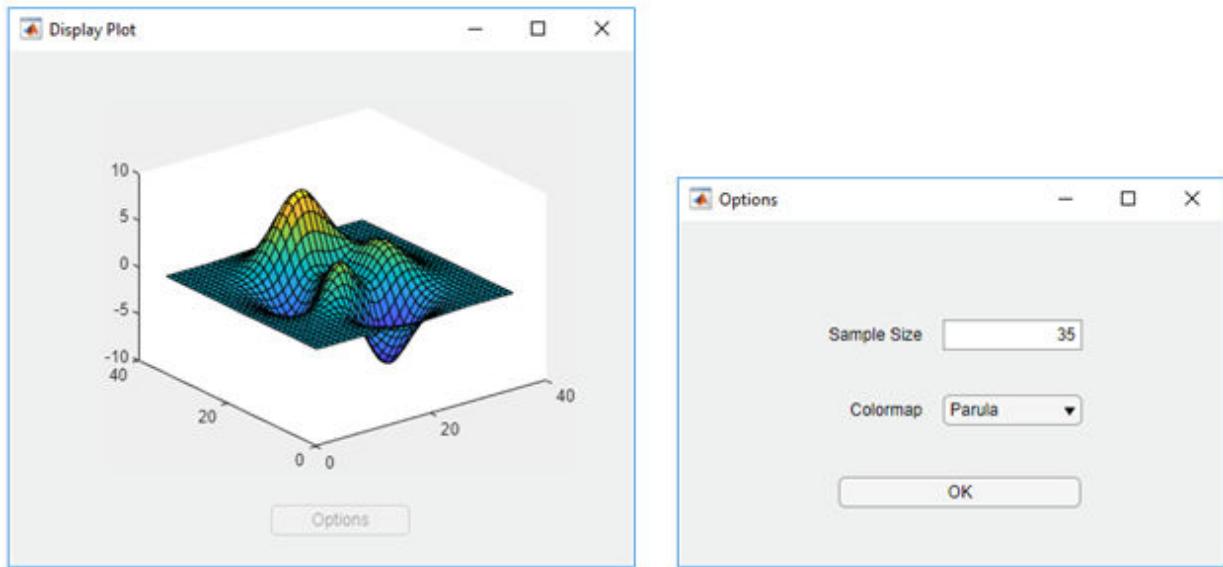
```
% Delete the dialog box  
delete(app)  
end
```

- 2 Open the main app into **Code View**, right-click the `app.UIFigure` object in the **Component Browser**, and select **Callbacks > Add CloseRequestFcn callback**. Then add code that deletes both apps.

```
function MainAppCloseRequest(app,event)  
    % Delete both apps  
    delete(app.DialogApp)  
    delete(app)  
end
```

Example: Plotting App That Opens a Dialog Box

This app consists of a main plotting app that has a button for selecting options in a dialog box. The **Options** button calls the dialog box app with input arguments. In the dialog box, the callback for the **OK** button sends the user's selections back to the main app by calling a public function in the main app.



See Also

More About

- “Write Callbacks in App Designer” on page 17-18
- “Startup Tasks and Input Arguments in App Designer” on page 17-8

Write Callbacks in App Designer

Note For information on callbacks in GUIDE, see “Write Callbacks in GUIDE” on page 7-2. If you are creating an app programmatically, see “Write Callbacks for Apps Created Programmatically” on page 10-5.

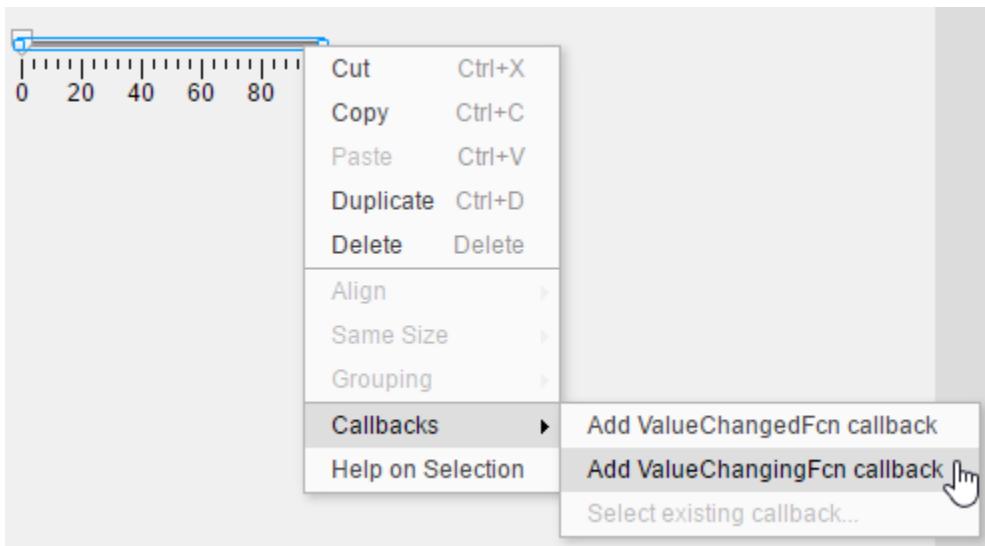
A callback is a function that executes when a user interacts with a UI component in your app. Most components can have at least one callback. However some components, such as labels and lamps, do not have callbacks because those components only display information.

To see the list of callbacks that a component supports, select the component and click the **Callbacks** tab in the component **Properties** pane.

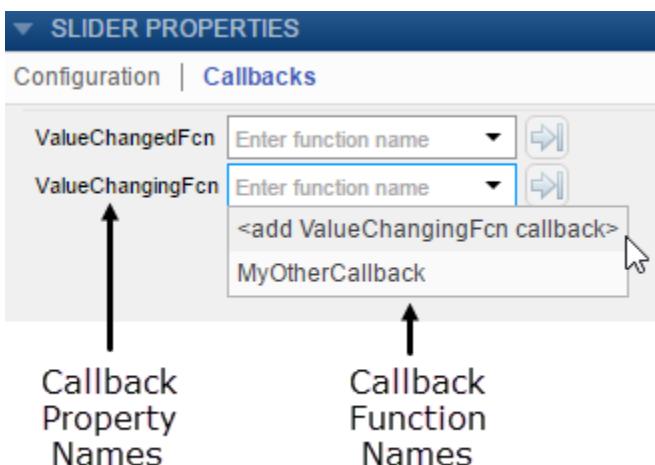
Create a Callback Function

There are several ways to create a callback for a UI component. You might use different approaches depending on what part of App Designer you are working in. Choose the most convenient approach from the following list.

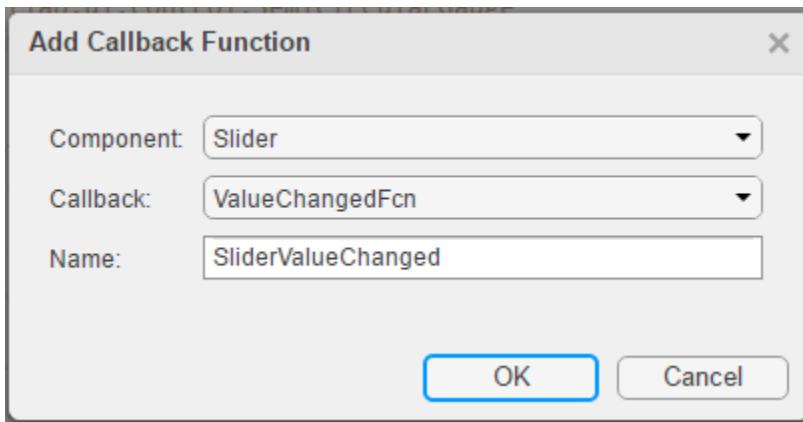
- Right-click a component in the canvas, **Component Browser**, or **App Layout** pane, and select **Callbacks > Add (callback property) callback**.



- Select the **Callbacks** tab in the component **Properties** pane. The left side of the **Callbacks** tab shows a list of supported callback properties. The text field next to each callback property allows you to specify a name for the callback function. The down-arrow next to the text field allows you to select a default name in angle brackets <>. If your app has existing callbacks, the drop-down includes those callbacks. Select an existing callback when you want multiple UI components to execute the same code.



- In code **Code View**, in the **Editor** tab, click **Callbacks** . Or in the **Code Browser** on the **Callbacks** tab, click the button.



Specify the following options in the **Add Callback Function** dialog box:

- **Component** — Specify the UI component that executes the callback.
- **Callback** — Specify the callback property. The callback property maps the callback function to a specific interaction. Some components have more than one callback property available. For example, sliders have two callback properties: `ValueChangedFcn` and `ValueChangingFcn`. The `ValueChangedFcn` property executes after the user moves the slider and releases the mouse. The `ValueChangingFcn` property for the same component executes repeatedly while the user moves the slider.
- **Name** — Specify a name for the callback function. App Designer provides a default name, but you can change it in the text field. If your app has existing callbacks, the **Name** field has a down-arrow next to it, indicating that you can select an existing callback from a list.

Using Callback Function Input Arguments

All callbacks in App Designer have the following input arguments in the function signature:

- `app` — The `app` object. Use this object to access UI components in the app as well as other variables stored as properties.

- **event** — An object that contains specific information about the user's interaction with the UI component.

The **app** argument provides the **app** object to your callback. You can access any component (and all component-specific properties) within any callback by using this syntax:

```
app.Component.Property
```

For example, this command sets the **Value** property of a gauge to 50. In this case, the name of the gauge is **PressureGauge**.

```
app.PressureGauge.Value = 50;
```

The **event** argument provides an object that has different properties, depending on the specific callback that is executing. The object properties contain information that is relevant to the type of interaction that the callback is responding to. For example, the **event** argument in a **ValueChangingFcn** callback of a slider contains a property called **Value**. That property stores the slider value as the user moves the thumb (before they release the mouse). Here is a slider callback function that uses the **event** argument to make a gauge track the value of the slider.

```
function SliderValueChanged(app, event)
    latestvalue = event.Value; % Current slider value
    app.PressureGauge.Value = latestvalue; % Update gauge
end
```

To learn more about the **event** argument for a specific component's callback function, see the property page for that component. Right-click the component, and select **Help on Selection** to open the property page. For a list of property pages for all UI components, see “Designing Apps in App Designer”.

Searching for Callbacks in Your Code

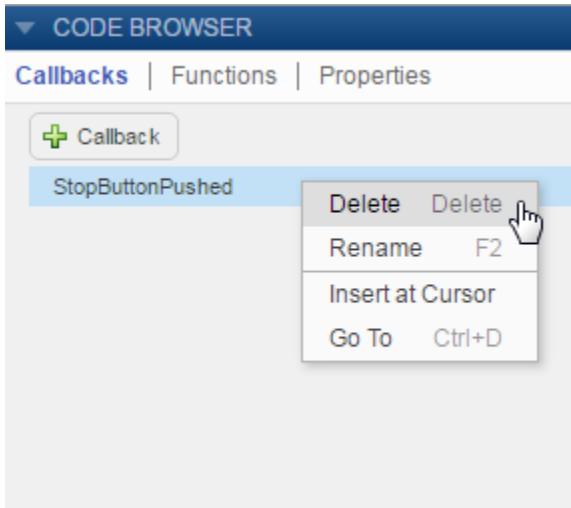
If your app has a lot of callbacks, you can quickly search and navigate to a specific callback by typing part of the name in the search bar at the top of the **Callbacks** tab in the **Code Browser**. After you begin typing, the **Callbacks** pane clears, except for the callbacks that match your search.



Click a search result to scroll the callback into view. Right-clicking a search result and selecting **Go To** places your cursor in the callback function.

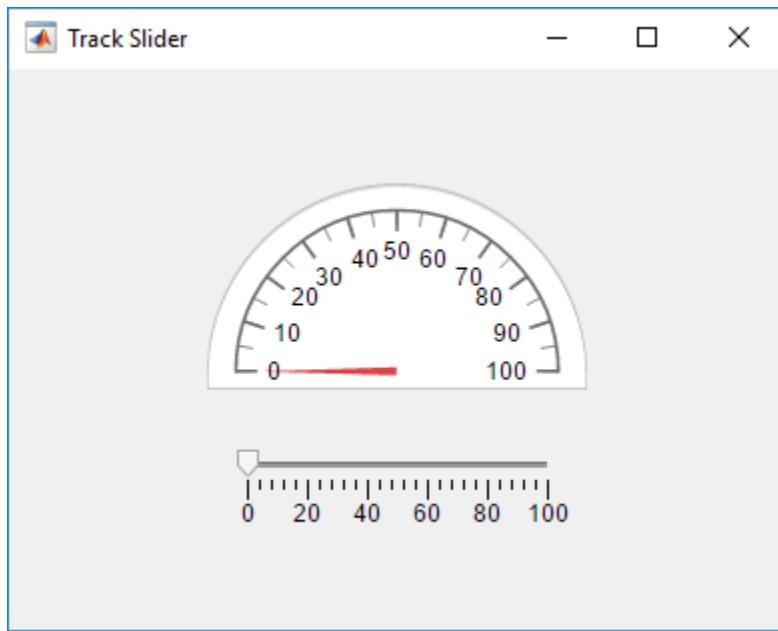
Deleting Callbacks

Delete a callback by right-clicking the callback in the **Callbacks** tab of the **Code Browser** and selecting **Delete** from the context menu.



Example: App with a Slider Callback

This app contains a gauge that tracks the value of a slider as the user moves the thumb. The `ValueChangingFcn` callback for the slider gets the current value of the slider from the `event` argument. Then it moves the gauge needle to that value.



See Also

Related Examples

- “Share Data Within App Designer Apps” on page 17-28
- “Use One Callback for Multiple App Designer Components” on page 17-35

Create Helper Functions in App Designer

Helper functions are MATLAB functions that you define in your app so that you can call them at different places in your code. For example, you might want to update a plot after the user changes a number in an edit field or selects an item in a drop-down list. Creating a helper function allows you to single-source the common commands and avoid having to maintain redundant code.

There are two types of helper functions: private functions, which you can call only inside your app, and public functions, which you can call either inside or outside your app. Private functions are commonly used in single-window apps, while public functions are commonly used in multiwindow apps.

Create a Helper Function

Code View provides a few different ways to create a helper function:

- Expand the drop-down menu from the bottom half of the **Function** button in the **Editor** tab. Select **Private Function** or **Public Function**.



- Select the **Functions** tab in the **Code Browser**, expand the drop-down list on the button, and select **Private Function** or **Public Function**.



When you make your selection, App Designer creates a template function and places your cursor in the body of that function. Then you can update the function name and its arguments, and add your code to the function body. The `app` argument is required, but you can add more arguments after the `app` argument. For example, this function creates a surface plot of the `peaks` function. It accepts an additional argument `n` for specifying the number of samples to display in the plot.

```
methods (Access = private)

    function updateplot(app,n)
        surf(app.UIAxes,peaks(n));
        colormap(app.UIAxes,winter);
    end

end
```

Call the function from within any callback. For example, this code calls the `updateplot` function and specifies 50 as the value for `n`.

```
updateplot(app,50);
```

Managing Helper Functions

Managing helper functions in the **Code Browser** is similar to managing callbacks. You can change the name of a helper function by double-clicking the name in the **Functions** tab of the **Code Browser** and typing a new name. App Designer automatically updates all references to the function when you change its name.

If your app has numerous helper functions, you can quickly search and navigate to a specific function by typing part of the name in the search bar at the top of the **Functions**

tab. After you begin typing, the **Functions** tab clears, except for the items that match your search.

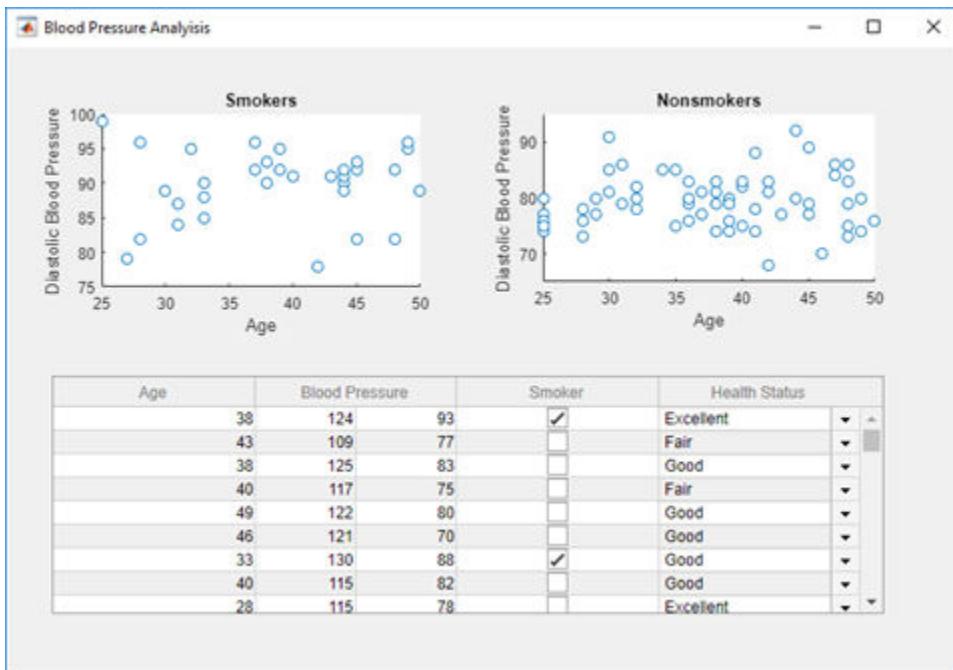


Click a search result to scroll the function into view. Right-clicking a search result and selecting **Go To** places your cursor in the function.

To delete a helper function, select its name in the **Functions** tab and press the **Delete** key.

Example: Helper Function that Initializes and Updates Two Plots

This app shows how to create a helper function that initializes two plots and updates them in a component callback. The app calls the `updateplot` function at the end of the `StartupFcn` callback when the app starts up. The `UITableCellEdit` callback calls the same function to update the plot when the user changes a value in the table.



See Also

Related Examples

- “Write Callbacks in App Designer” on page 17-18
- “Creating Multiwindow Apps in App Designer” on page 17-12

Share Data Within App Designer Apps

Note For information on sharing data in apps you create using GUIDE, see “Share Data Among Callbacks” on page 11-2.

Using properties is the best way to share data within an app because properties are accessible to all functions and callbacks in an app. All UI components are properties, so you can use this syntax to access and update UI components within your callbacks:

`app.Component.Property`

For example, these commands get and set the `Value` property of a gauge. In this case, the name of the gauge is `PressureGauge`.

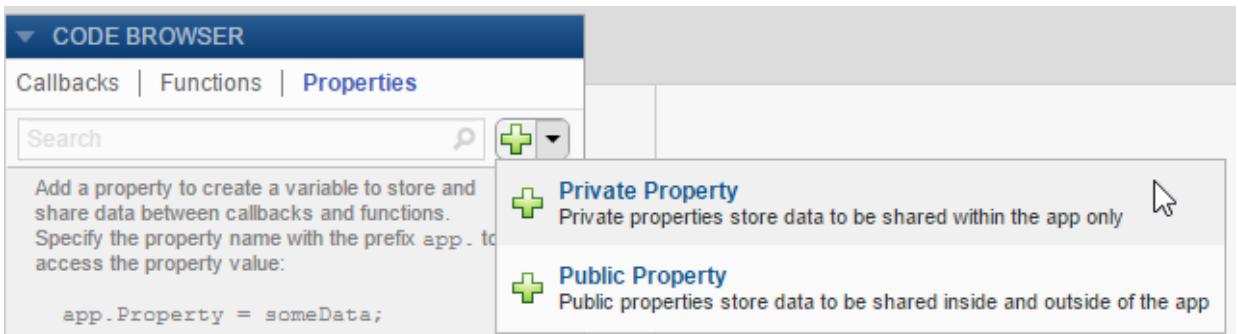
```
x = app.PressureGauge.Value; % Get the gauge value  
app.PressureGauge.Value = 50; % Set the gauge value to 50
```

However, if you want to share an intermediate result, or data that multiple callbacks need to access, then define a public or private property to store your data. Public properties are accessible both inside and outside of the app, whereas private properties are only accessible inside of the app. **Code View** provides a few different ways to create a property:

- Expand the drop-down menu from the bottom half of the **Properties** button in the **Editor** tab. Select **Private Property** or **Public Property**.



- Click on the **Properties** tab in the **Code Browser**, expand the drop-down list on the  button, and select **Private Property** or **Public Property**.



After you select an option to create a property, App Designer adds a property definition and a comment to a **properties** block.

```
properties (Access = public)
    Property % Description
end
```

The **properties** block is editable, so you can change the name of the property and edit the comment to describe the property. For example, this property stores a value for average cost:

```
properties (Access = public)
    X % Average cost
end
```

If your code needs to access a property value when the app starts, you can initialize its value in the **properties** block or in the **StartupFcn** callback.

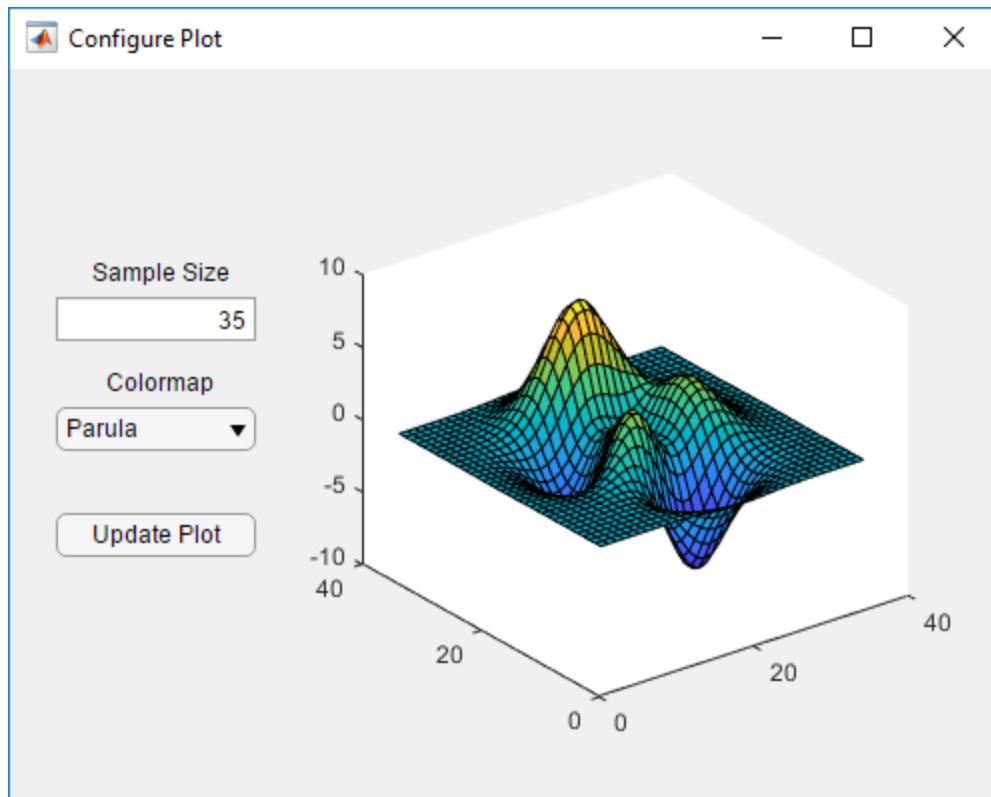
```
properties (Access = public)
    X = 5; % Average cost
end
```

Elsewhere in your code, use dot notation to get or set the value of a property:

```
y = app.X % Get the value of X
app.X = 5; % Set the value of X
```

Example: Share Plot Data and a Drop-Down List Selection

This app shows how to share data in a private property and a drop-down list. It has a private property called Z that stores plot data. The callback function for the edit field updates Z when the user changes the sample size. The callback function for the **Update Plot** button gets the value of Z and the colormap selection to update the plot.



See Also

Related Examples

- “Write Callbacks in App Designer” on page 17-18

- “Creating Multiwindow Apps in App Designer” on page 17-12

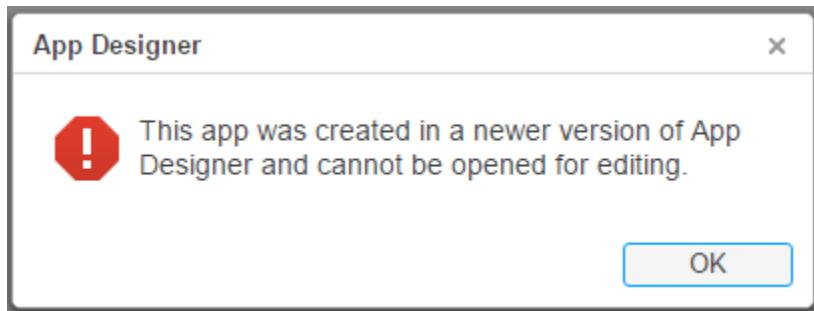
Compatibility Between Different Releases of App Designer

Starting in R2018a, the apps you save in App Designer have a new format. This new file format might impact your ability to edit newer apps in previous releases, but it has no impact on your ability to run them in previous releases.

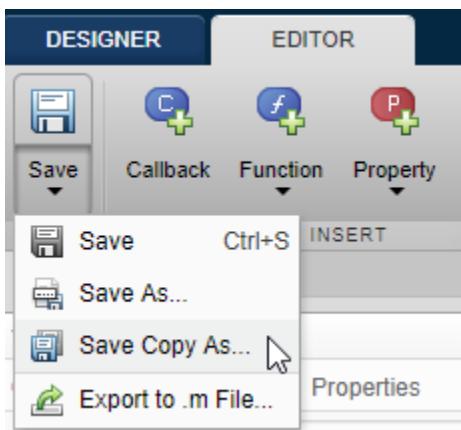
If you open an app for editing that was created in a previous release, App Designer updates the app, and displays a message such as this one.



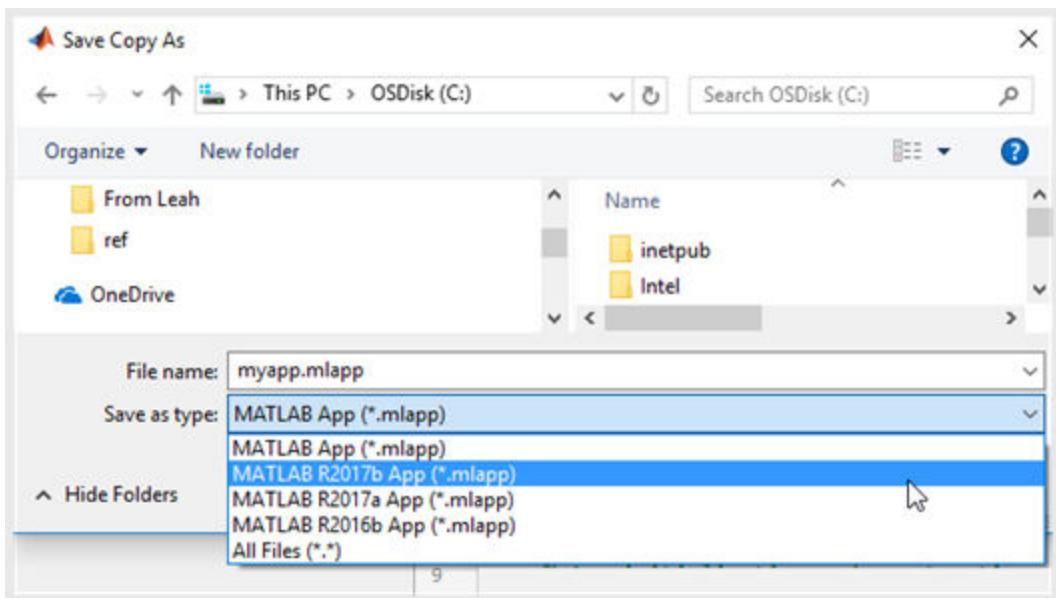
After saving your changes, the new format is not recognized if you try to edit the app in a previous release of App Designer. You might see a message such as this.



To enable editing of newer apps in a previous release (R2017b, R2017a, or R2016b), save the app in the release-specific format. Select **Save > Save Copy As** from any of the tabs in the toolbar.



In the Save Copy As window, select a type from the **Save as Type** drop-down list.



Save Copy As Versus Save As

The **Save Copy As** and **Save As** options serve different purposes, and their behavior is also different.

- To save your app in a format that can be edited in earlier releases, use **Save Copy As**. When you use this option, App Designer saves the copy of the app in the specified folder, but it does not replace the app in your current session.
- To save a copy of your app that is editable only with the current release, use **Save As**. When you use this option, App Designer saves the copy of the app in the specified folder and replaces the app in your current session.

See Also

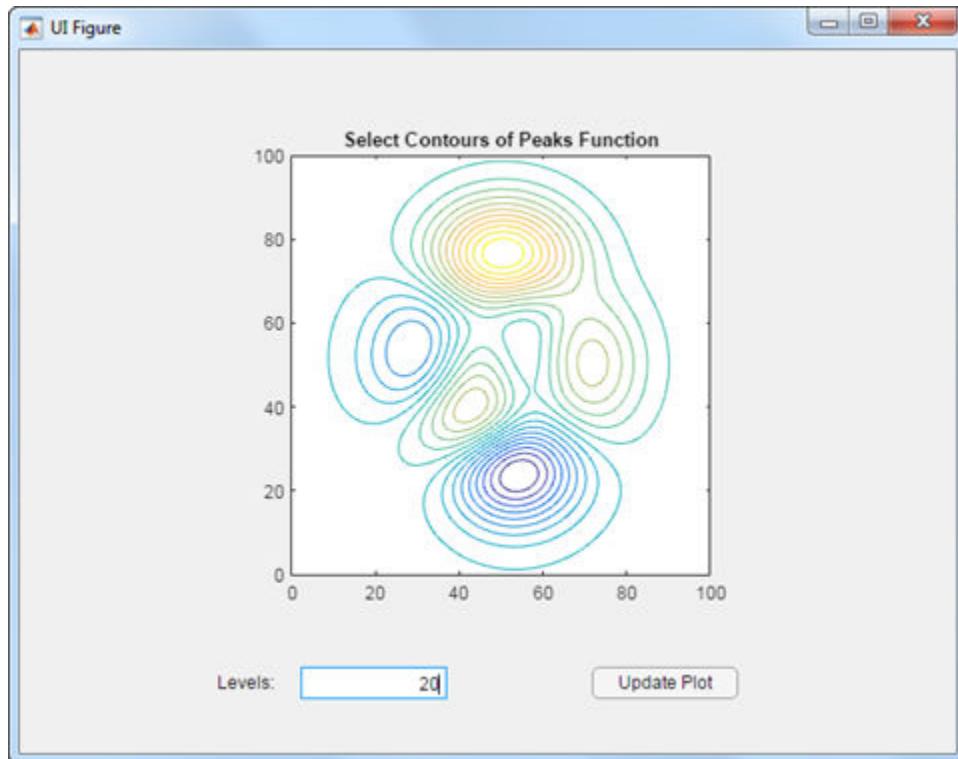
`appdesigner`

Use One Callback for Multiple App Designer Components

Sharing callbacks between components is useful when you want to offer multiple ways of doing something in your app. For example, you might want your app respond the same way when the user clicks a button or presses the **Enter** key in an edit field.

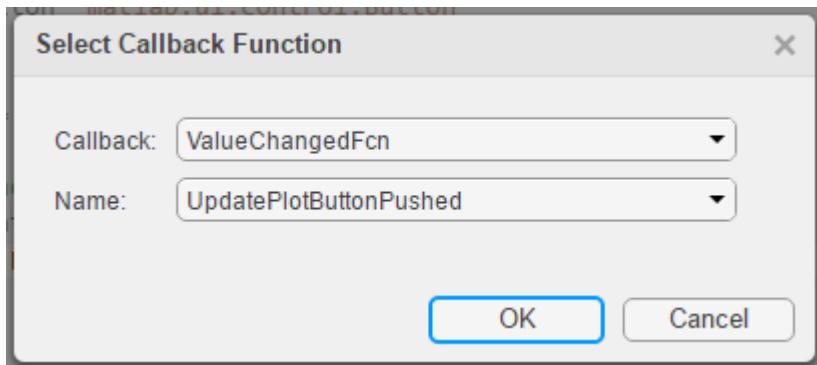
Example of a Shared Callback

This example shows how to create an app containing two UI components that share a callback. The app displays a contour plot with the specified number of levels. When the user changes the value in the edit field, they can press **Enter** or click the **Update Plot** button to update the plot.



- 1 In App Designer, drag an **Axes** component from the **Component Library** onto the canvas. Then make these changes:

- Double-click the title, and change it to **Select Contours of Peaks Function**.
 - Double-click the X and Y axis labels, and press the **Delete** key to remove them.
- 2 Drag an **Edit Field (Numeric)** component below the axes on the canvas. Then make these changes:
- Double-click the label next to the edit field and change it to **Levels**:
 - Double-click the edit field and change the default value to **20**.
- 3 Drag a **Button** component next to the edit field on the canvas. Then double-click its label and change it to **Update Plot**.
- 4 Add a callback function that executes when the user clicks the button. Right-click the **Update Plot** button and select **Callbacks > Add ButtonPushedFcn callback**.
- 5 App Designer switches to the **Code View**. Paste this code into the body of the **UpdatePlotButtonPushed** callback:
- ```
Z = peaks(100);
nlevels = app.LevelsEditField.Value;
contour(app.UIAxes,Z,nlevels);
```
- 6 Next, share the callback with the edit field. In the **Component Browser**, right-click the `app.LevelsEditField` component and select **Callbacks > Select existing callback...**. When the Select Callback Function dialog box displays, select **UpdatePlotButtonPushed** from the **Name** drop-down menu.



Sharing this callback allows the user to update the plot after changing the value in the edit field and pressing **Enter**. Alternatively, they can change the value and press the **Update Plot** button.

- 7 Next, set the axes aspect ratio and limits. In the **Component Browser**, select the app.UIAxes component. Then make the following changes in the **UIAxes Properties** panel:
  - Set **PlotBoxAspectRatio** to 1,1,1.
  - Set **XLim** and **YLim** to 0,100.
- 8 Click **Run** to save and run the app.

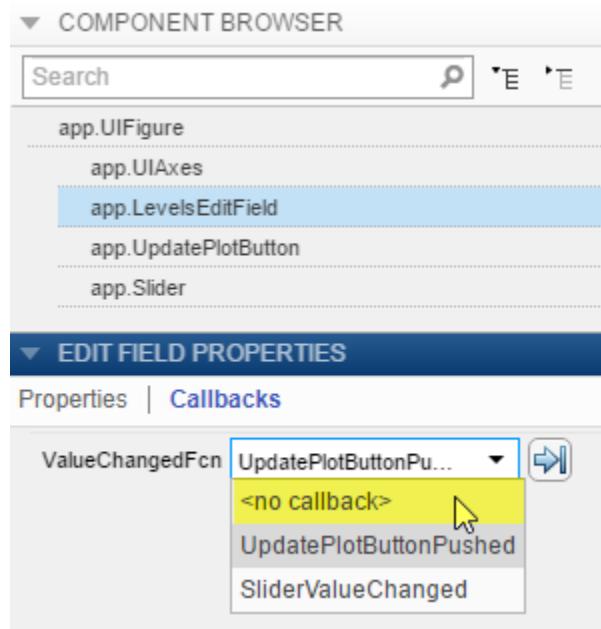


## Change or Disconnect a Callback

To assign a different callback to a component, select the component in the **Component Browser**. Then click the **Callbacks** tab in the properties panel and select a different callback from the drop-down menu. The drop-down displays only the existing callbacks.

The screenshot shows the MATLAB App Designer interface. The **COMPONENT BROWSER** panel on the left lists components: app.UIFigure, app.UIAxes, app.LevelsEditField (selected), app.UpdatePlotButton, and app.Slider. The **EDIT FIELD PROPERTIES** panel on the right has tabs for **Properties** and **Callbacks**. The **Callbacks** tab is active, showing a dropdown menu for the **ValueChangedFcn** property. The dropdown menu contains three options: **<no callback>**, **UpdatePlotButtonPushed** (which is highlighted with a red border), and **SliderValueChanged**. A mouse cursor is visible over the **SliderValueChanged** option.

To disconnect a callback that is shared with a component, select the component in the **Component Browser**. Then click the **Callbacks** tab in the properties panel and select **<no callback>** from the drop-down menu. Selecting this option only disconnects the callback from the component. It does not delete the function definition from your code, nor does it disconnect the callback from any other components.



After you disconnect a callback, you can create a new callback for the component or leave the component without a callback function.

## See Also

### Related Examples

- “Write Callbacks in App Designer” on page 17-18

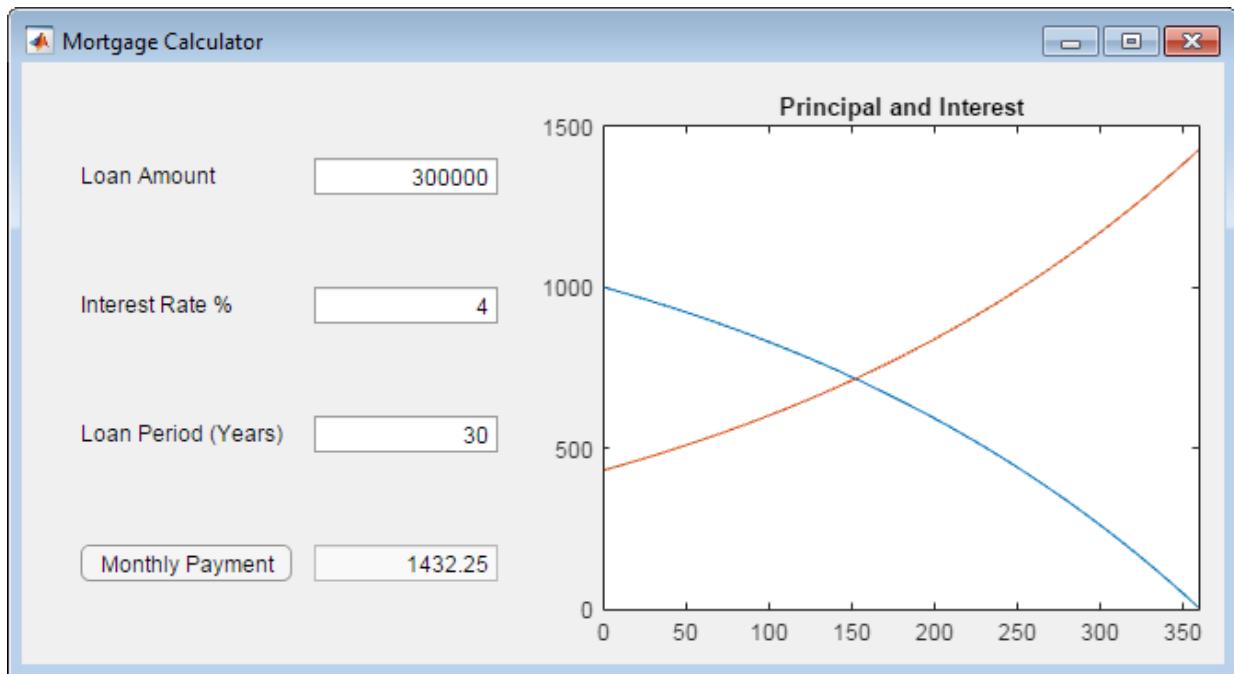
# App Designer Examples

---

## Mortgage Calculator App in App Designer

This app shows how to use numeric edit fields to create a simple mortgage amortization calculator. It includes the following components to collect user input, calculate monthly payments, and plot the principal and interest amounts over time:

- Numeric edit fields — allow users to enter values for the loan amount, interest rate, and loan period. MATLAB® automatically checks to make sure the values are numeric and within the range specified by the app. A fourth numeric edit field displays the resulting monthly payment amount based on the inputs.
- Push button — executes a callback function to calculate the monthly payment value.
- Axes — used to plot the principal and interest amounts versus mortgage installment.



### See Also

UIAxes

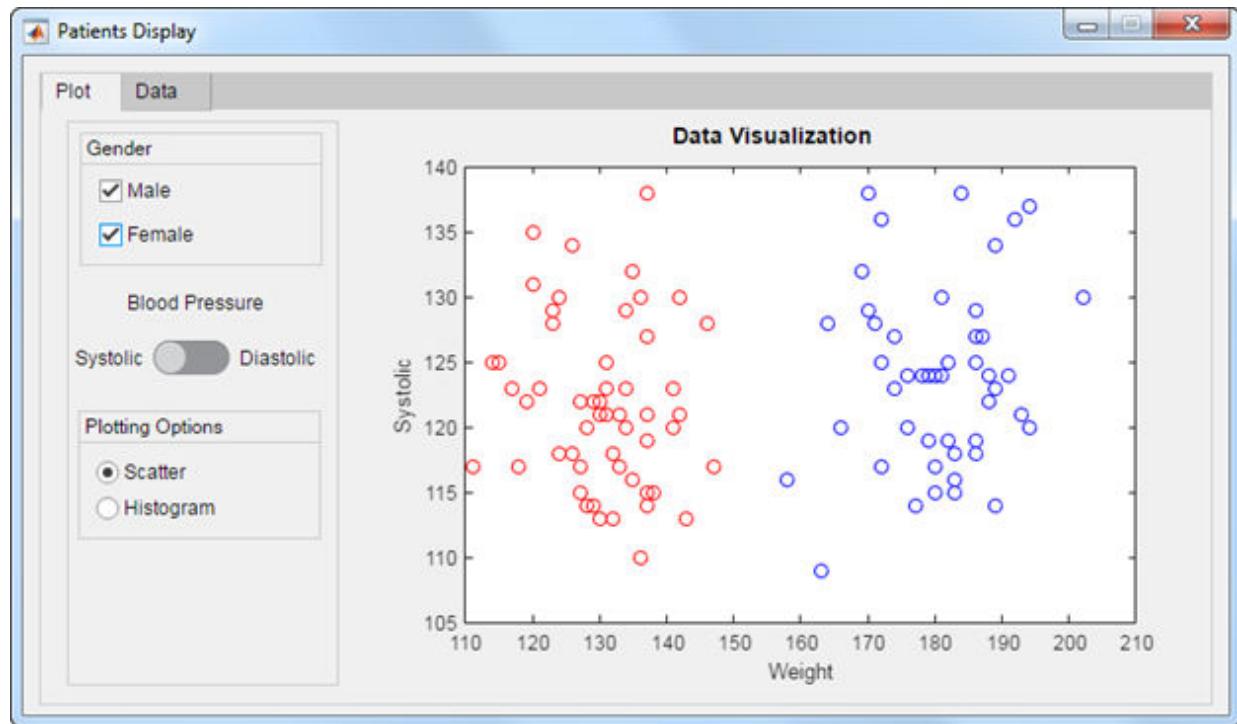
## Related Examples

- “Write Callbacks in App Designer” on page 17-18

## Data Analysis App in App Designer

This app shows how to define two tabs to list and plot data. One tab contains a chart and a few user interface components for adjusting the chart. The other tab displays a table that lists the data used to make the chart. The app includes these components:

- Check boxes — used to update the plot with male or female data when the user toggles a check box.
- Switch — used to switch between visualizing systolic and diastolic data.
- Button group containing radio buttons — used to manage exclusive selection of radio buttons. When the user selects a radio button, the button group executes a callback function to update the plot with the appropriate data.
- Slider — used to adjust histogram bin width. This slider only appears when the **Histogram** plotting option is selected in the button group.
- Table — used to view the data associated with the chart.



## See Also

[Table](#) | [UIAxes](#)

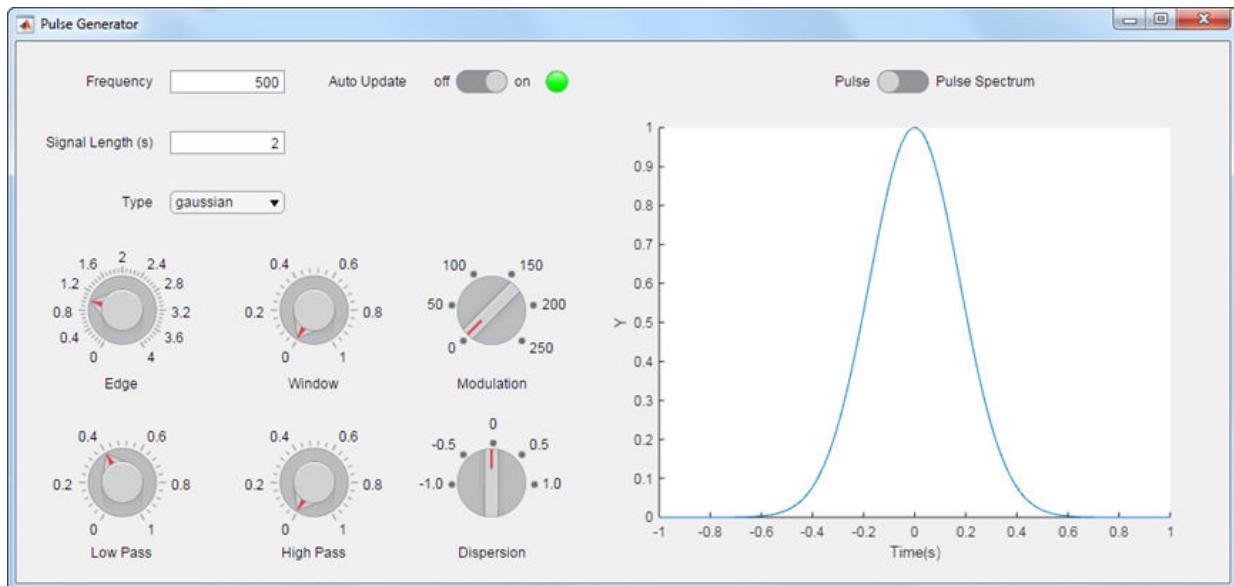
## Related Examples

- “Write Callbacks in App Designer” on page 17-18

## App with Instrumentation Controls in App Designer

This app shows how to generate a pulse from instrumentation controls. It uses the following components to gather user input and plot the resulting wave form:

- Numeric edit fields — allow users to enter the pulse frequency and length. MATLAB® automatically checks to make sure the values are numeric and within the range specified by the app.
- Switches — allow users to control automatic plot updates and toggle between plots in the time and frequency domains.
- Drop-down menu — allows users to select from a list of pulse shapes, such as Gaussian, sinc, and square.
- Knobs — allow users to modify the pulse by specifying a window function, modulating the signal, or applying other enhancements.



### See Also

UIAxes

## Related Examples

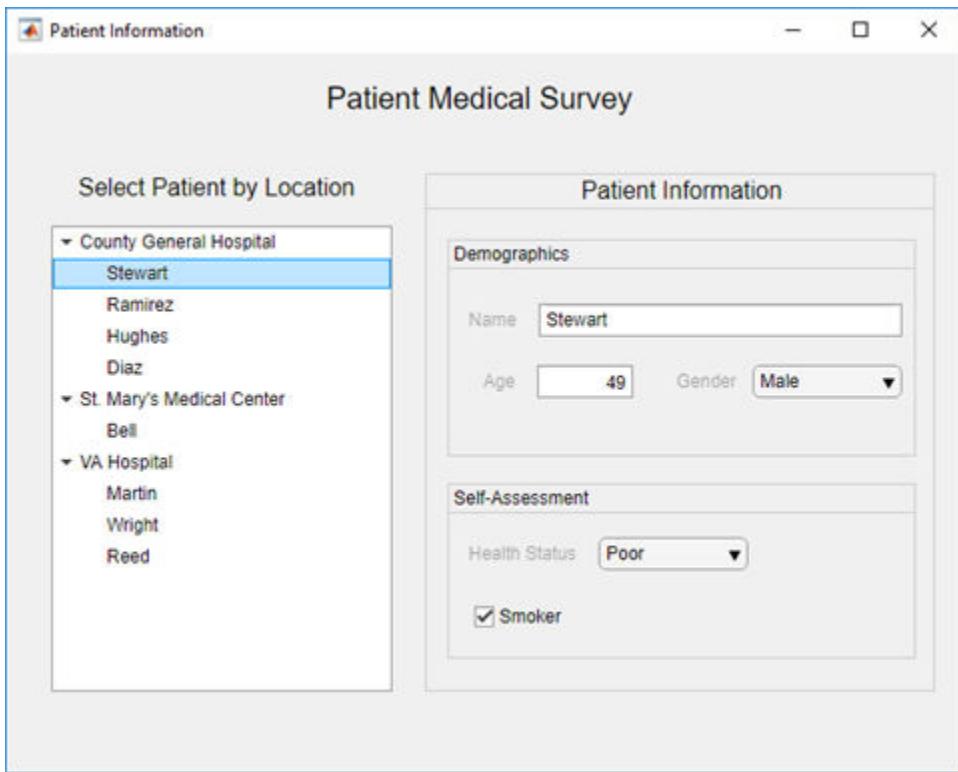
- “Write Callbacks in App Designer” on page 17-18

## App That Displays a Hierarchical Tree in App Designer

This app shows how to add a tree to an App Designer app. The app selects data from `patients.xls` and displays it in a hierarchy using a tree. The tree contains three nodes that display hospital names. Each hospital node contains nodes that display patient names. When the user clicks a patient name in the tree, the **Patient Information** panel displays data such as age, gender, and health status. The app stores changes to the data in the table variable.

In addition to the tree and **Patient Information** panel, the app also contains the following UI components:

- Read-only text field — Used to display the patient's name
- Numeric edit field — Used to display and accept changes to the patient's age
- Drop-down list — Used to display and accept changes to the patient's gender and health status
- Check box — Used to display and accept changes to the patient's smoking history



## See Also

[readtable](#) | [table](#) | [uitree](#) | [uitreenode](#)

## Related Examples

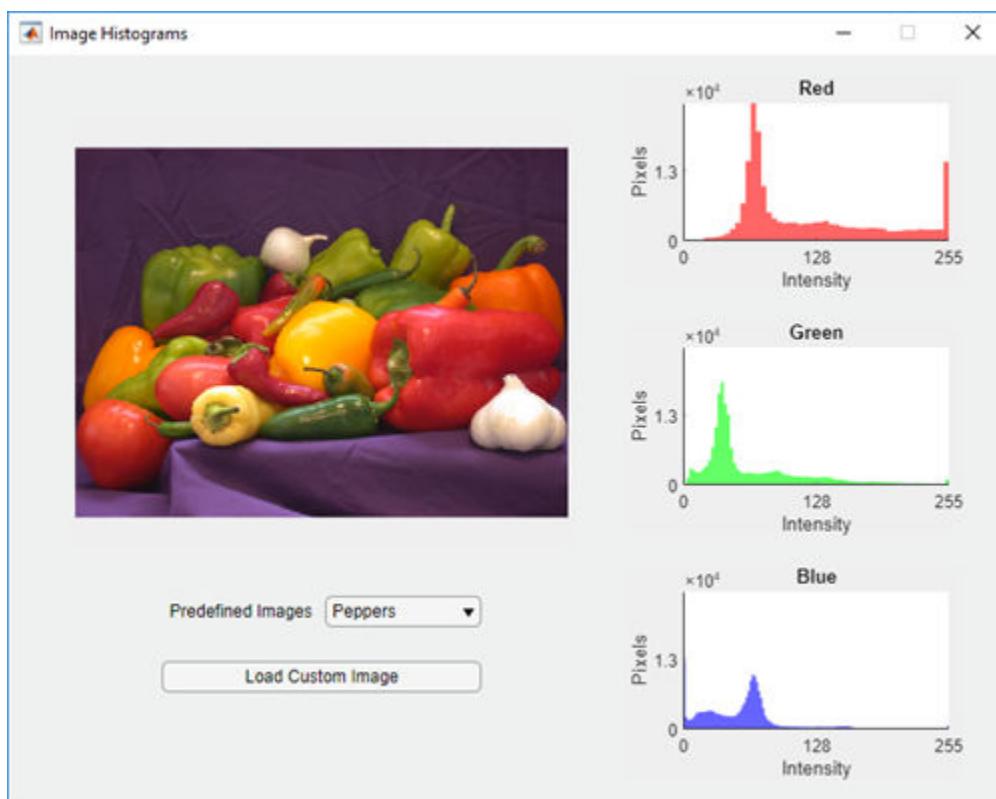
- “Add UI Components to App Designer Programmatically” on page 15-24

## Image Histogram App in App Designer

This app shows how to configure multiple axes components in App Designer. The app displays an image in one axes component, and displays histograms of the red, green, and blue pixels in the other three.

This example also demonstrates the following app building tasks:

- Managing multiple axes
- Reading and displaying images
- Browsing the user's file system using the `uigetfile` function
- Displaying an in-app alert for invalid input (in this case, an unsupported image file)
- Writing a `StartupFcn` callback to initialize the app with a default image



## See Also

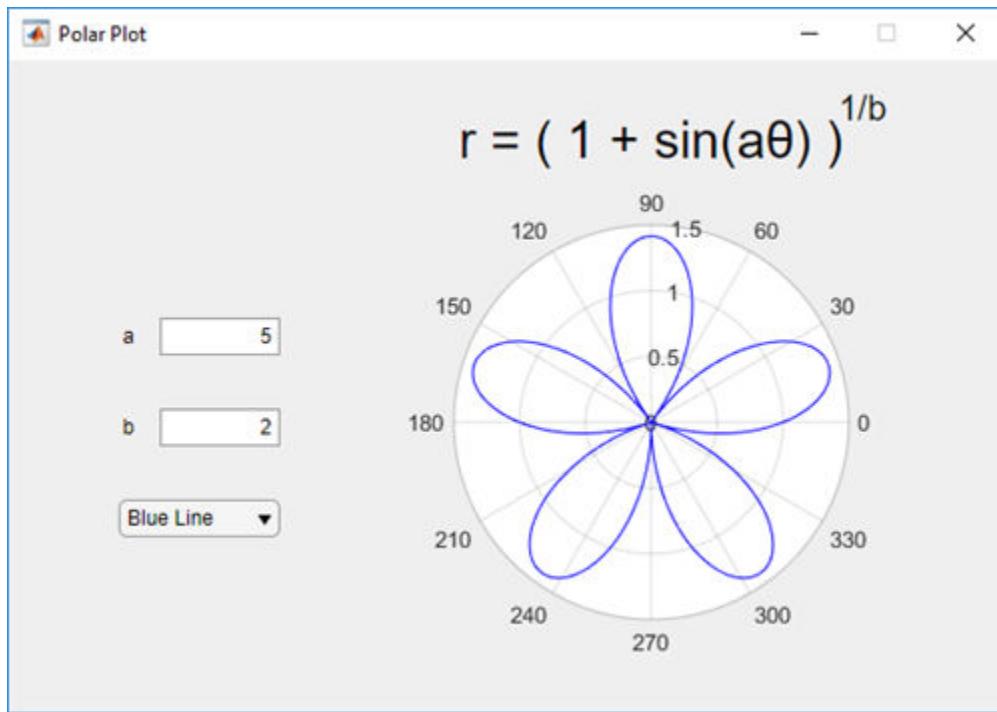
[UIAxes](#) | [imagesc](#) | [imread](#) | [uialert](#)

## Polar Plotting App in App Designer

This app shows how to display a plot by creating the axes programmatically before calling a plotting function. In this case, the app plots a polar equation using the `polaraxes` and `polarplot` functions. When the user changes the value of  $a$  or  $b$ , or when they select a different line color, the plot updates to reflect their changes.

This example also demonstrates the following app building concepts:

- Creating different types of axes programmatically to display plots that `uiaxes` does not support
- Calling a plotting function in App Designer
- Sharing a callback with multiple components
- Displaying Unicode® characters in a label



## See Also

[polaraxes](#) | [polarplot](#)

## Related Examples

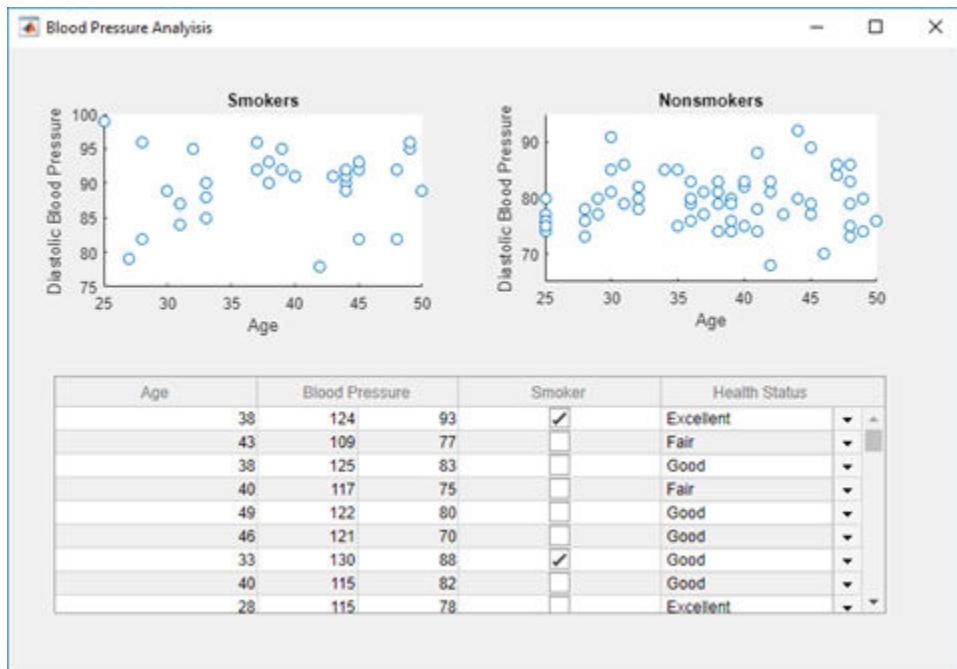
- “Displaying Graphics in App Designer” on page 14-9

## App with an Interactive Table in App Designer

This app shows how to display data in a **Table** UI component. The app loads a spreadsheet into a table array when the app starts up. Then it displays and plots a subset of the data from the spreadsheet. The plots update when the user edits values in the **Table** UI component at run time.

This example demonstrates the following app building tasks:

- Displaying the contents of a table array in a **Table** UI component
- Enabling some of the interactive features of a **Table** UI component



### See Also

[readtable](#) | [table](#)

## Related Examples

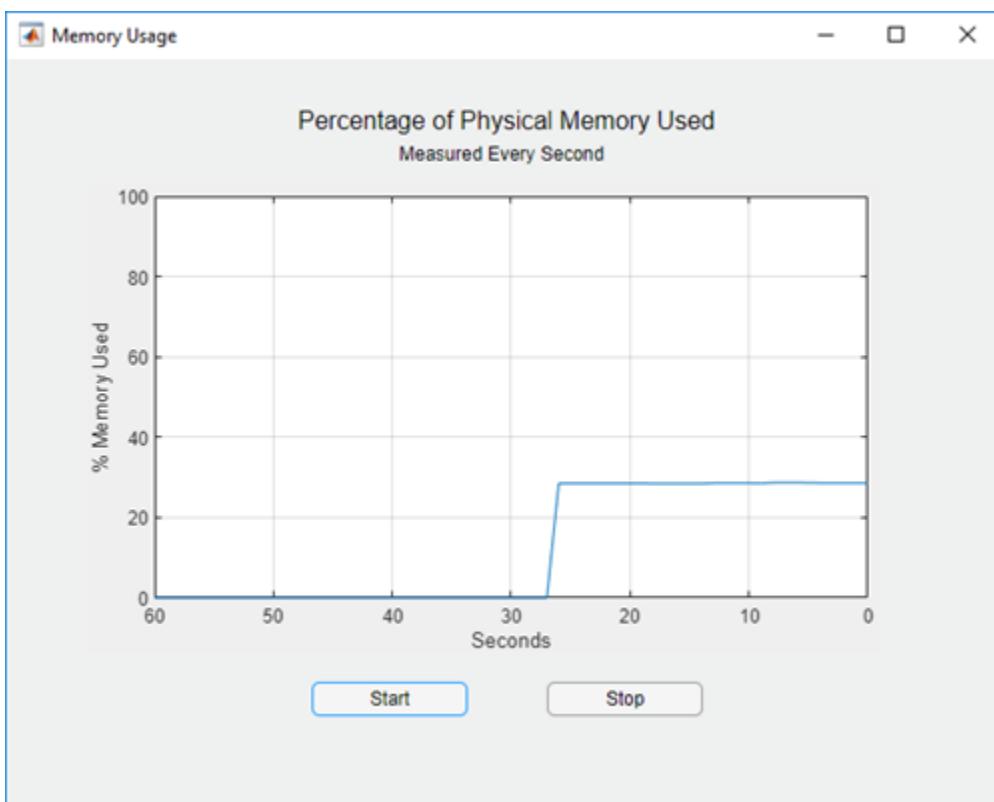
- “Table Array Data Types in App Designer Apps” on page 15-17

## Memory Monitor That Uses Timer Object in App Designer

This app shows how to create a timer object in App Designer that executes a function at regular time intervals. In this case, the app queries the available system memory every second and plots the percentage of memory that the system is using.

This example also demonstrates the following app building tasks:

- Writing a callback for an object created programmatically (in this case, the timer object)
- Configuring a timer object to execute its callback at regular intervals
- Starting the timer when the user clicks the **Start** button
- Stopping the timer when the user clicks the **Stop** button
- Deleting the timer when the app closes



## See Also

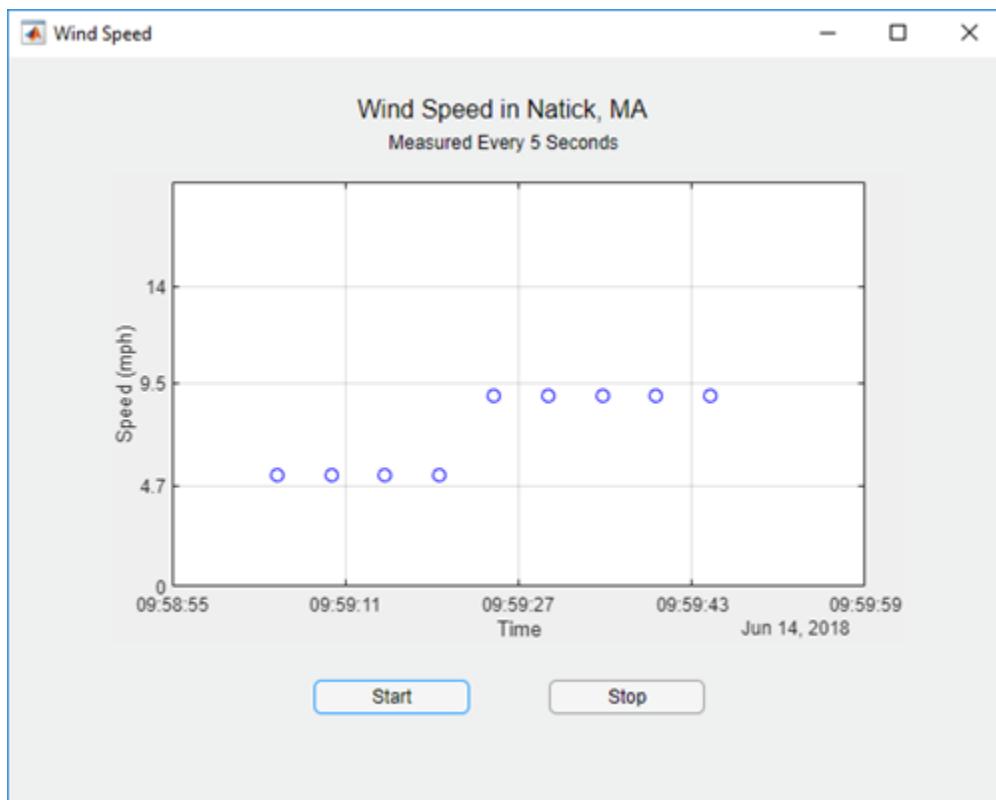
UIAxes | memory | timer

## Wind Speed App That Uses Timer Object in App Designer

This app shows how to create a timer object in App Designer that executes a function at regular time intervals. In this case, the app queries the wind speed from a web site every five seconds and plots the returned value.

This example also demonstrates the following app building tasks:

- Writing a callback for an object created programmatically (in this case, the timer object)
- Configuring a timer object to execute its callback at regular intervals
- Starting the timer when the user clicks the **Start** button
- Stopping the timer when the user clicks the **Stop** button
- Deleting the timer when the app closes



## See Also

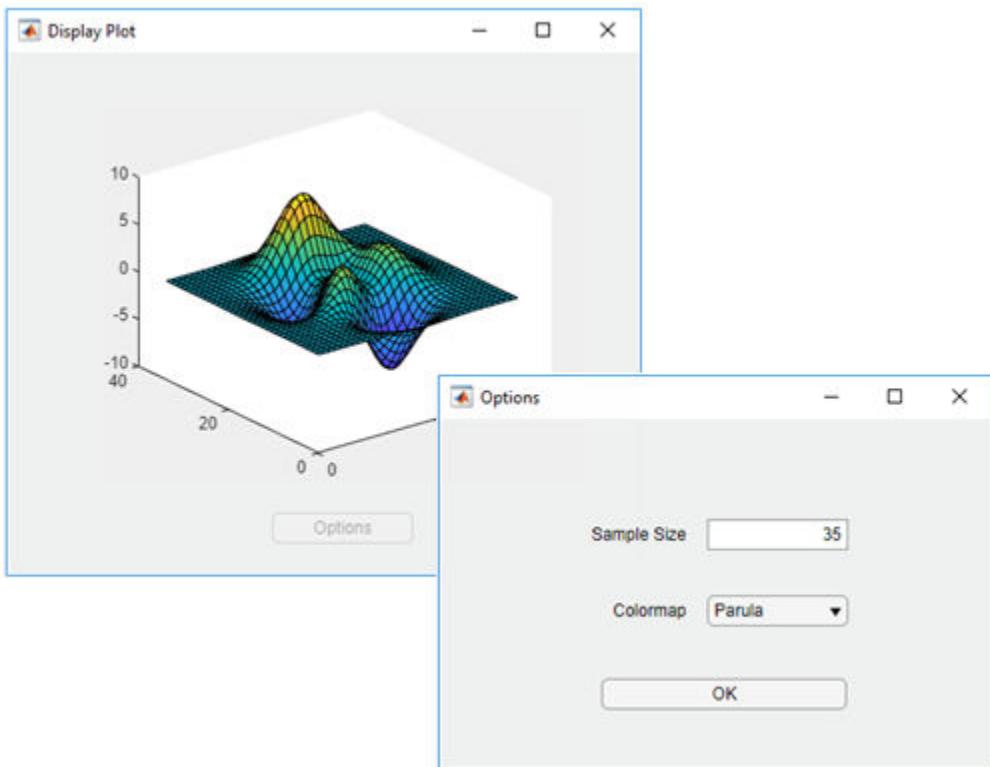
[UIAxes](#) | [timer](#) | [webread](#)

## Multiwindow App in App Designer

This example shows how pass data from one app to another. This multiwindow app consists of a main app that calls a dialog box app with input arguments. The dialog box displays a set of options for modifying aspects of the main app. When the user closes it, the dialog box sends their selections back to the main app.

This example demonstrates the following app building tasks:

- Calling an app with input arguments
- Calling an app with a return argument that is the app object
- Passing values to an app by calling a public function in the app
- Writing `CloseRequestFcn` callbacks to perform maintenance tasks when each app closes



## See Also

### Related Examples

- “Creating Multiwindow Apps in App Designer” on page 17-12
- “Startup Tasks and Input Arguments in App Designer” on page 17-8
- “Create Helper Functions in App Designer” on page 17-24



# Keyboard Shortcuts

---

## App Designer Keyboard Shortcuts

### In this section...

["Shortcuts Available Throughout App Designer" on page 19-2](#)

["Component Browser Shortcuts" on page 19-2](#)

["Design View Shortcuts" on page 19-3](#)

["Code View Shortcuts" on page 19-8](#)

### Shortcuts Available Throughout App Designer

| Action                                                                  | Key or Keys                                                                                   |
|-------------------------------------------------------------------------|-----------------------------------------------------------------------------------------------|
| Run the active app.                                                     | <b>F5</b>                                                                                     |
| Save the active app.                                                    | <b>Ctrl+S</b>                                                                                 |
| Save the active app, allowing you to specify a new file name. (Save as) | <b>Ctrl+Shift+S</b>                                                                           |
| Open a previously saved app.                                            | <b>Ctrl+O</b>                                                                                 |
| Redo an undone modification, returning it to the changed state.         | <b>Ctrl+Y</b> or, in the design area only, <b>Ctrl +Shift+Z</b>                               |
| Undo a modification, returning it to the previous state.                | <b>Ctrl+Z</b>                                                                                 |
| Alternate between design and code view.                                 | <b>Shift + F7</b><br><br>If debugging is in progress, this shortcut does not change the view. |
| Quit App Designer.                                                      | <b>Ctrl+Q</b>                                                                                 |

### 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 <b>Ctrl</b> key as you click each component that you want to include in the multiselection. |
| Deselect a component from multiselection.                                                     | Hold down the <b>Ctrl</b> 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. | <b>Up Arrow</b> and <b>Down Arrow</b>                                                                     |
| Edit code name of clicked component in the code browser.                                      | <b>F2</b> on Windows and Linux<br><b>Enter</b> on Mac                                                     |

## Design View Shortcuts

These shortcuts are available from the App Designer design view only.

- “Add Component Shortcuts” on page 19-4
- “Component, Group, and Text Selection Shortcuts” on page 19-4
- “Group and Ungroup Components Shortcuts” on page 19-4
- “Component and Group Move Shortcuts” on page 19-5
- “Component Resize Shortcuts” on page 19-5
- “Component Copy, Duplicate, and Delete Shortcuts” on page 19-6
- “Design Area Grid Shortcuts” on page 19-6
- “Component Alignment Shortcuts” on page 19-6
- “Change Font Characteristics Shortcuts” on page 19-7
- “Menu Component Shortcuts” on page 19-8
- “Tab Component Shortcuts” on page 19-8

### Add Component Shortcuts

| Action                                                 | Shortcut                                                                                                                                         |
|--------------------------------------------------------|--------------------------------------------------------------------------------------------------------------------------------------------------|
| Add component and associated label (if any) to canvas. | <b>Click</b> the component and hold down the mouse key to drag the component from the <b>Component Library</b> on the left into the design area. |
| Add component only to canvas.                          | Hold down the <b>Ctrl</b> key, click the component, and drag it from the <b>Component Library</b> 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.                                                                    | <b>Tab</b>                              |
| Move the selection to the previous component or container in the design area tab key navigation sequence.                                                                 | <b>Shift+Tab</b>                        |
| Selects all components on the canvas, with one exception. If any of the components are grouped, the group is selected, not the individual components within the grouping. | <b>Ctrl+A</b>                           |
| Clear a component selection. Press again to reselect the component.                                                                                                       | <b>Shift+Click</b> or <b>Ctrl+Click</b> |
| In the property editor or in-place editing, select all text in a text input field.                                                                                        | <b>Ctrl+A</b>                           |
| Select group containing a component.                                                                                                                                      | <b>Alt + Click</b> 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.            | <b>Ctrl+G</b>       |
| Ungroup components in selected group. | <b>Ctrl+Shift+G</b> |

### 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.               | <b>Down Arrow</b>        |
| Move left 1 pixel.               | <b>Left Arrow</b>        |
| Move right 1 pixel.              | <b>Right Arrow</b>       |
| Move up 1 pixel.                 | <b>Up Arrow</b>          |
| Move down 10 pixels.             | <b>Shift+Down Arrow</b>  |
| Move left 10 pixels.             | <b>Shift+Left Arrow</b>  |
| Move right 10 pixels.            | <b>Shift+Right Arrow</b> |
| Move up 10 pixels.               | <b>Shift+Up Arrow</b>    |
| Cancel an in-progress operation. | <b>Escape</b>            |

### Component Resize Shortcuts

| Action                                                                                 | Key                                                                                                             |
|----------------------------------------------------------------------------------------|-----------------------------------------------------------------------------------------------------------------|
| Resize component while maintaining aspect ratio.                                       | Press and hold down the <b>Shift</b> key before you begin to drag the component resize handle.                  |
| Resize component while keeping center location unchanged.                              | Press and hold down the <b>Ctrl</b> key before you begin to drag the component resize handle.                   |
| Resize component while maintaining aspect ratio and keeping center location unchanged. | Press and hold down the <b>Ctrl</b> and <b>Shift</b> keys before you begin to drag the component resize handle. |
| Cancel an in-progress resize operation.                                                | <b>Escape</b>                                                                                                   |

**Component Copy, Duplicate, and Delete Shortcuts**

| Action                                                                                                                                                                                                                                          | Key or Keys                                                              |
|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|--------------------------------------------------------------------------|
| Copy selected components and groups to the clipboard.                                                                                                                                                                                           | <b>Ctrl+C</b>                                                            |
| Duplicate the selected components and groups (without copying them to the clipboard).                                                                                                                                                           | <b>Ctrl+D</b> , or hold down the <b>Ctrl</b> key and drag the component. |
| Cut the selected components and groups from the design area onto the clipboard.                                                                                                                                                                 | <b>Ctrl+X</b>                                                            |
| Delete the selected components and groups from the design area.                                                                                                                                                                                 | <b>Backspace</b> or <b>Delete</b>                                        |
| Paste components and groups from the clipboard into the design area or a 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. | <b>Ctrl+V</b>                                                            |

**Design Area Grid Shortcuts**

| Action                              | Keys                 |
|-------------------------------------|----------------------|
| Toggle grid on and off.             | <b>Alt+G</b>         |
| Toggle snap to grid on and off.     | <b>Alt+P</b>         |
| Increase grid interval by 5 pixels. | <b>Alt+Page Up</b>   |
| Decrease grid interval by 5 pixels. | <b>Alt+Page Down</b> |

**Component Alignment Shortcuts**

| Action                                                            | Keys              |
|-------------------------------------------------------------------|-------------------|
| Align selected components and groups on their left edges.         | <b>Ctrl+Alt+1</b> |
| Align selected components and groups on their horizontal centers. | <b>Ctrl+Alt+2</b> |

| Action                                                         | Keys              |
|----------------------------------------------------------------|-------------------|
| Align selected components and groups on their right edges.     | <b>Ctrl+Alt+3</b> |
| Align selected components and groups on their top edges.       | <b>Ctrl+Alt+4</b> |
| Align selected components and groups on their vertical middle. | <b>Ctrl+Alt+5</b> |
| Align selected components and groups on their bottom edges.    | <b>Ctrl+Alt+6</b> |

### Change Font Characteristics Shortcuts

| Action                                                                                                                                                                                                         | Key or Keys   |
|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|---------------|
| Toggle the font weight of selected components <i>and their children</i> between normal and bold.                                                                                                               | <b>Ctrl+B</b> |
| Toggle the font angle of selected components <i>and their children</i> between normal and italic.                                                                                                              | <b>Ctrl+I</b> |
| Decrease the value of the <b>FontSize</b> property of the selected components <i>and their children</i> by one step.<br><br>Font size steps are: 8, 9, 10, 11, 12, 14, 16, 18, 20, 22, 24, 26, 28, 36, 48, 72. | <b>Ctrl+[</b> |
| Increase the value of the <b>FontSize</b> property of the selected components <i>and their children</i> by one step.<br><br>Font size steps are: 8, 9, 10, 11, 12, 14, 16, 18, 20, 22, 24, 26, 28, 36, 48, 72. | <b>Ctrl+]</b> |

### Menu Component Shortcuts

| Action                                                                                       | Key or Keys                                                   |
|----------------------------------------------------------------------------------------------|---------------------------------------------------------------|
| Add a menu item below the current item.<br>The new menu item appears at the end of the list. | <b>Enter</b>                                                  |
| Add an item to the right of selected item.                                                   | <b>Shift+Enter</b>                                            |
| Delete the current item.                                                                     | <b>Delete</b>                                                 |
| Commit text changes and navigate to the next item.                                           | Any <b>Arrow</b> key                                          |
| Select the first or last item at the level of the selected item.                             | <b>Home</b><br><b>End</b>                                     |
| Move the selected child menu item higher or lower in the list.                               | <b>Ctrl+Shift+Up Arrow</b><br><b>Ctrl+Shift+Down Arrow</b>    |
| Move the selected top-level menu item to the left or right.                                  | <b>Ctrl+Shift+Left Arrow</b><br><b>Ctrl+Shift+Right Arrow</b> |
| Move the selected item to the beginning or end of the list.                                  | <b>Ctrl+Shift+Home</b><br><b>Ctrl+Shift+End</b>               |

### Tab Component Shortcuts

| Action                                         | Key or Keys                                                   |
|------------------------------------------------|---------------------------------------------------------------|
| Move the selected tab to the left or right.    | <b>Ctrl+Shift+Left Arrow</b><br><b>Ctrl+Shift+Right Arrow</b> |
| Move the selected tab to the beginning or end. | <b>Ctrl+Shift+Home</b><br><b>Ctrl+Shift+End</b>               |

### Code View Shortcuts

These shortcuts are available only from the App Designer code view, within the editor.

- “Code Indenting Shortcuts” on page 19-9
- “Cut, Copy, and Paste Code Shortcuts” on page 19-9
- “Find Code Shortcuts” on page 19-9
- “Code Browser Shortcuts” on page 19-9

- “Other App Designer Code Editor Shortcuts” on page 19-10

### **Code Indenting Shortcuts**

| Action                                                              | Key or Keys   |
|---------------------------------------------------------------------|---------------|
| Smart indent selected code.                                         | <b>Ctrl+I</b> |
| Increase indent on current line of code or currently selected code. | <b>Ctrl+]</b> |
| Decrease indent on current line of code or currently selected code. | <b>Ctrl+[</b> |

### **Cut, Copy, and Paste Code Shortcuts**

| Action               | Key or Keys   |
|----------------------|---------------|
| Cut selected code.   | <b>Ctrl+X</b> |
| Copy selected code.  | <b>Ctrl+C</b> |
| Paste selected code. | <b>Ctrl+V</b> |

### **Find Code Shortcuts**

| Action          | Key or Keys     |
|-----------------|-----------------|
| Find.           | <b>Ctrl+F</b>   |
| Find next.      | <b>F3</b>       |
| Find previous.  | <b>Shift+F3</b> |
| Find selection. | <b>Ctrl+F3</b>  |

### **Code Browser Shortcuts**

| Action                                     | Key or Keys   |
|--------------------------------------------|---------------|
| Delete callback.                           | <b>Delete</b> |
| Rename callback.                           | <b>F2</b>     |
| Bring callback to focus and insert cursor. | <b>Ctrl+D</b> |

**Other App Designer Code Editor Shortcuts**

| Action                        | Key or Keys   |
|-------------------------------|---------------|
| Add comment to selected code. | <b>Ctrl+R</b> |
| Evaluate selection.           | <b>F9</b>     |
| Open selection.               | <b>Ctrl+D</b> |
| Go to specified line number.  | <b>Ctrl+G</b> |
| Set or clear breakpoint.      | <b>F12</b>    |

# **App Packaging**



# Packaging GUIs as Apps

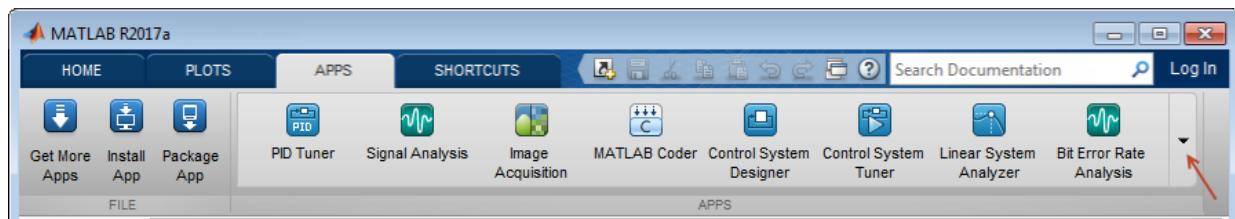
---

- “Apps Overview” on page 20-2
- “Package Apps From the MATLAB Toolstrip” on page 20-5
- “Package Apps in App Designer” on page 20-8
- “Modify Apps” on page 20-11
- “Ways to Share Apps” on page 20-13
- “MATLAB App Installer File — mlappinstall” on page 20-17
- “Dependency Analysis” on page 20-18

# 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 toolbar.



**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 descriptive text.

## 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 any required input arguments. However, you can define optional input arguments. One way to define optional input arguments is by using `varargin`.

## See Also

### Related Examples

- “Package Apps From the MATLAB Toolstrip” on page 20-5
- “Modify Apps” on page 20-11
- “Ways to Share Apps” on page 20-13

## Package Apps From the MATLAB Toolstrip

You can package any MATLAB app you create into a single file that can be easily shared with others. When you package an app, 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 if there are other files used in the app. For more information, see “Dependency Analysis” on page 20-18.

---

**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 If your app requires additional files that are not listed under **Files included through analysis**, add them by clicking **Add files/folders**.

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 screenshot 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**. Keep in mind that your users must have all of the dependent products installed on their systems.

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 your app in the MATLAB Central File Exchange, the same information also displays there. The screenshot 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 Add-Ons Manually”.

## See Also

### Related Examples

- “Modify Apps” on page 20-11
- “Ways to Share Apps” on page 20-13
- “MATLAB App Installer File — `mlappinstall`” on page 20-17
- “Dependency Analysis” on page 20-18

## Package Apps in App Designer

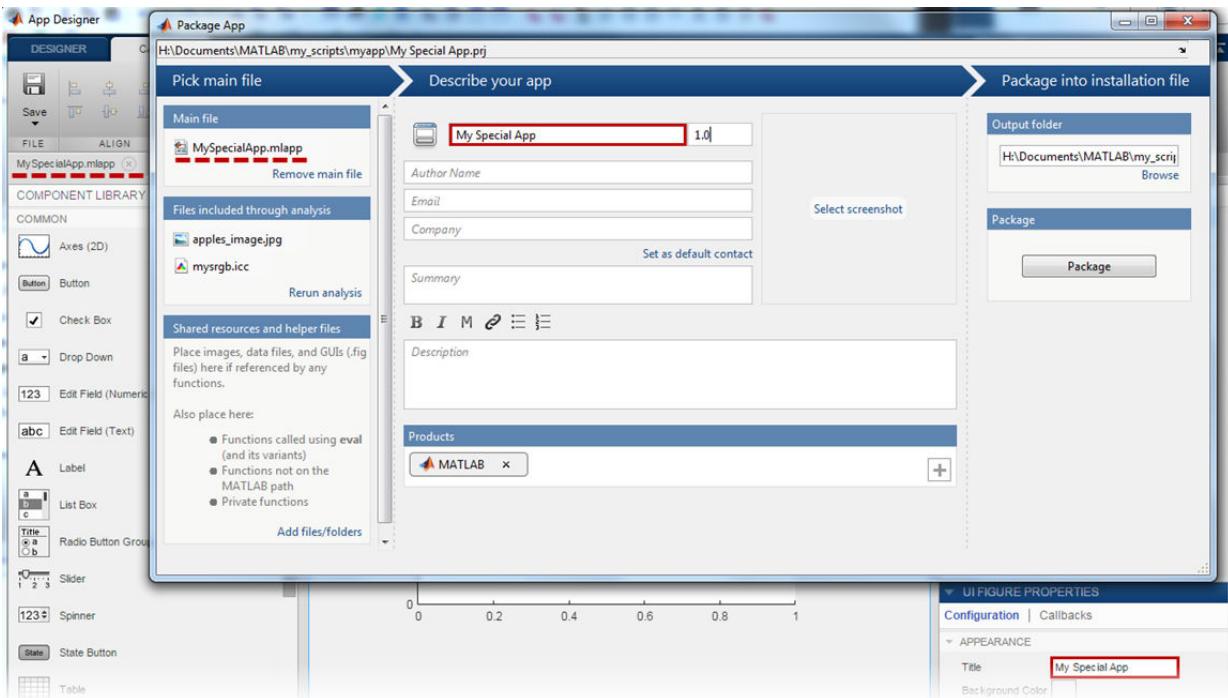
After creating an app in App Designer, you can package it into a single installer file that you can easily share with others. The underlying functionality for packaging apps in App Designer is the same as the functionality that underlies the **Add-Ons > Package App** option in the MATLAB Toolstrip.

- 1 In App Designer, select the **Designer** tab. Then select **Share > MATLAB App**.



MATLAB opens the Package App dialog box.

- 2 The Package App dialog box has the following items pre-populated:
  - The application name matches the title assigned to the figure in App Designer.
  - The **Main file** is the MLAPP file you currently have selected for editing.
  - The **Output folder** is the folder that contains the MLAPP file.
  - The files listed under **Files included through analysis** include any files MATLAB detected as dependent files. You can add additional files by clicking **Add files/folders** under **Shared resources and helper files**.



- 3 Specify details to display in the apps gallery. Enter the appropriate information in these fields: **Author Name**, **Email**, **Company**, **Summary**, and **Description**.
- 4 In the **Products** section, select the products that are required to run the app. Keep in mind that your users must have all of the dependent products installed on their systems.
- 5 Click **Select screenshot** to specify an icon to display in the apps gallery.
- 6 Click **Package** to create the **.mlappinstall** file to share with your users. Later, if you click the **Package App** button in the App Designer Toolbar again, the Package App dialog box opens the most recently modified **.prj** file for the MLAPP file.

## See Also

### Related Examples

- “Package Apps From the MATLAB Toolbar” on page 20-5

- “Ways to Share Apps” on page 20-13
- “MATLAB App Installer File — `mlappinstall`” on page 20-17
- “Dependency Analysis” on page 20-18

## 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. Keep in mind that your users must have all of the dependent products installed on their systems.

- 4 Click **Package**.

## See Also

### Related Examples

- “Ways to Share Apps” on page 20-13
- “MATLAB App Installer File — `mlappinstall`” on page 20-17
- “Dependency Analysis” on page 20-18

# Ways to Share Apps

There are several ways to share your apps.

- “Share MATLAB Files Directly” on page 20-13 — This approach is the simplest way to share an app, but your users must have MATLAB installed on their systems, as well as other MathWorks products that your app depends on. They must also be familiar with executing commands in the MATLAB Command Window and know how to manage the MATLAB path.
- “Package Your App” on page 20-14 — This approach uses the app packaging tool provided with MATLAB. When your users install a packaged app, the app appears in the **Apps** tab in the MATLAB Toolstrip. This approach is useful for sharing apps with larger audiences, or when your users are less familiar with executing commands in the MATLAB Command Window or managing the MATLAB path. As in the case of sharing MATLAB files directly, your users must have MATLAB installed on their systems (as well as other MathWorks products that your app depends on).
- “Create a Deployed Web App” on page 20-15 — This approach lets you create apps that users within an organization can run in their web browsers. To deploy a web app, you must have MATLAB Compiler installed on your system. Your users must have a web browser installed that can access your intranet, but they do not need to have MATLAB installed.
- “Create a Standalone Desktop Application” on page 20-16 — This approach lets you share desktop apps with users that do not have MATLAB installed on their systems. To create the standalone application, you must have MATLAB Compiler installed on your system. To run the application, your users must have MATLAB Runtime installed on their systems. For more information, see <https://www.mathworks.com/products/compiler/mcr.html>.

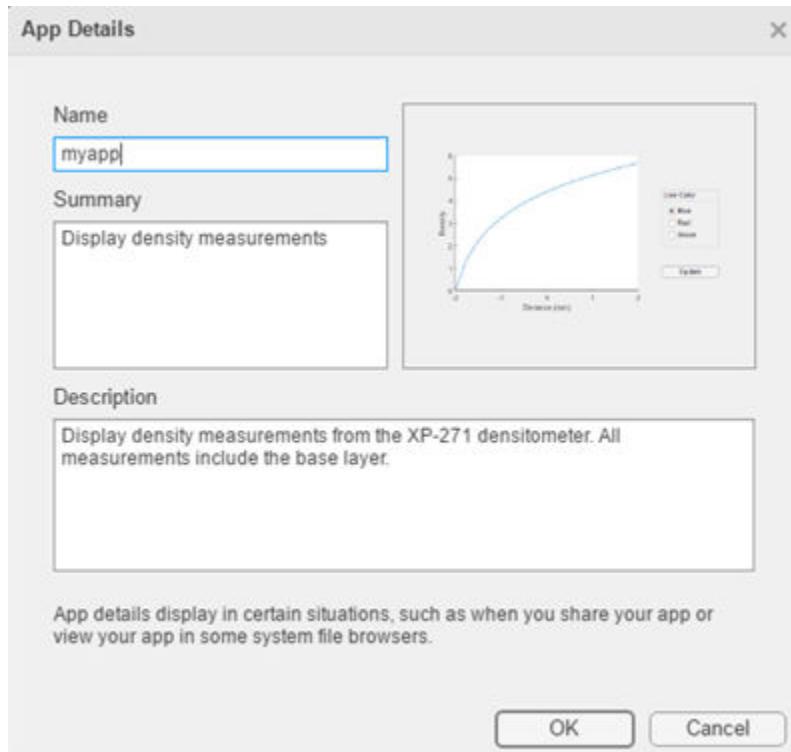
## Share MATLAB Files Directly

If you created your app in GUIDE, share the **.fig** file, the **.m** file, and all other dependent files with your users.

If you created your app programmatically, share all **.m** files and other dependent files with your users.

If you created your app in App Designer, share the **.mlapp** file and all other dependent files with your users. To provide a richer file browsing experience for your users, provide a name, summary, and description by clicking **App Details**  in the **Designer** tab of the

App Designer toolbar. The **App Details** dialog box also provides an option for specifying a screen shot. If you do not specify a screen shot, App Designer captures and updates a screen shot automatically when you run the app.



MATLAB provides your app details to some operating systems for display in their file browsers. Specifying apps details also makes it easier to package and compile your apps. The `.mlapp` file provides those details automatically to those interfaces.

## Package Your App

To package your app and make it accessible in the MATLAB **Apps** tab, create an `.mlappinstall` file by following the steps in “Package Apps in App Designer” on page 20-8 or “Package Apps From the MATLAB Toolbar” on page 20-5. The resulting `.mlappinstall` file includes all dependent files.

You can share the `.mlappinstall` file directly with your users. To install it, they must double-click the `.mlappinstall` file in the MATLAB **Current Folder** browser.

Alternatively, you can share your app as an add-on by uploading the `.mlappinstall` file to MATLAB Central File Exchange. Your users can find and install your add-on from the MATLAB Toolstrip by performing these steps:

- 1 In the MATLAB Toolstrip, on the **Home** tab, in the **Environment** section, click the **Add-Ons**  icon.
- 2 Find the add-on by browsing through available categories on the left side of the Add-On Explorer window. Use the search bar to search for an add-on using a keyword.
- 3 Click the add-on to open its detailed information page.
- 4 On the information page, click **Add** to install the add-on.

---

**Note** Although `.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 files:

- MEX-files
  - Other binary executable files, such as DLLs or ActiveX controls. (Data and image files are typically acceptable.)
- 

## Create a Deployed Web App

Web apps are MATLAB apps that can run in a web browser. You create an app in App Designer, package it using the **Web App Compiler**, and then use the MATLAB Web Apps Manager to serve the app in a web browser. Then you can share the app within your organization by sharing a URL. Creating deployed web apps requires MATLAB Compiler, and only App Designer apps can be deployed as web apps.

Once you have MATLAB Compiler on your system, you can open the **Web App Compiler** from within App Designer by clicking **Share**  in the **Designer** tab and selecting **Web App**. For more information, see “Web Apps” (MATLAB Compiler).

## Create a Standalone Desktop Application

Creating a standalone desktop application lets you share an app with users who do not have MATLAB on their systems. However, you must have MATLAB Compiler installed on your system to create the standalone application. Your users must have MATLAB Runtime on their systems to run the app.

Once you have MATLAB Compiler on your system, you can open the Application Compiler from within App Designer by clicking **Share**  in the **Designer** tab and selecting **Standalone Desktop App**.

If you used GUIDE or created your app programmatically, you can open the Application Compiler from the MATLAB Toolstrip, on the **Apps** tab, by clicking the **Application Compiler** icon.

See “Create Standalone Application from MATLAB” (MATLAB Compiler) for instructions on using the Application Compiler.

## See Also

### Related Examples

- “Apps Overview” on page 20-2
- “Ways to Build Apps” on page 1-2

## 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.

## See Also

### Related Examples

- “Package Apps From the MATLAB Toolstrip” on page 20-5

## Dependency Analysis

When you create an app package, MATLAB analyzes your main file and attempts to include all the files that your app uses. However, MATLAB does not guarantee to find every dependent file. It does not find files for functions that your code references as character vectors (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 the following types of files:

- Executable files, such as MATLAB program files, P-files, Fig-files, and MEX-files.
- Files that your app accesses by calling standard and low-level I/O functions. These dependent files include text files, spreadsheets, images, audio, video, and XML files.
- Files that your app accesses by calling any of these functions: `audioinfo`, `audioread`, `csvread`, `daqread`, `dlmread`, `fileread`, `fopen`, `imfinfo`, `importdata`, `imread`, `load`, `matfile`, `mmfileinfo`, `open`, `readtable`, `type`, `VideoReader`, `xlsfinfo`, `xlsread`, `xmlread`, and `xslt`.

Dependency analysis does not search for Java classes, .jar files, or files stored in a scientific format such as NetCDF or HDF. Click **Add files/folders** in the Package Apps dialog box to add these types of files manually.

## See Also

`matlab.codetools.requiredFilesAndProducts`