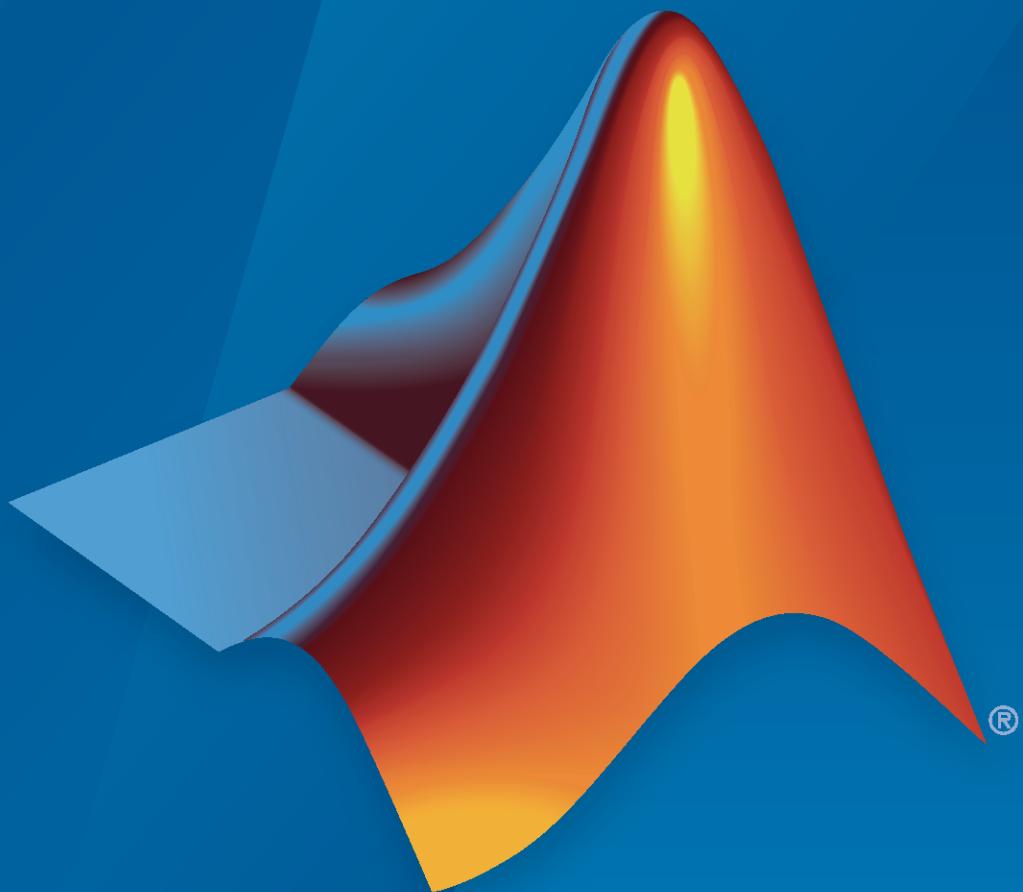


MATLAB®

App Building



MATLAB®

R2021a

 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.
1 Apple Hill Drive
Natick, MA 01760-2098

MATLAB® App Building

© COPYRIGHT 2000–2021 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)
March 2019	Online Only	Revised for MATLAB 9.6 (Release 2019a)
September 2019	Online Only	Revised for MATLAB 9.7 (Release 2019b)
March 2020	Online Only	Revised for MATLAB 9.8 (Release 2020a)
September 2020	Online Only	Revised for MATLAB 9.9 (Release 2020b)
March 2021	Online Only	Revised for MATLAB 9.10 (Release 2021a)

Introduction to Creating Apps

About Apps in MATLAB Software

1

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

A Simple Programmatic App

2

Create and Run a Simple figure-Based App	2-2
Create a Code File	2-2
Create the Figure Window	2-3
Add Components to the UI	2-3
Code the App Behavior	2-5
Verify Code and Run the App	2-7

How to Create a App with GUIDE

3

Create a Simple App Using GUIDE	3-2
Open a New UI in the GUIDE Layout Editor	3-2
Set the Window Size in GUIDE	3-3
Layout the UI	3-4
Code the Behavior of the App	3-10
Run the App	3-15
Files Generated by GUIDE	3-16
Code Files and FIG-Files	3-16
Code File Structure	3-16
Adding Callback Templates to an Existing Code File	3-17
About GUIDE-Generated Callbacks	3-17

App Designer Basics

4

Create and Run a Simple App Using App Designer	4-2
Run the Tutorial	4-2
Tutorial Steps for Creating the App	4-2
GUIDE Migration Strategies	4-5
Export GUIDE App to MATLAB File	4-5
Migrate GUIDE App to App Designer	4-6
Display Graphics in App Designer	4-12
App Designer Graphics Overview	4-12
Display Graphics on Existing Axes	4-12
Display Graphics in Container	4-13
Create Axes Programmatically	4-13
Use Functions With No Target Argument	4-14
Use Functions That Don't Support Automatic Resizing	4-14
Unsupported Functionality	4-15
App Designer Preferences	4-17

Component Choices and Customizations

5

App Building Components	5-2
Common Components	5-2
Axes	5-6
Containers and Figure Tools	5-7
Dialogs and Notifications	5-9
Instrumentation	5-12
Extensible Components	5-13
Toolbox Components	5-14
Table Array Data Types in App Designer Apps	5-15
Logical Data	5-15
Categorical Data	5-15
Datetime Data	5-16
Duration Data	5-16
Nonscalar Data	5-17
Missing Data Values	5-18
Example: App that Displays a Table Array	5-18
Add UI Components to App Designer Programmatically	5-20
Create the Component and Assign the Callback	5-20
Write the Callback	5-20
Example: Confirmation Dialog Box with a Close Function	5-21

Example: App that Populates Tree Nodes Based on a Data File	5-21
---	------

Create HTML File That Can Trigger or Respond to Data Changes

.....	5-23
Include Setup Function in Your HTML File	5-23
Sample HTML File	5-23
Debug an HTML File	5-25

App Layout

6

Lay Out Apps in App Designer Design View	6-2
Customize Components	6-3
Align and Space Components	6-4
Group Components	6-6
Reorder Components	6-6
Arrange Components in Containers	6-7
Create and Edit Context Menus	6-8
Manage Resizable Apps in App Designer	6-11
Resizing Graphics Objects with Normalized Position Units	6-11
Alternatives to Default Auto-Resize Behaviors	6-11
Use Grid Layout Managers	6-13
Add and Configure Grid Layout Manager	6-13
Convert Components from Pixel-Based Positions to Grid Layout Manager	6-13
Example: Convert Components to Use Grid Layout Manager Instead of Pixel-Based Positions	6-14
Apps with Auto-Reflow	6-16
What is Auto-Reflow?	6-16
Create New App with Auto-Reflow	6-17
Convert Existing App to Use Auto-Reflow	6-17
Example: App with Auto-Reflow	6-18

App Programming

7

Manage Code in App Designer Code View	7-2
Manage Components, Functions, and Properties	7-2
Identify Editable Sections of Code	7-3
Program Your App	7-3
Fix Code Problems and Run-Time Errors	7-6
Personalize Code View Appearance	7-6
Startup Tasks and Input Arguments in App Designer	7-8
Create a startupFcn Callback	7-8
Define Input App Arguments	7-8

Create Multiwindow Apps in App Designer	7-11
Overview of the Process	7-11
Send Information to the Dialog Box	7-11
Return Information to the Main App	7-13
Manage Windows When They Close	7-13
Example: Plotting App That Opens a Dialog Box	7-14
Write Callbacks in App Designer	7-15
Create a Callback Function	7-15
Using Callback Function Input Arguments	7-16
Searching for Callbacks in Your Code	7-17
Deleting Callbacks	7-18
Example: App with a Slider Callback	7-18
Reuse Code Using Helper Functions	7-20
Create a Helper Function	7-20
Managing Helper Functions	7-21
Example: Helper Function that Initializes Plots and Displays Updated Data	7-21
Share Data Within App Designer Apps	7-23
Example: Share Plot Data and a Drop-Down List Selection	7-24
Compatibility Between Different Releases of App Designer	7-26
Save Copy As Versus Save As	7-27
Opening Apps for Editing in a Newer Release	7-27
Use One Callback for Multiple App Designer Components	7-28
Example of a Shared Callback	7-28
Change or Disconnect a Callback	7-29

8

App Designer Examples

App that Calculates and Plots Data Based on Numerical Input	8-2
App with Auto-Reflow That Updates Plot Based on User Selections	8-3
App that Uses Grid Layout to Manage Component Positions and Resizing	8-4
App That Displays Data in a Hierarchy Using Tree	8-5
Create App that Uses Multiple Axes to Display Results of Image Analysis	8-6
Create Polar Axes Programmatically in an App	8-7
Create App with a Table That Can Be Sorted and Edited Interactively	8-8

Create App with Timer Object Configured Programmatically	8-10
Create App with Timer Object that Queries Website Data	8-12
Share Data in Multiwindow Apps	8-13
Display HTML Elements Styled by a Cascading Style Sheet	8-14

Keyboard Shortcuts

9

App Designer Keyboard Shortcuts	9-2
Shortcuts Available Throughout App Designer	9-2
Component Browser Shortcuts	9-2
Design View Shortcuts	9-3
Code View Shortcuts	9-7

Create UIs Programmatically

Lay Out a Programmatic UI

10

Lay Out a UI Programmatically	10-2
Component Placement and Sizing	10-2
Managing the Layout in Resizable UIs	10-6
Manage the Stacking Order of Grouped Components	10-8
DPI-Aware Behavior in MATLAB	10-10
Visual Appearance	10-10
Using Object Properties	10-12
Using print, getframe, and publish Functions	10-13

Code a Programmatic App

11

Write Callbacks for Apps Created Programmatically	11-2
Callbacks for Different User Actions	11-2
How to Specify Callback Property Values	11-3

Manage Application-Defined Data

12

Share Data Among Callbacks	12-2
Overview of Data Sharing Techniques	12-2
Store Data in UserData or Other Object Properties	12-2
Store Data as Application Data	12-3
Create Nested Callback Functions	12-5
Store Data Using the guidata Function	12-5

Manage Callback Execution

13

Interrupt Callback Execution	13-2
How to Control Interruption	13-2
Callback Behavior When Interruption is Allowed	13-2
Example	13-2

Examples of Programmatic Apps

14

Programmatic App that Displays a Table	14-2
Create a Table UI Component Within a Figure	14-2
Create a Table Containing Mixed Data Types	14-2
Customize the Display	14-2
Enable Column Sorting and Restrict Editing of Cell Values	14-5
Create a Callback	14-6
Get All Table Properties	14-6

Developing Classes of UI Component Objects

15

Custom UI Component Development Overview	15-2
Structure of a UI Component Class	15-2
Constructor Method	15-3
Public and Private Property Blocks	15-3
Event Block	15-4
Setup Method	15-5
Update Method	15-5
Example: Color Selector UI Component	15-6
Manage Properties of Custom UI Components	15-9
Initialize Property Values	15-9
Validate Property Values	15-9
Customize the Property Display	15-10

Optimize the update Method	15-11
Example: Optimized Polynomial Fit UI Component with Customized Property Display	15-12
Configure Custom UI Components for App Designer	15-17
Custom UI Component Class Prerequisites	15-17
Create a UI Component Class to Configure	15-18
Configure App Designer Metadata	15-19
View Configured UI Component in App Designer	15-20
Reconfigure UI Component	15-21
Remove UI Component From App Designer	15-23
Share Configured UI Component	15-23
Customize Properties of HTML UI Component	15-25
Class Construction Overview	15-25
RoundButton Class Implementation	15-26

Create UIs with GUIDE

16

What Is GUIDE?

GUIDE: Getting Started	16-2
UI Layout	16-2
UI Programming	16-2

17

GUIDE Preferences and Options

GUIDE Preferences	17-2
Set Preferences	17-2
Confirmation Preferences	17-2
Backward Compatibility Preference	17-4
All Other Preferences	17-4
GUIDE Options	17-8
The GUI Options Dialog Box	17-8
Resize Behavior	17-8
Command-Line Accessibility	17-9
Generate FIG-File and MATLAB File	17-10
Generate FIG-File Only	17-11

Set the UI Window Size in GUIDE	18-2
Prevent Existing Objects from Resizing with the Window	18-2
Set the Window Position or Size to an Exact Value	18-2
Maximize the Layout Area	18-3
Add Components to the GUIDE Layout Area	18-4
Place Components	18-4
User Interface Controls	18-8
Panels and Button Groups	18-22
Axes	18-26
Table	18-29
Resize GUIDE UI Components	18-37
Create Menus for GUIDE Apps	18-40
Menus for the Menu Bar	18-40
Context Menus	18-47

Programming a GUIDE App

Write Callbacks in GUIDE	19-2
Callbacks for Different User Actions	19-2
GUIDE-Generated Callback Functions and Property Values	19-4
GUIDE Callback Syntax	19-4
Share Data Among GUIDE Callbacks	19-5
GUIDE Example: Share Slider Data Using guidata	19-10
GUIDE Example: Share Data Between Two Apps	19-10
GUIDE Example: Share Data Among Three Apps	19-11
Renaming and Removing GUIDE-Generated Callbacks	19-13
Callbacks for Specific Components	19-14
How to Use the Example Code	19-14
Push Button	19-14
Toggle Button	19-14
Radio Button	19-15
Check Box	19-15
Edit Text Field	19-16
Slider	19-17
List Box	19-17
Pop-Up Menu	19-18
Panel	19-19
Button Group	19-20
Menu Item	19-21
Table	19-23
Axes	19-24
Examples of GUIDE Apps	19-27

GUIDE App With Parameters for Displaying Plots	20-2
Open and Run the Example	20-2
Examine the Code	20-3
Interactive List Box App in GUIDE	20-6
Open and Run The Example	20-6
Examine the Layout and Callback Code	20-7
Automatically Refresh Plot in a GUIDE App	20-9
Open and Run the Example	20-9
Examine the Code	20-10

App Packaging

Get and Create Apps	21-2
What Is an App?	21-2
Where to Get Apps	21-2
Why Create an App?	21-3
Best Practices and Requirements for Creating an App	21-3
Package Apps From the MATLAB Toolstrip	21-5
Package Apps in App Designer	21-7
Modify Apps	21-9
Ways to Share Apps	21-10
Share MATLAB Files Directly	21-10
Package Your App	21-12
Create a Deployed Web App	21-13
Create a Standalone Desktop Application	21-13
MATLAB App Installer File — mlappinstall	21-14
App Packaging Dependency Analysis	21-15

Introduction to Creating Apps

About Apps in MATLAB Software

Ways to Build Apps

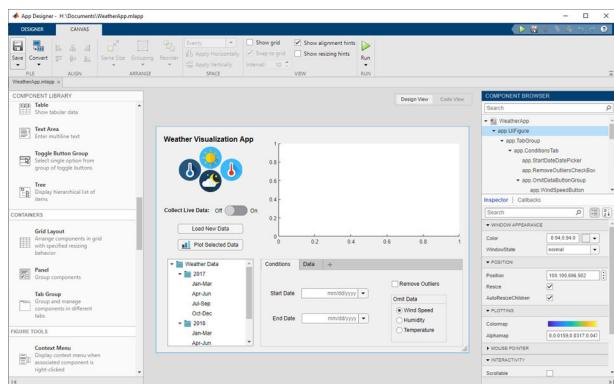
There are different ways to build MATLAB apps:

- Interactively, using App Designer
- Programmatically, using MATLAB functions

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.

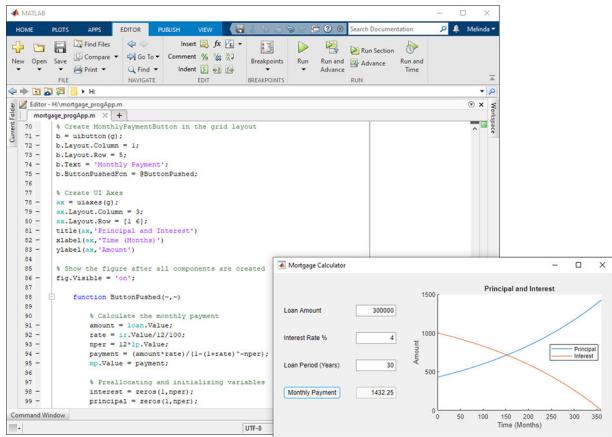
Use App Designer

App Designer is a rich interactive environment introduced in R2016a, and it is the recommended environment for building apps in MATLAB. 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 components is available, including date picker, tree, and image. There are also features like a grid layout manager and automatic reflow options to make your app detect and adapt to changes in screen size. For more information, see “Develop Apps Using App Designer”.



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 figure to serve as the container for your UI using either the `uifigure` or `figure` function. Then, you add components to it programmatically. Each type of figure supports different components and properties. Select the workflow that best suits your project requirements. For more information, see “Develop Apps Programmatically”.



See Also

Related Examples

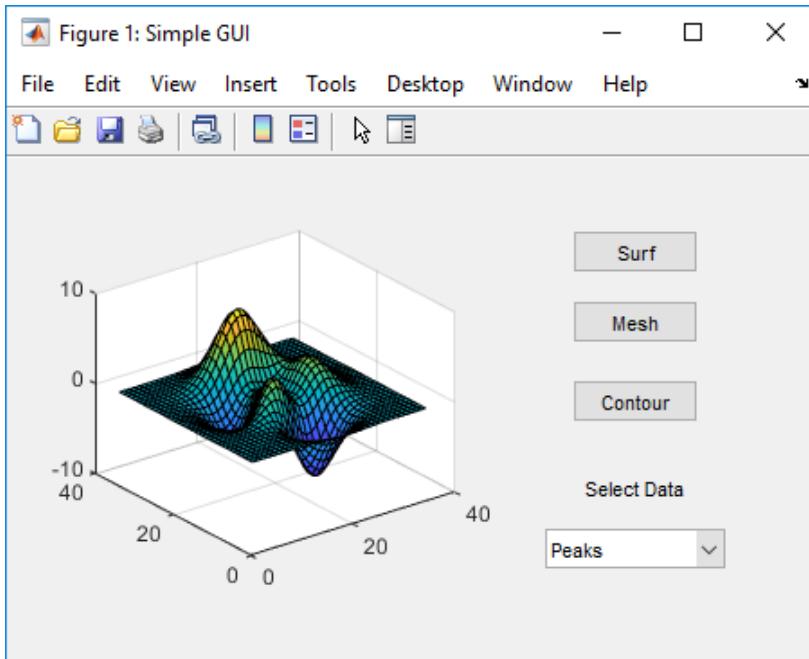
- “Create and Run a Simple App Using App Designer” on page 4-2
- “Create and Run a Simple figure-Based App” on page 2-2
- “Display Graphics in App Designer” on page 4-12
- “GUIDE Migration Strategies” on page 4-5

A Simple Programmatic App

Create and Run a Simple figure-Based App

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
```

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 pop-up menu.

```

htext = uicontrol('Style','text','String','Select Data',...
    'Position',[325,90,60,15]);
hpopup = uicontrol('Style','popupmenu',...
    'String',{'Peaks','Membrane','Sinc'},...
    'Position',[300,50,100,25]);

```

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

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

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

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

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

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

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

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

- 5 Add this command following the **align** command.

```
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]);

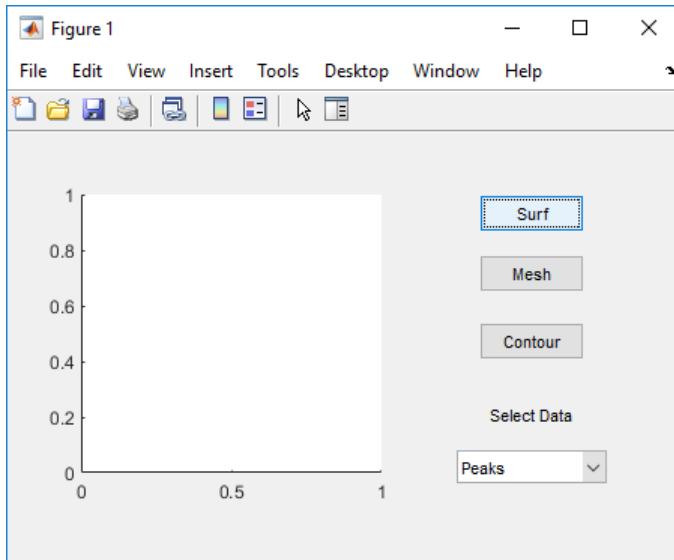
% Construct the components.
hsurf = uicontrol('Style','pushbutton','String','Surf',...
    'Position',[315,220,70,25]);
hmesh = uicontrol('Style','pushbutton','String','Mesh',...
    'Position',[315,180,70,25]);
hcontour = uicontrol('Style','pushbutton',...
    'String','Contour',...
    'Position',[315,135,70,25]);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

```
% 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 11-2.

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 $(1.0, 1.0)$.

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.

- 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.

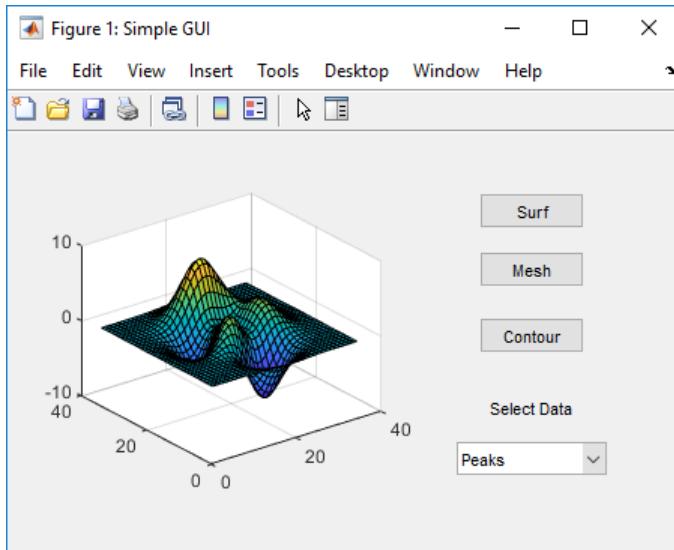
function surfbutton_Callback(source,eventdata)
% Display surf plot of the currently selected data.
    surf(current_data);
end

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
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 and Run a Simple App Using App Designer” on page 4-2

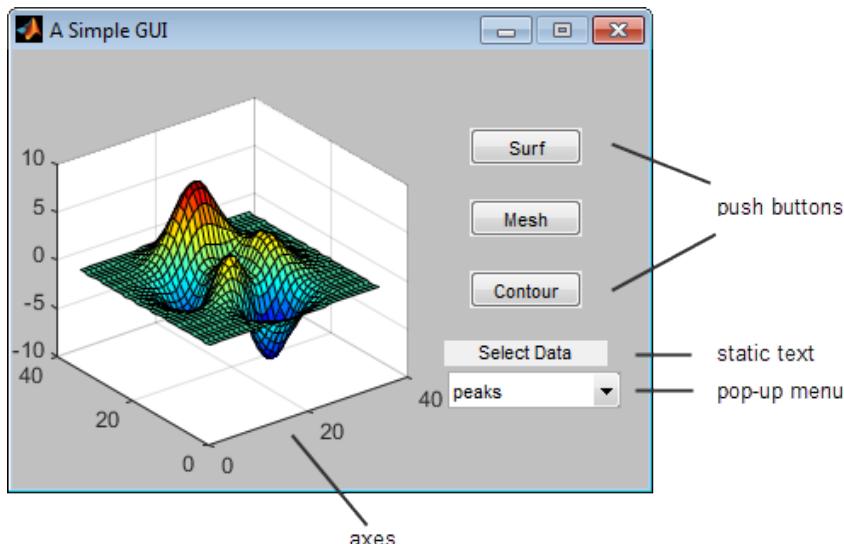
How to Create a App with GUIDE

Create a Simple App Using GUIDE

Note The GUIDE environment will be removed in a future release. After GUIDE is removed, existing GUIDE apps will continue to run in MATLAB but they will not be editable in GUIDE.

To continue editing an existing GUIDE app, see “GUIDE Migration Strategies” on page 4-5 for information on how to help maintain compatibility of the app with future MATLAB releases. To create new apps interactively, “Develop Apps Using App Designer” instead.

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.

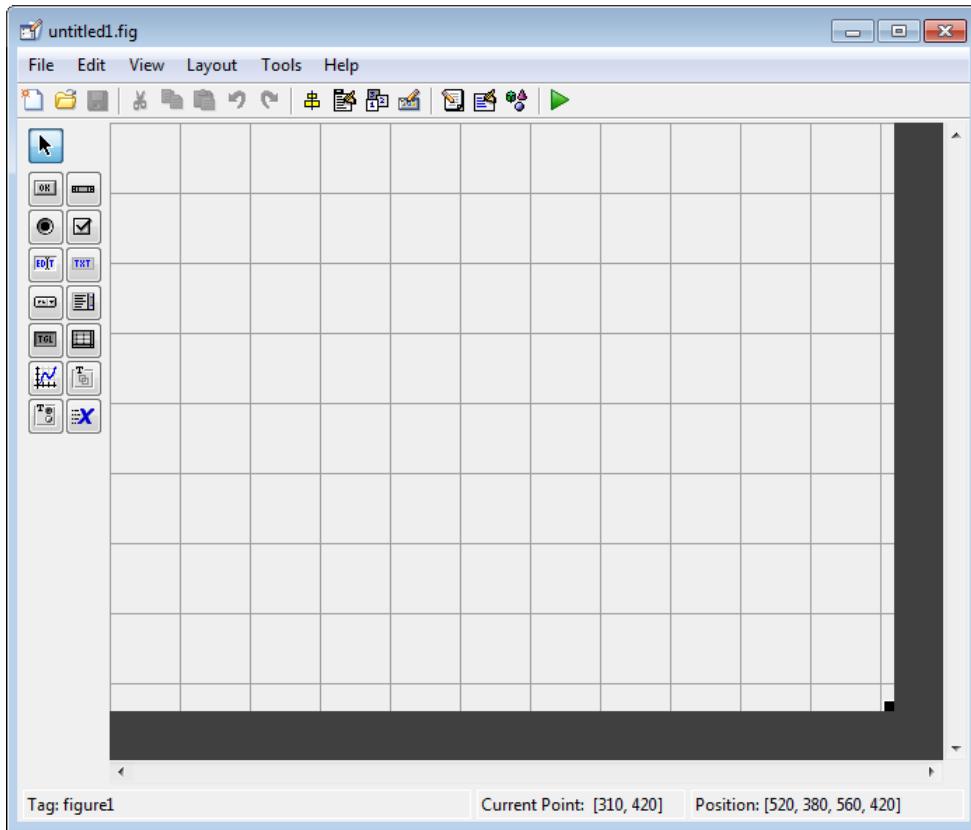
If you only want to view and run the code that created this app, set your current folder to one to which you have write access. Copy the example code and open it in the Editor by issuing the following MATLAB commands:

```
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

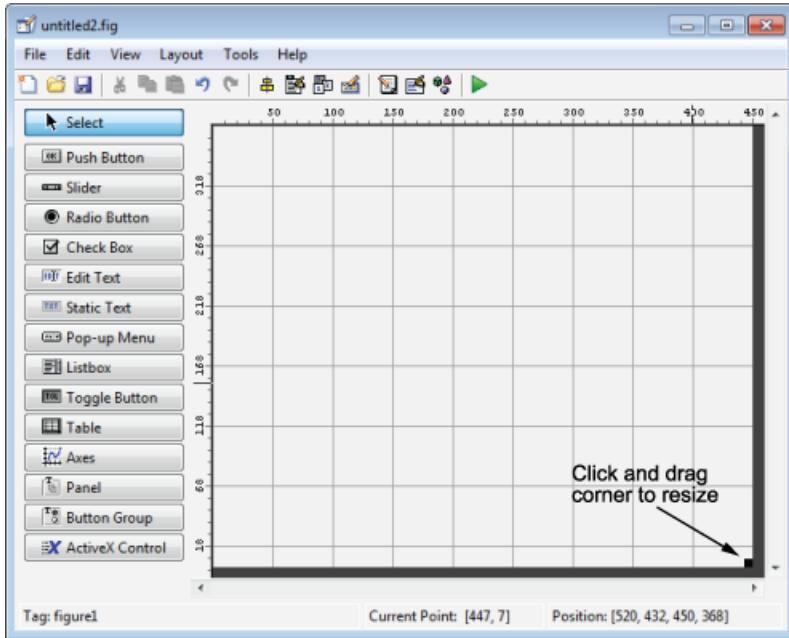
- 1 Open a new blank GUI in GUIDE by typing `guide` at the MATLAB prompt.



- 2 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

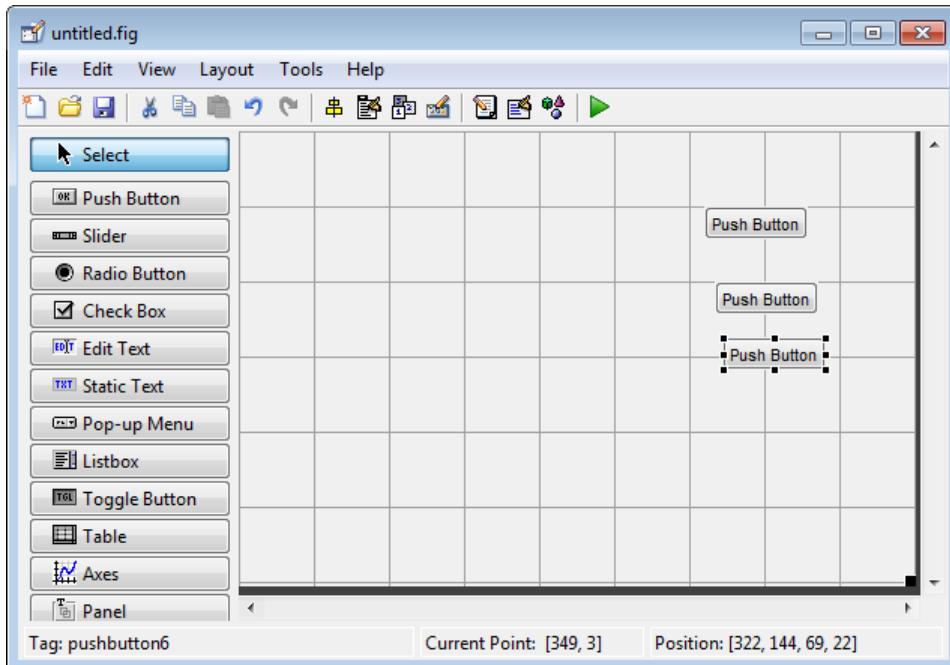
Set the size of the window by resizing the grid area in the Layout Editor. Click the lower-right corner and drag it until the canvas is approximately 3 inches high and 4 inches wide. If necessary, make the canvas larger.



Layout the UI

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

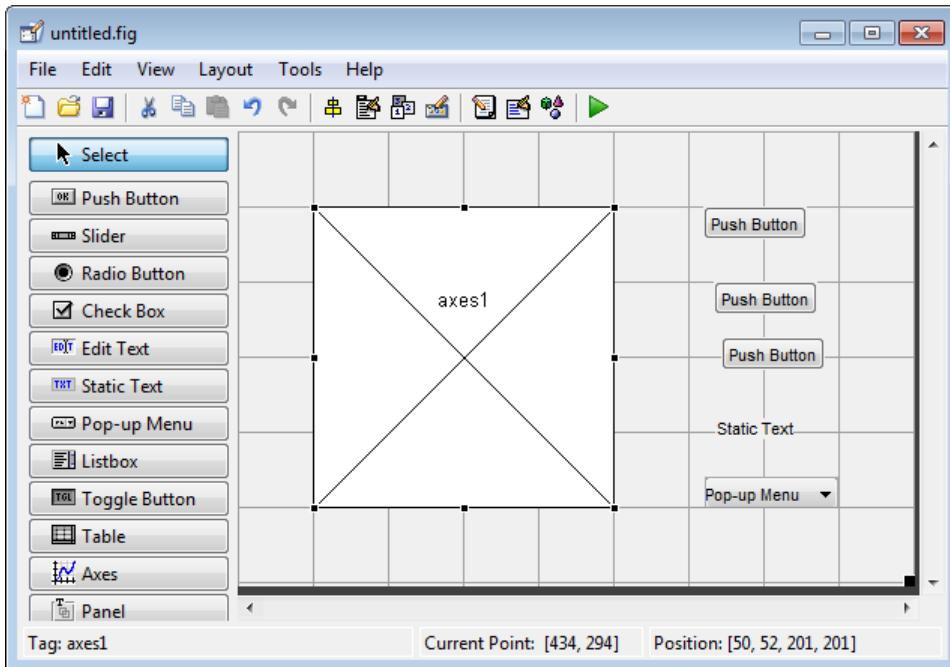
- 1 Add the three push buttons to the UI. Select the push button tool from the component palette at the left side of the Layout Editor and drag it into the layout area. Create three buttons, positioning them approximately as shown in the following figure.



- 2 Add the remaining components to the UI.

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

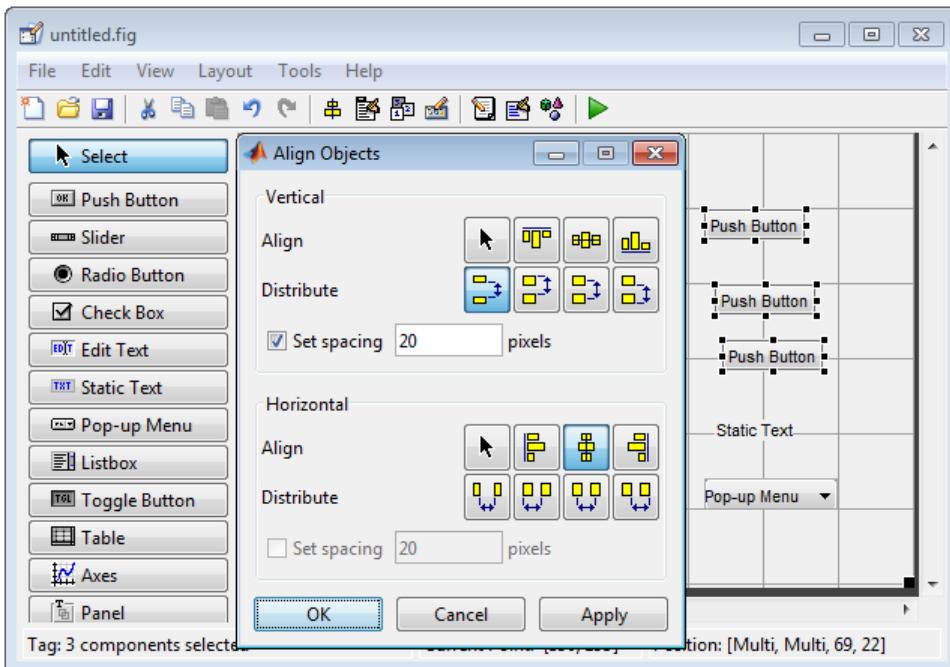
Arrange the components as shown in the following figure. Resize the axes component to approximately 2-by-2 inches.



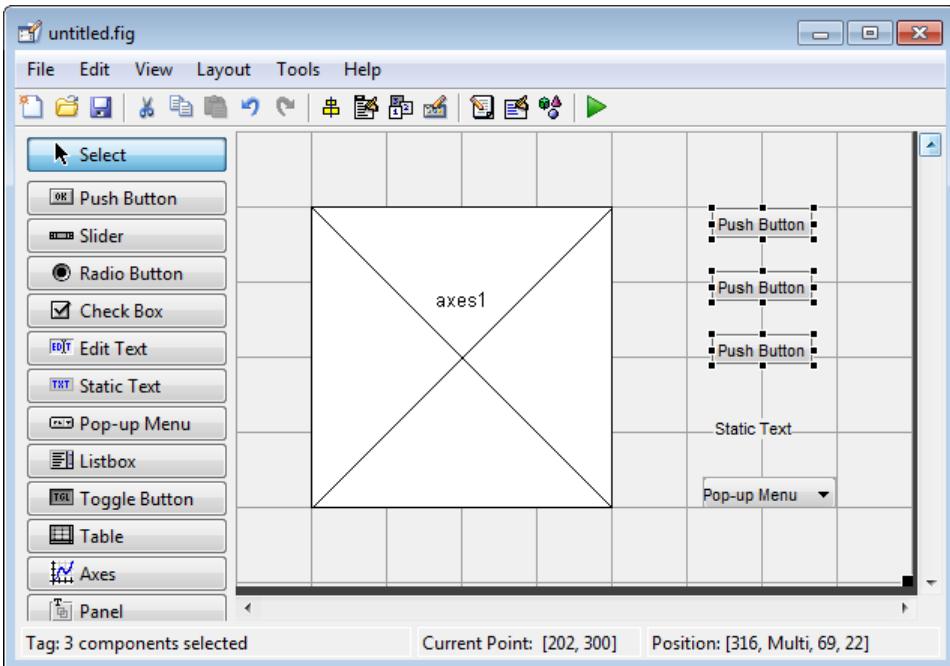
Align the Components

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.



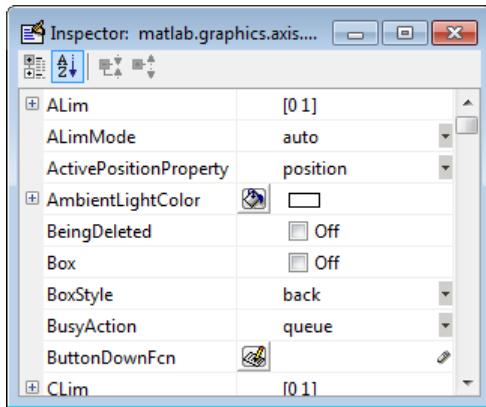
- 4 Click **OK**.



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**.

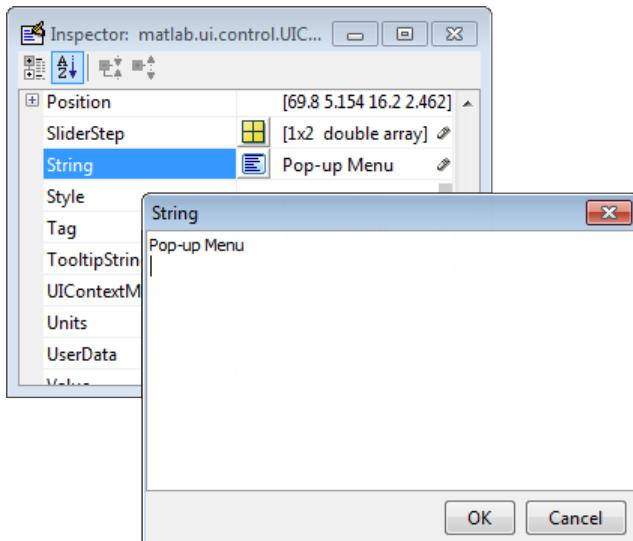


- 5 Click each of the remaining push buttons in turn and repeat steps 3 and 4. Label the middle push button **Mesh**, and the bottom button **Contour**.

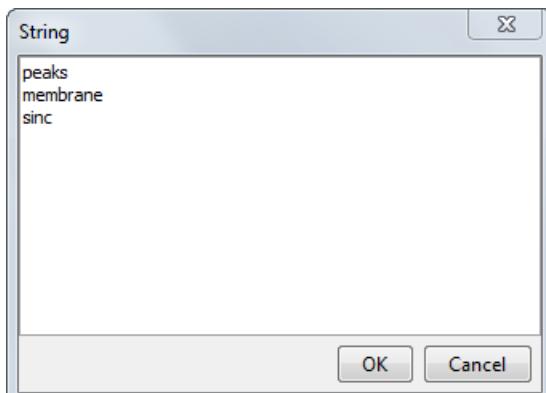
List Pop-Up Menu Items

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.

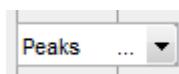


- 3 Replace the existing text with the names of the three data sets: peaks, membrane, and sinc. Press **Enter** to move to the next line.



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

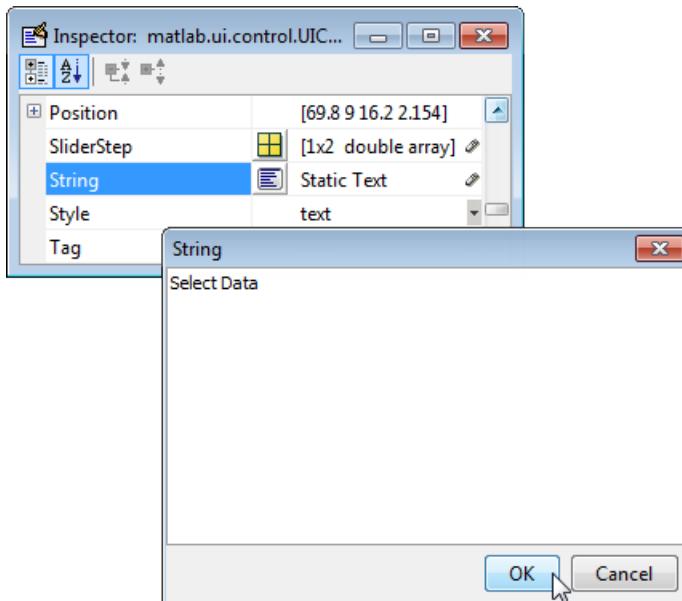
The first item in your list, **peaks**, appears in the pop-up menu in the layout area.



Modify the Static Text

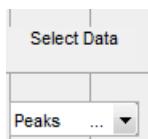
In this UI, the static text serves as a label for the pop-up menu. This section shows you how to change the static text to read **Select Data**.

- 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.



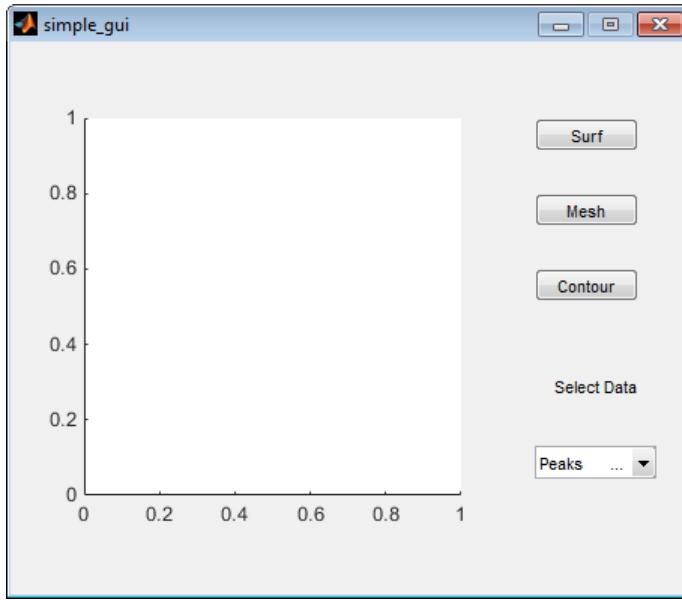
Save the Layout

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 3-9, 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_gui is 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);

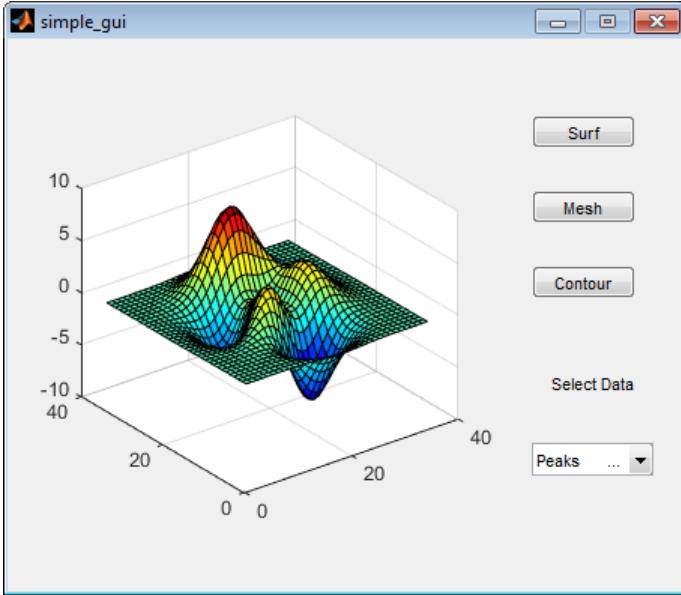
% UIWAIT makes simple_gwait for user response (see UIRESUME)
% uiwait(handles.figure1);
```

- 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.

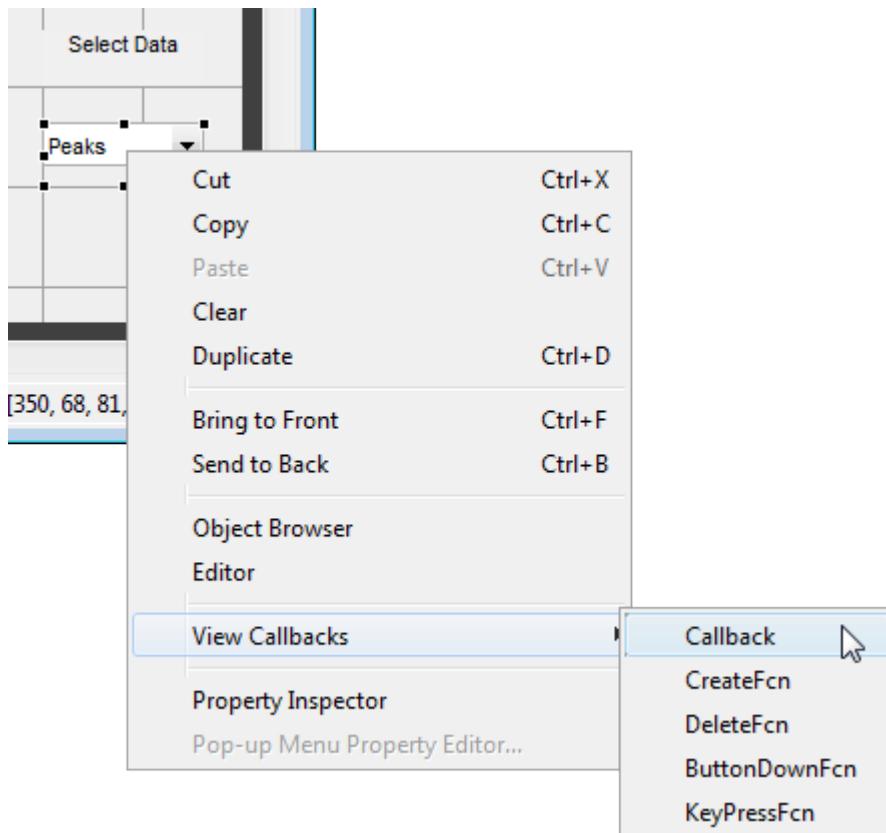
The last two lines create a current data value and set it to `peaks`, and then display the surf plot for `peaks`. The following figure shows how the app looks when it first displays.



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.

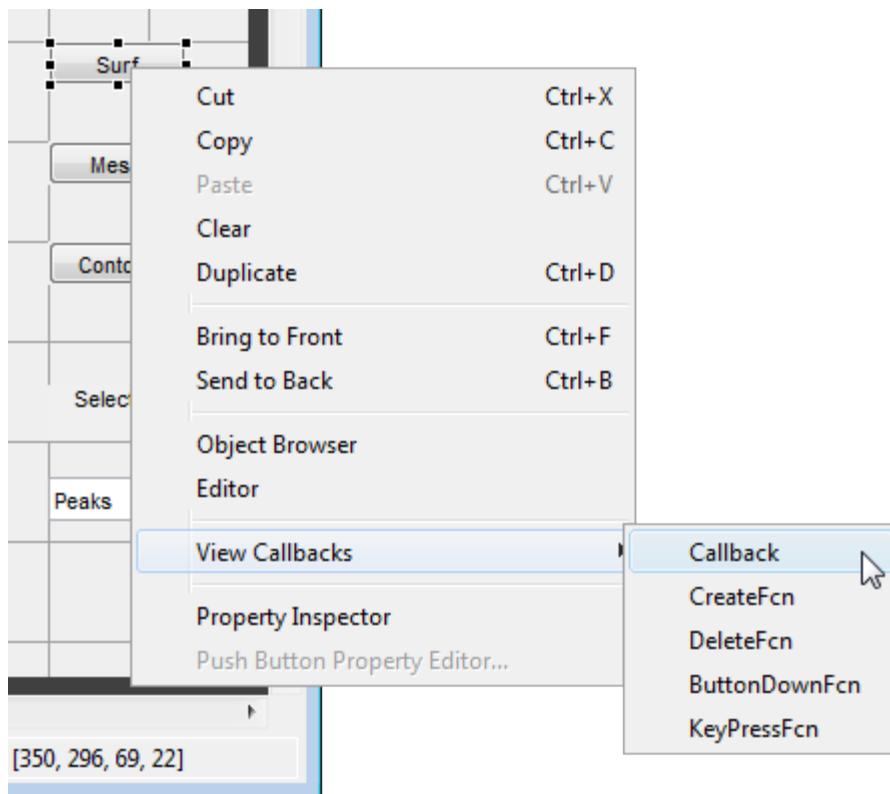
```
% Determine the selected data set.
str = get(hObject, 'String');
val = get(hObject, 'Value');
% Set current data to the selected data set.
switch str{val};
case 'peaks' % User selects peaks.
    handles.current_data = handles.peaks;
case 'membrane' % User selects membrane.
    handles.current_data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

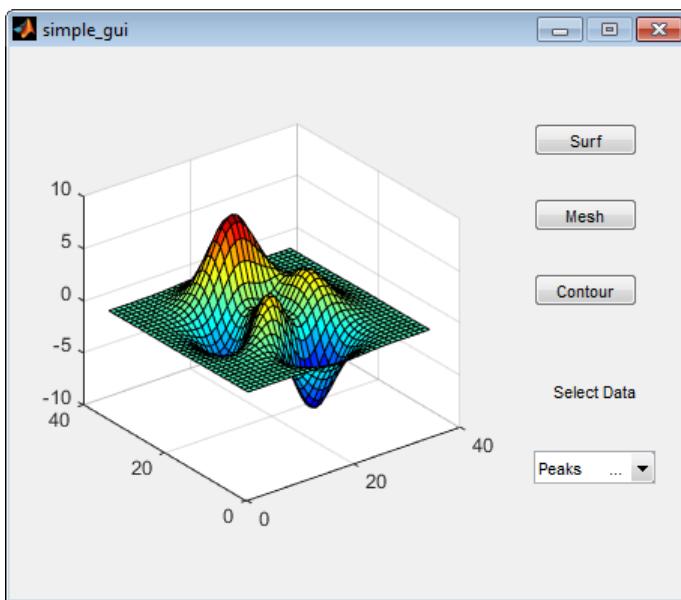
```
% Display contour plot of the currently selected data.
contour(handles.current_data);
```

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

Run the App

In “Code the Behavior of the App” on page 3-10, 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 and Run a Simple App Using App Designer” on page 4-2
- “Create and Run a Simple figure-Based App” on page 2-2

Files Generated by GUIDE

In this section...

- "Code Files and FIG-Files" on page 3-16
- "Code File Structure" on page 3-16
- "Adding Callback Templates to an Existing Code File" on page 3-17
- "About GUIDE-Generated Callbacks" on page 3-17

Note The GUIDE environment will be removed in a future release. After GUIDE is removed, existing GUIDE apps will continue to run in MATLAB but they will not be editable in GUIDE.

To continue editing an existing GUIDE app, see "GUIDE Migration Strategies" on page 4-5 for information on how to help maintain compatibility of the app with future MATLAB releases. To create new apps interactively, "Develop Apps Using App Designer" instead.

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 are stored in the FIG-file. When you program the app, your code is stored in the corresponding code file.

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 19-4.

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 19-4 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 19-13.

See Also

Related Examples

- “Write Callbacks in GUIDE” on page 19-2

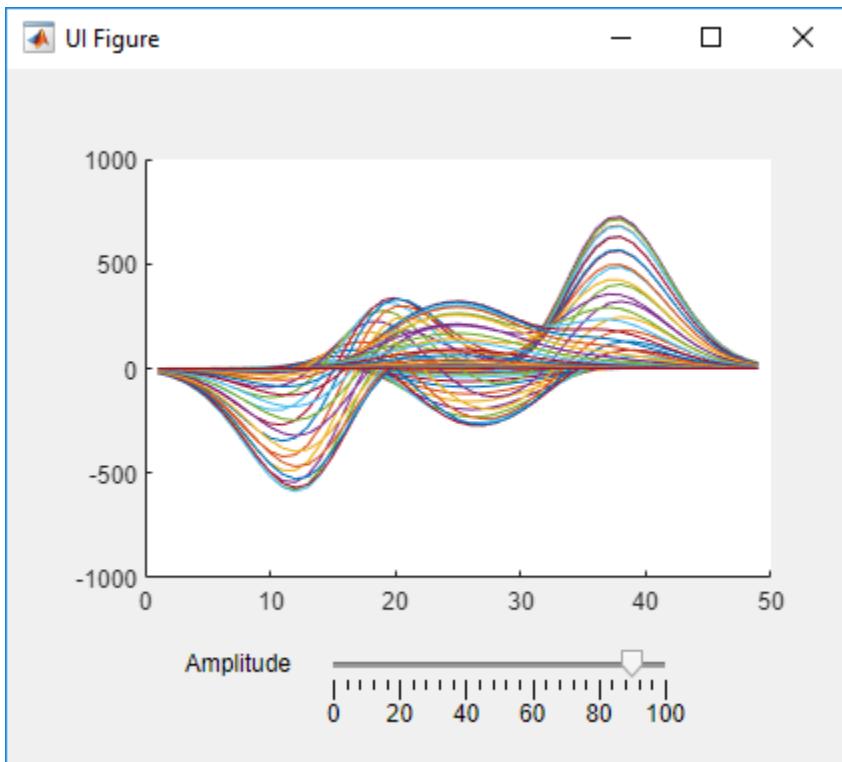
App Designer

App Designer Basics

- “Create and Run a Simple App Using App Designer” on page 4-2
- “GUIDE Migration Strategies” on page 4-5
- “Display Graphics in App Designer” on page 4-12
- “App Designer Preferences” on page 4-17

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 here.



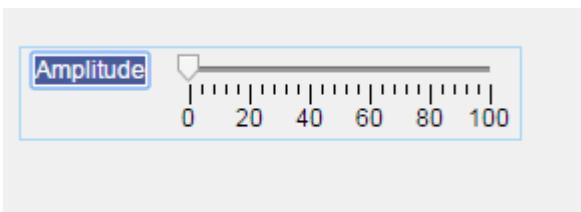
Run the Tutorial

To run the tutorial in App Designer, open the App Designer Start Page and expand the **Examples: General** section. Then, select **Interactive Tutorial**.

Tutorial Steps for Creating the App

Perform the following steps in App Designer.

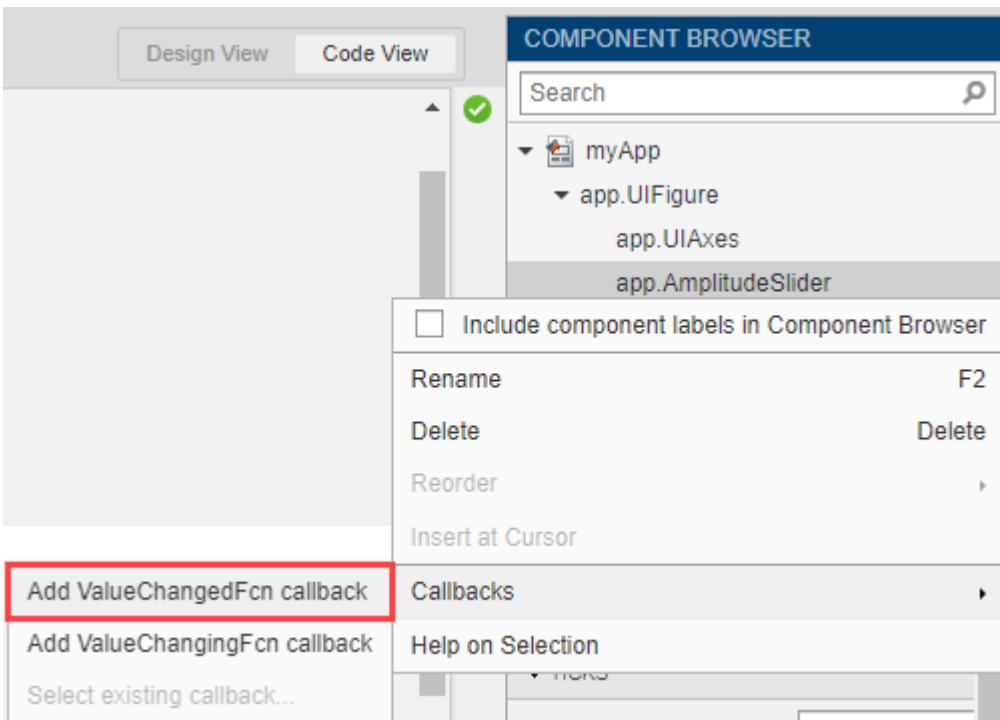
- 1 Drag an **Axes** component from the **Component Library** onto the canvas.
- 2 Drag a **Slider** component from the **Component Library** onto the canvas. Place it below the axes, as in the preceding image.
- 3 Replace the slider label text. Double-click the label and replace the word **Slider**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

- “Manage Code in App Designer Code View” on page 7-2
- “Write Callbacks in App Designer” on page 7-15
- “Display Graphics in App Designer” on page 4-12

GUIDE Migration Strategies

In R2019b, MathWorks announced that GUIDE, the original drag-and-drop environment for building apps in MATLAB, will be removed in a future release. After GUIDE is removed, existing GUIDE apps (GUIs) will continue to run in MATLAB, and app program files will still be editable if you need to change the behavior of an app.

To continue editing the *layout* of an existing GUIDE app and help maintain its compatibility with future MATLAB releases, you must use one of the suggested migration strategies listed in this table.

App Development Needs	Migration Strategy	How to Migrate
Occasional editing	Export your app to a single MATLAB file to manage your app layout and code using MATLAB functions.	Open the app in GUIDE and select File > Export to MATLAB-file . In the GUIDE Removal Options dialog, click Export .
Ongoing development	Migrate your app to App Designer.	Open the app in GUIDE and select File > Migrate to App Designer . In the GUIDE Removal Options dialog, click Migrate .

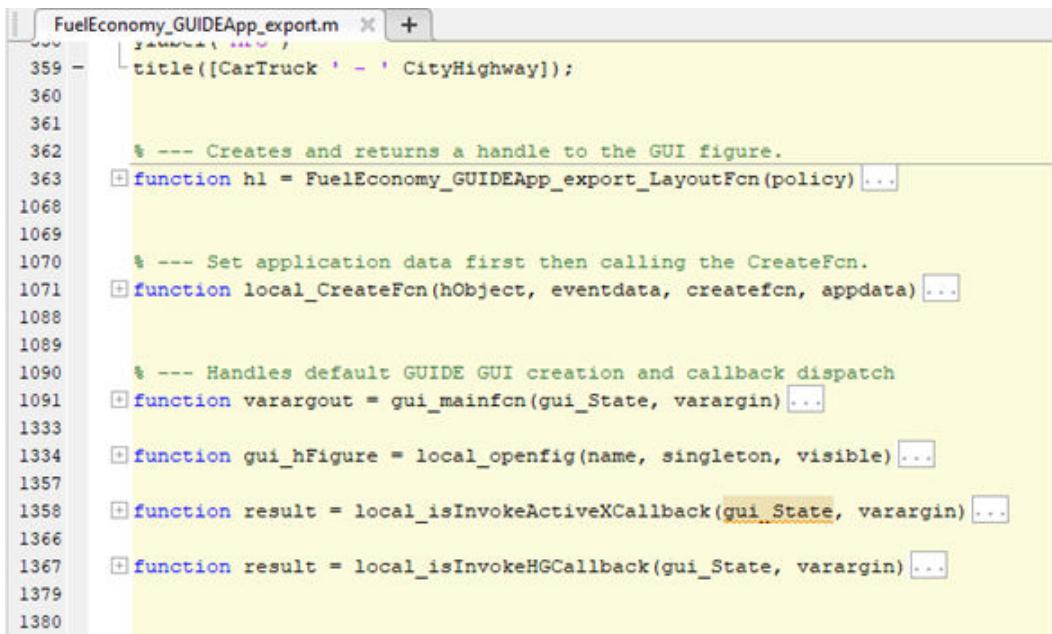
Export GUIDE App to MATLAB File

Exporting a GUIDE app converts it into a programmatic app by recreating the GUIDE FIG and program files together in a single MATLAB program file.

Use this option if you plan to:

- Make minor changes to the layout or behavior of your app.
- Develop your app programmatically, not interactively.

To export your app, open it in GUIDE and select **File > Export to MATLAB-file**, or right-click the FIG file in the MATLAB **Current Folder** browser and select **Export to MATLAB-file**. This brings up the GUIDE Removal Options dialog. Verify that the correct FIG file is selected and then click **Export**. MATLAB creates a program file with `_export` appended to the file name. The new file contains your original callback code plus auto-generated functions that handle the creation and layout of the app. An example of these added functions is shown here.



```

FuelEconomy_GUIDEApp_export.m
359 -     title([CarTruck ' - ' CityHighway]);
360
361
362     % ---- Creates and returns a handle to the GUI figure.
363     function hl = FuelEconomy_GUIDEApp_export_LayoutFcn(policy) ...
1068
1069
1070     % ---- Set application data first then calling the CreateFcn.
1071     function local_CreateFcn(hObject, eventdata, createfcn, appdata) ...
1088
1089
1090     % ---- Handles default GUIDE GUI creation and callback dispatch
1091     function varargout = gui_mainfcn(gui_State, varargin) ...
1333
1334     function gui_hFigure = local_openfig(name, singleton, visible) ...
1357
1358     function result = local_isInvokeActiveXCallback(gui_State, varargin) ...
1366
1367     function result = local_isInvokeHGCallback(gui_State, varargin) ...
1379
1380

```

Migrate GUIDE App to App Designer

Migrating your GUIDE app to App Designer allows you to continue developing the layout of your app interactively. It also allows you to take advantage of features like an enhanced UI component set and auto-reflow options to make your app responsive to changes in screen size. And it gives you the ability to create and share your app as a web app (requires MATLAB Compiler™).

The GUIDE to App Designer Migration Tool for MATLAB was first released in R2018a to ease the conversion process. It is available through the Add-On Explorer in the MATLAB desktop or through File Exchange on MATLAB Central™.

Starting in R2020a, the migration tool has significant improvements that drastically reduce the time, and the number of manual code updates, required to get your app running in App Designer. For details about these enhancements, see “Callback Code” on page 4-7.

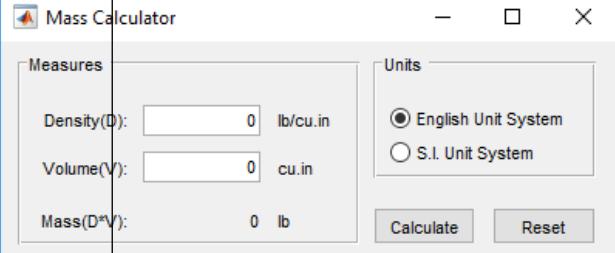
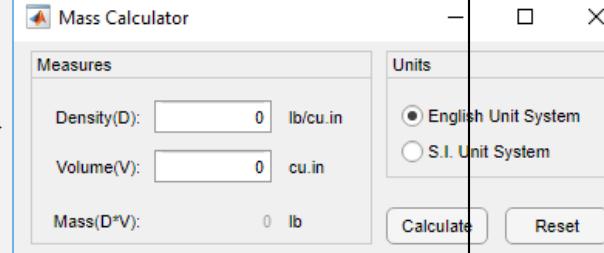
Use this option for GUIDE apps that require significant or ongoing feature development.

There are several ways to migrate your app, depending on which environment you begin in.

- Open the GUIDE Removal Options dialog by opening your app in GUIDE and selecting **File > Migrate to App Designer**, or right-clicking the FIG file in the MATLAB **Current Folder** browser and selecting **Migrate to App Designer**.
 - If you do not already have the GUIDE to App Designer Migration Tool installed, click **Install Support Package**. This opens the Add-On Explorer, where you can install the migration tool. Once you have installed the tool, reopen the GUIDE Removal Options dialog.
 - Once you have installed the GUIDE to App Designer Migration Tool, choose the correct FIG file and then click **Migrate**. The app migrates and automatically opens in App Designer.
- From App Designer, open any app and go to the **Designer** tab. In the **File** section, click **Open > Open GUIDE to App Designer Migration Tool**.

Features of the Migration Tool

The migration tool helps you convert your apps by reading in a GUIDE FIG file and automatically generating the App Designer equivalent components and layout in an MLAPP file. Your GUIDE callback code and other user-defined functions are copied into the MLAPP file. This semi-automated code conversion also creates a migration report that suggests actions for any manual code updates that are needed. Some features of the tool are described in this table.

Migration Tool Features	Description
File Conversion	Read in a GUIDE FIG file and associated code and then generate an App Designer MLAPP file. The App Designer file name takes the form <i>guideFileName_App.mlapp</i> .
Components and App Layout	Convert components and property configurations to App Designer equivalents, and preserve the layout of the app. GUIDE App:  App Designer App: 
Callback Code	Retain a copy of the GUIDE callback code and user-defined functions in the MLAPP file.
Tutorial	Step through the changes made to your migrated app.
Migration Report	Summarize the actions successfully completed by the migration tool. List any limitations or unsupported functionality, specific to your app, with suggested actions if available.

Callback Code

In order to make your GUIDE-style callback code compatible with the App Designer UI components in your app, the migration tool uses a function called `convertToGUIDECallbackArguments`. This function converts App Designer callback arguments into the GUIDE-style callback arguments that your code requires. The `convertToGUIDECallbackArguments` function is added to the beginning of each migrated callback function. It takes the App Designer callback arguments `app` and `event` and returns the GUIDE-style callback arguments `hObject`, `eventdata`, and `handles`. For example:

```

205 % Button pushed function: showcode
206 function showcode_Callback(app, event)
207 % Create GUIDE-style callback args - Added by Migration Tool
208 [hObject, eventdata, handles] = convertToGUIDECallbackArguments(app, event);
209
210 % hObject    handle to showcode (see GCBO)
211 % eventdata   reserved - to be defined in a future version of MATLAB
212 % handles    structure with handles and user data (see GUIDATA)
213 open(handles.scriptPath);
214 end

```

`(hObject` is the handle of the object whose callback is executing. For components from your GUIDE app that were `UIControl` or `ButtonGroup` objects, `(hObject` is a handle to a `UIControlPropertiesConverter` or `ButtonGroupPropertiesConverter` object. These objects are created to make your GUIDE-style code work in your App Designer callback functions.

`eventdata` is usually empty, but can be a structure containing specific information about the callback event.

`handles` is a structure that contains the migrated child components of the UI figure that have a 'Tag' property value set. Child components that were `UIControl` objects in your GUIDE app are `UIControlPropertiesConverter` objects in the migrated app. Similarly, child `ButtonGroup` objects are `ButtonGroupPropertiesConverter` objects in the migrated app.

The `UIControlPropertiesConverter` and `ButtonGroupPropertiesConverter` objects act like adapters between the GUIDE-style code and the App Designer components and callbacks. A `UIControlPropertiesConverter` object is created for each component in your GUIDE app that was a `UIControl` object. These converter objects are associated with an App Designer UI component in your migrated app. The converter object has the same properties and values as the original `UIControl` from your GUIDE app, but it applies them to its associated App Designer UI component.

Similarly, for `ButtonGroup` objects from GUIDE, a `ButtonGroupPropertiesConverter` object is created in App Designer. This object makes it possible to set the `SelectedObject` property to a `UIControlPropertiesConverter` object so that button group `SelectionChangedFcn` callback logic will function.

Special Considerations

There are some circumstances that require you to take extra steps before or after you migrate your app. This table lists common scenarios and coding patterns that require extra steps or manual code updates. This is not intended to be a comprehensive list.

GUIDE App Feature	Description	Suggested Actions
Multiwindow apps (that is, two or more apps that share data)	Multiwindow apps require each app to be migrated separately. Migrated app file names are appended with <code>_App</code> . Calls to these apps from other apps must be updated.	Migrate each app separately. In the calling app, update the name of the app that is being called to the new file name.

GUIDE App Feature	Description	Suggested Actions
Radio buttons and radio button callbacks	The migration tool does not migrate radio buttons that are not parented to a radio button group, or callback functions for individual radio buttons.	Create a button group in App Designer and add radio buttons to it. To execute behavior when radio button selection is changed, create a <code>SelectionChangedFcn</code> callback function for the button group. For more information, see <code>uiradiobutton</code> and <code>ButtonGroup</code> Properties.
<code>gco</code> , <code>movie</code> , <code>uistack</code> , and <code>clf</code> with the 'reset' argument	Calling these functions in App Designer is not supported.	Determine if this functionality is critical to your app before migrating. There is no workaround in App Designer.
<code>findobj</code> , <code>findall</code> , and <code>gcbo</code>	Using <code>findobj</code> , <code>findall</code> , or <code>gcbo</code> to reference components and set properties can error. <code>UIControl</code> objects are migrated to the equivalent App Designer UI component. To access and set properties on these migrated components, you must set it on the <code>UIControlPropertiesConverter</code> objects. Or, you can update your code to use its associated App Designer component, properties, and values.	Reference components using the <code>handles</code> structure instead, or update your code to use the associated App Designer component, properties, and values.
<code>nargin</code> and <code>nargchk</code>	Helper functions are migrated to app methods and have <code>app</code> as an additional input argument. This can cause incorrect <code>nargin</code> or <code>nargchk</code> logic.	Increment check values by 1.

GUIDE App Feature	Description	Suggested Actions
OutputFcn(varargout) and Figure output	<p>There is no equivalent functionality in App Designer.</p> <p>When you instantiate a migrated App Designer app, the output is always the app object, not the Figure object.</p>	<p>If your OutputFcn function includes initialization code that is critical to your app, then add it to the end of the OpeningFcn instead.</p> <p>If your OutputFcn function specifies output to be assigned to the workspace when you instantiate the app, such as the Figure object, then you need to create a function that instantiates the app. For example:</p> <pre>function out = MyGUIDEApp(varargin) app = MyMigratedApp(varargin{:}); out = app.UIFigure; end</pre>

If your GUIDE app integrates third-party components using functions like `actxcontrol`, see Recommendations for MATLAB Apps Using Java and ActiveX.

Aids for Adding New Features or Fully Adopting App Designer Code Style

App Designer and GUIDE have different code structures, callback syntaxes, and techniques for accessing UI components and sharing data. Understanding these differences is useful if you plan to add new App Designer features to your migrated app or want to update it to use App Designer code style and conventions. This table summarizes some of these 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 axes for displaying plots.</p> <p>All MATLAB graphics functions are supported. There is no need to specify the target axes.</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 axes for displaying plots.</p> <p>Most MATLAB graphics functions are supported.</p>	"Display Graphics in App Designer" on page 4-12
Using Components	GUIDE creates most components with the <code>uicontrol</code> function. Fewer components are available.	App Designer creates each UI component with its own dedicated function. More components are available, including Tree, Gauge, TabGroup, and DatePicker.	"App Building Components" on page 5-2

Difference	GUIDE	App Designer	More Information
Accessing Component Properties	GUIDE uses <code>set</code> and <code>get</code> to access component properties, and uses <code>handles</code> to specify a component. For example, <code>name = get(handles.Fig, 'Name')</code>	App Designer supports <code>set</code> and <code>get</code> , but encourages the use of dot notation to access component properties, and uses <code>app</code> to specify a component. For example, <code>name = app.UIFigure.Name</code>	"Write Callbacks in App Designer" on page 7-15
Managing App Code	The code is defined as a main function that can call local functions. All code is editable.	The code is defined as a MATLAB class. Only callbacks, helper functions, and custom properties are editable.	"Manage Code in App Designer Code View" on page 7-2
Writing Callbacks	Required callback input arguments are <code>handles</code> , <code>hObject</code> , and <code> eventdata</code> . For example, <code>myCallback(hObject, eventdata, handles)</code>	Required callback input arguments are <code>app</code> and <code>event</code> . For example, <code>myCallback(app, event)</code>	"Write Callbacks in App Designer" on page 7-15
Sharing Data	To store and share data between callbacks and functions, use the <code>UserData</code> property, the <code>handles</code> structure, or the <code>guidata</code> , <code>setappdata</code> , or <code>getappdata</code> function. For example, <code>handles.currSelection = selection; guidata(hObject, handles);</code>	To store and share data between callbacks and functions, use custom properties to create variables. For example, <code>app.currSelection = selection</code>	"Share Data Within App Designer Apps" on page 7-23

See Also

Related Examples

- “Create and Run a Simple App Using App Designer” on page 4-2
- “Display Graphics in App Designer” on page 4-12
- “Ways to Build Apps” on page 1-2

Display Graphics in App Designer

In this section...

- "App Designer Graphics Overview" on page 4-12
- "Display Graphics on Existing Axes" on page 4-12
- "Display Graphics in Container" on page 4-13
- "Create Axes Programmatically" on page 4-13
- "Use Functions With No Target Argument" on page 4-14
- "Use Functions That Don't Support Automatic Resizing" on page 4-14
- "Unsupported Functionality" on page 4-15

App Designer Graphics Overview

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 this argument. The reason is that, in most contexts, MATLAB defaults to using the `gcf` or `gca` functions to get the target object for an operation. But these functions depend on the `HandleVisibility` property of the parent figure being '`on`', and the `HandleVisibility` property of App Designer figures is set to '`off`' by default. This means that `gcf` and `gca` do not work as normal. As a result, omitting the argument for a target axes or parent object can produce unexpected results.

Depending on the graphics function you call, you might need to specify:

- A `UIAxes` component on the canvas
- A parent container in your app
- An axes component that you create programmatically in your app code

There are a number of ways to specify the target component for a graphics function. Some examples of the most common syntaxes are given below. To determine the correct target and syntax in your context, see the documentation for the specific graphics function you are using.

Display Graphics on Existing Axes

The most common way to display graphics in App Designer is to specify a `UIAxes` object on the App Designer canvas as the graphics function target. When you drag an axes component from the **Component Library** onto the canvas, this creates a `UIAxes` object in your app. The default name for an App Designer axes object is `app.UIAxes`. To determine or change the name of a specific axes on your canvas, select the axes component. Its name is listed and can be edited in the **Component Browser**.

Specify Axes as First Argument

Many graphics functions have an optional first input argument to specify the target axes object. For example, both the `plot` function and the `hold` function take a target axes object in this way. To plot two lines on a set of axes on the canvas, specify the name of the axes object as the first argument to each function you call.

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

Specify Axes as Name-Value Argument

Some graphics functions require the target axes object to be specified as a name-value argument. For example, when you call the `imshow` and `triplot` functions, specify the axes object to display on using the `'Parent'` name-value argument. This code displays an image on an existing set of axes on your canvas:

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

Display Graphics in Container

Some graphics functions display in a container component, such as a figure, panel, or grid layout, instead of an axes object. For example, the `heatmap` function has an optional first argument for specifying the container that the chart will display in.

Every App Designer app has a figure object, by default named `app.UIFigure`, that is a container for the components that make up the main app window. Specify `app.UIFigure` as the parent container argument to display graphics in the main app window. For example, to create a heat map in your app, use this syntax:

```
h = heatmap(app.UIFigure, rand(10));
```

To further organize and compartmentalize graphics that take a parent container input argument, drag a container component such as a panel, tab, or grid layout from the **Component Library** onto the canvas. Determine the name of the component by selecting it and viewing its name in the **Component Browser**. You can then specify this container as the parent when you call the graphics function.

Other commonly used graphics functions that take a parent container as input include `annotation`, `geobubble`, `parallelplot`, `scatterhistogram`, `stackedplot`, and `wordcloud`.

Create Axes Programmatically

Some graphics functions plot data on specialized axes. For example, functions that plot polar data must do so on a `PolarAxes` object. Unlike `UIAxes` objects, which you can add to your app from the **Component Library**, you must add specialized axes to your app *programmatically* in your code. To create an axes object programmatically, create a `StartupFcn` callback for your app. Within it, call the appropriate graphics function and specify a parent container in your app as the target.

Plot on Polar Axes

Functions such as `polarplot`, `polarhistogram`, and `polarscatter` take a polar axes object as a target. Create a polar axes programmatically by calling the `polaraxes` function. For example, to plot a polar equation in a panel, first drag a panel component from the **Component Library** onto your canvas. In the code for your app, create the polar axes object by calling the `polaraxes` function and specifying the panel as the parent container. Then, plot your equation with the `polarplot` function, specifying the polar axes as the target axes.

```
theta = 0:0.01:2*pi;
rho = sin(2*theta).*cos(2*theta);
```

```
pax = polaraxes(app.Panel);
polarplot(pax,theta,rho)
```

Plot on Geographic Axes

Functions such as `geoplot`, `geoscatte`r, and `geodensityplot` take a geographic axes object as a target. Create a geographic axes programmatically by calling the `geoaxes` function. For example, to plot geographic data in a panel, use the following code:

```
latSeattle = 47 + 37/60;
lonSeattle = -(122 + 20/60);
gx = geoaxes(app.Panel);
geoplot(gx,latSeattle,lonSeattle)
```

Create Tiled Chart Layout

To tile multiple charts using the `tiledlayout` function, create a tiled chart layout in a panel and programmatically create axes in it using the `nexttile` function. Return the axes object from the `nexttile` function and use it to specify the axes for your charts or plots.

```
t = tiledlayout(app.Panel,2,1);
[X,Y,Z] = peaks(20)

% Tile 1
ax1 = nexttile(t);
surf(ax1,X,Y,Z)

% Tile 2
ax2 = nexttile(t);
contour(ax2,X,Y,Z)
```

Use Functions With No Target Argument

Some graphics functions, such as `ginput` and `gtext`, do not have an argument for specifying a target. As a result, you must set the `HandleVisibility` property of the App Designer figure to '`callback`' or '`on`' before calling these functions. After you call these functions, you can set the `HandleVisibility` property back to '`off`'. For example, this code shows how to define a callback that allows you to identify the coordinates of two points using the `ginput` function.

```
function pushButtonCallback(app,event)
    app.UIFigure.HandleVisibility = 'callback';
    ginput(2)
    app.UIFigure.HandleVisibility = 'off';
end
```

Use Functions That Don't Support Automatic Resizing

App Designer figures are resizable by default. This means that when you run an app and resize the figure window, components in the figure are automatically resized and repositioned to fit. However, some graphics functions do not support automatic resizing. To use these functions in App Designer, create a panel in which to display the output of the function and set the `AutoResizeChildren` property of the panel to '`off`'. You can set this property in the **Inspector** tab of the **Component Browser** or in your code.

For example, the `subplot` function does not support automatic resizing. To use this function in your app:

- 1 Drag a panel component from the **Component Library** onto your canvas.
- 2 Set the `AutoResizeChildren` property of the panel to '`off`'.
- 3 Specify the panel as the parent container using the '`Parent`' name-value argument when you call `subplot`. Also, specify an output argument to store the axes.
- 4 Call the plotting function with the axes as the first input argument.

```
app.Panel.AutoResizeChildren = 'off';
ax1 = subplot(1,2,1,'Parent',app.Panel);
ax2 = subplot(1,2,2,'Parent',app.Panel);
plot(ax1,[1 2 3 4])
plot(ax2,[10 9 4 7])
```

Other commonly used functions that do not support automatic resizing include `pareto` and `plotmatrix`.

For more information about managing resize behavior, see “Alternatives to Default Auto-Resize Behaviors” on page 6-11.

Unsupported Functionality

As of R2021a, some graphics functionality is not supported in App Designer. This table lists the unsupported functionality that is most relevant to app building workflows.

Category	Not Supported
Animation	The <code>movie</code> function is not supported.
Retrieving and Saving Data	<p>These functions are not supported: <code>hgexport</code>, <code>hglload</code>, <code>hgsave</code>, <code>save</code>, <code>load</code>, <code>savefig</code>, <code>openfig</code>, and <code>saveas</code>.</p> <p>Instead of the <code>saveas</code> function, use the <code>exportapp</code> function to save the content of an app window. To save plots in an app, use the <code>exportgraphics</code> or <code>copygraphics</code> functions.</p> <p>Figures created programmatically with <code>uifigure</code> do support the <code>save</code>, <code>load</code>, <code>savefig</code>, and <code>openfig</code> functions.</p>
Utilities	<p>The <code>clf</code> function with the '<code>reset</code>' argument and the <code>print</code> function are not supported.</p> <p>Instead of the <code>print</code> function, use the <code>exportapp</code> function to save the content of an app window. To save plots in an app, use the <code>exportgraphics</code> or <code>copygraphics</code> functions.</p>
Web Apps	<p>If you are using App Designer to create a deployed web app (requires MATLAB Compiler), additional graphics limitations apply.</p> <p>For more information, see “Limitations and Unsupported Functionality” (MATLAB Compiler).</p>

See Also

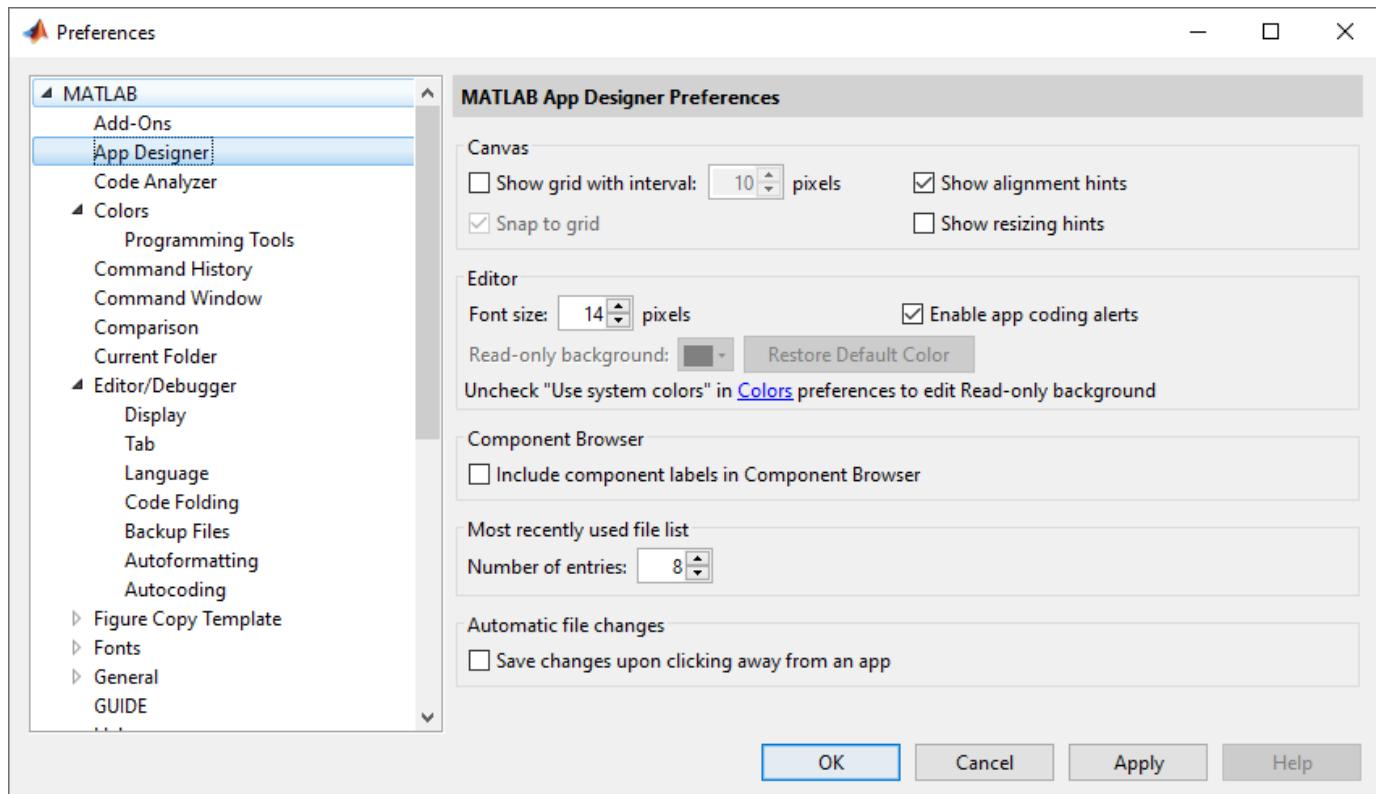
[UI Figure](#) | [UIAxes](#)

More About

- “Create Polar Axes Programmatically in an App” on page 8-7
- “App Building Components” on page 5-2
- “Add UI Components to App Designer Programmatically” on page 5-20
- “Manage Resizable Apps in App Designer” on page 6-11

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
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.

Option	Description
Font Size	You can change the font size that displays in App Designer Code View . The font size can range from 14–48 pixels. The default font size is 14 pixels.
Enable app coding alerts	When selected, App Designer flags coding problems in the editor as you write code.
Read-only background	You can change the background color of the uneditable code sections in App Designer Code View . To change the background color, clear the Use system colors check box in the MATLAB Colors preferences. Then, select a new color from the color drop-down in the App Designer preferences. The default background color is gray.
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.
Automatic File Changes	When selected, App Designer automatically saves changes to an app when you click away from it to switch between apps or to bring another window into focus. If an app has not already been saved at least once, autosave has no effect.

To customize the App Designer canvas and **Component Browser** settings programmatically, use `matlab.appdesigner.Settings`.

See Also

Related Examples

- “Lay Out Apps in App Designer Design View” on page 6-2
- “Manage Code in App Designer Code View” on page 7-2

Component Choices and Customizations

- “App Building Components” on page 5-2
- “Table Array Data Types in App Designer Apps” on page 5-15
- “Add UI Components to App Designer Programmatically” on page 5-20
- “Create HTML File That Can Trigger or Respond to Data Changes” on page 5-23

App Building Components

App Designer and UI figures support a large set of components for designing modern, full-featured applications. The tables below list the components that are available.

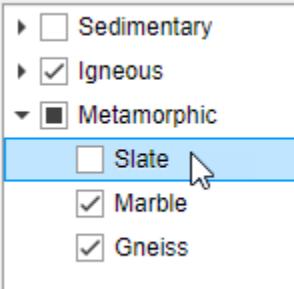
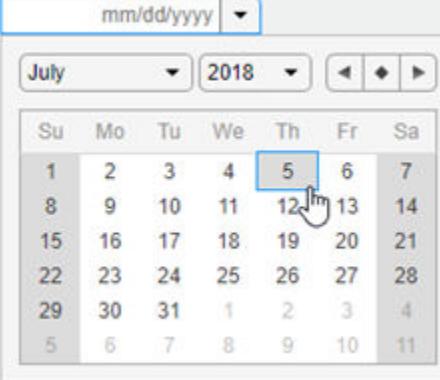
- Common Components — Include components that respond to interactions, such as buttons, sliders, drop-down lists, and trees.
- Axes — Include axes to create plots for data visualization and exploration.
- Containers and Figure Tools — Include panels and tabs for grouping components, as well as menu bars.
- Instrumentation Components — Include gauges and lamps for visualizing status, as well as knobs and switches for selecting input parameters.
- Extensible Components — Include custom UI components that you author. Interface with third-party libraries to display content like widgets or data visualizations.
- Toolbox Components — Include toolbox authored UI components. Requires additional toolbox license and installation.

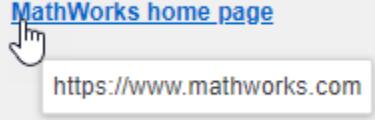
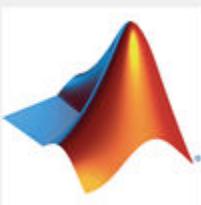
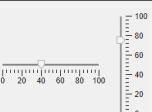
All components are available programmatically. Most UI components are also available in the App Designer **Component Library** for you to drag and drop onto the canvas. To add components to an App Designer app that are not available in the **Component Library**, or that you want to add dynamically to the running app, see “Add UI Components to App Designer Programmatically” on page 5-20.

When calling graphics functions in App Designer, the workflow is slightly different than you typically use at the MATLAB command line. For more information about how to call graphics functions in App Designer, see “Display Graphics in App Designer” on page 4-12.

Common Components

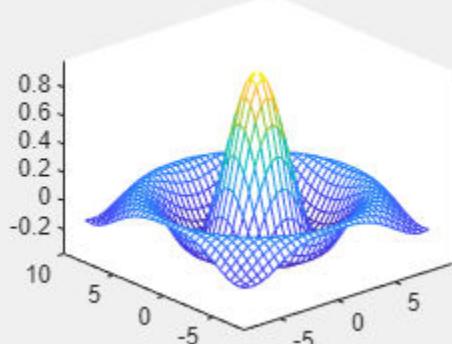
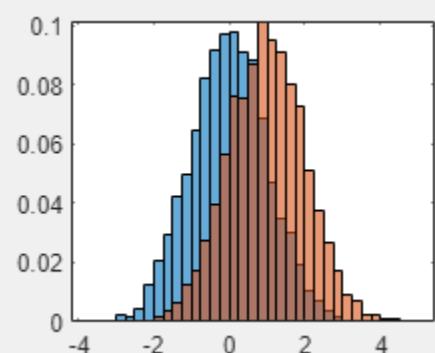
Component Information	Example
Button	
CheckBox	<input type="checkbox"/> Remove Outliers <input checked="" type="checkbox"/> Add Trendline

Component Information	Example
CheckBoxTree Properties TreeNode	
DatePicker Properties	
DropDown	
NumericEditField	
EditField	

Component Information	Example
Hyperlink Properties	
Image Properties	
Label	Select an Option
ListBox	
ButtonGroup RadioButton	
Slider	
Spinner	

Component Information	Example
StateButton	
Table	
TextArea	
ButtonGroup ToggleButton	
Tree TreeNode	

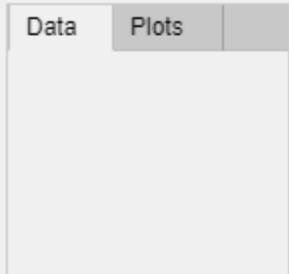
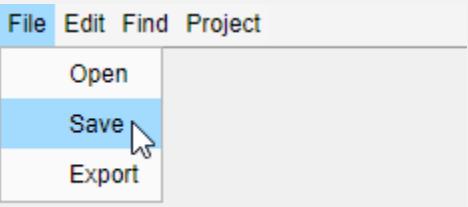
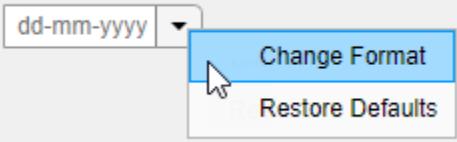
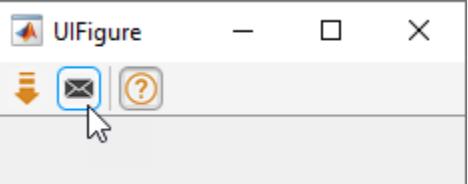
Axes

Axes Information	Example
UIAxes	
Axes Properties This object can be added programmatically only.	
GeographicAxes Properties This object can be added programmatically only.	

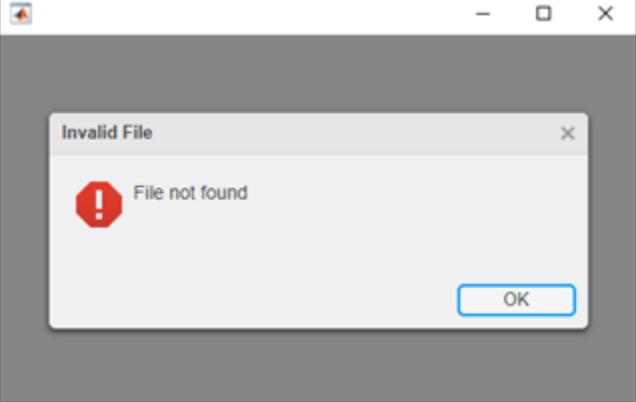
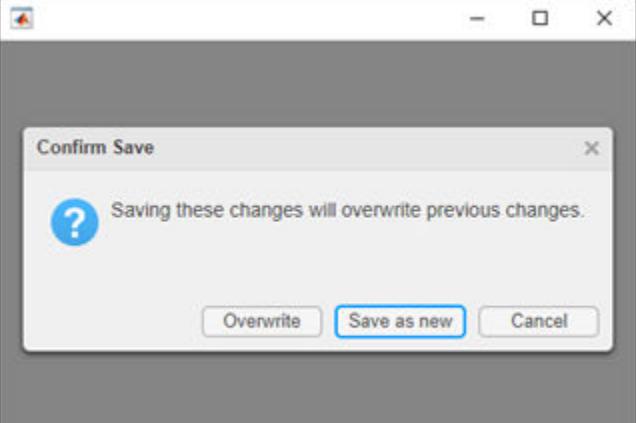
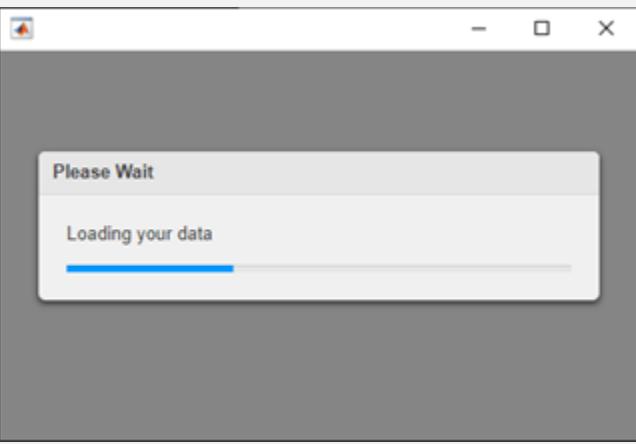
Axes Information	Example
PolarAxes Properties This object can be added programmatically only.	

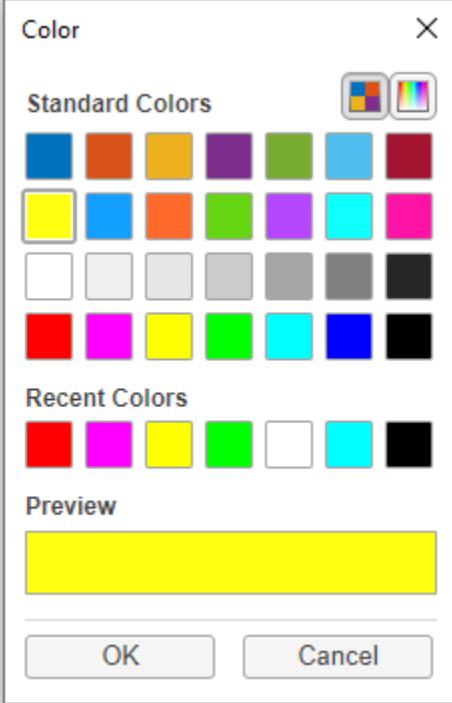
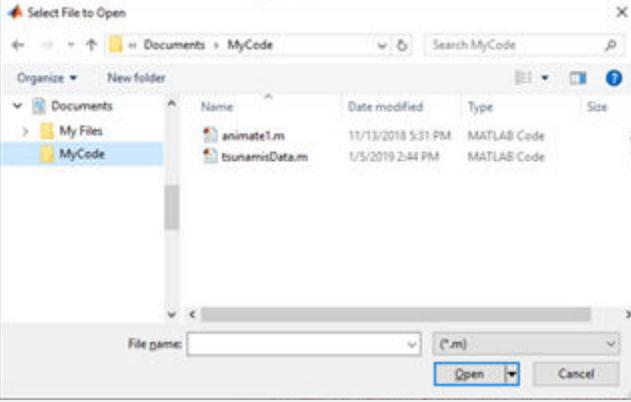
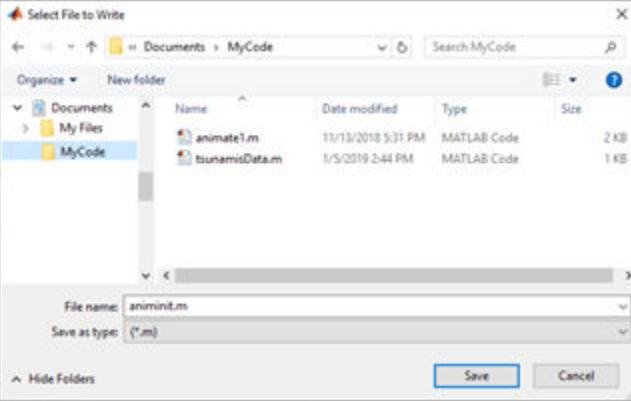
Containers and Figure Tools

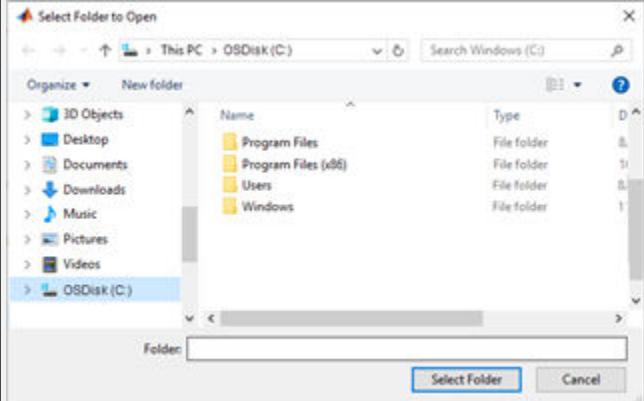
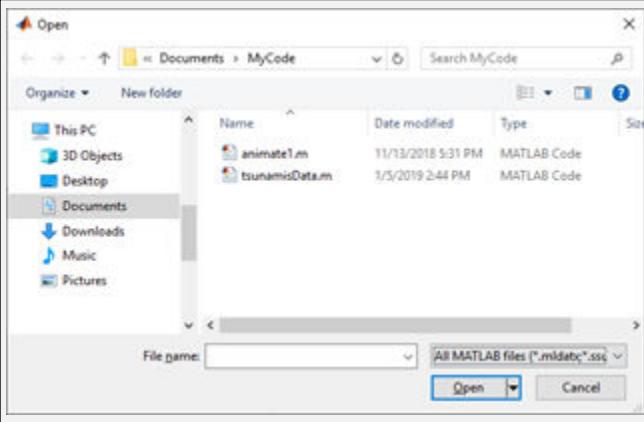
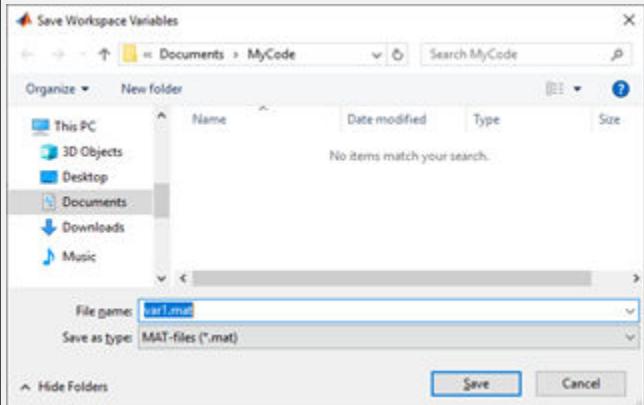
Container Information	Example
GridLayout Properties	
Panel	

Container Information	Example
TabGroup Tab	
Menu	
ContextMenu Properties	
Toolbar Properties PushTool Properties ToggleTool Properties	

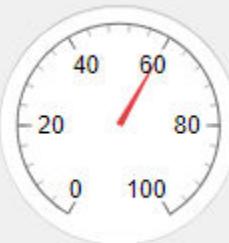
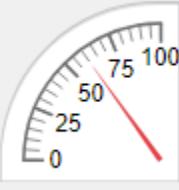
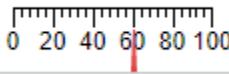
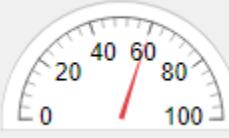
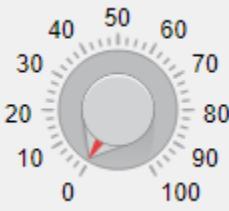
Dialogs and Notifications

Dialog Information	Example
<code>uialert</code> This object can be added programmatically only.	
<code>uiconfirm</code> This object can be added programmatically only.	
<code>uiprogressdlg</code> This object can be added programmatically only.	

Dialog Information	Example
uisetcolor This object can be added programmatically only.	
uigetfile This object can be added programmatically only.	
uiputfile This object can be added programmatically only.	

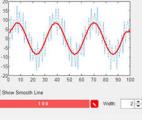
Dialog Information	Example
uigetdir This object can be added programmatically only.	
uiopen This object can be added programmatically only.	
uisave This object can be added programmatically only.	

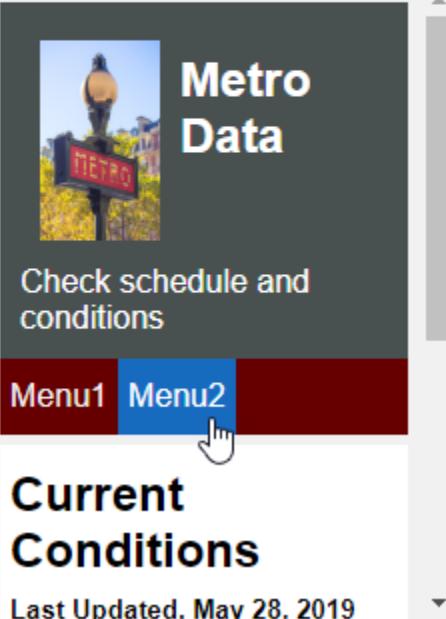
Instrumentation

Component Information	Example
Gauge	
NinetyDegreeGauge	
LinearGauge	
SemicircularGauge	
Knob	
DiscreteKnob	

Component Information	Example
Lamp	
Switch	
RockerSwitch	
ToggleSwitch	

Extensible Components

Component Information	Example
<p><code>matlab.ui.componentcontainer.ComponentContainer</code> Class</p> <p><code>matlab.graphics.chartcontainer.ChartContainer</code> Class</p>	

Component Information	Example
HTML Properties	<p>Use the <code>uihtml</code> function to:</p> <ul style="list-style-type: none"> • Display HTML markup • Embed HTML, JavaScript®, or CSS content 

Toolbox Components

Apps created in App Designer or with the `uifigure` function support Aerospace Toolbox components. For more information, see “Flight Instruments” (Aerospace Toolbox). To use toolbox components, a valid license and installation of the associated toolbox is required.

See Also

Related Examples

- “Ways to Build Apps” on page 1-2
- “Display Graphics in App Designer” on page 4-12
- “Create and Run a Simple App Using App Designer” on page 4-2
- “Add UI Components to App Designer Programmatically” on page 5-20
- “Create and Run a Simple figure-Based App” on page 2-2

Table Array Data Types in App Designer Apps

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. Only App Designer apps and figures created with the `uifigure` function support 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.

```
fig = uifigure;
tdata = table([true; true; false]);
uit = uitable(fig,'Data',tdata);
uit.Position(3) = 130;
uit.RowName = 'numbered';
```

	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.

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

Color	Wavelength
Blue	400
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).

```
fig = 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(fig, 'Data', tdata);
```

Date	Measurement
01/17/2016	10
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
17/01/2016	10
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).

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

Time	Distance
00:01:20	10.5100
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
80 sec	10.5100
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.

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

Array	Structure
3×3×3 double	1×1 struct
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`.

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

ROI	RGB		
1	128	122	16
2	0	66	155
3	255	0	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.

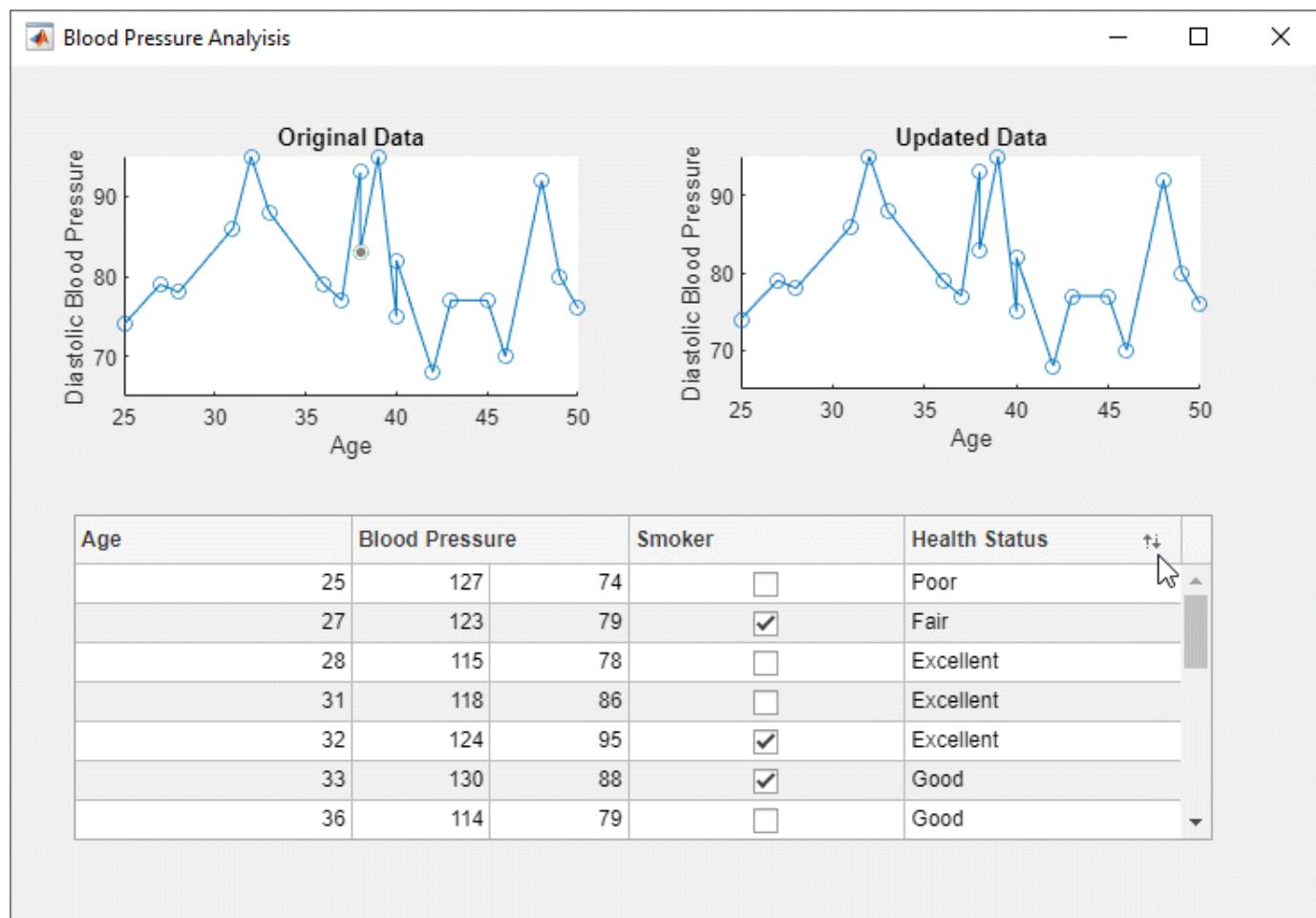
```
fig = uifigure;
sz = categorical([1; 3; 4; 2],1:3,{'Large','Medium','Small'});
num = [NaN; 10; 12; 15];
tdata = table(sz,num,'VariableNames',{'Size','Number'});
uit = uitable(fig,'Data',tdata,'ColumnEditable',true);
```

Size	Number
Large	NaN
Small	10
<undefined>	12
Medium	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. One plot displays the original table data. The other plot initially shows the same table data, and then updates when the user edits a value or sorts a column in the `Table` UI component.



See Also

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

Related Examples

- “Write Callbacks in App Designer” on page 7-15
- “Reuse Code Using Helper Functions” on page 7-20

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 UI component functions, see “[uifigure-Based Apps](#)”). 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

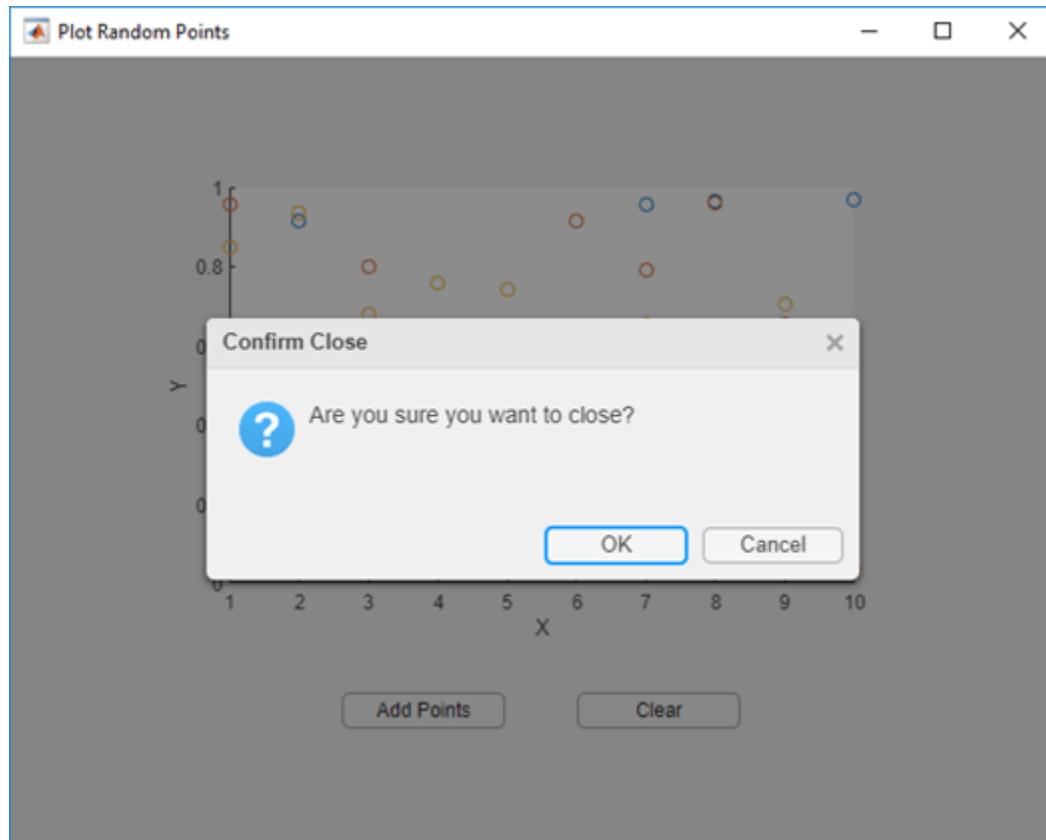
To assign this callback to a component, specify the component's callback property as cell array. The first element in the cell array must be the function handle. Subsequent elements must be the additional input values. For example:

```
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.

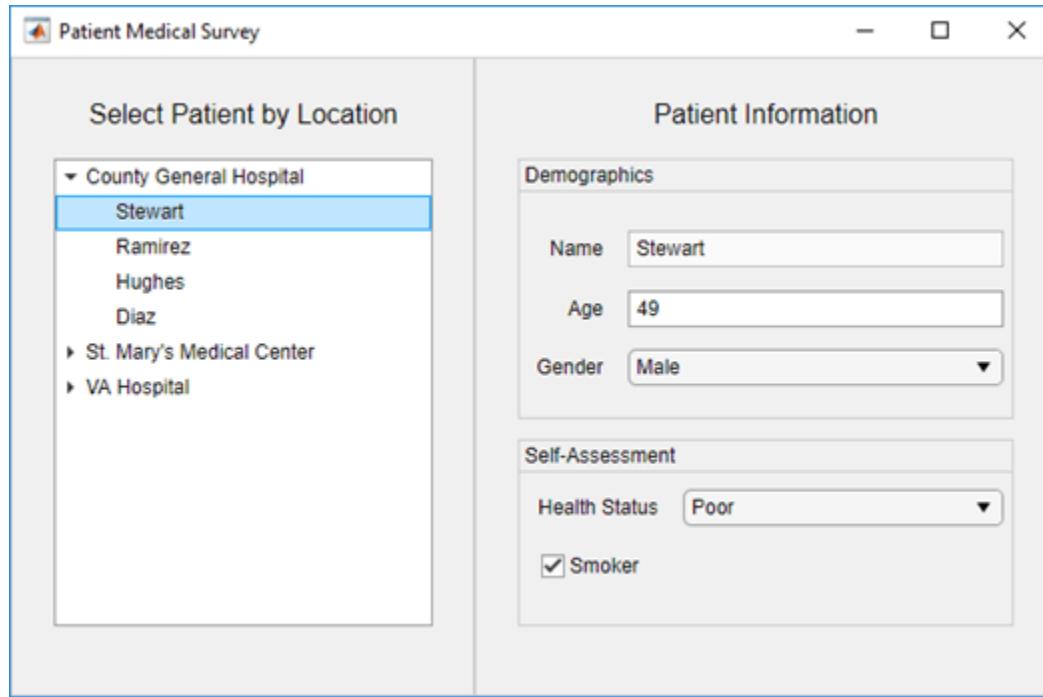
When the user clicks the window's close button (**X**), a dialog box displays to confirm that the user wants to close the app. When the user dismisses the dialog box, the `CloseFcn` callback executes.



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 7-15
- “Reuse Code Using Helper Functions” on page 7-20

Create HTML File That Can Trigger or Respond to Data Changes

You can include third-party visualizations or widgets in your app by creating an HTML UI component in it that displays HTML, JavaScript, or CSS content from an HTML file. When you add an HTML UI component to your app, to enable the component to set data or respond to data changes between MATLAB and JavaScript, include a `setup` function in your HTML file. Within the `setup` function you can connect the HTML content to the HTML UI component in MATLAB.

Include Setup Function in Your HTML File

To connect the MATLAB HTML UI component in your app to the content in your HTML file, create a `setup` function that defines and initializes a local `htmlComponent` JavaScript object. The HTML UI component in MATLAB and the `htmlComponent` JavaScript object have `Data` properties that synchronize with each other. The `setup` function is required if you want to set data from either MATLAB or JavaScript and respond to changes in data that occur on the opposite side.

The `setup` function is called when one of these events happens:

- The HTML UI component is created in the figure and the content has fully loaded.
- The `HTMLSource` property changes to a new value.

The `setup` function is called only if it is defined. The `htmlComponent` JavaScript object is accessible only from within the `setup` function.

The `htmlComponent` JavaScript object also has `addEventListener` and `removeEventListener` properties. Use these properties to listen for `DataChanged` events from MATLAB. The event data from `DataChanged` events provides the source `htmlComponent` JavaScript object with the old and new data. For more information about the `addEventListener` and `removeEventListener` methods, see `EventTarget.addEventListener()` and `EventTarget.removeEventListener()` on Mozilla® MDN web docs.

Sample HTML File

This example shows an HTML file with the required `setup` function for enabling MATLAB and JavaScript to respond to data changes from one another.

Within the `setup` function, once the `htmlComponent` JavaScript object has been initialized, you define the behavior of the component. For example:

- Get the initial value of the `Data` property from the HTML UI component in MATLAB.
- Initialize your HTML or JavaScript by updating DOM elements or JavaScript widgets.
- Listen for "DataChanged" events in MATLAB and code a JavaScript response. For example, you can update your HTML or JavaScript with the new data that triggered the event.
- Create a function that sets the `Data` property of the `htmlComponent` JavaScript object and triggers a `DataChangedFcn` callback in MATLAB.

After the `setup` function, you can use your third-party JavaScript libraries as the library documentation recommends.

Here is a sample HTML file, `sampleHTMLFile.html`.

```
<!DOCTYPE html>
<html>
<head>
    <script type="text/javascript">

        function setup(htmlComponent) {
            console.log("Setup called:", htmlComponent);

            // Get the initial 'Data' value from MATLAB

            var initialValue = htmlComponent.Data;
            console.log("Initial MATLAB Data", initialValue);

            // Initialize your HTML or JavaScript here
            // Update things like DOM elements or JavaScript widgets

            var dom = document.getElementById("Content");
            dom.textContent = initialValue;

            // Code response to data changes in MATLAB

            htmlComponent.addEventListener("DataChanged", function (event) {
                var changedData = htmlComponent.Data;
                console.log("New Data from MATLAB", changedData);

                // Update your HTML or JavaScript with the new data
                var dom = document.getElementById("Content");
                dom.textContent = changedData;

            });

            // Update 'Data' in MATLAB and trigger
            // the 'DataChangedFcn' callback function

            function updateData(newData) {
                htmlComponent.Data = newData;
                console.log("Changing Data in HTML", newData)
            }
        }
    </script>
</head>

<body>
    <div style="font-family:sans-serif;">
        <span style="font-weight:bold;"> The data from MATLAB will display here:</span><br />
        <div id ="Content"></div>
    </div>

    <!Reference supporting files here>

    <script src=""></script>
    <link rel="stylesheet" type="text/css" href="">
    <link rel="icon" type="image/png" href="">

</body>
```

```
</html>
```

Debug an HTML File

If you create an HTML component that is not working as expected, or if you want to know what your data looks like after conversion is complete between MATLAB and JavaScript, open the HTML file in your system browser. Using your browser Developer Tools (DevTools), you can set breakpoints to test portions of your `setup` function. When you debug your HTML file through the system browser, you must simulate the connection between MATLAB and JavaScript that the `setup` function provides.

Simulate Sending Data from MATLAB to JavaScript

This example shows how to simulate the way MATLAB sends data to JavaScript so that you can debug the HTML file.

Open this example in MATLAB. From the **Current Folder** browser, right-click the file called `sampleHTMLFile.html` and select **Open Outside MATLAB**. The HTML file opens in your system browser.

- 1 In MATLAB, run this code to convert a MATLAB cell array of character vectors to a JSON string. Copy the returned string value to your clipboard.

```
value = {'one'; 'two'; 'three'};
jsontxt = jsonencode(value)

jsontxt =
['"one", "two", "three"]'
```

- 2 In the DevTools of your system browser, open the file to view the code. Create a breakpoint at line 16, where `dom.textContent = initialData;`.
- 3 Open the DevTools console and create the `htmlComponent` JavaScript object. Use the `JSON.parse` method to convert the JSON string you just generated in MATLAB to a JavaScript object and store it in the `Data` property of the `htmlComponent` object.

```
var htmlComponent = {
  Data: JSON.parse('["one", "two", "three"]'), // JSON formatted text from MATLAB data
  addEventListener: function() {console.log("addEventListener called with: ", arguments)}
};
```

- 4 While still in the DevTools console, call the `setup` function. When you resume execution of the `setup` function, the data appears in the HTML page within DevTools.

```
setup(htmlComponent)
```

You can also simulate the "DataChanged" listener callback by JSON encoding and parsing data from MATLAB into your JavaScript code.

Simulate Sending Data from JavaScript to MATLAB

If you want to debug how data is sent from JavaScript to MATLAB, use the `JSON.stringify` method to convert a JavaScript object into a JSON-formatted string. Then, in MATLAB, use the `jsondecode` function to convert that string to MATLAB data.

See Also

Functions

`jsondecode` | `jsonencode` | `uihtml`

Properties

HTML Properties

More About

- “Display HTML Elements Styled by a Cascading Style Sheet” on page 8-14

App Layout

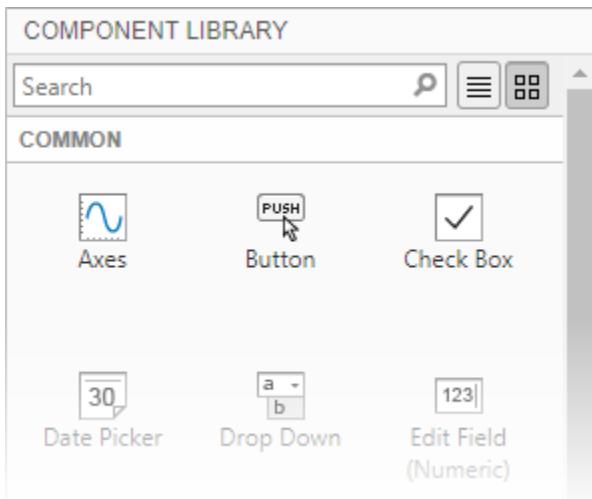
- “Lay Out Apps in App Designer Design View” on page 6-2
- “Manage Resizable Apps in App Designer” on page 6-11
- “Use Grid Layout Managers” on page 6-13
- “Apps with Auto-Reflow” on page 6-16

Lay Out Apps in App Designer Design View

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 various 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, use one of these methods:

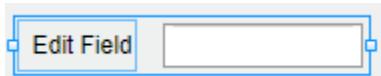
- Drag a component from the **Component Library** and drop it on the canvas.
- Click a component in the **Component Library** and then move your cursor over the canvas. The cursor changes to a crosshair. Click your mouse to add the component to the canvas in its default size, or click and drag to size the component as you add it. Some components can only be added in their default size.



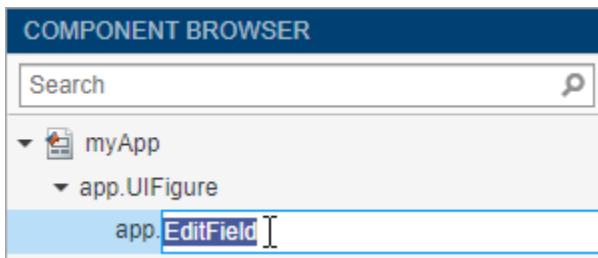
The name of the component appears in the **Component Browser** after you add it to the canvas. You can select components in either the canvas or the **Component Browser**. The selection occurs in both places simultaneously.



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 the canvas.

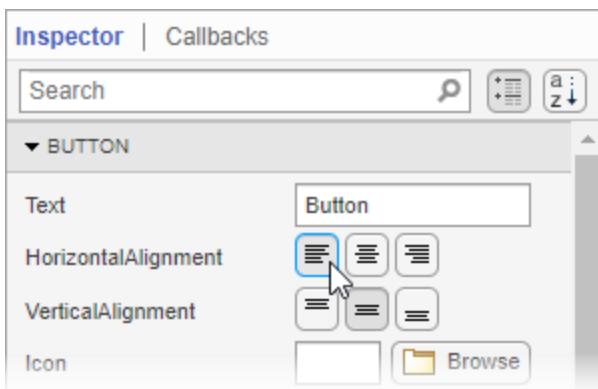


If a component has a label, and you change the label text, the name of the component in the **Component Browser** changes to match that text. You can customize the name of the component by double-clicking it and typing a new name.

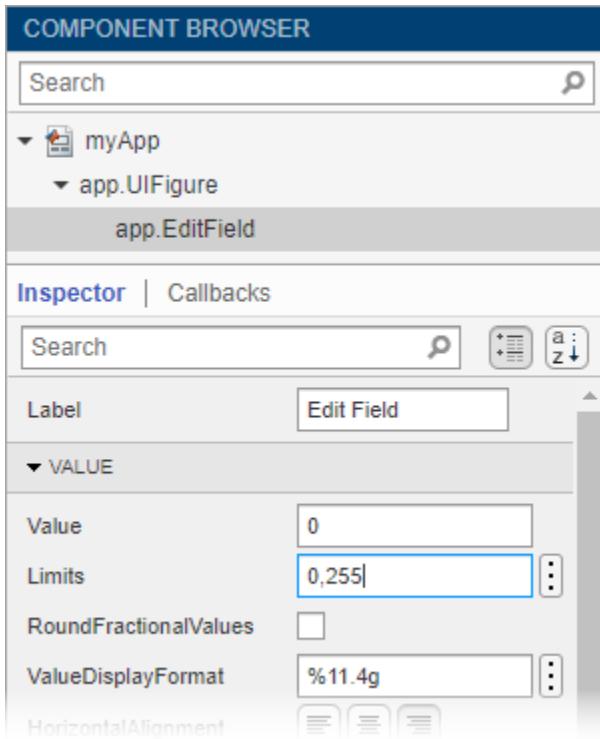


Customize Components

You can customize the appearance of a component by selecting it and then editing its properties in the **Inspector** tab of the **Component Browser**. For example, from this tab you can change the alignment of the text that displays on a button.



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.



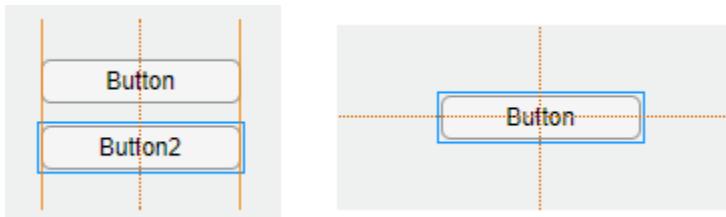
You can edit some properties directly in the canvas by double-clicking the component. For example, you can edit a button label by double-clicking it and typing the desired text. To add multiple lines of text, hold down the **Shift** key and press **Enter**.



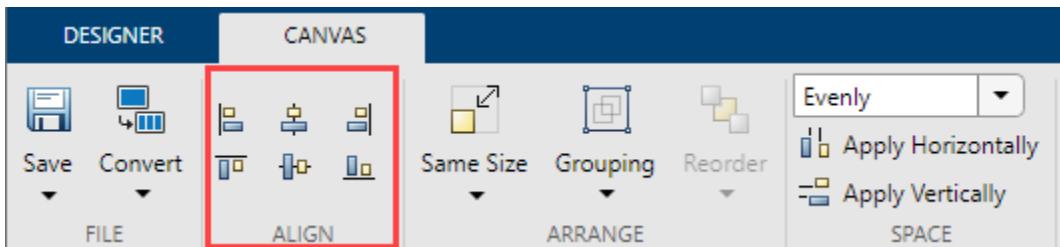
Align and Space Components

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.

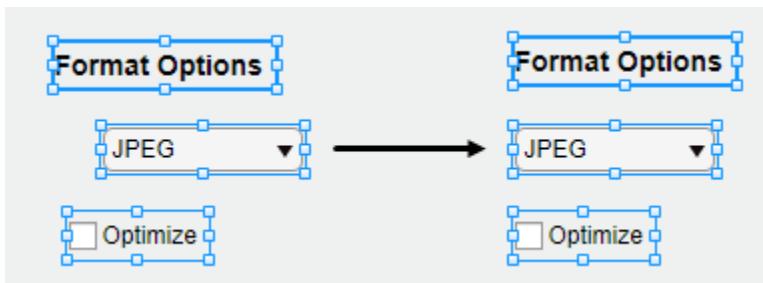
App Designer provides alignment hints to help you align components as you drag them in the canvas. Orange dotted lines passing through the centers of multiple components indicate that their centers are aligned. Orange solid lines at the edges indicate that the edges are aligned. Perpendicular lines indicate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



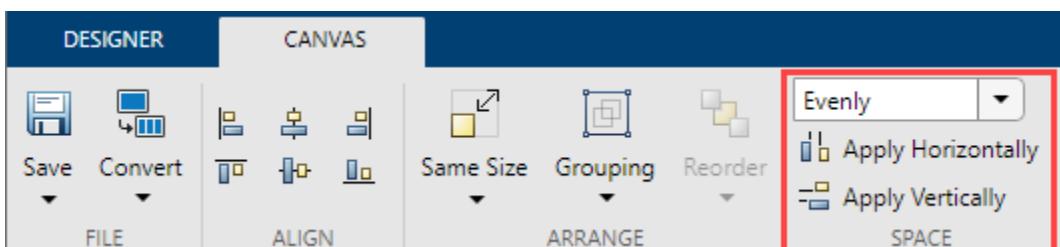
As an alternative to dragging components on the canvas, you can align components using the tools in the **Align** section of the toolbar.



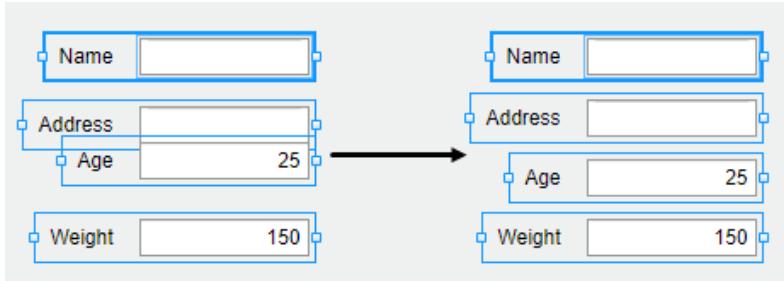
When you use an alignment tool, the selected components align to an anchor component. The anchor component is the last component selected, and it has a thicker selection border than the other components. To select a different anchor, hold down the **Ctrl** or **Shift** key and click the desired component 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 drop-down 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.



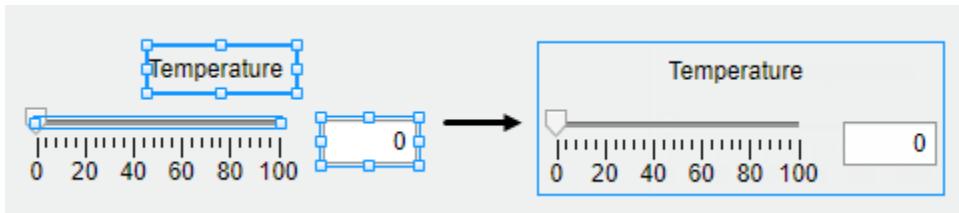
Next, click **Apply Horizontally**  or **Apply Vertically** . For example, select **Evenly** and then click **Apply Vertically**  to distribute the space among a vertical stack of components.



Group Components

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

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



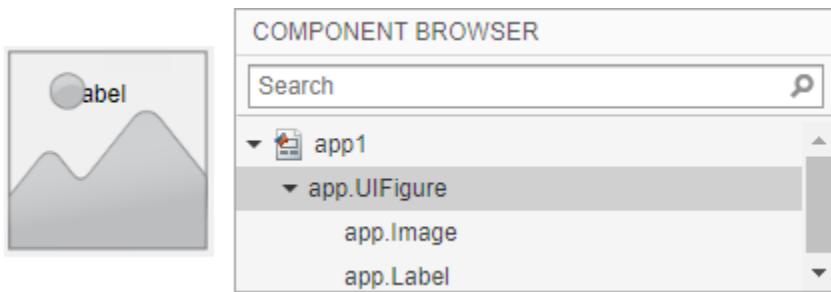
The **Grouping** tool also provides functionality for these common tasks:

- 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**.

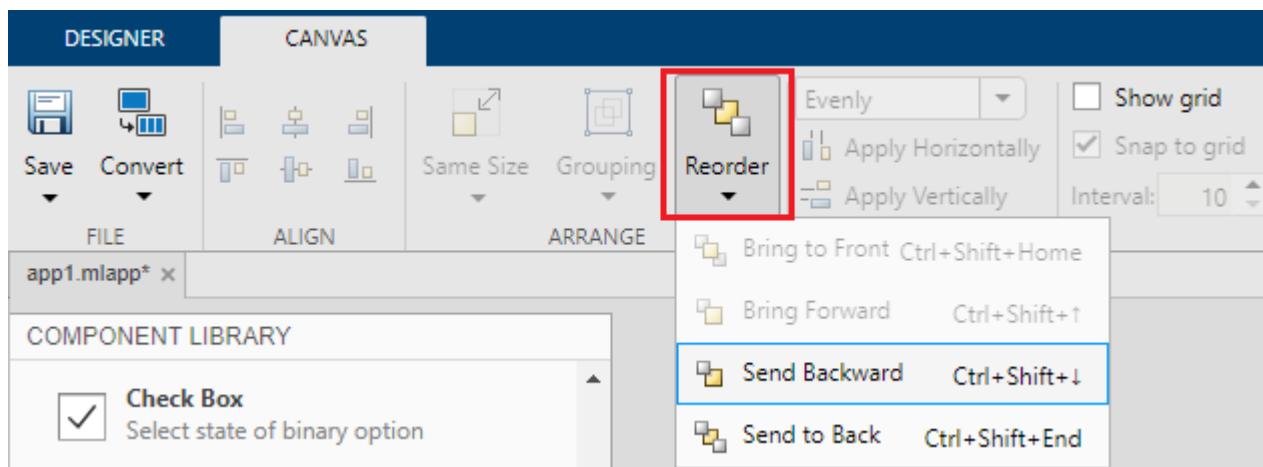
Reorder Components

You can reorder components by using the **Reorder** tool in **Design View**.

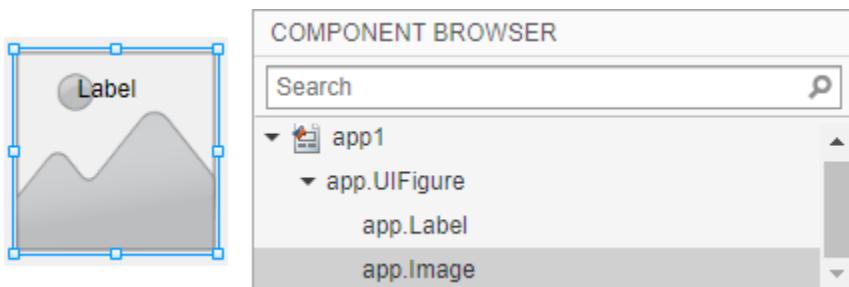
For example, create a label and then create an image. By default, the image appears on top of the label. The **Component Browser** shows the components based on their stacking order, with the image first since it is on top and the label second.



To reorder the components so that the label is on top of the image, select the image on the canvas, and then select **Reorder** in the toolbar. You can also right-click the image and select the **Reorder** tool. Send the image backward by choosing **Send Backward**.

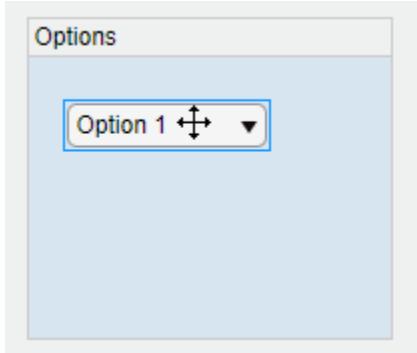


The image now is behind the label. When you reorder components, the order of the components inside the **Component Browser** also changes.



Arrange 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.

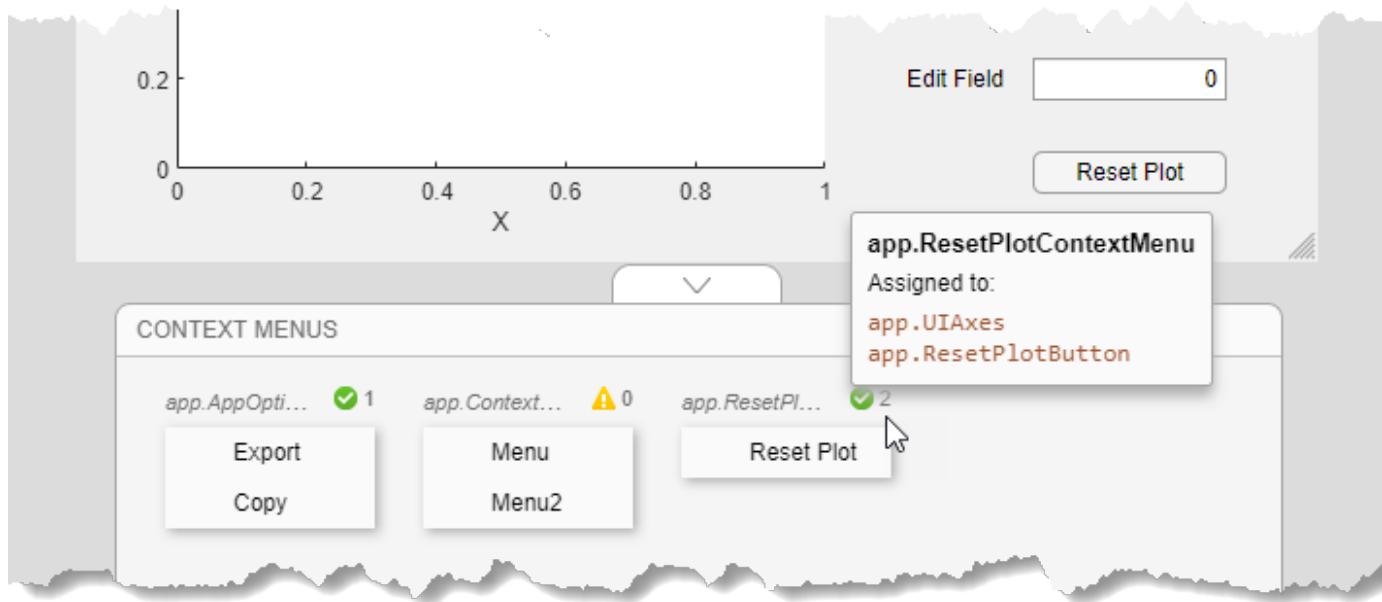


Create and Edit Context Menus

There are several ways to create context menus in App Designer. Since context menus are visible only when you right-click a component in the running app, they do not appear in the figure when you are in **Design View**. This makes the workflow for editing context menus slightly different than for other components. These sections describe the ways to create and edit context menus.

Create Context Menus

To create a context menu, drag it from the **Component Library** onto the UI figure or another component. This assigns the context menu to the **ContextMenu** property of that component. When you create a context menu it appears in an area on the canvas below the figure. This **Context Menus** area gives you a preview of each context menu you created and indicates how many components each one is assigned to. For example, this is how one set of context menus might appear on the canvas:



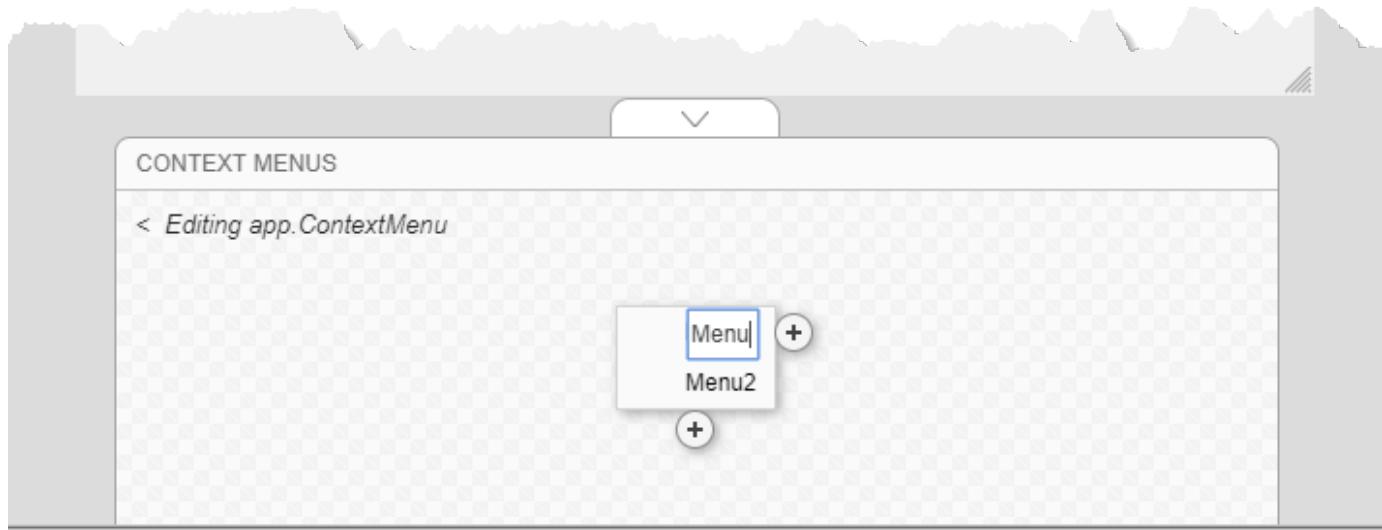
If you want to create a context menu without assigning it to a component, drag it to the **Context Menus** area instead.

Alternatively, create and assign a context menu to a specific component by right-clicking on that component and selecting **Context Menu > Add New Context Menu**.

All context menus are created as children of the UI figure and are added to the **Component Browser**, even if they are not assigned to a component.

Edit Context Menus

Edit a context menu by double-clicking it in the **Context Menus** area or by right-clicking it and selecting the edit option for the name of your menu. This brings the context menu into the **Context Menus** editing area where you can edit and add menu items and submenus.



When you are finished editing, click the back arrow (\leftarrow) to exit the edit area.

Change Context Menu Assignments

To disassociate a context menu from a component, right-click on the component and select **Context Menu > Unassign Context Menu**.

To replace the context menu that is assigned to a component with another one, you can drag the context menu onto the component, or you can right-click on the component, click **Context Menu > Replace With**, and select one of the other context menus you have created. If you only created one context menu, then the **Replace With** option does not appear.

Alternatively, select a component in the **Component Browser** and select **Interactivity** from the **Inspector** tab. Then, expand the **ContextMenu** drop-down list and select a different context menu to assign to the component.

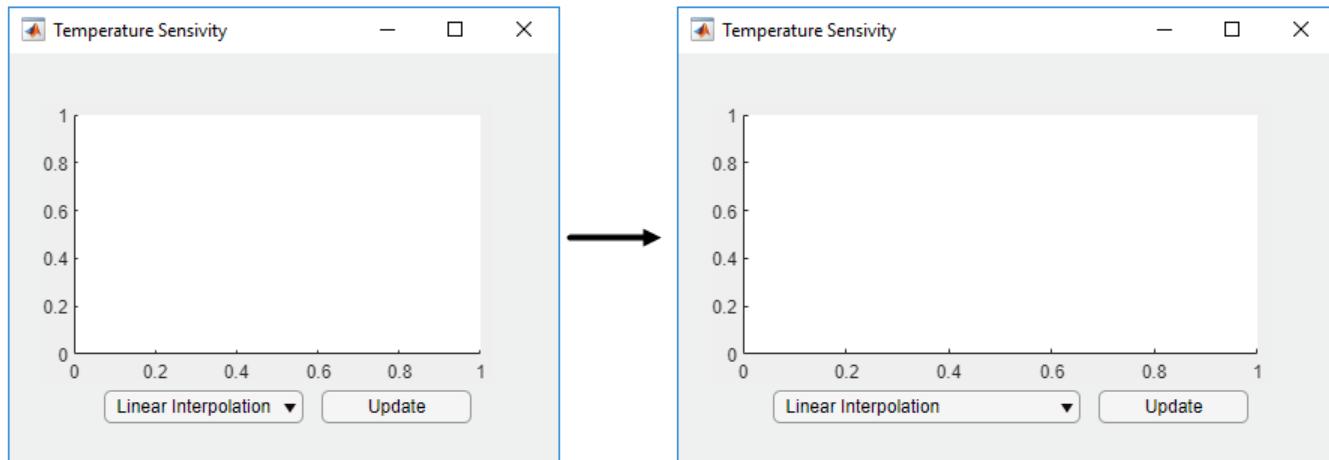
See Also

More About

- “App Building Components” on page 5-2
- “App Designer Keyboard Shortcuts” on page 9-2
- “Manage Resizable Apps in App Designer” on page 6-11

Manage 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. The `AutoResizeChildren` property controls this automatic resize behavior. By default, App Designer enables this property for the UI figure and all its child containers such as panels and tabs. 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.



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. To ensure that the alignment of components relative to one another (like a grouping of buttons) is preserved when your app is resized, parent the grouping of components to a panel, instead of directly to the figure.

Resizing Graphics Objects with Normalized Position Units

When graphics objects, like axes or charts, use normalized position units and are the child of a resizable container, certain properties of the graphics object are affected after the parent container is resized. For example, if axes or charts use a value of '`normalized`' for the `Units` property and are parented to a container with the `AutoResizeChildren` property set to '`on`', then:

- The value of the `OuterPosition` property for the axes or chart changes when the app is resized.
- The axes or chart does not shrink smaller than a minimum size when the app is resized.

If you want to avoid either of these behaviors, set the `AutoResizeChildren` property of the container to '`off`'.

Alternatives to Default Auto-Resize Behaviors

If you want more flexibility over how your app automatically resizes, use grid layout managers or the auto-reflow options in App Designer instead of the `AutoResizeChildren` property. For more information about these options, see:

- “Use Grid Layout Managers” on page 6-13

- “Apps with Auto-Reflow” on page 6-16

If the resize behaviors supported by `AutoResizeChildren`, grid layout managers, or auto-reflow options are not the behaviors you want, then you can create custom resize behaviors by writing a `SizeChangedFcn` callback function for the container. For more information, see the “`SizeChangedFcn`” property.

See Also

UI Figure

More About

- “Lay Out Apps in App Designer Design View” on page 6-2
- “Write Callbacks in App Designer” on page 7-15

Use Grid Layout Managers

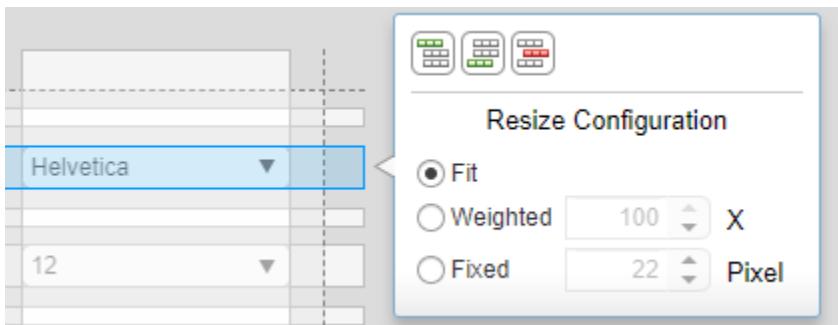
Grid layout managers provide a way to lay out your app without having to set pixel positions of UI components in **Position** vectors. For resizable apps, grid layout managers provide more flexibility than the automatic resize behavior in App Designer. They are also easier to configure than it is to code **SizeChangedFcn** callback functions.

Add and Configure Grid Layout Manager

In App Designer, you can add a grid layout manager to a blank app or to empty container components within the figure.

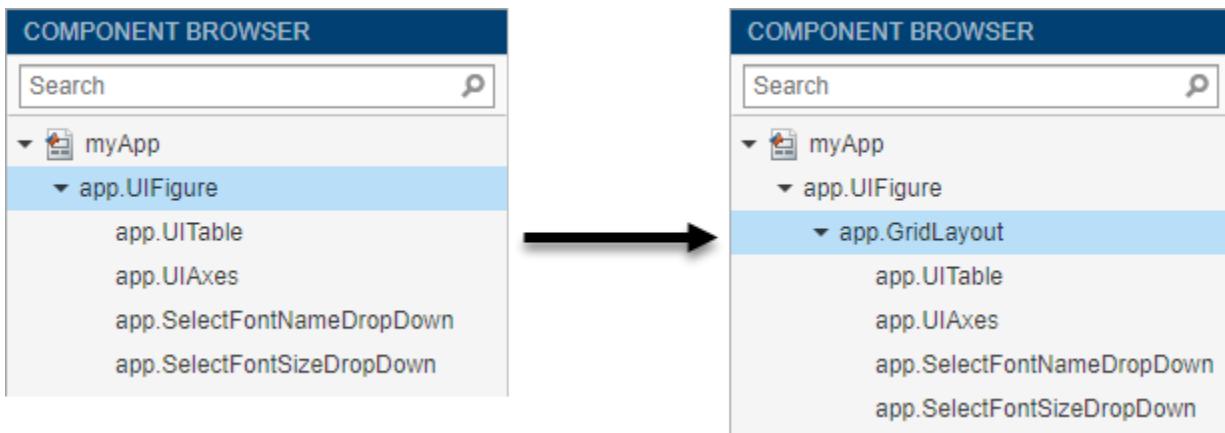
To use a grid layout manager, drag a grid layout from the **Component Library** onto the canvas. Alternatively, you can right-click the figure or container and select **Apply Grid Layout** from the context menu. A grid layout manager spans the entire app window or container that you place it in. It is invisible unless you are actively configuring it on the App Designer canvas.

To configure the grid layout manager, in **Design View**, bring the grid layout into focus by clicking in the area where you added it. Then, select the  button from the upper-left corner of the grid layout manager, or right-click the grid layout and select **Configure Grid Layout**. Then, select a row or column and from the **Resize Configuration** menu, specify **Fit**, **Weighted**, or **Fixed**. For more information about these options, see **GridLayout Properties**. You can also add or remove rows and columns.



Convert Components from Pixel-Based Positions to Grid Layout Manager

You can also convert the components within a UI figure or container from pixel-based positioning to a grid layout manager. When you apply a grid layout manager to a UI figure or container that has components in it, the components get added to the grid layout manager and their **Position** vectors get replaced by **Layout.Row** and **Layout.Column** values that specify their location in the grid. The component hierarchy also updates in the **Component Browser**.



Grid layout managers support different properties than other container components. In some cases, you might need to update your callback code if it sets these types of properties, or if it sets component properties that are not available when they are managed by the grid layout. If your callbacks or other behaviors do not work as expected, then look for code patterns like the ones listed in this table.

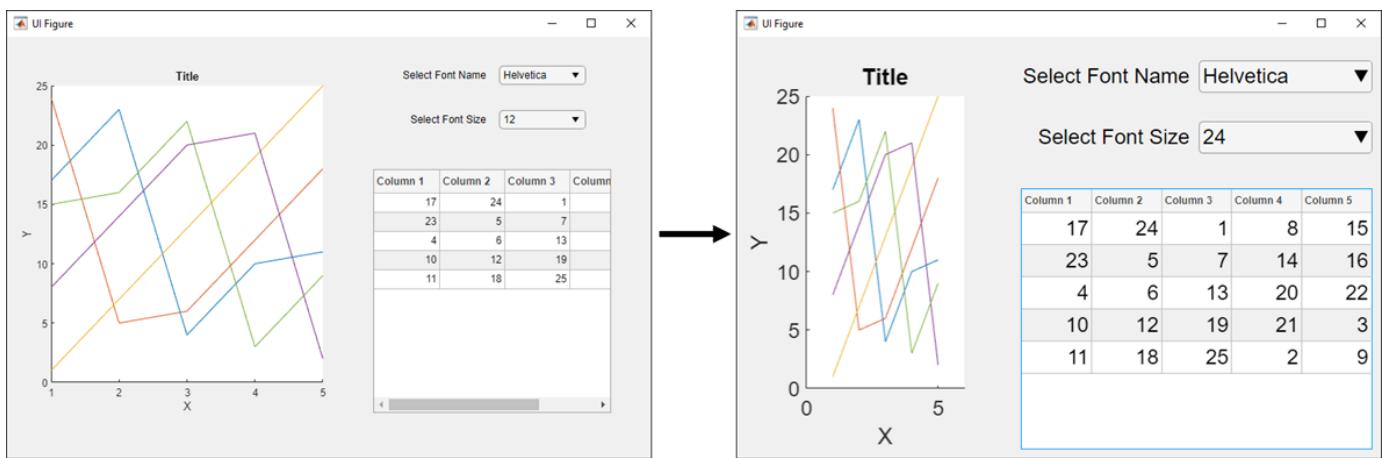
Symptom or Warning	Explanation	Suggested Action
Warning: Unable to set 'Position', 'InnerPosition', or 'OuterPosition' for components in 'GridLayout'.	You cannot set the Position property on components in a grid layout manager.	Specify a grid location for the component by setting the Layout property with appropriate Row and Column values.
Error using matlab.ui.container.GridLayout/set There is no FontSize property on the GridLayout class.	Properties you set on other container components might not be supported on the grid layout manager.	Update your code so that it sets properties on the intended container.
A context menu assigned to a container does not open in the running app.	When you add a grid layout manager to a container, it spans the entire container. This means that click events happen on the grid, instead of the container.	Reassign the context menu to the grid layout.

Example: Convert Components to Use Grid Layout Manager Instead of Pixel-Based Positions

This app shows how to apply a grid layout manager to the figure of an app that already has components in it. It also shows how to configure the grid layout manager so that the rows and columns automatically adjust to accommodate changes in size of text-based components.

- 1 Open the app in App Designer. In **Design View**, drag a grid layout manager into the figure.
- 2 Right-click the grid layout manager that you just added to the figure and select **Configure Grid Layout** from the context menu.

- 3 One-by-one, select the rows and columns of the grid that contain the drop-down menus and the table and change their resize configurations to **Fit**. When you are finished, verify that in the **Inspector** tab of the **Component Browser**, the **ColumnWidth** values are **12.64x,1.89x,fit,fit,fit,fit** and the **RowHeight** values are **1x,fit,1.93x,fit,3.07x,fit**.
- 4 Switch to **Code View**. Update each of the `DropDownValueChanged` callbacks so that the `allchild` functions set the font name and font size on components in `app.GridLayout`, instead of in `app.UIFigure`.
- 5 Now run the app to see how the grid adjusts to accommodate the components as their sizes change.



See Also

Functions
[uigridlayout](#)

Properties
[GridLayout Properties](#)

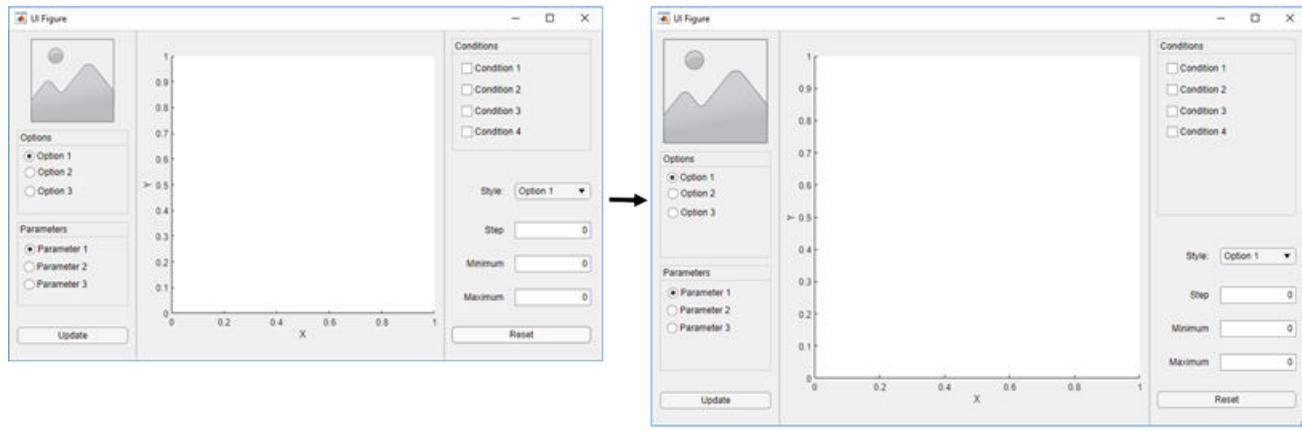
Apps with Auto-Refow

Apps with auto-reflow are preconfigured app types that optimize the viewing experience by automatically adjusting the size, location, and visibility of the app content in response to screen size, orientation, and platform. Use apps with auto-reflow if you expect to run or share your apps across multiple environments or desktop resolutions.

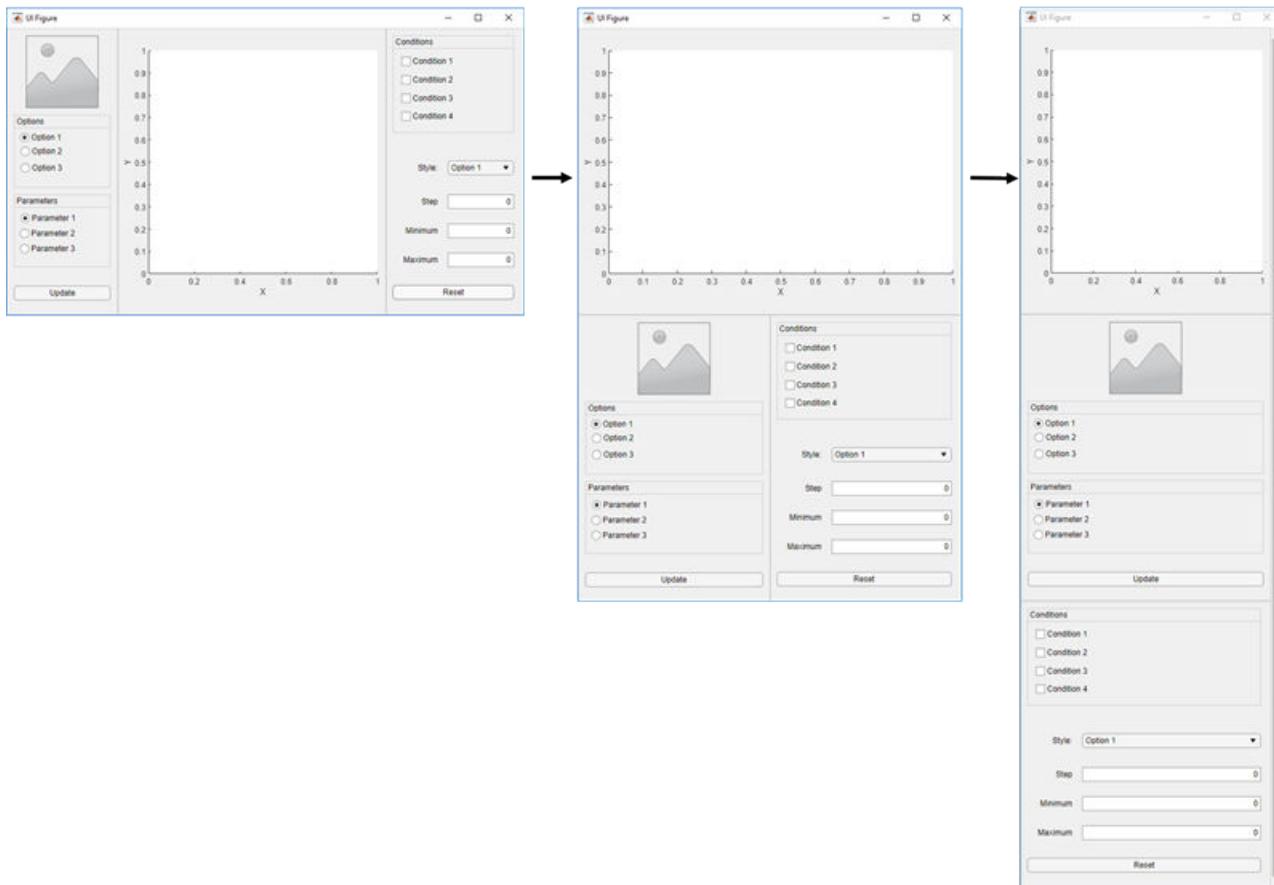


What is Auto-Refow?

Apps with auto-reflow extend the existing auto-resize behaviors that are on by default in all App Designer apps. These apps detect and adapt to the available screen size when they are first displayed. Both 2- and 3-panel apps have a large flexible-size panel, intended for visualizations like plots. As the app changes size, the large panel grows or shrinks, depending on the space available.



When an app is resized beyond a certain predefined threshold, the panels in the app reflow and reorder to make the best use of the space. As panels reorder themselves, they and the components in them dynamically adjust in size while extra space between components (whitespace) is also reduced.



When an app becomes very small, auto-resize stops eliminating whitespace and resizing components. This can put some components outside the visible part of the window. To access these components, set the **Scrollable** property of the panels to 'on'. This enables scroll bars to appear when necessary.

Create New App with Auto-Reflow

The App Designer Start Page includes options to create new 2-panel and 3-panel apps with auto-resize and auto-reflow, and canvas interactions to guide app building. No additional code is needed to achieve the refloowing and resizing behavior.

Convert Existing App to Use Auto-Reflow

You can also convert an existing app into an app with auto-reflow by expanding the **Convert**  drop-down menu from the **File** section of the **Canvas** tab and selecting **2-Panel App with Auto-Reflow** or **3-Panel App with Auto-Reflow**.

When you convert an existing app to an app with auto-reflow, App Designer:

- Creates a duplicate of your app with `_autoreflow` appended to the file name. Your original app file is not changed.

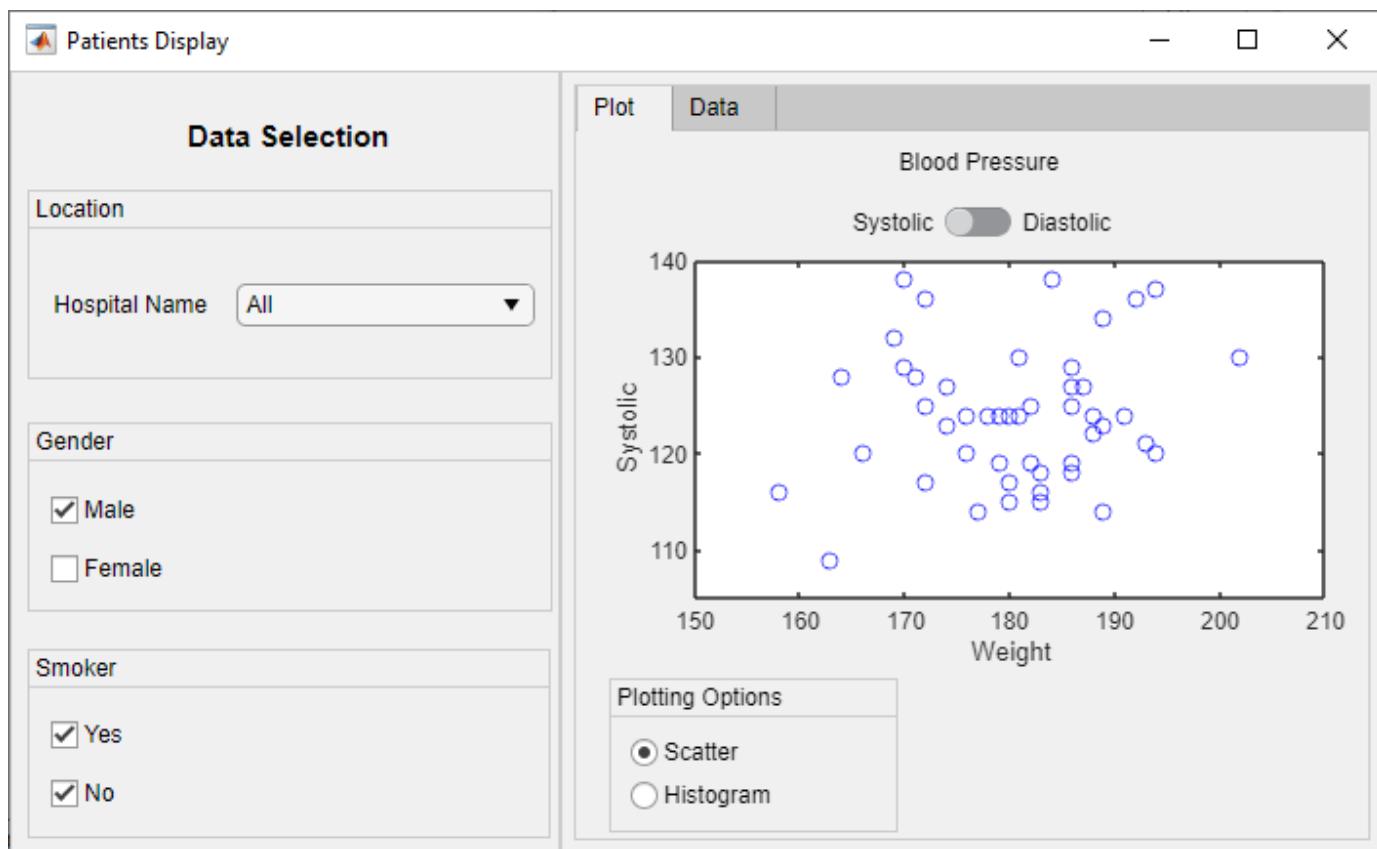
- Automatically adds preconfigured panels to your app to provide the automatic reflow and resize behaviors.
- Creates a `SizeChangedFcn` callback function in order to control the layout of the app as the figure is resized.

In some cases, after App Designer has converted your app, you may need to update your callback code or the position of some components. This table describes some examples of adjustments you that you may need to make.

Symptom	Explanation	Suggested Action
Components overlap	App Designer tries to maintain the relative positions of your components, but you may need to make some minor adjustments.	Adjust the position of components as needed.
Callback code does not behave as expected	When the preconfigured panels are added to your app the hierarchy of the components in your app changes. If your callbacks reference components based on their parent, they may need to be updated.	Update the parent of the components in your callbacks.
Existing <code>SizeChangedFcn</code> callback on the UI figure does not behave as expected	Apps with auto-reflow generate their own <code>SizeChangedFcn</code> callback for the figure. If your app already had a <code>SizeChangedFcn</code> callback for the figure, App Designer disconnects it from the figure, but it does not remove the code.	After your app has been converted, modify or remove the <code>SizeChangedFcn</code> callback that was disconnected from the figure. You can assign it to another container component, or remove it if it is no longer needed.

Example: App with Auto-Reflow

This app has components within panels that have auto-reflow behavior. Controls for data selection are parented to the left panel and data visualizations are parented to two tabs in the right panel. Run the app and change the size of the app window. The app content resizes and refloows based on the app window size.



See Also

appdesigner

App Programming

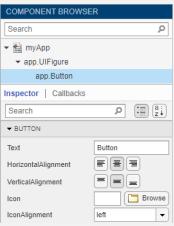
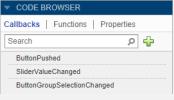
- “Manage Code in App Designer Code View” on page 7-2
- “Startup Tasks and Input Arguments in App Designer” on page 7-8
- “Create Multiwindow Apps in App Designer” on page 7-11
- “Write Callbacks in App Designer” on page 7-15
- “Reuse Code Using Helper Functions” on page 7-20
- “Share Data Within App Designer Apps” on page 7-23
- “Compatibility Between Different Releases of App Designer” on page 7-26
- “Use One Callback for Multiple App Designer Components” on page 7-28

Manage 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.

Manage Components, Functions, and Properties

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

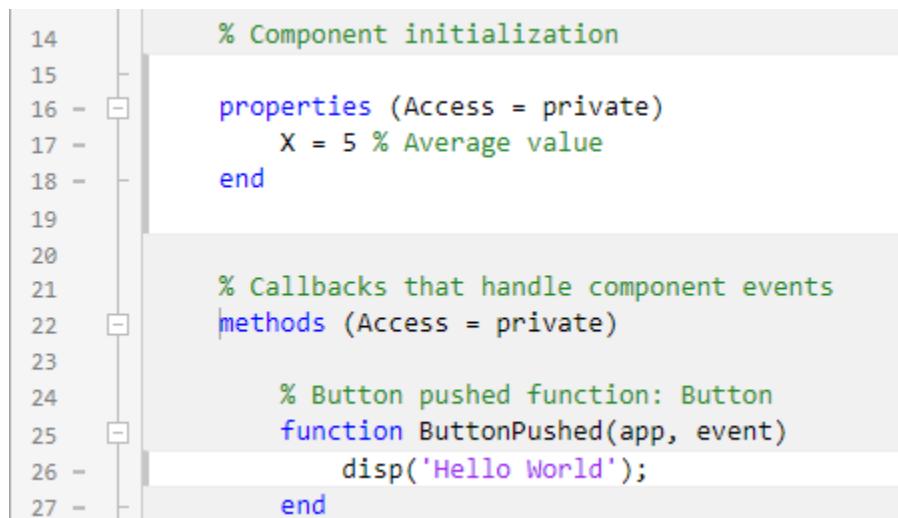
Pane Name	Pane Appearance	Pane Features
Component Browser		<ul style="list-style-type: none"> Context menu — Right-click a component in the list to display a context menu that has options for deleting or renaming the component, adding a callback, or displaying help. Select the Include Component Labels 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. Inspector 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.
Code Browser		<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. Rearrange the order of callbacks by selecting the callback you want to move and then, drag and drop the callback into its new position in the list. This also repositions the callback in the editor. Search bar — Quickly locate a callback, helper function, or property by typing part of its name in the search bar.
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.

Identify Editable Sections of Code

In the **Code View** editor, some sections of code are editable and some are not. Uneditable sections are generated and managed by App Designer, whereas editable sections correspond to:

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

In the default color scheme, uneditable sections of code are gray and editable sections of code are white.



```

14 % Component initialization
15
16 - properties (Access = private)
17 -     X = 5 % Average value
18 - end
19
20
21 % Callbacks that handle component events
22 methods (Access = private)
23
24     % Button pushed function: Button
25     function ButtonPushed(app, event)
26         disp('Hello World');
27     end

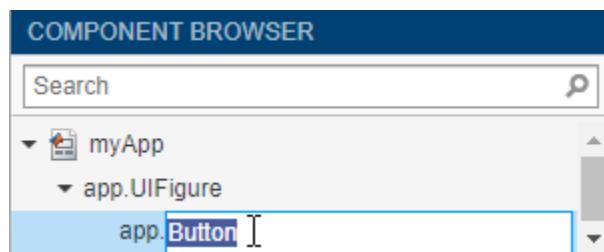
```

Program Your App

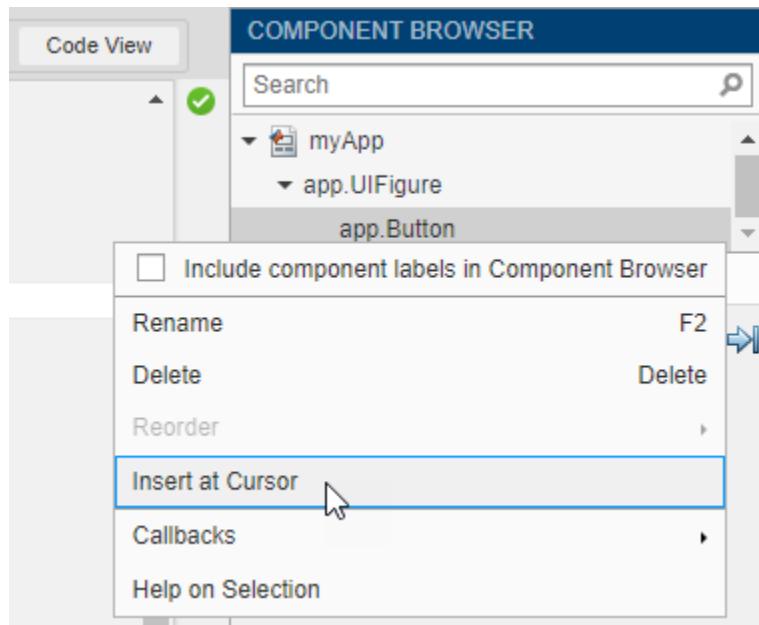
App Designer defines your app as a MATLAB class. You do not need to understand classes or object-oriented programming to create an app because App Designer manages those aspects of the code. However, programming in App Designer requires a different workflow than working strictly with functions. You can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

Manage 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**. Place your cursor in an editable area of the code where you want to add the component name. Then, 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.

Manage 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 7-15.

Share 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 7-23.

Single-Source 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 “Reuse Code Using Helper Functions” on page 7-20.

Create Input Arguments

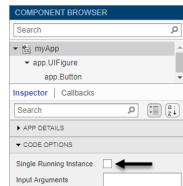
To add input arguments to your app, click **App Input Arguments**  in the **Editor** tab. Input arguments are commonly used for creating apps that have multiple windows. For more information, see “Startup Tasks and Input Arguments in App Designer” on page 7-8.

Limit Your App to Only One Running Instance at a Time

When you create an app in App Designer you have the option to select between two run behaviors for the app:

- Allow only a single running instance of the app at a time.
- Allow multiple instances of the app to run at the same time. This is the default behavior.

To change the run behavior of your app, select the app node from the **Component Browser**. Then, from the **Code Options** section of the **Inspector** tab, select or clear **Single Running Instance**.

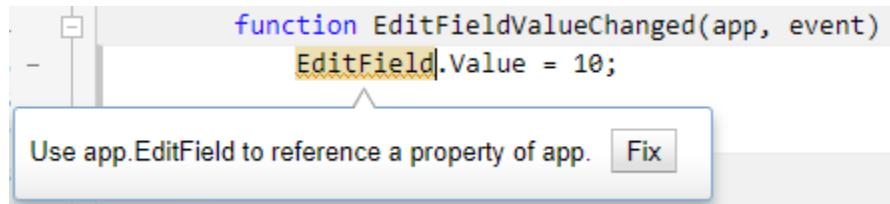


When **Single Running Instance** is selected and you run the app multiple times, MATLAB reuses the existing instance and brings it to the front rather than creating a new one. When this option is cleared, MATLAB creates a new app instance each time you run it and continues to run the existing instances. These run behaviors apply to apps that you run from the **Apps** tab on the MATLAB Toolbar or from the Command Window.

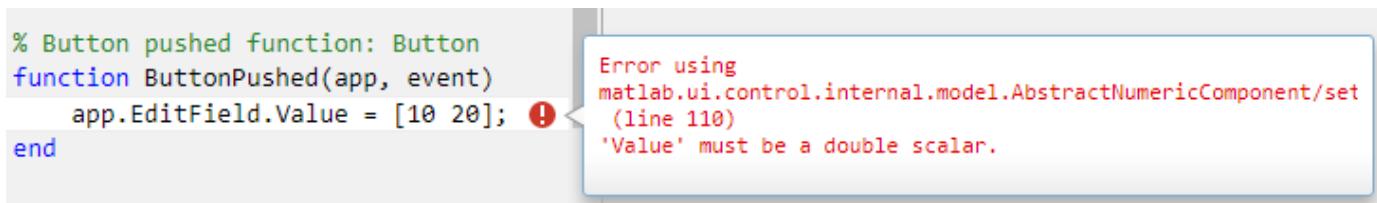
When you run apps from App Designer their behavior doesn't change whether this option is selected or cleared. App Designer always closes the existing app instance before creating a new one.

Fix Code 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.



Personalize Code View Appearance

You can customize how your code appears in the **Code View** editor. To change your code view preferences, go to the **Home** tab of the MATLAB Desktop. In the **Environment** section, click **Preferences**.

Change Color Settings

To change the color settings for editable sections of code and to customize syntax highlighting, select **MATLAB > Colors** and adjust the desktop tool colors and the MATLAB syntax highlighting colors. These settings affect both the App Designer **Code View** editor and the MATLAB Editor. For more information, see “Change Desktop Colors”.

To change the background color of uneditable sections of code, select **MATLAB > App Designer** and adjust the read-only background color. This setting can be changed only if the **Use system colors** option in **MATLAB > Color Preferences** is unchecked.

Change Tab Preferences

To specify the size of tabs and indents in the **Code View** editor, select **MATLAB > Editor/Debugger > Tab**. From here, you can specify the size of tabs and indents, as well as details about how tabs behave. These preferences affect both the App Designer **Code View** editor and the MATLAB Editor. For more information, see “Editor/Debugger Tab Preferences”.

See Also

Related Examples

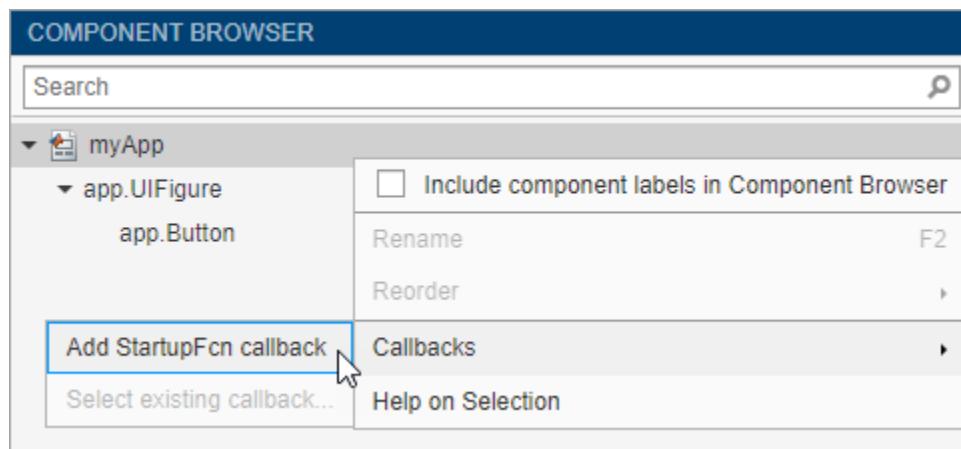
- “Write Callbacks in App Designer” on page 7-15
- “Share Data Within App Designer Apps” on page 7-23
- “Reuse Code Using Helper Functions” on page 7-20
- “Startup Tasks and Input Arguments in App Designer” on page 7-8

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 app node from the top of the **Component Browser** hierarchy, and select **Callbacks > Add StartupFcn callback**. The app node has the same name as your MLAPP file.



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.

```

9 % Callbacks that handle component events
10 methods (Access = private)
11
12 % Code that executes after component creation
13 function startupFcn(app)
14
15 end

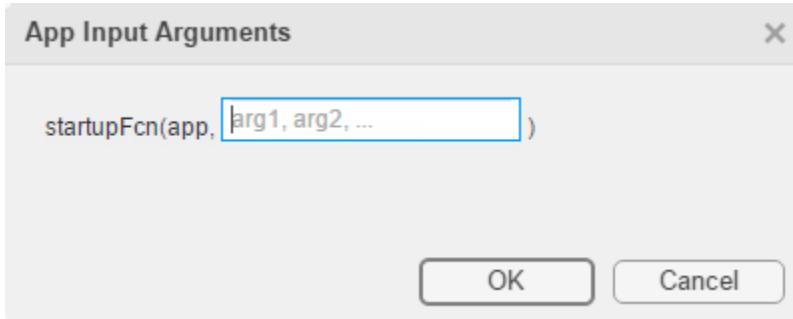
```

See “App with Auto-Reflow That Updates Plot Based on User Selections” on page 8-3 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.

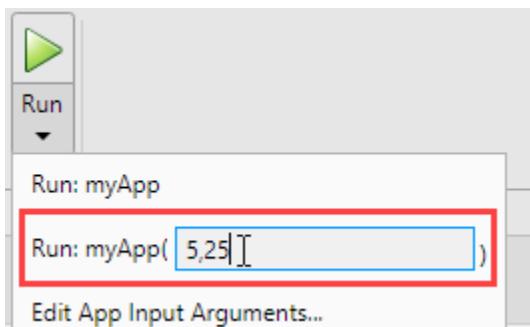
To add input arguments to an app, open the app in App Designer and click **Code View**. Then click **App Input Arguments** in the **Editor** tab.



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.

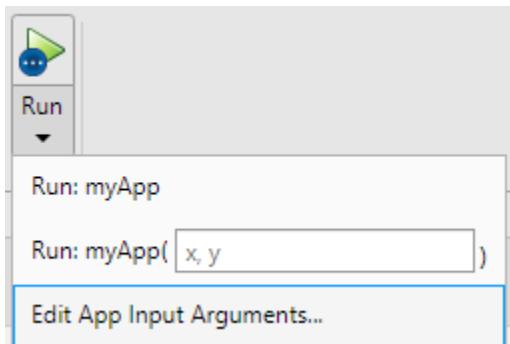
After successfully running the app with a set of input arguments, the **Run** button icon contains a blue circle.



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 “Create Multiwindow Apps in App Designer” on page 7-11 for an example of an app that uses input arguments.

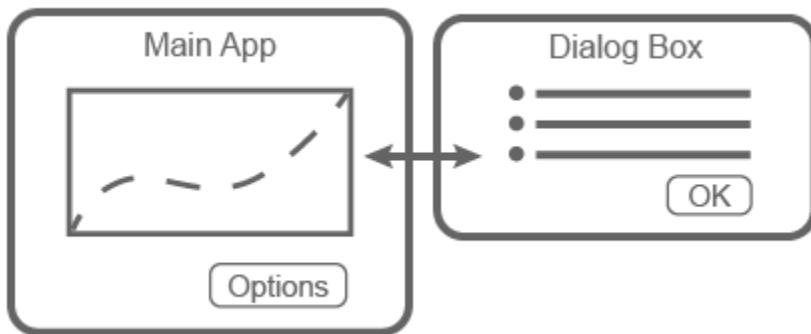
See Also

Related Examples

- “Write Callbacks in App Designer” on page 7-15
- “Create Multiwindow Apps in App Designer” on page 7-11

Create 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.

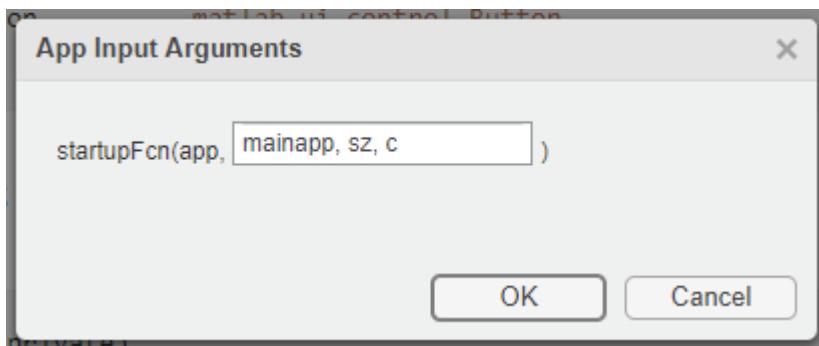
- “Send Information to the Dialog Box” on page 7-11 — Write a `startupFcn` callback in the dialog box app that accepts input arguments. One of the input arguments must be the main app object. Then, in the main app, call the dialog box app with the input arguments.
- “Return Information to the Main App” on page 7-13 — 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 7-13 — 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 7-14.

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 7-14.

- 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
```

- 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 7-14.

- 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';

    % Delete the dialog box
```

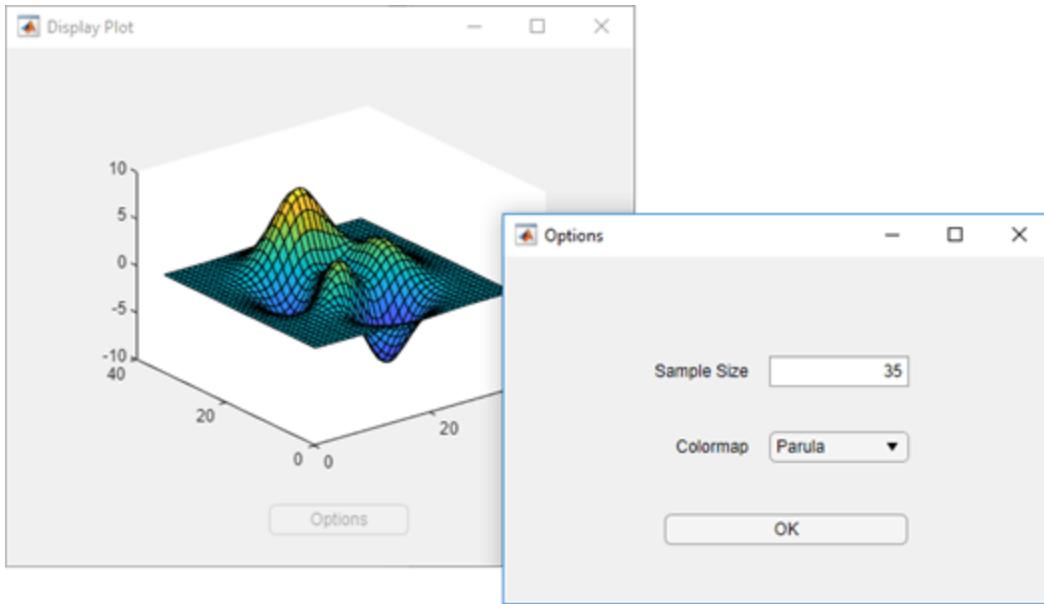
```
    delete(app)
end
```

- 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 7-15
- “Startup Tasks and Input Arguments in App Designer” on page 7-8

Write Callbacks in App Designer

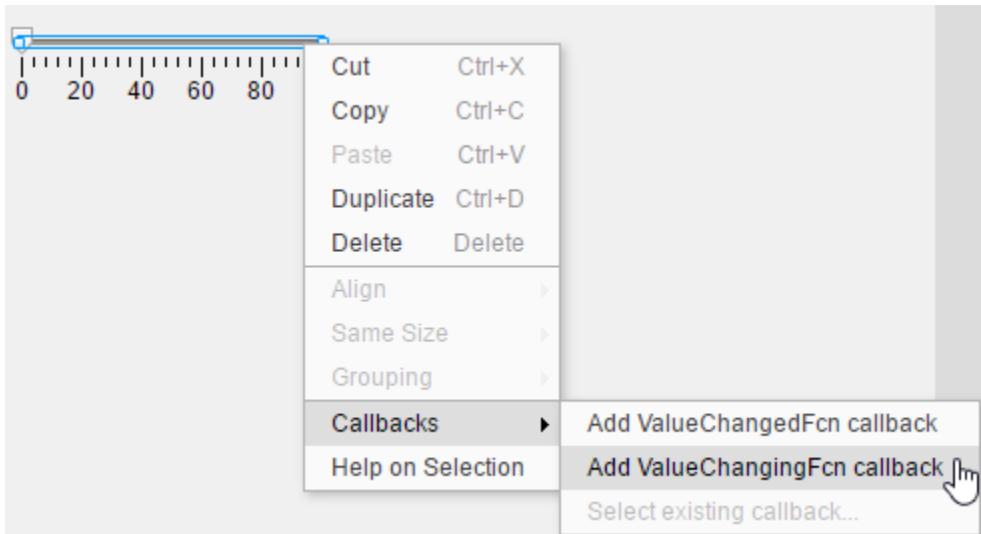
A callback is a function that executes when a user interacts with a UI component in your app. Most components can have at least one callback. However some components, such as labels and lamps, do not have callbacks because those components only display information.

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

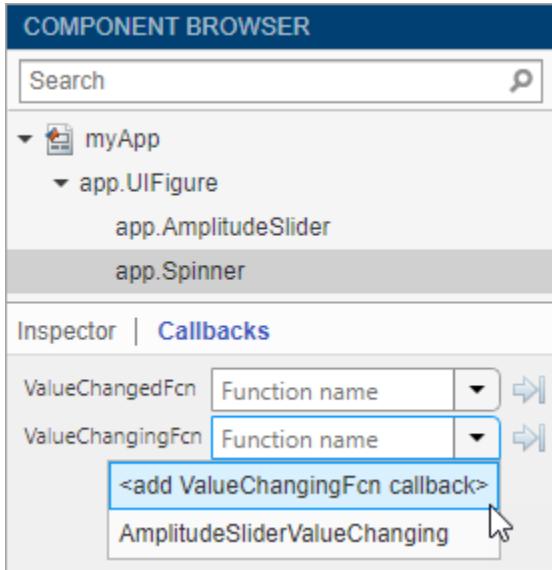
Create a Callback Function

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.

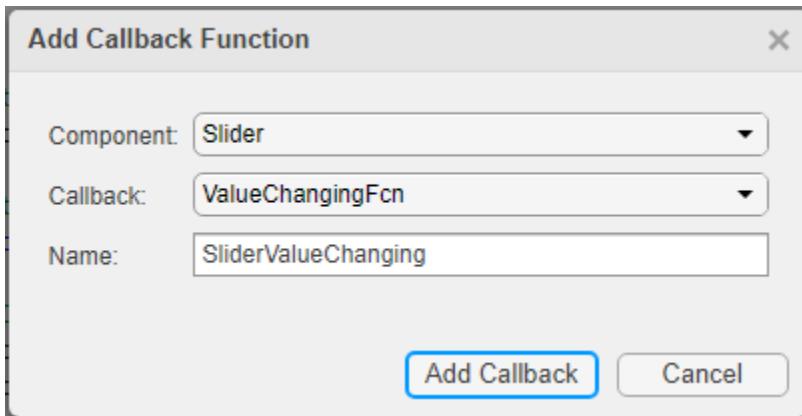
- Right-click a component in the canvas, **Component Browser**, or **App Layout** pane, and select **Callbacks > Add (callback property) callback**.



- Select the **Callbacks** tab in the **Component Browser**. The left side of the **Callbacks** tab shows a list of supported callback properties. The drop-down menu 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 SliderValueChanging(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 “App Building Components” on page 5-2.

Searching for Callbacks in Your Code

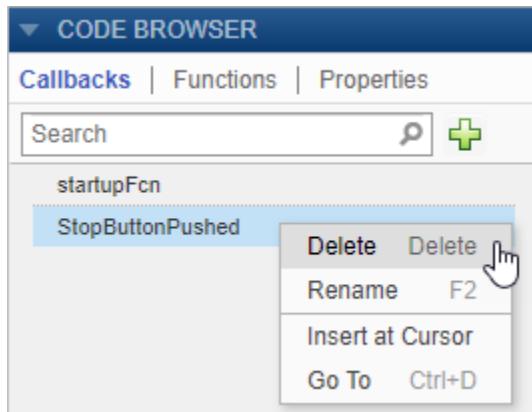
If your app has a lot of callbacks, you can quickly search and navigate to a specific callback by typing part of the name in the search bar at the top of the **Callbacks** tab in the **Code Browser**. After you begin typing, the **Callbacks** pane clears, except for the callbacks that match your search.



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.

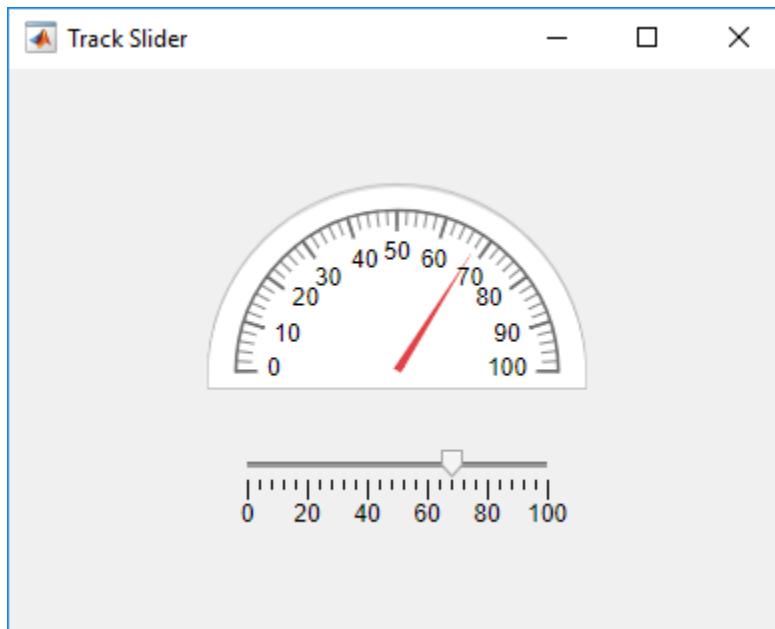
Deleting Callbacks

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 7-23

- “Use One Callback for Multiple App Designer Components” on page 7-28
- “Write Callbacks for Apps Created Programmatically” on page 11-2

Reuse Code Using Helper Functions

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.

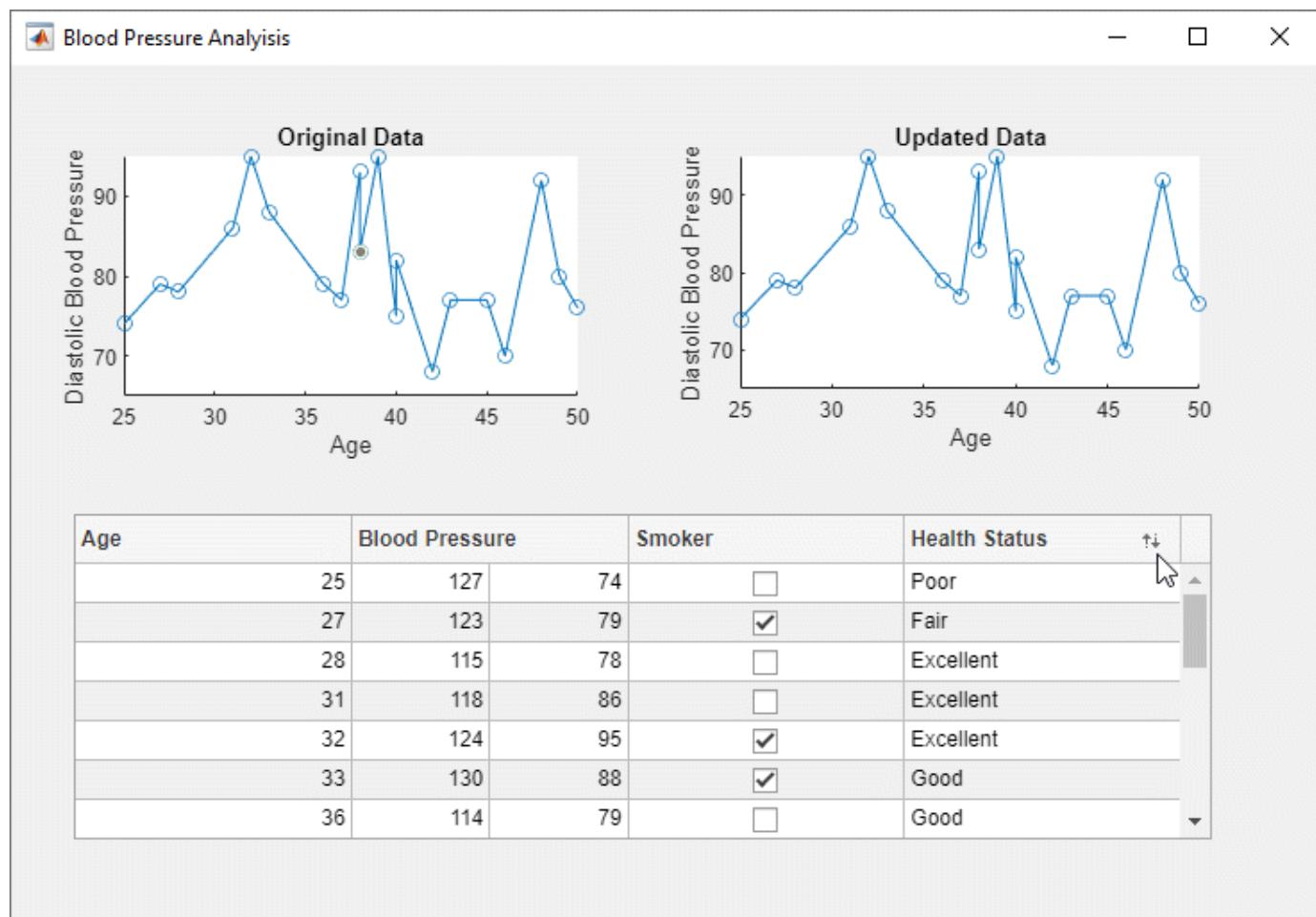


Click a search result to scroll the function into view. Right-clicking a search result and selecting **Go To** places your cursor in the function.

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

Example: Helper Function that Initializes Plots and Displays Updated Data

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



See Also

Related Examples

- “Write Callbacks in App Designer” on page 7-15
- “Create Multiwindow Apps in App Designer” on page 7-11

Share Data Within App Designer Apps

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`.

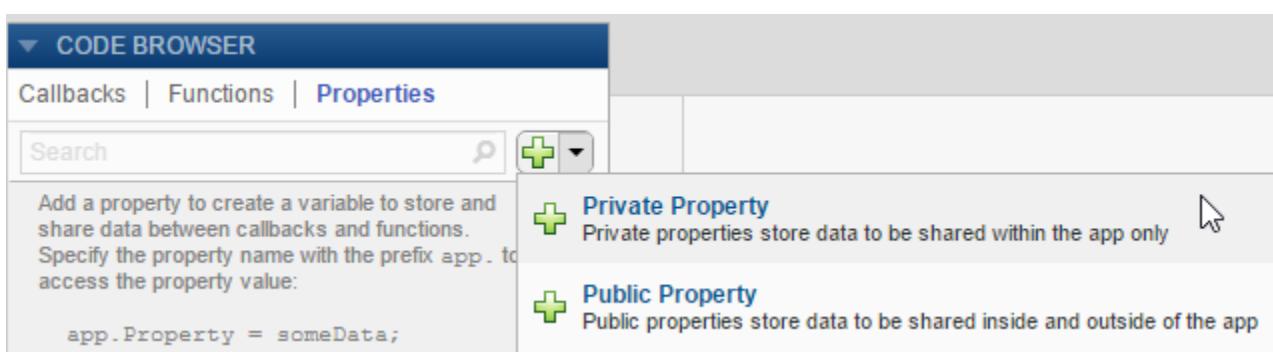
```
x = app.PressureGauge.Value; % Get the gauge value
app.PressureGauge.Value = 50; % Set the gauge value to 50
```

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.

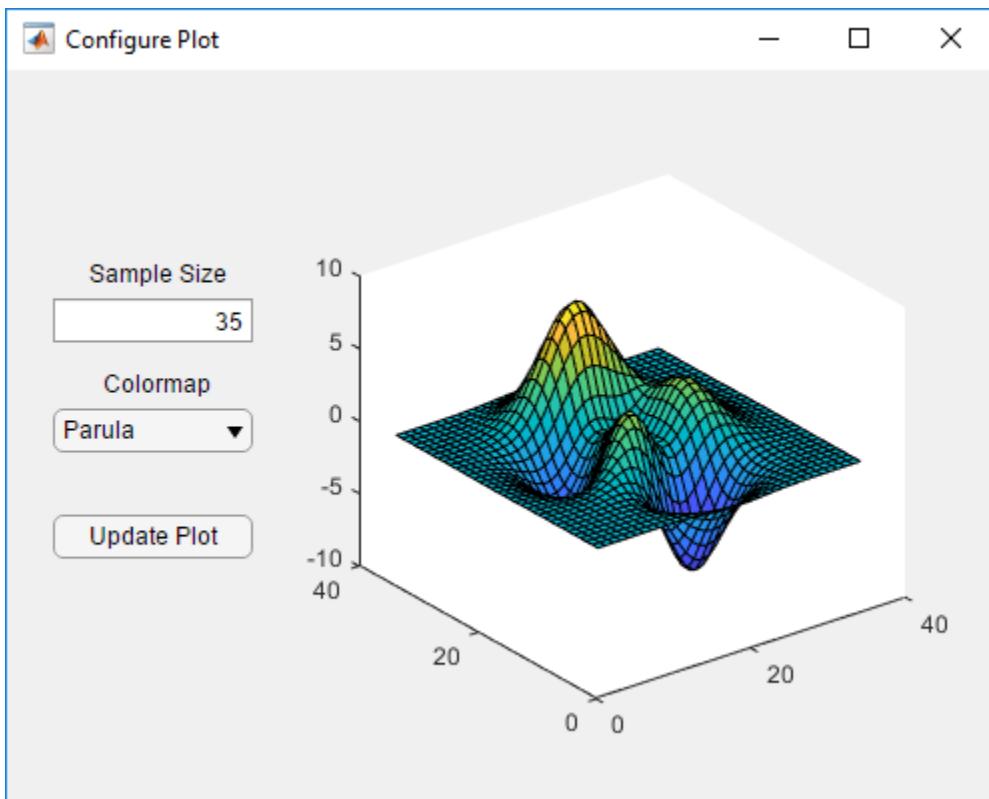
```
properties (Access = public)
    X = 5; % Average cost
end
```

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

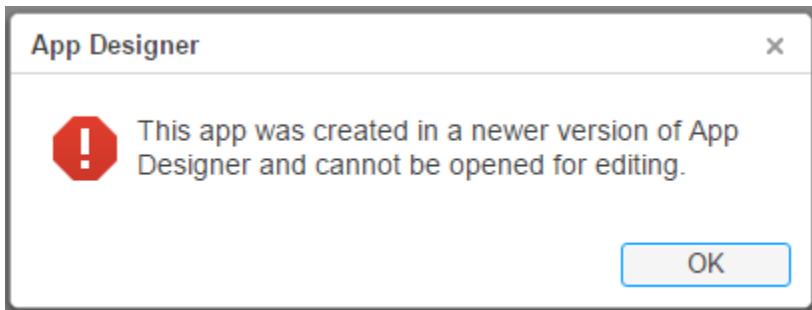
Related Examples

- “Write Callbacks in App Designer” on page 7-15
- “Create Multiwindow Apps in App Designer” on page 7-11

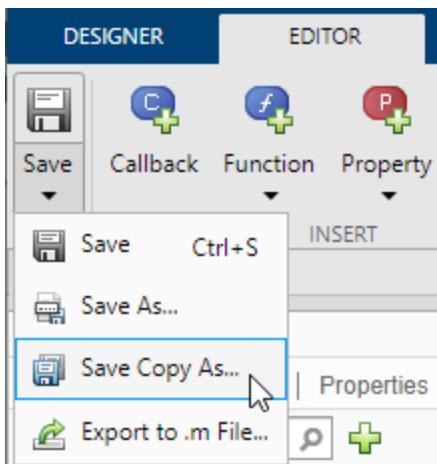
Compatibility Between Different Releases of App Designer

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.

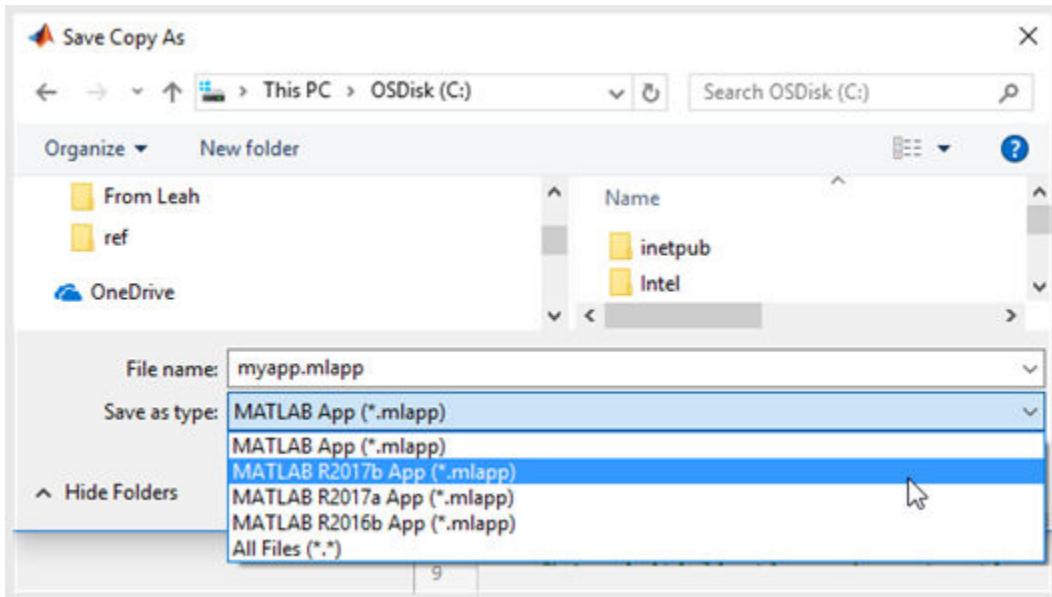
If you try to edit an app, created in R2018a or later, in an earlier release of App Designer, the new format is not recognized after saving your changes. You see a message such as this.



To enable editing of newer apps in a previous release, 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.



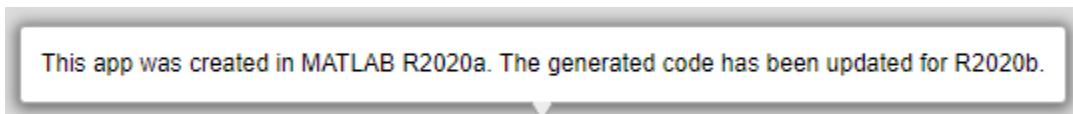
Save Copy As Versus Save As

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.

Opening Apps for Editing in a Newer Release

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.



See Also

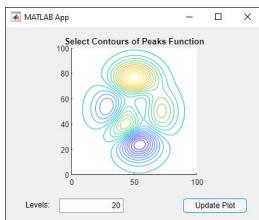
[appdesigner](#)

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

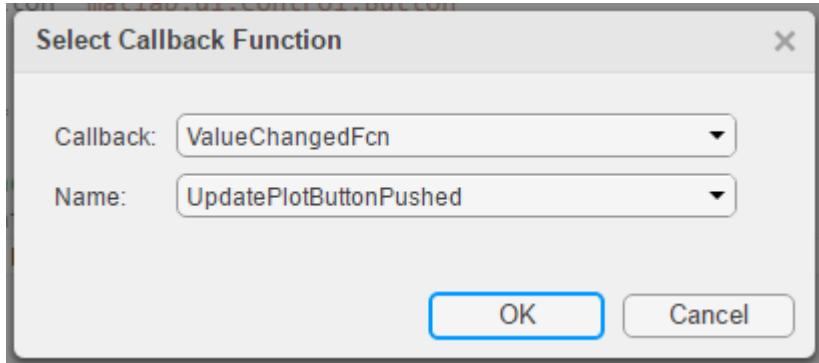
This example shows how to create an app containing two UI components that share a callback. The app displays a contour plot with the specified number of levels. When the user changes the value in the edit field, they can press **Enter** or click the **Update Plot** button to update the plot.



- 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.



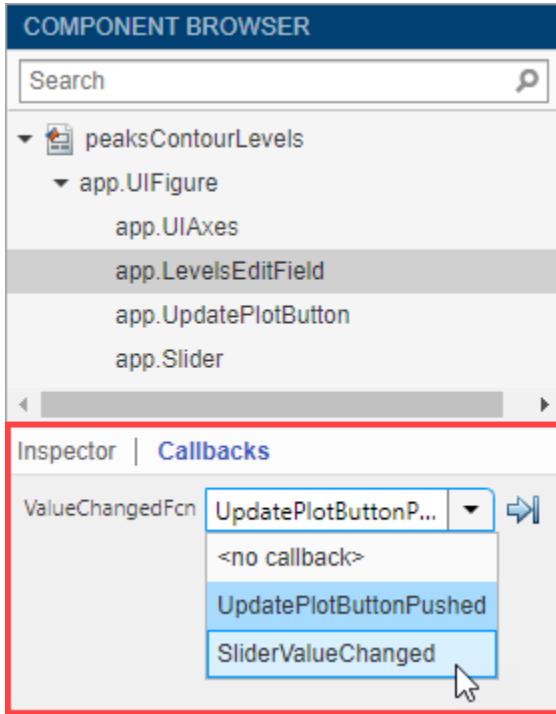
Sharing this callback allows the user to update the plot after changing the value in the edit field and pressing **Enter**. Alternatively, they can change the value and press the **Update Plot** button.

- 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 **Inspector** tab:
 - 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 and select a different callback from the drop-down menu. The drop-down displays only the existing callbacks.



To disconnect a callback that is shared with a component, select the component in the **Component Browser**. Then click the **Callbacks** tab 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.

To delete a callback function definition from your code, go to the **Callbacks** tab in the **Code Browser** and right-click the callback you want to remove. Then, select **Delete** from the context menu.

See Also

Related Examples

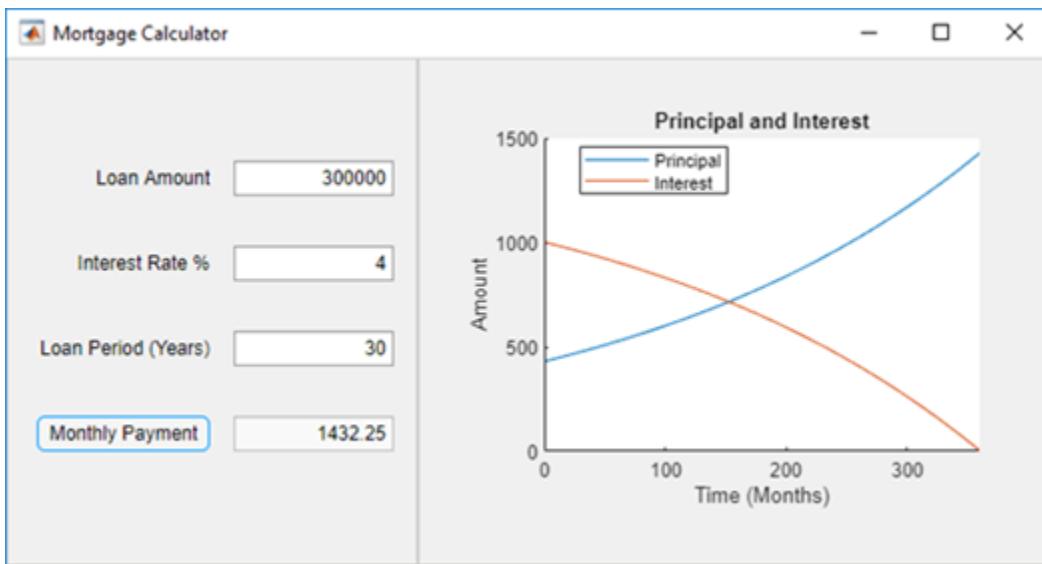
- “Write Callbacks in App Designer” on page 7-15

App Designer Examples

App that Calculates and Plots Data Based on Numerical Input

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

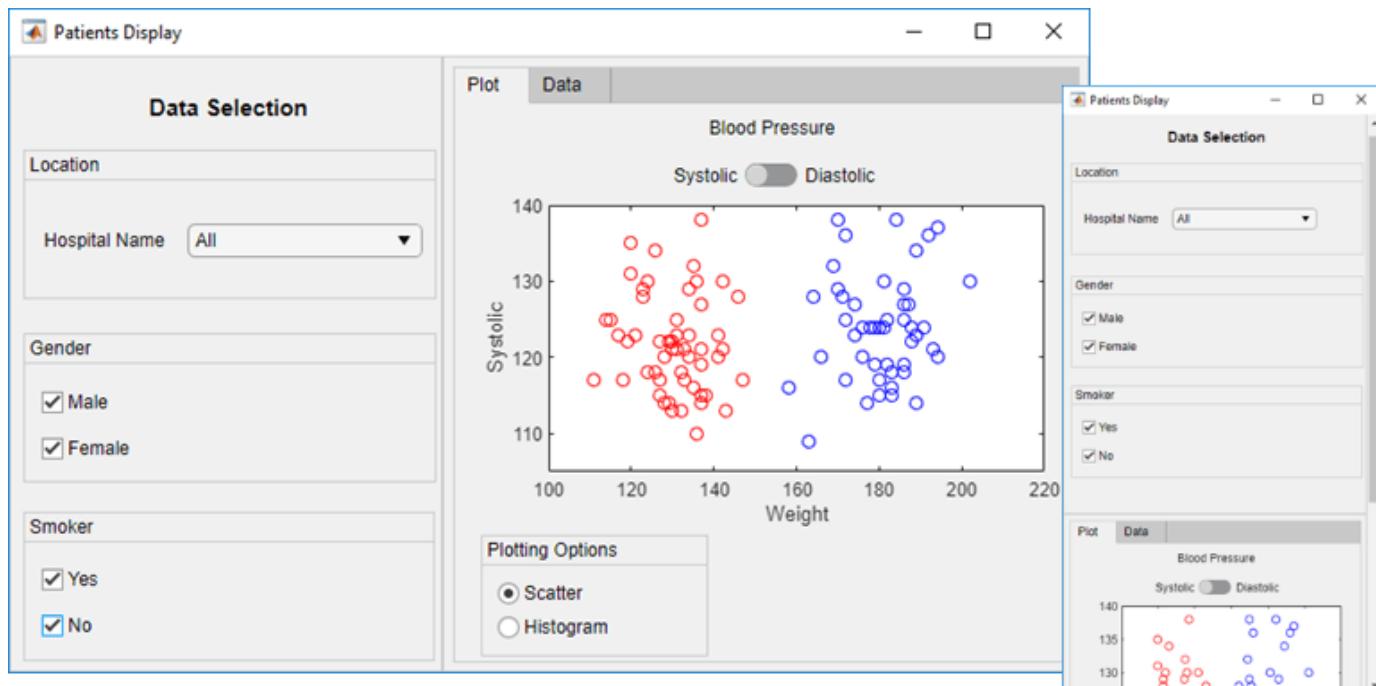
- “Write Callbacks in App Designer” on page 7-15

App with Auto-Reflow That Updates Plot Based on User Selections

This app shows how to define controls and tabs within the panels of an app with auto-reflow. The controls are in an anchored panel on the left. The right panel that reflows contains two tabs. One tab displays a chart and user interface components for adjusting the chart. The other tab contains a table with the data used to make the chart. User selections update both the plot and the table. The app responds to resizing by automatically growing, shrinking, and reflowing the app content.

The app includes these components:

- Check boxes — used to update the plot and table when the user selects or clears a check box.
- Switch — used to toggle the data that is visualized in the chart
- 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](#)

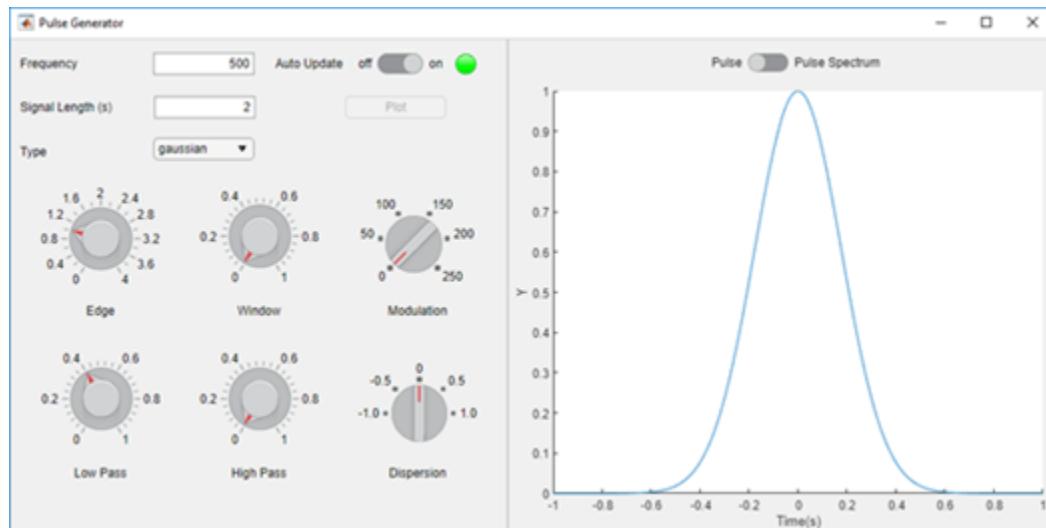
Related Examples

- “Write Callbacks in App Designer” on page 7-15

App that Uses Grid Layout to Manage Component Positions and Resizing

This app shows how to use a grid layout manager to control the alignment and resizing of knobs when the app is resized. The app also 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

Functions
uigridlayout

Properties
UIAxes

Related Examples

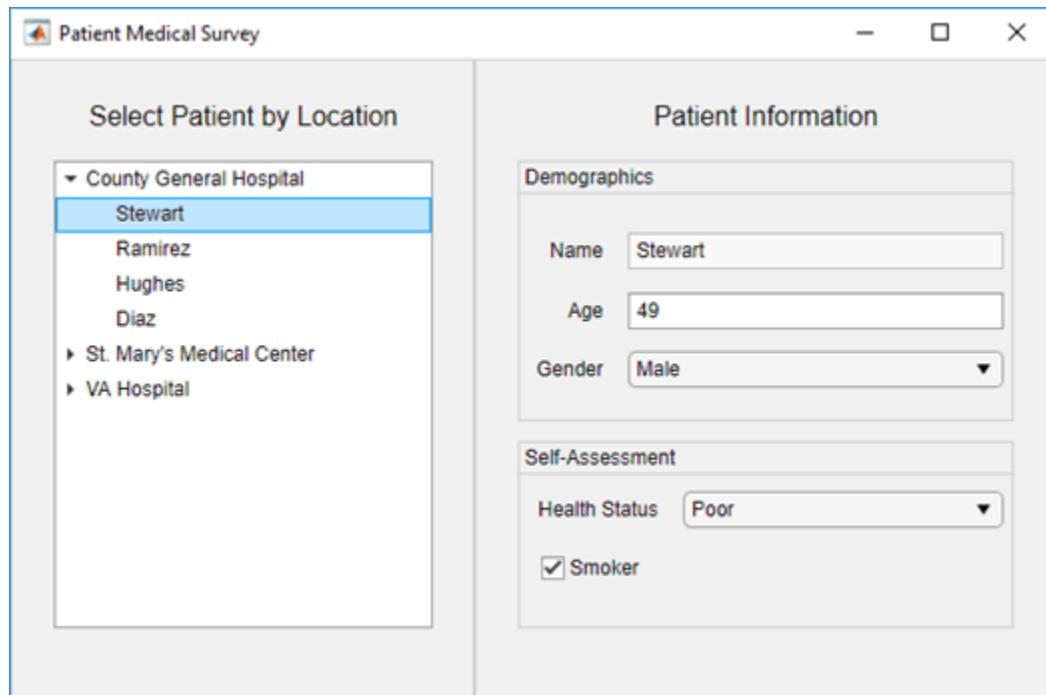
- “Write Callbacks in App Designer” on page 7-15

App That Displays Data in a Hierarchy Using Tree

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. When the user edits the patient data, the app asks the user to confirm the change and then stores the change in the table variable.

In addition to the tree and **Patient Information** panel, this 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
- Confirmation dialog box — Used to confirm changes to patient data



See Also

[readtable](#) | [table](#) | [uitree](#) | [uitreenode](#)

Related Examples

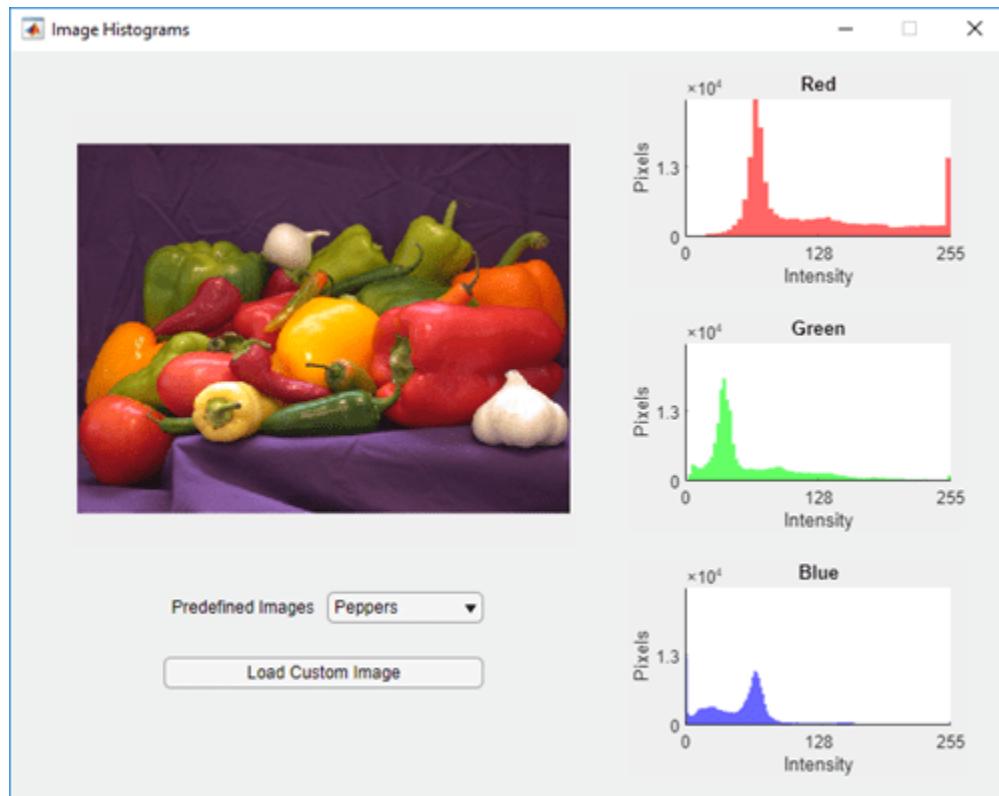
- “Add UI Components to App Designer Programmatically” on page 5-20

Create App that Uses Multiple Axes to Display Results of Image Analysis

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

Functions

`imagesc` | `imread` | `uialert`

Properties

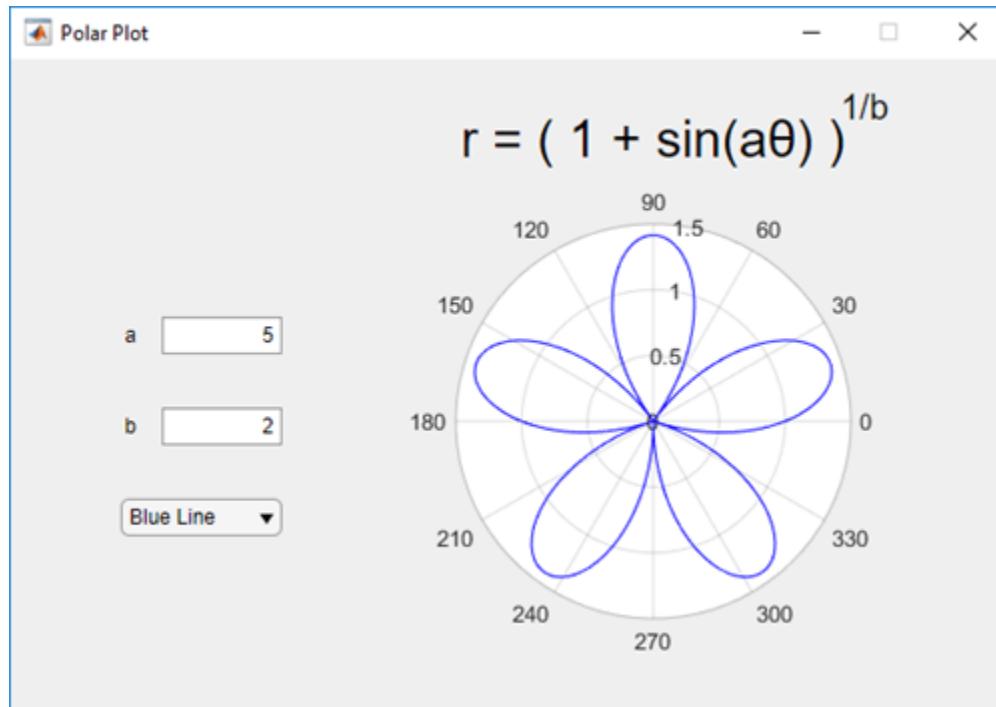
`UIAxes`

Create Polar Axes Programmatically in an App

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 these 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

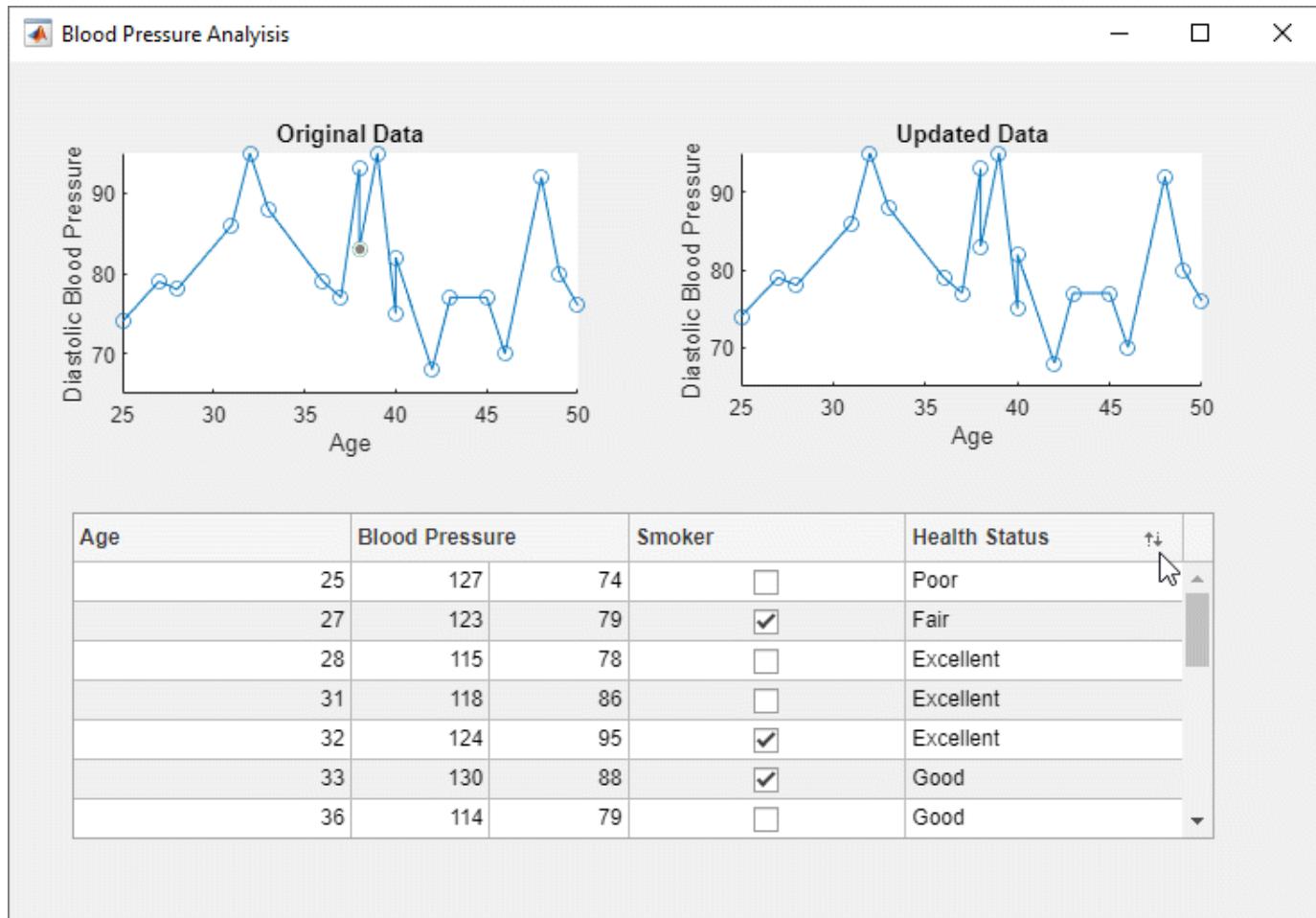
- “Display Graphics in App Designer” on page 4-12

Create App with a Table That Can Be Sorted and Edited Interactively

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. One of the plots updates when the user edits values or sorts columns 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](#) | [uitable](#)

Related Examples

- “Table Array Data Types in App Designer Apps” on page 5-15

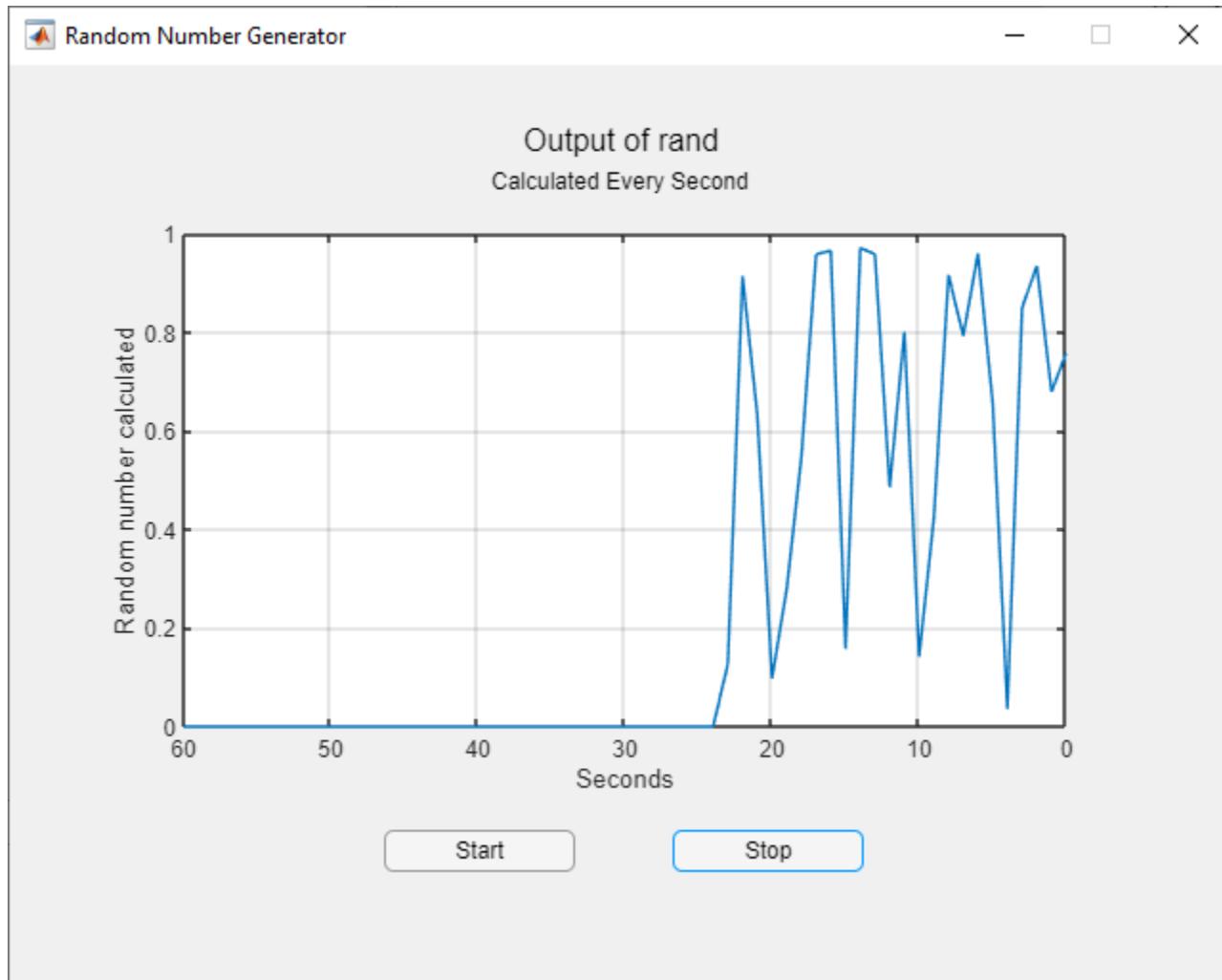
- “Programmatic App that Displays a Table” on page 14-2

Create App with Timer Object Configured Programmatically

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 generates random data every second and plots the result.

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

Functions
memory | timer

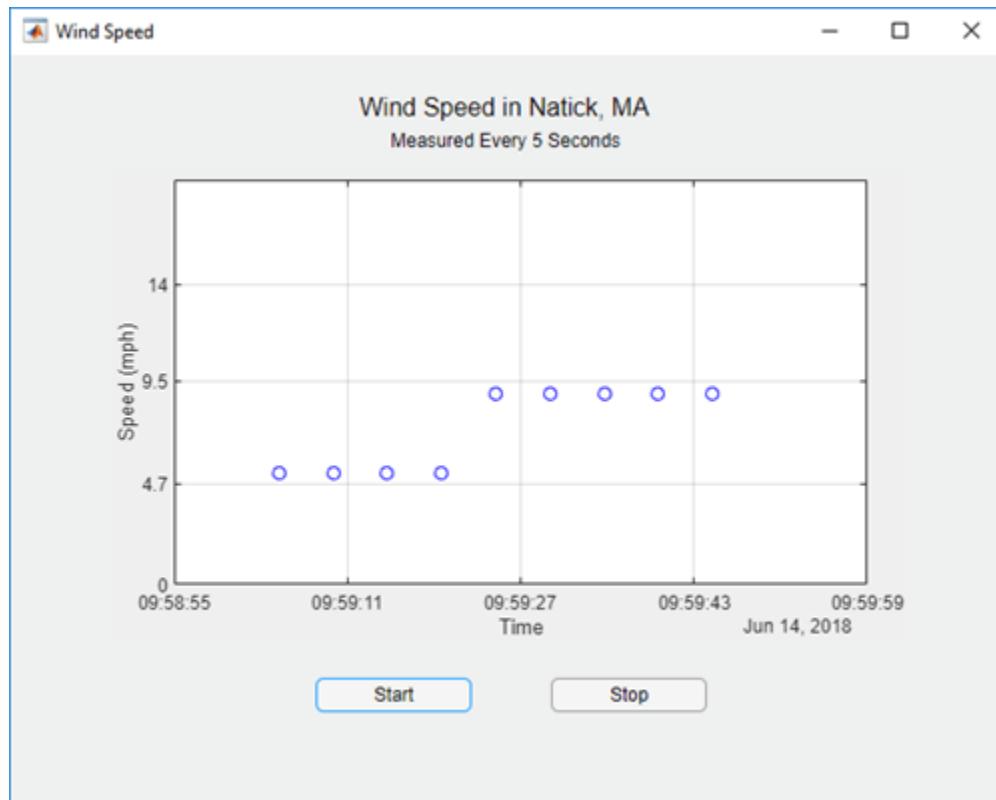
Properties
UIAxes

Create App with Timer Object that Queries Website Data

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

Classes
timer

Functions
webread

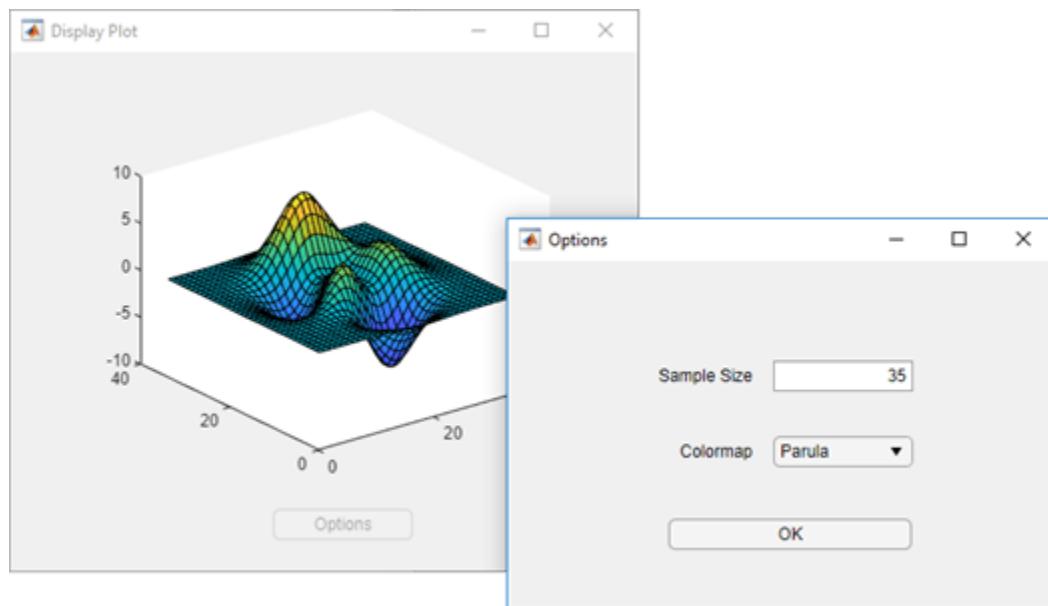
Properties
UIAxes

Share Data in Multiwindow Apps

This example shows how to 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



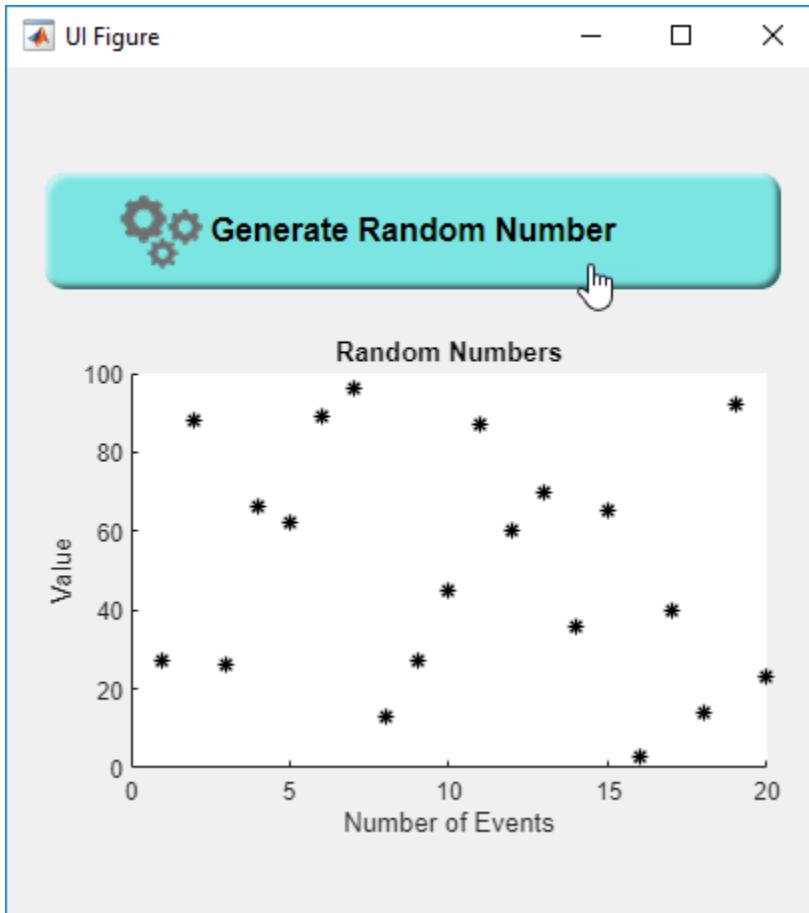
See Also

Related Examples

- “Create Multiwindow Apps in App Designer” on page 7-11
- “Startup Tasks and Input Arguments in App Designer” on page 7-8
- “Reuse Code Using Helper Functions” on page 7-20

Display HTML Elements Styled by a Cascading Style Sheet

This app shows how to reference supporting files from your HTML file, like a Cascading Style Sheet and an image used by the CSS file. This app also demonstrates how to plot data in MATLAB® that is generated in JavaScript® when an HTML button is clicked.



See Also

Functions
uihtml

Properties
HTML Properties

More About

- “Create HTML File That Can Trigger or Respond to Data Changes” on page 5-23

Keyboard Shortcuts

App Designer Keyboard Shortcuts

In this section...

- “Shortcuts Available Throughout App Designer” on page 9-2
- “Component Browser Shortcuts” on page 9-2
- “Design View Shortcuts” on page 9-3
- “Code View Shortcuts” on page 9-7

Shortcuts Available Throughout App Designer

Action	Key or Keys
Run the active app.	F5
Save the active app.	Ctrl+S
Save the active app, allowing you to specify a new file name. (Save as)	Ctrl+Shift+S
Open a previously saved app.	Ctrl+O
Redo an undone modification, returning it to the changed state.	Ctrl+Y or, in the design area only, Ctrl+Shift+Z
Undo a modification, returning it to the previous state.	Ctrl+Z
Alternate between design and code view.	Shift + F7 If debugging is in progress, this shortcut does not change the view.
Quit App Designer.	Ctrl+Q

Component Browser Shortcuts

These shortcuts are available in the **Component Browser**, in both code view and design view

Action	Key or Keys
Select multiple components.	Hold down the Ctrl key as you click each component that you want to include in the multiselection.
Deselect a component from multiselection.	Hold down the Ctrl key as you click each component that you want to remove from a multiselection.
Navigate from clicked component to the previous or next component listed in the code browser.	Up Arrow and Down Arrow
Edit code name of clicked component in the code browser.	F2 on Windows® and Linux® Enter on Mac

Design View Shortcuts

These shortcuts are available from the App Designer design view only.

- “Add Component Shortcuts” on page 9-3
- “Component, Group, and Text Selection Shortcuts” on page 9-3
- “Group and Ungroup Components Shortcuts” on page 9-4
- “Component and Group Move Shortcuts” on page 9-4
- “Component Resize Shortcuts” on page 9-4
- “Component Copy, Duplicate, and Delete Shortcuts” on page 9-4
- “Design Area Grid Shortcuts” on page 9-5
- “Component Alignment Shortcuts” on page 9-5
- “Change Font Characteristics Shortcuts” on page 9-5
- “Menu Component Shortcuts” on page 9-6
- “Tab Component Shortcuts” on page 9-6
- “Navigate Canvas Shortcuts” on page 9-6

Add Component Shortcuts

Action	Shortcut
Add component and associated label (if any) to canvas.	Click the component and hold down the mouse key to drag the component from the Component Library on the left into the design area.
Add component only to canvas.	Hold down the Ctrl key, click the component, and drag it from the Component Library on the left into the design area.

Component, Group, and Text Selection Shortcuts

Action	Key or Keys
Move the selection to the next component, or container in the design area tab key navigation sequence.	Tab
Move the selection to the previous component or container in the design area tab key navigation sequence.	Shift+Tab
Selects all components on the canvas, with one exception. If any of the components are grouped, the group is selected, not the individual components within the grouping.	Ctrl+A
Clear a component selection. Press again to reselect the component.	Shift+Click or Ctrl+Click
In the property editor or in-place editing, select all text in a text input field.	Ctrl+A
Select group containing a component.	Alt+Click a component

Group and Ungroup Components Shortcuts

Select the components that you want to group, and then press **Ctrl+G**. All components to be grouped must have the same parent component.

Action	Key or Keys
Group selected components.	Ctrl+G
Ungroup components in selected group.	Ctrl+Shift+G

Component and Group Move Shortcuts

This table summarizes the keyboard shortcuts for moving selected components and groups.

Action	Key or Keys
Move down 1 pixel.	Down Arrow
Move left 1 pixel.	Left Arrow
Move right 1 pixel.	Right Arrow
Move up 1 pixel.	Up Arrow
Move down 10 pixels.	Shift+Down Arrow
Move left 10 pixels.	Shift+Left Arrow
Move right 10 pixels.	Shift+Right Arrow
Move up 10 pixels.	Shift+Up Arrow
Cancel an in-progress operation.	Escape

Component Resize Shortcuts

Action	Key
Resize component while maintaining aspect ratio.	Press and hold down the Shift key before you begin to drag the component resize handle.
Resize component while keeping center location unchanged.	Press and hold down the Ctrl 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 Ctrl and Shift keys before you begin to drag the component resize handle.
Cancel an in-progress resize operation.	Escape

Component Copy, Duplicate, and Delete Shortcuts

Action	Key or Keys
Copy selected components and groups to the clipboard.	Ctrl+C
Duplicate the selected components and groups (without copying them to the clipboard).	Ctrl+D , or hold down the Ctrl key and drag the component.
Cut the selected components and groups from the design area onto the clipboard.	Ctrl+X

Action	Key or Keys
Delete the selected components and groups from the design area.	Backspace or Delete
Paste components and groups from the clipboard into the design area or a 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.	Ctrl+V

Design Area Grid Shortcuts

Action	Keys
Toggle grid on and off.	Alt+G
Toggle snap to grid on and off.	Alt+P
Increase grid interval by 5 pixels.	Alt+Page Up
Decrease grid interval by 5 pixels.	Alt+Page Down

Component Alignment Shortcuts

Action	Keys
Align selected components and groups on their left edges.	Ctrl+Alt+1
Align selected components and groups on their horizontal centers.	Ctrl+Alt+2
Align selected components and groups on their right edges.	Ctrl+Alt+3
Align selected components and groups on their top edges.	Ctrl+Alt+4
Align selected components and groups on their vertical middle.	Ctrl+Alt+5
Align selected components and groups on their bottom edges.	Ctrl+Alt+6

Change Font Characteristics Shortcuts

Action	Key or Keys
Toggle the font weight of selected components <i>and their children</i> between normal and bold.	Ctrl+B
Toggle the font angle of selected components <i>and their children</i> between normal and italic.	Ctrl+I

Action	Key or Keys
<p>Decrease the value of the <code>FontSize</code> property of the selected components <i>and their children</i> by one step.</p> <p>Font size steps are: 8, 9, 10, 11, 12, 14, 16, 18, 20, 22, 24, 26, 28, 36, 48, 72.</p>	Ctrl+ [
<p>Increase the value of the <code>FontSize</code> property of the selected components <i>and their children</i> by one step.</p> <p>Font size steps are: 8, 9, 10, 11, 12, 14, 16, 18, 20, 22, 24, 26, 28, 36, 48, 72.</p>	Ctrl+]

Menu Component Shortcuts

Action	Key or Keys
Add a menu item below the current item. The new menu item appears at the end of the list.	Enter
Add an item to the right of selected item.	Shift+Enter
Delete the current item.	Delete
Commit text changes and navigate to the next item.	Any Arrow key
Select the first or last item at the level of the selected item.	Home End
Move the selected child menu item higher or lower in the list.	Ctrl+Shift+Up Arrow Ctrl+Shift+Down Arrow
Move the selected top-level menu item to the left or right.	Ctrl+Shift+Left Arrow Ctrl+Shift+Right Arrow
Move the selected item to the beginning or end of the list.	Ctrl+Shift+Home Ctrl+Shift+End

Tab Component Shortcuts

Action	Key or Keys
Move the selected tab to the left or right.	Ctrl+Shift+Left Arrow Ctrl+Shift+Right Arrow
Move the selected tab to the beginning or end.	Ctrl+Shift+Home Ctrl+Shift+End

Navigate Canvas Shortcuts

Action	Shortcut
Zoom in on the canvas.	Ctrl+Plus
Zoom out on the canvas.	Ctrl+Minus
Reset the canvas zoom to default.	Ctrl+Alt+0

Action	Shortcut
Pan on the canvas.	Click and drag with the middle mouse button, or hold Space while clicking and dragging with the left mouse button.

Code View Shortcuts

These shortcuts are available only from the App Designer code view, within the editor.

- “Code Indenting Shortcuts” on page 9-7
- “Cut, Copy, and Paste Code Shortcuts” on page 9-7
- “Find Code Shortcuts” on page 9-7
- “Code Browser Shortcuts” on page 9-7
- “Code View Zoom Shortcuts” on page 9-8
- “Other App Designer Code Editor Shortcuts” on page 9-8

Code Indenting Shortcuts

Action	Key or Keys
Smart indent selected code.	Ctrl+I
Increase indent on current line of code or currently selected code.	Ctrl+]
Decrease indent on current line of code or currently selected code.	Ctrl+[

Cut, Copy, and Paste Code Shortcuts

Action	Key or Keys
Cut selected code.	Ctrl+X
Copy selected code.	Ctrl+C
Paste selected code.	Ctrl+V

Find Code Shortcuts

Action	Key or Keys
Find.	Ctrl+F
Find next.	F3
Find previous.	Shift+F3
Find selection.	Ctrl+F3

Code Browser Shortcuts

Action	Key or Keys
Delete callback.	Delete
Rename callback.	F2

Action	Key or Keys
Bring callback to focus and insert cursor.	Ctrl+D

Code View Zoom Shortcuts

Action	Shortcut
Zoom in on code editor.	Ctrl+Plus
Zoom out on code editor.	Ctrl-Minus
Reset code editor zoom to default.	Ctrl+Alt+0

Other App Designer Code Editor Shortcuts

Action	Key or Keys
Add comment to selected code.	Ctrl+R
Remove comment from selected code.	Ctrl+T
Evaluate selection.	F9
Open selection.	Ctrl+D
Go to specified line number.	Ctrl+G
Set or clear breakpoint.	F12

Create UIs Programmatically

- “Lay Out a UI Programmatically” on page 10-2
- “Write Callbacks for Apps Created Programmatically” on page 11-2
- “Create and Run a Simple figure-Based App” on page 2-2
- “Callbacks for Specific Components” on page 19-14
- “Share Data Among Callbacks” on page 12-2

Lay Out a Programmatic UI

- “Lay Out a UI Programmatically” on page 10-2
- “DPI-Aware Behavior in MATLAB” on page 10-10

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

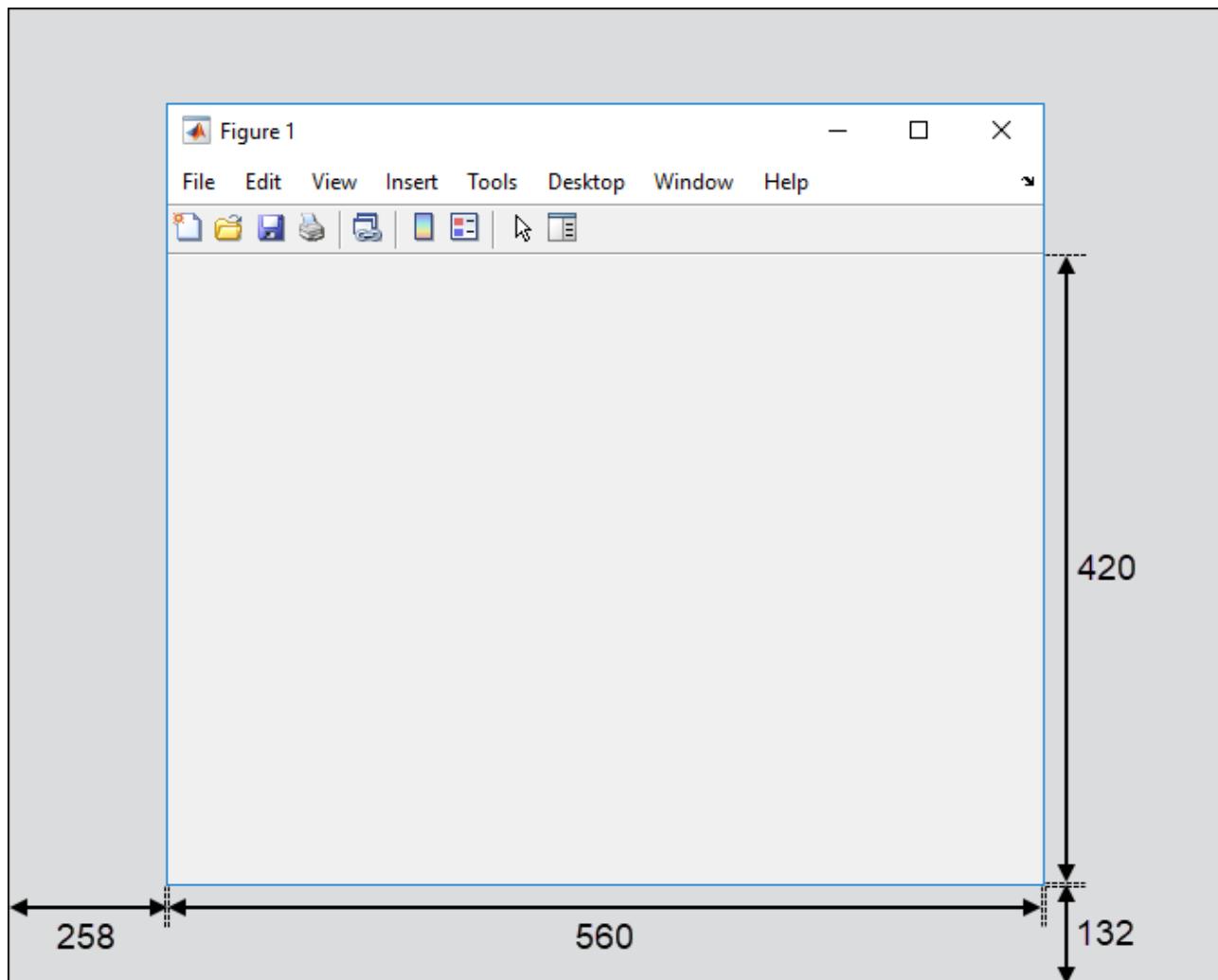
A UI layout consists of a figure and one or more components that you place inside the figure. Accurate placement and sizing of each component involves setting certain properties and understanding how the inner and outer boundaries of the figure relate to each other.

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:

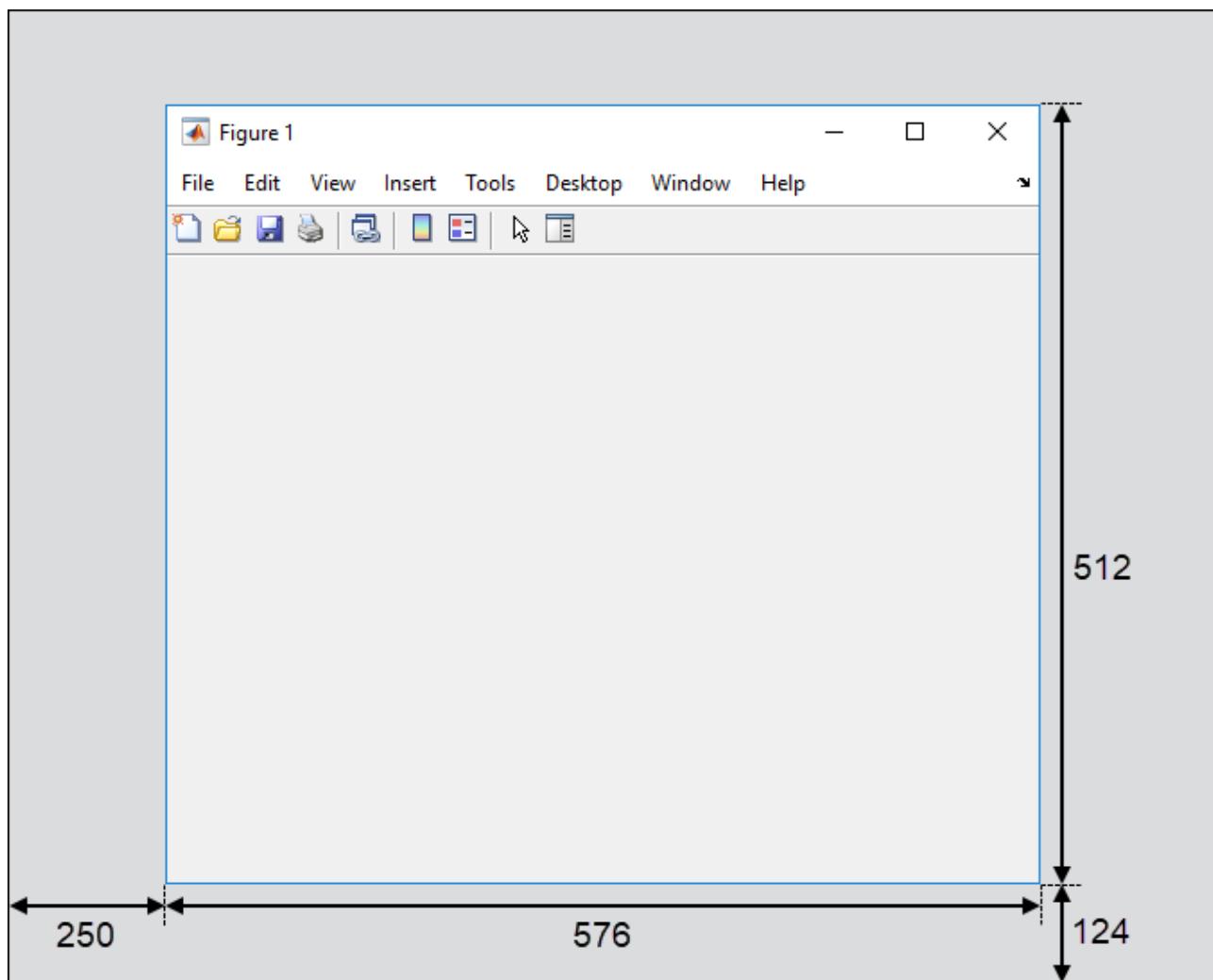
```
f = figure('Position',[258 132 560 420]);
```



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 `UIControl`, `Table`, and `Panel` objects 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';

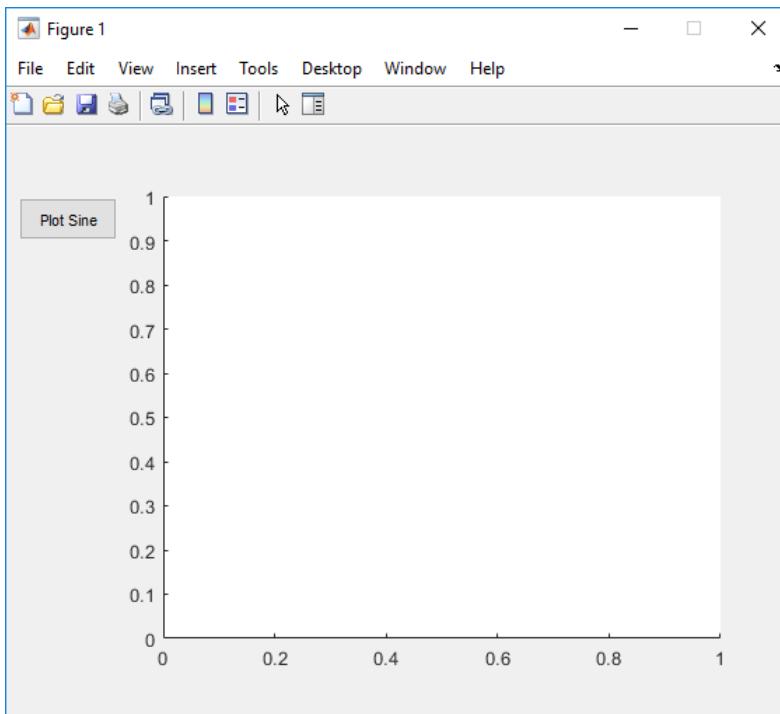
    function hs = addcomponents
        % add components, save handles in a struct
        hs.fig = figure('Visible','off',...
            'Resize','off',...
            'Tag','fig');
        hs.btn = uicontrol(hs.fig,'Position',[10 340 70 30],...
            'String','Plot Sine',...
            'Tag','button',...
            '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 panel or button group, 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 10-5. 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);
        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

```

```
    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 (panels, button groups, and tabs) 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 panel by setting the `Parent` property of the axes to be the panel.

The `Children` property of a `Panel`, `ButtonGroup`, or `Tab` object lists the child objects inside the container according to their stacking order.

See Also

Related Examples

- “[Lay Out Apps in App Designer Design View](#)” on page 6-2
- “[Manage Resizable Apps in App Designer](#)” on page 6-11
- “[DPI-Aware Behavior in MATLAB](#)” on page 10-10

DPI-Aware Behavior in MATLAB

In this section...

["Visual Appearance" on page 10-10](#)

["Using Object Properties" on page 10-12](#)

["Using print, getframe, and publish Functions" on page 10-13](#)

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

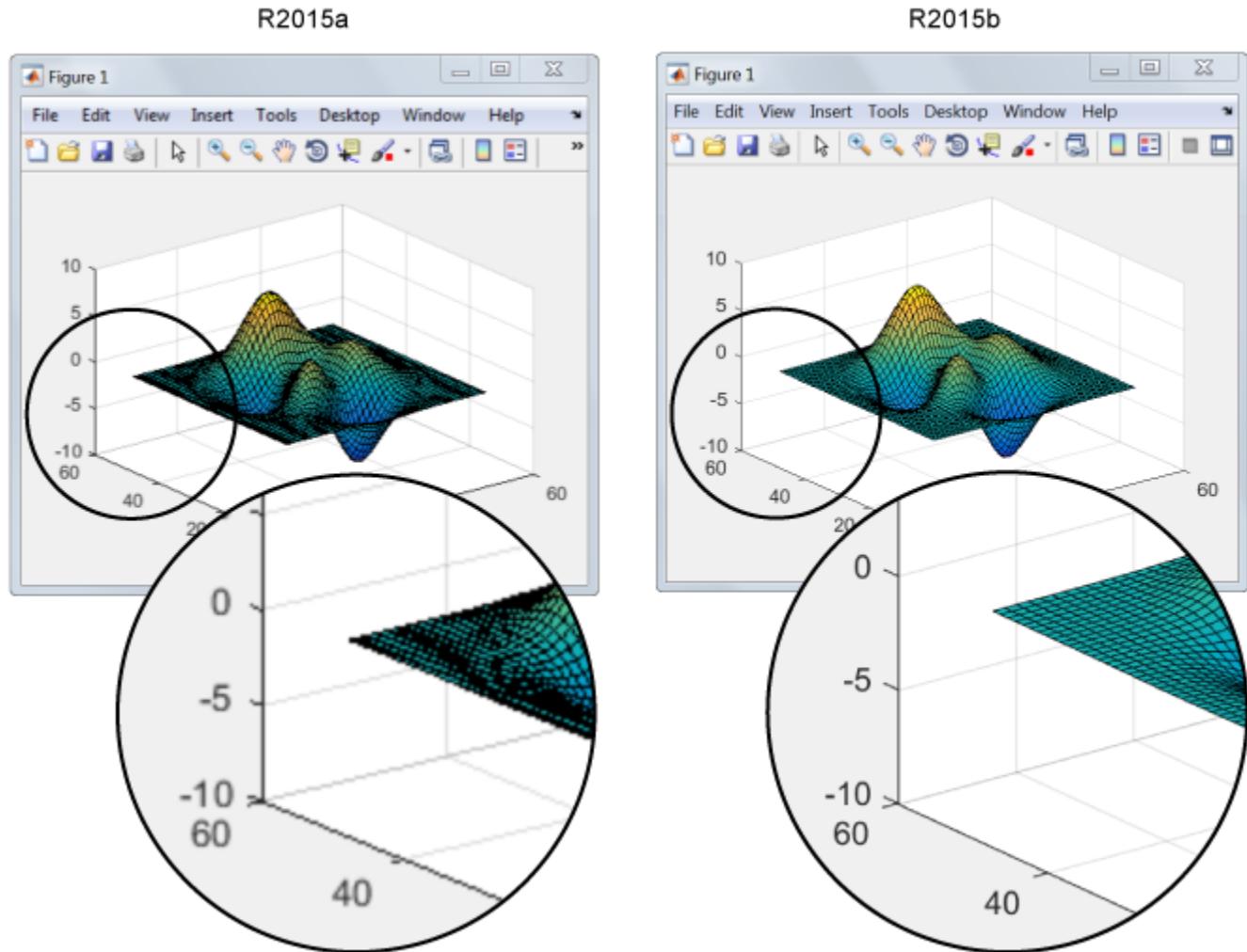
DPI-aware behavior does not apply to Linux systems.

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

Write Callbacks for Apps Created Programmatically

In this section...

[“Callbacks for Different User Actions” on page 11-2](#)

[“How to Specify Callback Property Values” on page 11-3](#)

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>
<code>CloseRequestFcn</code>	The figure closes.	<code>figure</code>
<code>CreateFcn</code>	Callback executes when MATLAB creates the object, but before it is displayed.	<code>axes</code> , <code>figure</code> , <code>uibuttongroup</code> , <code>uicontextmenu</code> , <code>uicontrol</code> , <code>uimenu</code> , <code>uipushtool</code> , <code>uipanel</code> , <code>uitable</code> , <code>uitoggletool</code> , <code>uicontrol</code>
<code>DeleteFcn</code>	Callback executes just before MATLAB deletes the figure.	<code>axes</code> , <code>figure</code> , <code>uibuttongroup</code> , <code>uicontextmenu</code> , <code>uicontrol</code> , <code>uimenu</code> , <code>uipushtool</code> , <code>uipanel</code> , <code>uitable</code> , <code>uitoggletool</code> , <code>uicontrol</code>
<code>KeyPressFcn</code>	End user presses a keyboard key while the pointer is on the object.	<code>figure</code> , <code>uicontrol</code> , <code>uipanel</code> , <code>uipushtool</code> , <code>uitable</code> , <code>uicontrol</code>

Callback Property	User Action	Components That Use This Property
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
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 11-3.
- “Specify a Cell Array” on page 11-4. 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 11-5.
- “Specify a Character Vector Containing MATLAB Commands (Not Recommended)” on page 11-5

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
<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>
<code>CellEditCallback</code>	<code>uitable</code>
<code>CellSelectionCallback</code>	<code>uitable</code>

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 11-3. 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 19-14
- “Share Data Among Callbacks” on page 12-2
- “Interrupt Callback Execution” on page 13-2
- “Anonymous Functions”
- “Write Callbacks in App Designer” on page 7-15

Manage Application-Defined Data

Share Data Among Callbacks

In this section...
"Overview of Data Sharing Techniques" on page 12-2
"Store Data in UserData or Other Object Properties" on page 12-2
"Store Data as Application Data" on page 12-3
"Create Nested Callback Functions" on page 12-5
"Store Data Using the guidata Function" on page 12-5

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 UserData or Other Object Properties" on page 12-2	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 12-3	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.
"Create Nested Callback Functions" on page 12-5	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. Can share multiple variables.
"Store Data Using the guidata Function" on page 12-5	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 User Data in Apps Created With Traditional Figures

Use dot notation to set and query properties of the traditional figure.

```

hfig = figure;
figname = hfig.Name;
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;
end

function button_callback(hObject,eventdata)
    h = findobj('Tag','slider1');
    data = 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.

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

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);
end

function button_callback(hObject,eventdata)
currentval = getappdata(hObject.Parent,'slidervalue');
diffval = getappdata(hObject.Parent,'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.

Create Nested Callback Functions

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;
        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 user clicks the push button, the `button_callback` function displays the values stored in `data`.

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** to Store and Share Data

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;
    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.

See Also

Related Examples

- “Nested Functions”
- “Interrupt Callback Execution” on page 13-2
- “Write Callbacks for Apps Created Programmatically” on page 11-2
- “Write Callbacks in App Designer” on page 7-15
- “Share Data Within App Designer Apps” on page 7-23

Manage Callback Execution

Interrupt Callback Execution

In this section...

- ["How to Control Interruption" on page 13-2](#)
- ["Callback Behavior When Interruption is Allowed" on page 13-2](#)
- ["Example" on page 13-2](#)

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

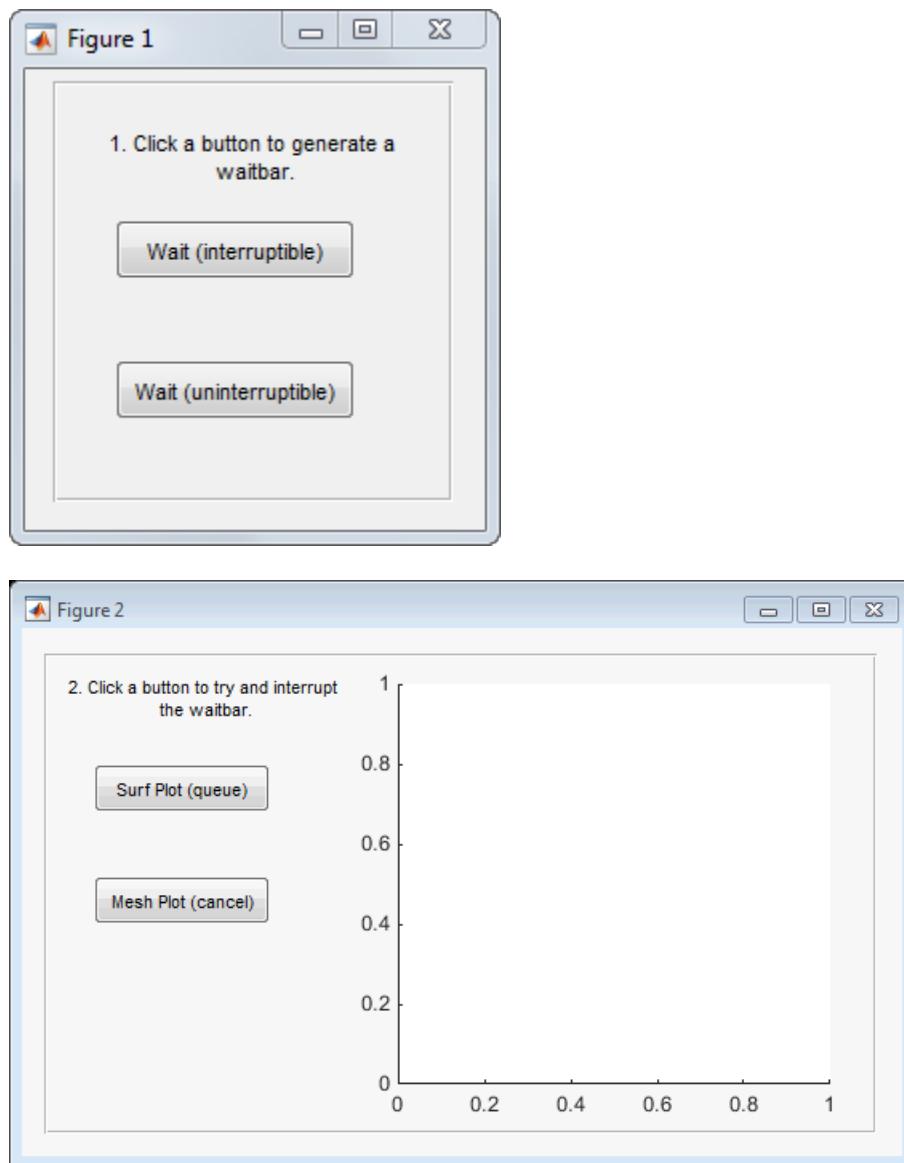
1 In MATLAB, set your current folder to one in which you have write access.

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 11-2
- “Schedule Command Execution Using Timer”
- “Finding Code Bottlenecks”

Examples of Programmatic Apps

Programmatic App that Displays a Table

This example shows how to display 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 UI Component Within a Figure

The `uitable` function creates an empty UI table in the figure.

```
fig = uifigure('Position',[100 100 752 250]);
uit = uitable('Parent',fig,'Position',[25 50 700 200]);
```

Create a Table Containing Mixed Data Types

Load sample patients data that contains mixed data types and store it in a table array. To have data appear as a drop-down list in the cells of the table component, convert a cell array variable to a categorical array. To display the data in the table UI component, specify the table array as the value of the `Data` property.

```
load patients
t = table(LastName,Age,Weight,Height,Smoker, ...
    SelfAssessedHealthStatus);
t.SelfAssessedHealthStatus = categorical(t.SelfAssessedHealthStatus, ...
    {'Poor','Fair','Good','Excellent'},'Ordinal',true);

uit.Data = t;
```

LastName	Age	Weight	Height	Smoker	SelfAssessedHealthStatus
Smith	38	176	71	<input checked="" type="checkbox"/>	Excellent
Johnson	43	163	69	<input type="checkbox"/>	Fair
Williams	38	131	64	<input type="checkbox"/>	Good
Jones	40	133	67	<input type="checkbox"/>	Fair
Brown	49	119	64	<input type="checkbox"/>	Good
Davis	46	142	68	<input type="checkbox"/>	Good
Miller	33	142	64	<input checked="" type="checkbox"/>	Good
Wilson	40	180	68	<input type="checkbox"/>	Good

Customize the Display

You can customize the display of a UI table in several ways. Use the `ColumnName` property to add column headings.

```
uit.ColumnName = {'Last Name','Age','Weight',...
    'Height','Smoker','Health Status'};
```

Last Name	Age	Weight	Height	Smoker	Health Status
Smith	38	176	71	<input checked="" type="checkbox"/>	Excellent
Johnson	43	163	69	<input type="checkbox"/>	Fair
Williams	38	131	64	<input type="checkbox"/>	Good
Jones	40	133	67	<input type="checkbox"/>	Fair
Brown	49	119	64	<input type="checkbox"/>	Good
Davis	46	142	68	<input type="checkbox"/>	Good
Miller	33	142	64	<input checked="" type="checkbox"/>	Good
Wilson	40	180	68	<input type="checkbox"/>	Good

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 use 'auto' to let MATLAB set the width based on the contents.

```
uit.ColumnWidth = {'auto',75,'auto','auto','auto',100};
```

Last Name	Age	Weight	Height	Smoker	Health Status
Smith	38	176	71	<input checked="" type="checkbox"/>	Excellent
Johnson	43	163	69	<input type="checkbox"/>	Fair
Williams	38	131	64	<input type="checkbox"/>	Good
Jones	40	133	67	<input type="checkbox"/>	Fair
Brown	49	119	64	<input type="checkbox"/>	Good
Davis	46	142	68	<input type="checkbox"/>	Good
Miller	33	142	64	<input checked="" type="checkbox"/>	Good
Wilson	40	180	68	<input type="checkbox"/>	Good

Add numbered row headings by setting the `RowName` property to 'numbered'.

```
uit.RowName = 'numbered';
```

	Last Name	Age	Weight	Height	Smoker	Health Status	
1	Smith	38	176	71	<input checked="" type="checkbox"/>	Excellent	
2	Johnson	43	163	69	<input type="checkbox"/>	Fair	
3	Williams	38	131	64	<input type="checkbox"/>	Good	
4	Jones	40	133	67	<input type="checkbox"/>	Fair	
5	Brown	49	119	64	<input type="checkbox"/>	Good	
6	Davis	46	142	68	<input type="checkbox"/>	Good	
7	Miller	33	142	64	<input checked="" type="checkbox"/>	Good	
8	Wilson	40	180	68	<input type="checkbox"/>	Good	

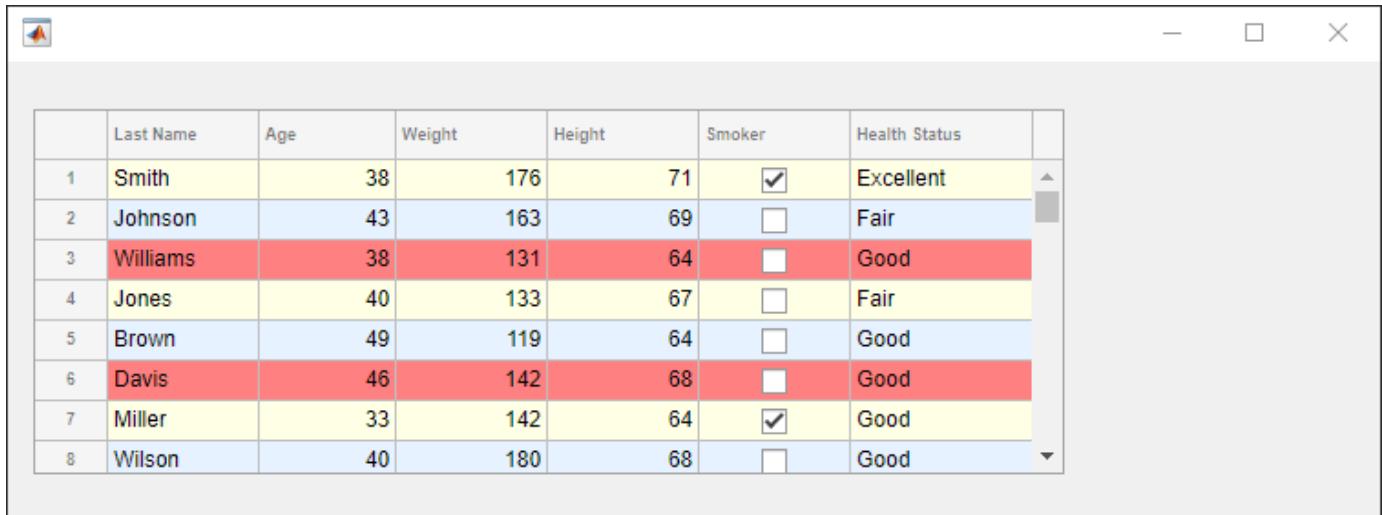
Reposition and resize the table using the `Position` property.

```
uit.Position = [15 25 565 200];
```

	Last Name	Age	Weight	Height	Smoker	Health Status	
1	Smith	38	176	71	<input checked="" type="checkbox"/>	Excellent	
2	Johnson	43	163	69	<input type="checkbox"/>	Fair	
3	Williams	38	131	64	<input type="checkbox"/>	Good	
4	Jones	40	133	67	<input type="checkbox"/>	Fair	
5	Brown	49	119	64	<input type="checkbox"/>	Good	
6	Davis	46	142	68	<input type="checkbox"/>	Good	
7	Miller	33	142	64	<input checked="" type="checkbox"/>	Good	
8	Wilson	40	180	68	<input type="checkbox"/>	Good	

By default, table UI components use row striping and cycle through the specified background colors until the end of the table is reached. If you set the `RowStriping` property to 'off', the table UI component will just use the first color specified in the `BackgroundColor` property for all rows. Here, leave row striping 'on' and set three different colors for the `BackgroundColor` property.

```
uit.BackgroundColor = [1 1 .9; .9 .95 1;1 .5 .5];
```



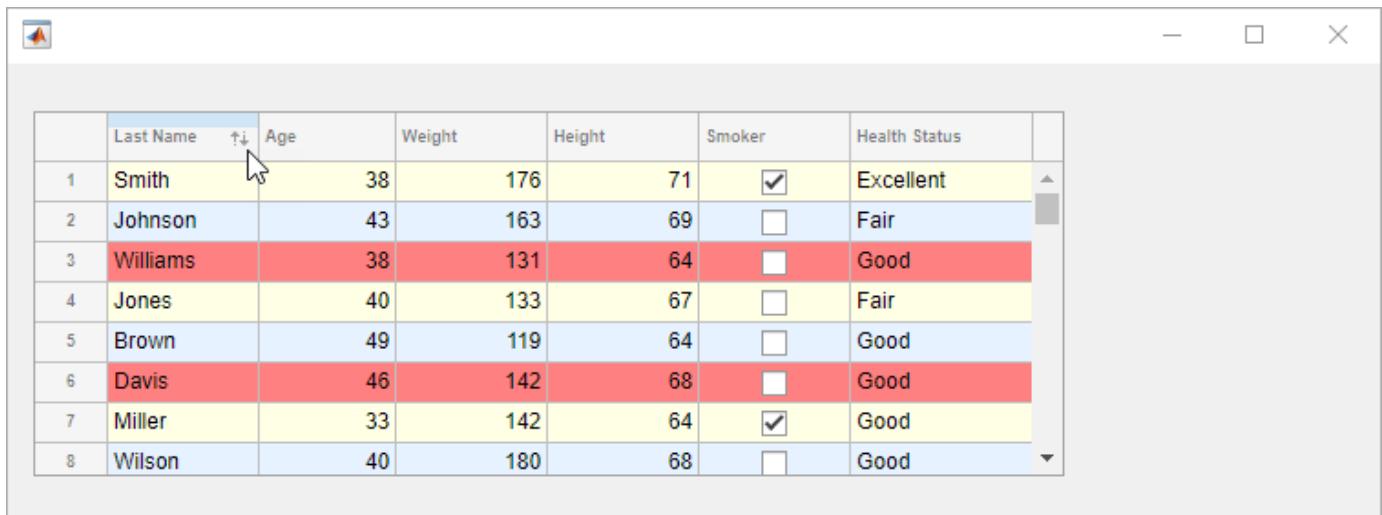
A screenshot of a MATLAB application window displaying a 8x7 table. The columns are labeled "Last Name", "Age", "Weight", "Height", "Smoker", and "Health Status". The "Smoker" column contains checkboxes. The "Health Status" column contains text labels: "Excellent", "Fair", "Good". The "Age" column has a sorting arrow icon above it. The table rows are numbered 1 through 8. The data is as follows:

	Last Name	Age	Weight	Height	Smoker	Health Status
1	Smith	38	176	71	<input checked="" type="checkbox"/>	Excellent
2	Johnson	43	163	69	<input type="checkbox"/>	Fair
3	Williams	38	131	64	<input type="checkbox"/>	Good
4	Jones	40	133	67	<input type="checkbox"/>	Fair
5	Brown	49	119	64	<input type="checkbox"/>	Good
6	Davis	46	142	68	<input type="checkbox"/>	Good
7	Miller	33	142	64	<input checked="" type="checkbox"/>	Good
8	Wilson	40	180	68	<input type="checkbox"/>	Good

Enable Column Sorting and Restrict Editing of Cell Values

Make all the columns sortable by setting the `ColumnSortable` property to `true`. If a column is sortable, arrows appear in the header when you hover your mouse over it.

```
uit.ColumnSortable = true;
```

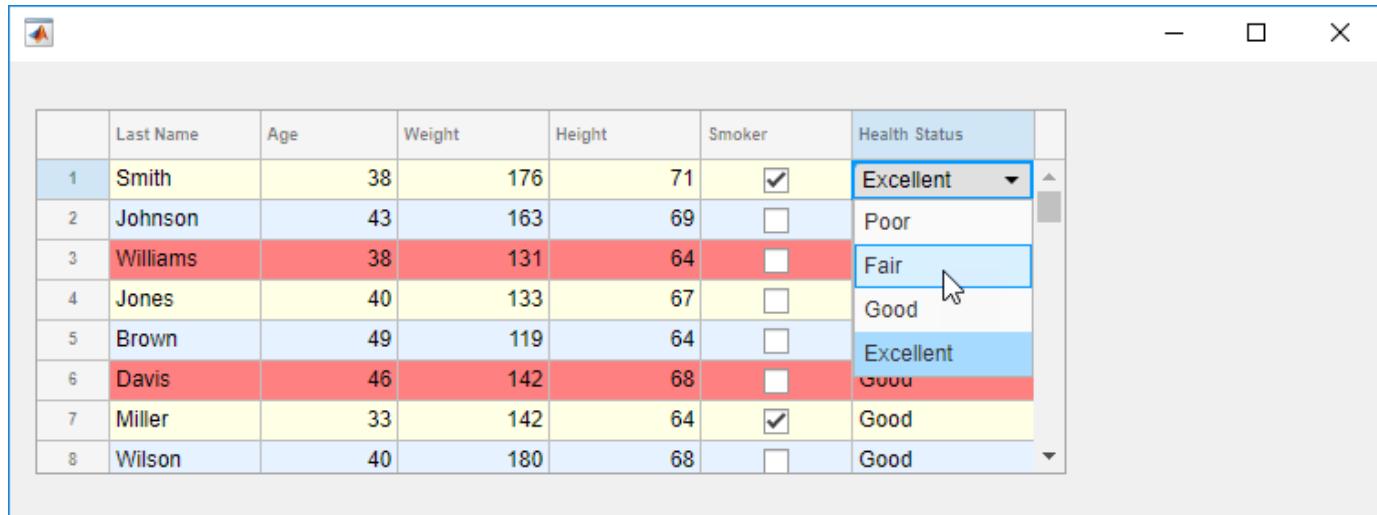


A screenshot of a MATLAB application window displaying a 8x7 table. The columns are labeled "Last Name", "Age", "Weight", "Height", "Smoker", and "Health Status". The "Smoker" column contains checkboxes. The "Health Status" column contains text labels: "Excellent", "Fair", "Good". The "Age" column has a sorting arrow icon above it. The table rows are numbered 1 through 8. The data is as follows:

	Last Name	Age	Weight	Height	Smoker	Health Status
1	Smith	38	176	71	<input checked="" type="checkbox"/>	Excellent
2	Johnson	43	163	69	<input type="checkbox"/>	Fair
3	Williams	38	131	64	<input type="checkbox"/>	Good
4	Jones	40	133	67	<input type="checkbox"/>	Fair
5	Brown	49	119	64	<input type="checkbox"/>	Good
6	Davis	46	142	68	<input type="checkbox"/>	Good
7	Miller	33	142	64	<input checked="" type="checkbox"/>	Good
8	Wilson	40	180	68	<input type="checkbox"/>	Good

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

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



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.

```
uit.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))
    tableData = src.Data;
    tableData{eventdata.Indices(1),eventdata.Indices(2)} = eventdata.PreviousData;
    src.Data = tableData;
    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(uit)

BackgroundColor: [3x3 double]
BeingDeleted: off
BusyAction: 'queue'
ButtonDownFcn: ''
CellEditCallback: @ageCheckCB
```

```
CellSelectionCallback: ''
    Children: [0x0 handle]
    ColumnEditable: [0 0 1 1 1 1]
    ColumnFormat: {}
    ColumnName: {6x1 cell}
    ColumnSortable: 1
    ColumnWidth: {'auto' [75] 'auto' 'auto' 'auto' [100]}
    ContextMenu: [0x0 GraphicsPlaceholder]
    CreateFcn: ''
        Data: [100x6 table]
    DeleteFcn: ''
    DisplayData: [100x6 table]
DisplayDataChangedFcn: ''
    Enable: 'on'
    Extent: [0 0 0 0]
    FontAngle: 'normal'
    FontName: 'Helvetica'
   FontSize: 12
    FontUnits: 'pixels'
    FontWeight: 'normal'
    ForegroundColor: [0 0 0]
    HandleVisibility: 'on'
    InnerPosition: [15 25 565 200]
    Interruptible: on
    KeyPressFcn: ''
    KeyReleaseFcn: ''
        Layout: [0x0 matlab.ui.layout.LayoutOptions]
    OuterPosition: [15 25 565 200]
        Parent: [1x1 Figure]
    Position: [15 25 565 200]
RearrangeableColumns: off
    RowName: 'numbered'
    RowStriping: on
StyleConfigurations: [0x3 table]
    Tag: ''
    Tooltip: ''
        Type: 'uitable'
        Units: 'pixels'
    UserData: []
    Visible: on
```


Developing Classes of UI Component Objects

- “Custom UI Component Development Overview” on page 15-2
- “Manage Properties of Custom UI Components” on page 15-9
- “Configure Custom UI Components for App Designer” on page 15-17
- “Customize Properties of HTML UI Component” on page 15-25

Custom UI Component Development Overview

To create custom UIs and visualizations, you can combine multiple graphics and UI objects, change their properties, or call additional functions. In R2020a and earlier releases, a common way to store your customization code and share it with others is to write a script or a function.

Starting in R2020b, instead of a script or function, you can create a class implementation for your UI components by defining a subclass of the `ComponentContainer` base class. Creating a class has these benefits:

- Easy customization — When users want to customize an aspect of your UI component, they can set a property rather than having to modify and rerun your code. Users can modify properties at the command line or inspect them in the Property Inspector.
- Encapsulation — Organizing your code in this way allows you to hide implementation details from your users. You implement methods that perform calculations and manage the underlying graphics objects.

Structure of a UI Component Class

A UI component class has several required parts, and several more that are optional.

In the first line of a UI component class, specify the `matlab.ui.componentcontainer.ComponentContainer` class as the superclass. For example, the first line of a class called `ColorSelector` looks like this:

```
classdef ColorSelector < matlab.ui.componentcontainer.ComponentContainer
```

In addition to specifying the superclass, include the following components in your class definition. Some components are required, while other components are either recommended or optional.

Component	Description
Public property block on page 15-3 (recommended)	This block defines all the properties that users have access to. Together, these properties make up the user interface of your UI component.
Private property block on page 15-3 (recommended)	This block defines the underlying graphics objects and other implementation details that users cannot access. In this block, set these attribute values: <ul style="list-style-type: none">• <code>Access = private</code>• <code>Transient</code>• <code>NonCopyable</code>

Component	Description
Events block on page 15-4 (optional)	<p>This block defines the events that this UI component will trigger.</p> <p>In this block, set these attribute values:</p> <ul style="list-style-type: none"> • <code>HasCallbackProperty</code> • <code>NotifyAccess = protected</code> <p>When you set the <code>HasCallbackProperty</code> attribute, MATLAB creates a public property for each event in the block. The public property stores the user-provided callback to execute when the event fires.</p>
<code>setup</code> method on page 15-5 (required)	<p>This method sets the initial state of the UI component. It executes once when MATLAB constructs the object.</p> <p>Define this method in a protected block so that only your class can execute it.</p>
update method on page 15-5 (required)	<p>This method updates the underlying objects in your UI component. It executes under the following conditions:</p> <ul style="list-style-type: none"> • During the next <code>drawnow</code> execution after the user changes one or more property values • When an aspect of the user's graphics environment changes (such as the size) <p>Define this method in the same protected block as the <code>setup</code> method.</p>

Constructor Method

You do not have to write a constructor method for your class, because it inherits one from the `ComponentContainer` base class. The inherited constructor accepts optional input arguments: a parent container and any number of name-value pair arguments for setting properties on the UI component. For example, if you define a class called `ColorSelector` that has the public properties `Value` and `ValueChangedFcn`, you can create an instance of your class using this code:

```
f = uifigure;
c = ColorSelector(f, 'Value', [1 1 0], 'ValueChangedFcn', @(o,e)disp('Changed'))
```

If you want to provide a constructor that has a different syntax or different behavior, you can define a custom constructor method. For an example of a custom constructor, see “Write Constructors for Chart Classes”.

Public and Private Property Blocks

Divide your class properties between at least two blocks:

- A public block for storing the components of the user-facing interface
- A private block for storing the implementation details that you want to hide

The properties that go in the public block store the input values provided by the user. For example, a UI component that allows a user to pick a color value might store the color value in a public property. Since the property name-value pair arguments are optional inputs to the implicit constructor method, the recommended approach is to initialize the public properties to default values.

The properties that go in the private block store the underlying graphics objects that make up your UI component, in addition to any calculated values that you want to store. Eventually, your class will use the data in the public properties to configure the underlying objects. Set the `Transient` and `NonCopyable` attributes for the private block to avoid storing redundant information if the user copies or saves an instance of the UI component.

For example, here are the property blocks for a UI component that allows a user to pick a color value. The public property block stores the value that the user can control: the color value. The private property block stores the grid layout manager, button, and edit field objects.

```
properties
    Value {validateattributes(Value, ...
        {'double'},{'<='},1,'>=',0,'size',[1 3])} = [1 0 0];
end

properties(Access = private,Transient,NonCopyable)
    Grid matlab.ui.container.GridLayout
    Button matlab.ui.control.Button
    EditField matlab.ui.control.EditField
end
```

Event Block

You optionally can add a third block for events that the UI component fires.

Create a public property for each event in the block by specifying the `HasCallbackProperty` attribute. The public property stores the user-provided callback to execute when the event fires. The name of the public property is the name of the event appended with the letters `Fcn`. For example, a UI component that allows a user to pick a color value might define the event `ValueChanged`, which generates the corresponding public property `ValueChangedFcn`. Use the `notify` method to fire the event and execute the callback in the property.

For example, here is the event block for a UI component that allows a user to pick a color value.

```
events (HasCallbackProperty, NotifyAccess = protected)
    ValueChanged
end
```

When the user picks a color value, call the `notify` method to fire the `ValueChanged` event and execute the callback in the `ValueChangedFcn` property.

```
function getColorFromUser(obj)
    c = uisetcolor(obj.Value);
    if (isscalar(c) && (c == 0))
        return;
    end

    % Update the Value property
    oldValue = obj.Value;
    obj.Value = c;

    % Execute user callbacks and listeners
    notify(obj,'ValueChanged');
end
```

When a user creates an instance of the UI component, they can specify a callback to execute when the color value changes using the generated public property.

```
f = uifigure;
c = ColorSelector(f,'ValueChangedFcn',@(o,e)disp('Changed'))
```

For more information about specifying callbacks to properties, see “Write Callbacks for Apps Created Programmatically” on page 11-2.

Setup Method

Define a `setup` method for your class. A `setup` method executes once when MATLAB constructs the UI component object. Any property values passed as name-value pair arguments to the constructor method are assigned after this method executes.

Use the `setup` method to:

- Create graphics and UI objects that make up the component.
- Store the objects as private properties on the component object.
- Lay out and configure the objects.
- Wire up the objects to do something useful within the component.

To ensure that only your UI component class can execute the `setup` method, define it in a protected block.

Most UI object creation functions have an optional input argument for specifying the parent. When you call these functions from within a class method, you must specify the target parent. The `setup` method takes one argument that represents the instance of the UI component object being set up. Use this argument to specify the UI component as the target parent when calling an object creation function from within the `setup` method.

For example, consider a UI component that has these properties:

- One public property called `Value`
- Three private properties called `Grid`, `Button`, and `EditField`

The `setup` method calls the `uigridlayout`, `uieditfield`, and `uibutton` functions to create the underlying graphics object for each private property, specifying the instance of the UI component (`obj`) as the target parent.

```
function setup(obj)
    % Create grid layout to manage building blocks
    obj.Grid = uigridlayout(obj,[1 2], 'ColumnWidth', {'1x', 22}, ...
        'RowHeight', {'fit'}, 'ColumnSpacing', 2, 'Padding', 2);

    % Create edit field for entering color value
    obj.EditField = uieditfield(obj.Grid, 'Editable', false, ...
        'HorizontalAlignment', 'center');

    % Create button to confirm color change
    obj.Button = uibutton(obj.Grid, 'Text', char(9998), ...
        'ButtonPushedFcn', @(o,e) obj.getColorFromUser());
end
```

Update Method

Define an `update` method for your class. This method executes when your UI component object needs to change its appearance in response to a change in values.

Use the `update` method to reconfigure the underlying graphics objects in your UI component based on the new values of the public properties. Typically, this method does not determine which of the public properties changed. It reconfigures all aspects of the underlying graphics objects that depend on the public properties.

For example, consider a UI component that has these properties:

- One public property called `Value`
- Three private properties called `Grid`, `Button`, and `EditField`

The `update` method updates the `BackgroundColor` of the `EditField` and `Button` objects with the color stored in `Value`. The `update` method also updates the `EditField` object with a numeric representation of the color. This way, however `Value` is changed, the change becomes equally visible everywhere.

```
function update(obj)
    % Update edit field and button colors
    set([obj.EditField obj.Button], 'BackgroundColor', obj.Value, ...
        'FontColor', obj.getContrastingColor(obj.Value));

    % Update edit field display text
    obj.EditField.Value = num2str(obj.Value, '%0.2g');

end
```

There might be a delay between changing property values and seeing the results of those changes. The `update` method runs for the first time after the `setup` method runs and then it runs every time `drawnow` executes. The `drawnow` function automatically executes periodically, based on the state of the graphics environment in the user's MATLAB session. This periodic execution can lead to the potential delay.

Example: Color Selector UI Component

This example shows how to create a UI component for selecting a color, using the code discussed in other sections of this page. Create a class definition file named `ColorSelectorComponent.m` in a folder that is on the MATLAB path. Define the class by following these steps.

Step	Implementation
Derive from the <code>ComponentContainer</code> base class.	<code>classdef ColorSelector < matlab.ui.componentcontainer.ComponentContainer</code>
Define public properties.	<code>properties</code> <code>Value {validateattributes(Value, ...)</code> <code>{'double'},{'<='},1,'>=',0,'size',[1 3]}} = [1 0 0];</code> <code>end</code>
Define public events.	<code>events (HasCallbackProperty, NotifyAccess = protected)</code> <code>ValueChanged % ValueChangedFcn will be the generated callback property</code> <code>end</code>
Define private properties.	<code>properties (Access = private, Transient, NonCopyable)</code> <code>Grid matlab.ui.container.GridLayout</code> <code>Button matlab.ui.control.Button</code> <code>EditField matlab.ui.control.EditField</code> <code>end</code>

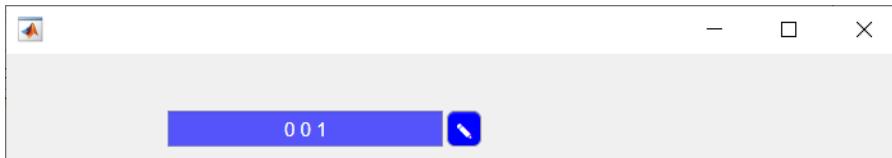
Step	Implementation
<p>Implement the <code>setup</code> method. In this case, call the <code>uigridlayout</code>, <code>uieditfield</code>, and <code>uibutton</code> functions to create <code>GridLayout</code>, <code>EditField</code>, and <code>Button</code> objects. Store those objects in the corresponding private properties.</p> <p>Specify the <code>getColorFromUser</code> method as the <code>ButtonPushedFcn</code> callback that is called when the button is pressed.</p>	<pre>methods (Access = protected) function setup(obj) % Grid layout to manage building blocks obj.Grid = uigridlayout(obj,[1,2],{'ColumnWidth',{'1x',22}, ... 'RowHeight',{'fit'},'ColumnSpacing',2,'Padding',2}); % Edit field for value display and button to launch uisetcolor obj.EditField = uieditfield(obj.Grid,'Editable',false, ... 'HorizontalAlignment','center'); obj.Button = uibutton(obj.Grid,'Text',char(9998), ... 'ButtonPushedFcn',@(o,e) obj.getColorFromUser()); end</pre>
<p>Implement the <code>update</code> method. In this case, update the background color of the underlying objects and the text in the edit field to show the color value.</p>	<pre>function update(obj) % Update edit field and button colors set([obj.EditField obj.Button], 'BackgroundColor',obj.Value, ... 'FontColor',obj.getContrastingColor(obj.Value)); % Update the display text obj.EditField.Value = num2str(obj.Value,'%0.2g'); end</pre>
<p>Wire the callbacks and other pieces together using private methods.</p> <p>When the <code>getColorFromUser</code> method is triggered by a button press, call the <code>uisetcolor</code> function to open the color picker and then call the <code>notify</code> function to execute the user callback and listener functions.</p> <p>When the <code>getContrastingColor</code> method is called by the <code>update</code> method, calculate whether black or white text is more readable on the new background color.</p>	<pre>methods (Access = private) function getColorFromUser(obj) c = uisetcolor(obj.Value); if (isscalar(c) && (c == 0)) return; end % Update the Value property obj.Value = c; % Execute user callbacks and listeners notify(obj, 'ValueChanged'); end function contrastColor = getContrastingColor(~,color) % Calculate opposite color c = color * 255; contrastColor = [1 1 1]; if (c(1)*.299 + c(2)*.587 + c(3)*.114) > 186 contrastColor = [0 0 0]; end end end</pre>

Next, create an instance of the UI component by calling the implicit constructor method with a few of the public properties. Specify a callback to display the words `Color changed` when the color value changes.

```
h = ColorSelector('Value', [1 1 0]);
h.ValueChangedFcn = @(o,e) disp('Color changed');
```



Click the button and select a color using the color picker. The component changes appearance and MATLAB displays the words `Color changed` in the Command Window.



See Also

Classes

`matlab.ui.componentcontainer.ComponentContainer`

Functions

`uibutton | uieditfield | uigridlayout`

More About

- “Class Syntax Guide”
- “Configure Custom UI Components for App Designer” on page 15-17

Manage Properties of Custom UI Components

When you develop a custom UI component as a subclass of the `ComponentContainer` base class, you can use certain techniques to make your code more robust, efficient, and user-friendly. These techniques focus on how you define and manage the properties of your class. Use any that are helpful for the type of component you want to create and the user experience you want to provide.

- “Initialize Property Values” on page 15-9 — Set the default state of the UI component in case your users call the implicit constructor without any input arguments.
- “Validate Property Values” on page 15-9 — Ensure that the values are valid before using them.
- “Customize the Property Display” on page 15-10 — Provide a customized list of properties in the Command Window when a user references the UI component object without a semicolon.
- “Optimize the update Method” on page 15-11 — Improve the performance of the `update` method when only a subset of your properties are used in a time-consuming calculation.

For an example of these techniques, see “Example: Optimized Polynomial Fit UI Component with Customized Property Display” on page 15-12.

In addition, there are certain considerations and limitations to keep in mind if you want to use your custom UI component in App Designer, or share your component with users who develop apps in App Designer. These considerations are listed on a separate page, in “Configure Custom UI Components for App Designer” on page 15-17.

Initialize Property Values

Assign default values for all of the public properties of your class. This allows MATLAB to create a valid UI component even if the user omits some name-value arguments when they call the constructor method.

For UI components that contain a chart and have properties that store coordinate data, set the initial values to `NaN` values or empty arrays so that the default chart is empty when the user does not specify the coordinates.

Validate Property Values

Before your code uses property values, confirm that they have the correct size and class. For example, this property block validates the size and class of three properties.

```
properties
    LineColor {validateattributes(LineColor,{'double'}, ...
        {'<=' ,1,'>=' ,0,'size',[1 3]})} = [1 0 0]
    XData (1,:) double = NaN
    YData (1,:) double = NaN
end
```

`LineColor` must be a 1-by-3 array of class `double`, where each value is in the range $[0, 1]$. Both `XData` and `YData` must be row vectors of class `double`.

You can also validate properties that store the underlying component objects in your UI component. To do this, you need to know the correct class name for each object. To determine the class name of an object, call the corresponding UI component function at the command line, and then call the `class` function to get the class name. For example, if you plan to create a drop-down component in

your `setup` method, call the `uidropdown` function at the command line with an output argument. Then, pass the output to the `class` function to get its class name.

```
dd = uidropdown;
class(d)

ans =
'matlab.ui.control.DropDown'
```

Use the output of the `class` function to validate the class for the corresponding property in your class. Specify the class after the property name. For example, the following property stores a `DropDown` object and validates its class.

```
properties (Access = private, Transient, NonCopyable)
    DropDown matlab.ui.control.DropDown
end
```

Occasionally, you might want to define a property that can store different shapes and classes of values. For example, if you define a property that can store a character vector, cell array of character vectors, or string array, omit the size and class validation or use a custom property validation method. For more information about validating properties, see “Validate Property Values”.

Customize the Property Display

One of the benefits of defining your UI component as a subclass of the `ComponentContainer` base class is that it also inherits from the `matlab.mixin.CustomDisplay` class. This lets you customize the list of properties MATLAB displays in the Command Window when you reference the UI component without a semicolon. To customize the property display, overload the `getPropertyGroups` method. Within that method, you can customize which properties are listed and the order of the list. For example, consider a `FitPlot` class that has the following public properties.

```
properties
    LineColor {validateattributes(LineColor,{'double'}, ...
        {'<=',1,'>=',0,'size',[1 3]})} = [1 0 0]
    XData (1,:) double = NaN
    YData (1,:) double = NaN
end
```

The following `getPropertyGroups` method specifies the scalar object property list as `XData`, `YData`, and `LineColor`.

```
function propgrp = getPropertyGroups(obj)
    if ~isscalar(obj)
        % List for array of objects
        propgrp = getPropertyGroups@matlab.mixin.CustomDisplay(obj);
    else
        % List for scalar object
        propList = {'XData','YData','LineColor'};
        propgrp = matlab.mixin.util.PropertyGroup(propList);
    end
end
```

When the user references an instance of this UI component without a semicolon, MATLAB displays the customized list.

```
p = FitPlot
p =
FitPlot with properties:

    XData: NaN
    YData: NaN
    LineColor: [1 0 0]
```

For more information about customizing the property display, see “Customize Property Display”.

Optimize the update Method

In most cases, the `update` method of your class reconfigures all the relevant aspects of your UI component that depend on the public properties. Sometimes, the reconfiguration involves an expensive calculation that is time consuming. If the calculation involves only a subset of the properties, you can design your class to execute that code only when it is necessary.

One way to optimize the `update` method is to add these elements to your class:

- A private property called `ExpensivePropChanged` that accepts a `logical` value. This property indicates whether any of the properties used in the expensive calculation have changed.
- A `set` method for each property involved in the expensive calculation. Within each `set` method, set the `ExpensivePropChanged` property to `true`.
- A protected method called `doExpensiveCalculation` that performs the expensive calculation.
- A conditional statement in the `update` method that checks the value of `ExpensivePropChanged`. If the value is `true`, execute `doExpensiveCalculation`.

The following code provides a template for this design.

```
classdef OptimizedUIComponent < matlab.ui.componentcontainer.ComponentContainer

    properties
        Prop1
        Prop2
    end
    properties(Access=private,Transient,NonCopyable)
        ExpensivePropChanged (1,1) logical = true
    end

    methods(Access = protected)
        function setup(obj)
            % Configure UI component
            % ...
        end
        function update( obj )
            % Perform expensive computation if needed
            if obj.ExpensivePropChanged
                doExpensiveCalculation(obj);
                obj.ExpensivePropChanged = false;
            end
            % Update other aspects of UI component
            % ...
        end
        function doExpensiveCalculation(obj)
            % Expensive code
            % ...
        end
    end
end
```

```
methods
    function set.Prop2(obj,val)
        obj.Prop2 = val;
        obj.ExpensivePropChanged = true;
    end
end
```

In this case, Prop2 is involved in the expensive calculation. The `set.Prop2` method sets the value of Prop2, and then it sets `ExpensivePropChanged` to `true`. The next time the `update` method runs, it calls `doExpensiveCalculation` only if `ExpensivePropChanged` is `true`. Then, the `update` method continues to update other aspects of the UI component.

Example: Optimized Polynomial Fit UI Component with Customized Property Display

This example defines a `FitPlot` class for interactively displaying best fit polynomials, and uses all four of these best practices. The properties defined in the properties block have default values and use size and class validation. The `getPropertyGroups` method defines a custom order for the property display. The `changeFit` method performs the potentially expensive polynomial fit calculation, and the `update` method executes `changeFit` only if the plotted data changed.

To define this class, save the `FitPlot` class definition to a file named `FitPlot.m` in a folder that is on the MATLAB path.

```
classdef FitPlot < matlab.ui.componentcontainer.ComponentContainer
    % Choose a fit method for your plotted data

    properties
        LineColor {validateattributes(LineColor,{ 'double'}, ...
            {'<=',1,'>=',0,'size',[1 3]})} = [1 0 0]
        XData (1,:) double = NaN
        YData (1,:) double = NaN
    end

    properties (Access = private, Transient, NonCopyable)
        DropDown matlab.ui.control.DropDown
        Axes matlab.ui.control.UIAxes
        GridLayout matlab.ui.container.GridLayout
        DataLine (1,1) matlab.graphics.chart.primitive.Line
        FitLine (1,1) matlab.graphics.chart.primitive.Line
        FitXData (1,:) double
        FitYData (1,:) double
        ExpensivePropChanged (1,1) logical = true
    end

    methods (Access=protected)
        function setup(obj)
            % Set the initial position of this component
            obj.Position = [100 100 300 300];

            % Create the grid layout, drop-down, and axes
            obj.GridLayout = uigridlayout(obj,[2,1], ...
                'RowHeight',{20,'lx'},...
                'ColumnWidth',{'lx'});
            obj.DropDown = uidropdown(obj.GridLayout, ...
                'Items',{'None','Linear','Quadratic','Cubic'}, ...
                'ValueChangedFcn',@(s,e) changeFit(obj));
            obj.Axes = uiaxes(obj.GridLayout);

            % Create the line objects
            obj.DataLine = plot(obj.Axes,NaN,NaN,'o');
            hold(obj.Axes,'on');
            obj.FitLine = plot(obj.Axes,NaN,NaN);
            hold(obj.Axes,'off');
        end
    end
```

```

function update(obj)
    % Update data points
    obj.DataLine.XData = obj.XData;
    obj.DataLine.YData = obj.YData;

    % Do an expensive operation
    if obj.ExpensivePropChanged
        obj.changeFit();
        obj.ExpensivePropChanged = false;
    end

    % Update the fit line
    obj.Fitline.Color = obj.LineColor;
    obj.Fitline.XData = obj.FitXData;
    obj.Fitline.YData = obj.FitYData;
end

function changeFit(obj)
    % Calculate the fit line based on the drop-down value
    if strcmp(obj.DropDown.Value,'None')
        obj.FitXData = NaN;
        obj.FitYData = NaN;
    else
        switch obj.DropDown.Value
            case 'Linear'
                f = polyfit(obj.XData,obj.YData,1);
            case 'Quadratic'
                f = polyfit(obj.XData,obj.YData,2);
            case 'Cubic'
                f = polyfit(obj.XData,obj.YData,3);
        end
        obj.FitXData = linspace(min(obj.XData),max(obj.XData));
        obj.FitYData = polyval(f,obj.FitXData);
    end
end

function propgrp = getPropertyGroups(obj)
    if ~isscalar(obj)
        % List for array of objects
        propgrp = getPropertyGroups@matlab.mixin.CustomDisplay(obj);
    else
        % List for scalar object
        propList = {'XData','YData','LineColor'};
        propgrp = matlab.mixin.util.PropertyGroup(propList);
    end
end

methods
    function set.XData(obj,val)
        obj.XData = val;
        obj.ExpensivePropChanged = true;
    end
    function set.YData(obj,val)
        obj.YData = val;
        obj.ExpensivePropChanged = true;
    end
end
end

```

Define some sample data and use it to create an instance of `FitPlot`.

```

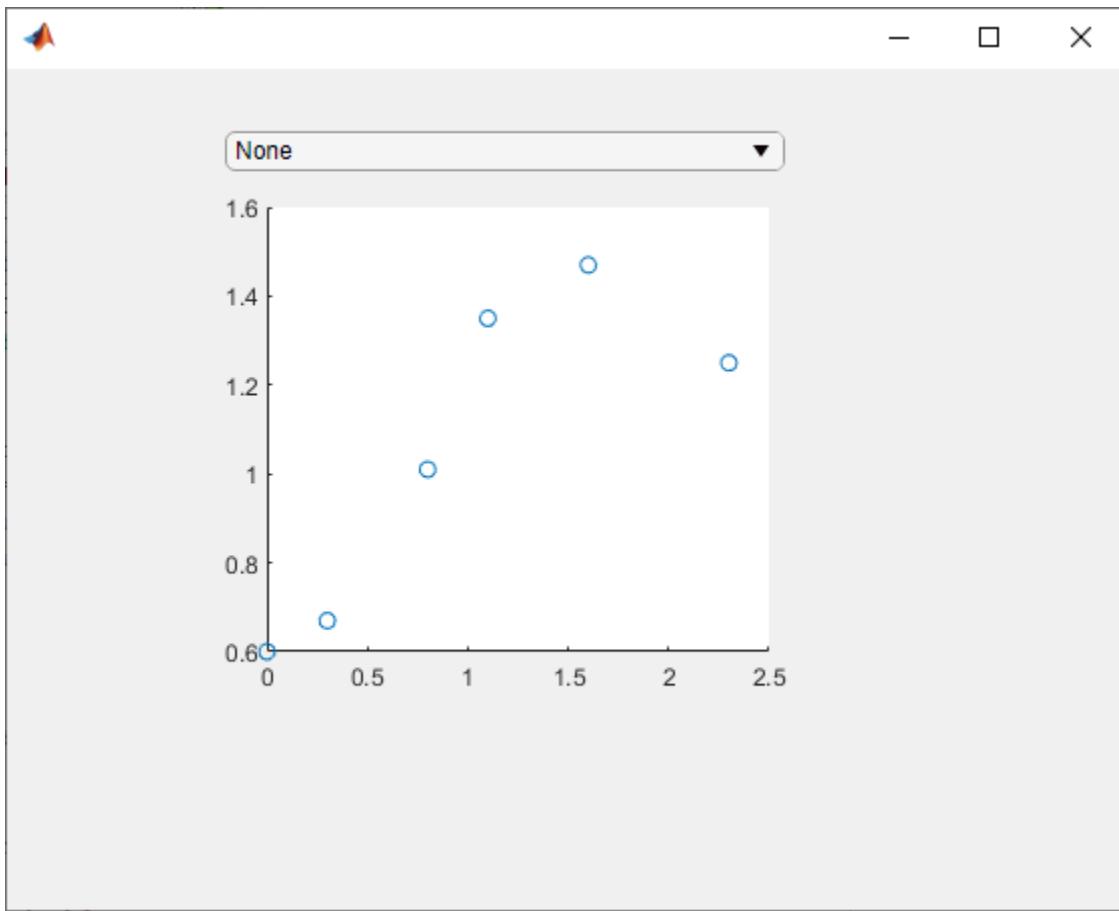
x = [0 0.3 0.8 1.1 1.6 2.3];
y = [0.6 0.67 1.01 1.35 1.47 1.25];
p = FitPlot('XData',x,'YData',y)

```

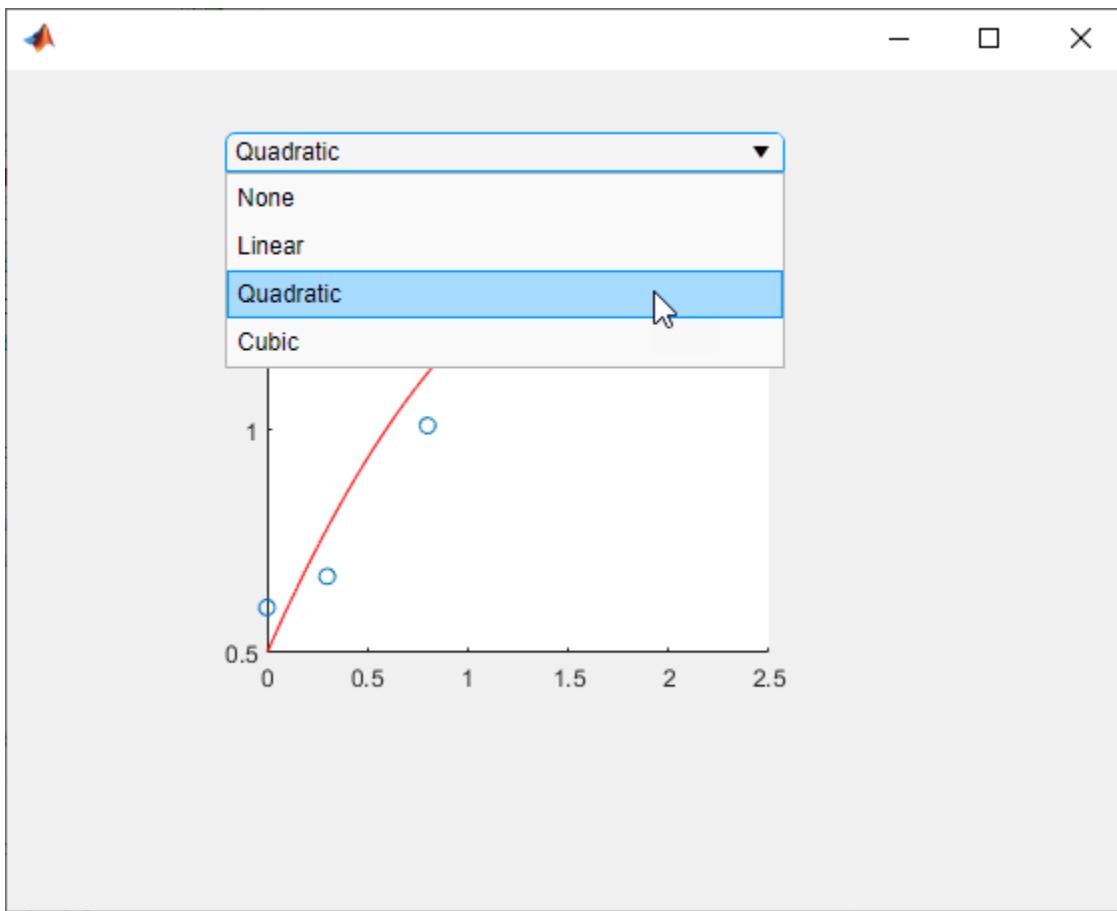
```
ans =
```

FitPlot with properties:

```
XData: [1×43 double]  
YData: [1×43 double]  
LineColor: [1 0 0]
```



Use the drop-down to display the quadratic best fit curve.



Set the `LineColor` property to change the color of the best fit curve to green.

```
p.LineColor = [0 0.5 0];
```



See Also

Classes

`matlab.ui.componentcontainer.ComponentContainer`

Functions

`polyfit` | `uidropdown` | `uigridlayout`

More About

- “Validate Property Values”
- “Customize Property Display”
- “Property Set Methods”
- “Custom UI Component Development Overview” on page 15-2

Configure Custom UI Components for App Designer

In this section...

- “Custom UI Component Class Prerequisites” on page 15-17
- “Create a UI Component Class to Configure” on page 15-18
- “Configure App Designer Metadata” on page 15-19
- “View Configured UI Component in App Designer” on page 15-20
- “Reconfigure UI Component” on page 15-21
- “Remove UI Component From App Designer” on page 15-23
- “Share Configured UI Component” on page 15-23

Starting in R2021a, when you create a custom UI component class, you can configure your component for app creators to use interactively in App Designer. After you configure a UI component, app creators can add the component to the **Component Library** and can interact with the component on the App Designer canvas and in the Property Inspector.

Follow these configuration steps if you have authored a UI component class defined as a subclass of the `matlab.ui.componentcontainer.ComponentContainer` base class, and you would like to use it in either of these ways:

- Access your UI component from the App Designer **Component Library** and interactively use it to create an App Designer app.
- Share your UI component for others to use interactively to create apps in App Designer.

Custom UI Component Class Prerequisites

To allow your custom UI component to be used interactively in App Designer, there are some requirements that your UI component class must satisfy.

To successfully configure your UI component, the `setup` method of your UI component class cannot have required input arguments. Also, the component class cannot dynamically add additional UI components to its parent container. The only exception is that the class can dynamically add a `ContextMenu` component in the parent figure.

For a public property of your component class to appear in the Property Inspector, you must specify its type or assign a default value to it. If the property is an enumeration, you must *both* specify its type and assign it a default value. In addition, the property type must belong to the list of types supported by App Designer. This table shows the allowable property types and their appearance in the Property Inspector.

Property Category	Supported Data Types	Property Inspector Input
Numerical	Scalars or arrays of type <code>single</code> , <code>double</code> , <code>int8</code> , <code>int32</code> , <code>int64</code> , <code>uint8</code> , <code>uint16</code> , <code>uint32</code> , or <code>uint64</code>	Edit field
Logical	<code>logical</code>	Check box

Property Category	Supported Data Types	Property Inspector Input
Text	Scalars of type <code>string</code> , scalars or row vectors of type <code>char</code> , and scalars or vectors of type <code>cell</code>	Text area
Enumeration	<code>enumeration</code>	Editable drop-down menu

For more information on specifying property types and assigning default values, see “Manage Properties of Custom UI Components” on page 15-9.

Create a UI Component Class to Configure

Create a folder named `MyComponents` in a particular location, for example, `C:\`. Copy the `ColorSelector` class definition and save it with the name `ColorSelector.m` in the folder `C:\MyComponents`.

```
classdef ColorSelector < matlab.ui.componentcontainer.ComponentContainer
    % UI component to select colors

    % Public properties
    properties
        Value {validateattributes(Value, ...
            {'double'},{'<='},1,'>=',0,'size',[1 3])} = [1 0 0];
    end

    % Events
    events (HasCallbackProperty, NotifyAccess = protected)
        ValueChanged % ValueChangedFcn will be the generated callback property
    end

    % Private properties
    properties (Access = private, Transient, NonCopyable)
        Grid matlab.ui.container.GridLayout
        Button matlab.ui.control.Button
        EditField matlab.ui.control.EditField
    end

    methods (Access = protected)
        function setup(obj)
            % Grid layout to manage building blocks
            obj.Grid = uigridlayout(obj,[1 2], 'ColumnWidth',{'1x',22}, ...
                'RowHeight',{'fit'}, 'ColumnSpacing',2, 'Padding',2);

            % Edit field for value display and button to launch uisetcolor
            obj.EditField = uieditfield(obj.Grid, 'Editable',false, ...
                'HorizontalAlignment','center');
            obj.Button = uibutton(obj.Grid, 'Text',char(9998), ...
                'ButtonPushedFcn',@(o,e) obj.getColorFromUser());

        end
        function update(obj)
            % Update edit field and button colors
            set([obj.EditField obj.Button], 'BackgroundColor', obj.Value, ...
                'FontColor', obj.getContrastingColor(obj.Value));

            % Update the display text
            obj.EditField.Value = num2str(obj.Value, '%0.2g ');
        end
    end

    methods (Access = private)
        function getColorFromUser(obj)
            c = uisetcolor(obj.Value);
            if (isscalar(c) && (c == 0))
                return;
            end
        end
    end
end
```

```
end

% Update the Value property
obj.Value = c;

% Execute user callbacks and listeners
notify(obj, 'ValueChanged');

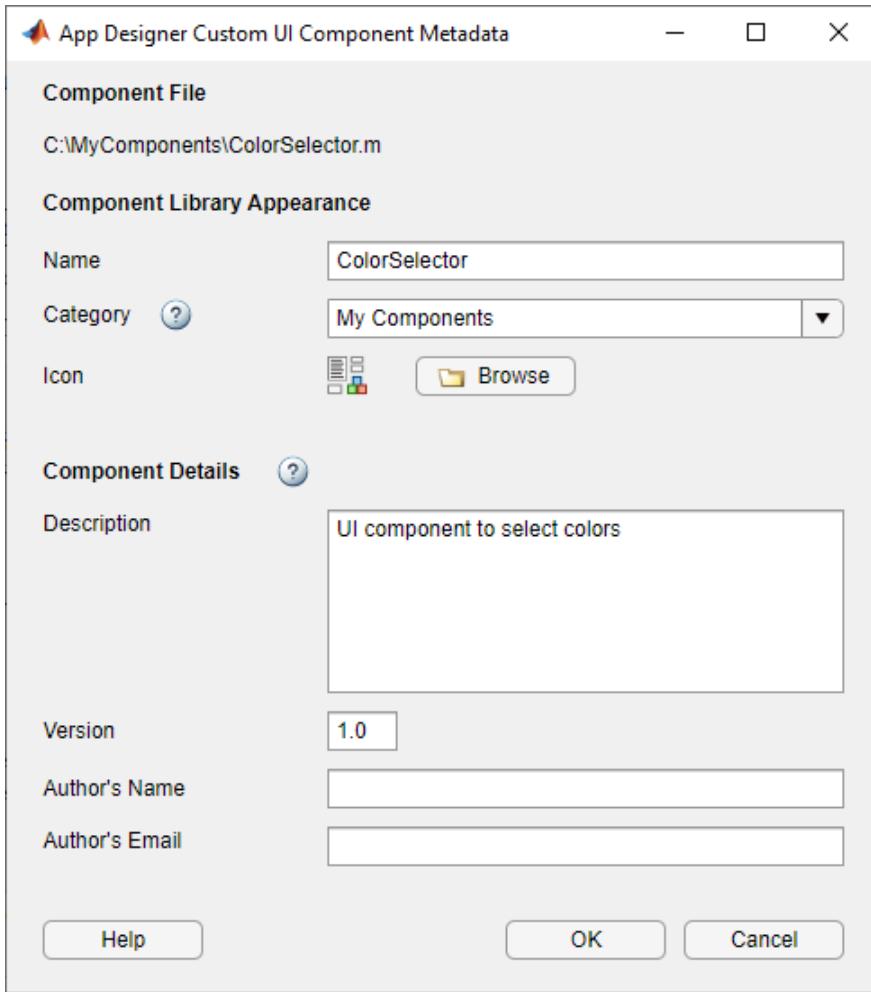
end
function contrastColor = getContrastingColor(~,color)
    % Calculate opposite color
    c = color * 255;
    contrastColor = [1 1 1];
    if (c(1)*.299 + c(2)*.587 + c(3)*.114) > 186
        contrastColor = [0 0 0];
    end
end
end
end
```

Configure App Designer Metadata

Configure your custom UI component for use in App Designer by using the `appdesigner.customcomponent.configureMetadata` function.

Call the function by passing it the path to your component class file. The function opens the App Designer Custom UI Component Metadata dialog. This dialog allows you to specify metadata about the component. App Designer uses this metadata to display the component in the **Component Library**.

```
appdesigner.customcomponent.configureMetadata('C:\MyComponents\ColorSelector.m');
```



The dialog prepopulates all of the required metadata from the component class definition. You can edit the prepopulated metadata using the form. Select **OK** to configure the `ColorSelector` UI component.

After you select **OK**, the function creates a folder named `resources` inside the `MyComponents` folder. Inside the `resources` folder, the function generates a file named `appDesigner.json`. This file contains the metadata you provided in the dialog, in addition to other metadata MATLAB needs to make your component available in App Designer.

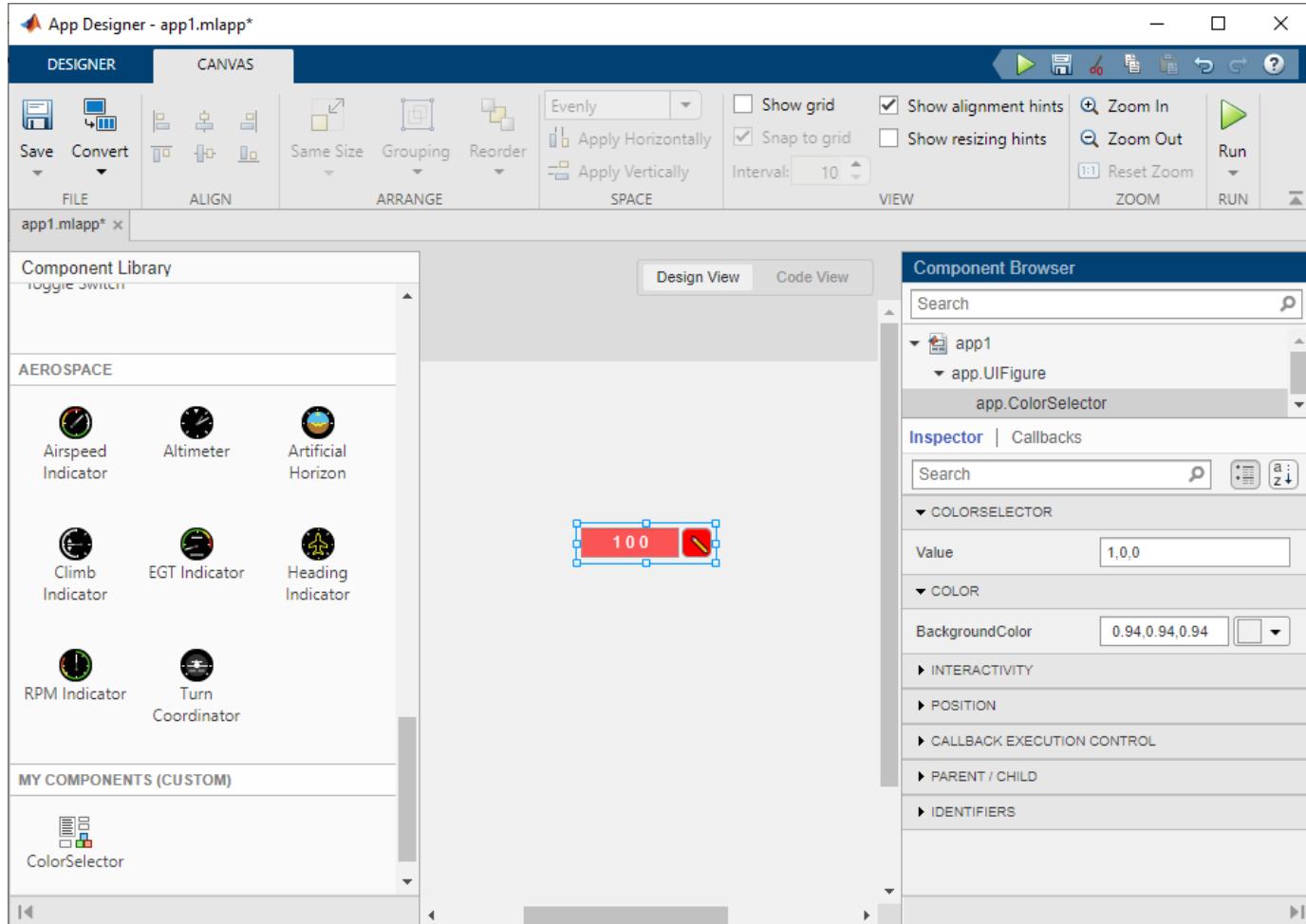
Note Do not modify the `appDesigner.json` file by hand. To change any custom UI component metadata, use the `appdesigner.customcomponent.configureMetadata` function.

View Configured UI Component in App Designer

After you configure your custom UI component class using the `appdesigner.customcomponent.configureMetadata` function, you can view and use it in App Designer. For the UI component to appear in the App Designer **Component Library**, you must add the folder containing the class file and generated `resources` folder to the MATLAB path.

For example, to use the `ColorSelector` UI component in App Designer, add the `MyComponents` folder to the MATLAB path by following the steps in “Change Folders on the Search Path”. Then, open App Designer by entering `appdesigner` at the MATLAB command line. When it opens, select **Blank App**. The `ColorSelector` UI component appears at the bottom of the **Component Library** in the **My Components** section.

Drag an instance of the `ColorSelector` UI component onto the App Designer canvas. Notice that the Property Inspector lists the public property `Value` and the `ValueChangedFcn` callback created in the UI component class definition.



Note Avoid making changes to public properties and events in your UI component class definition while using your component in App Designer, as doing so might lead to errors or unexpected behavior.

Reconfigure UI Component

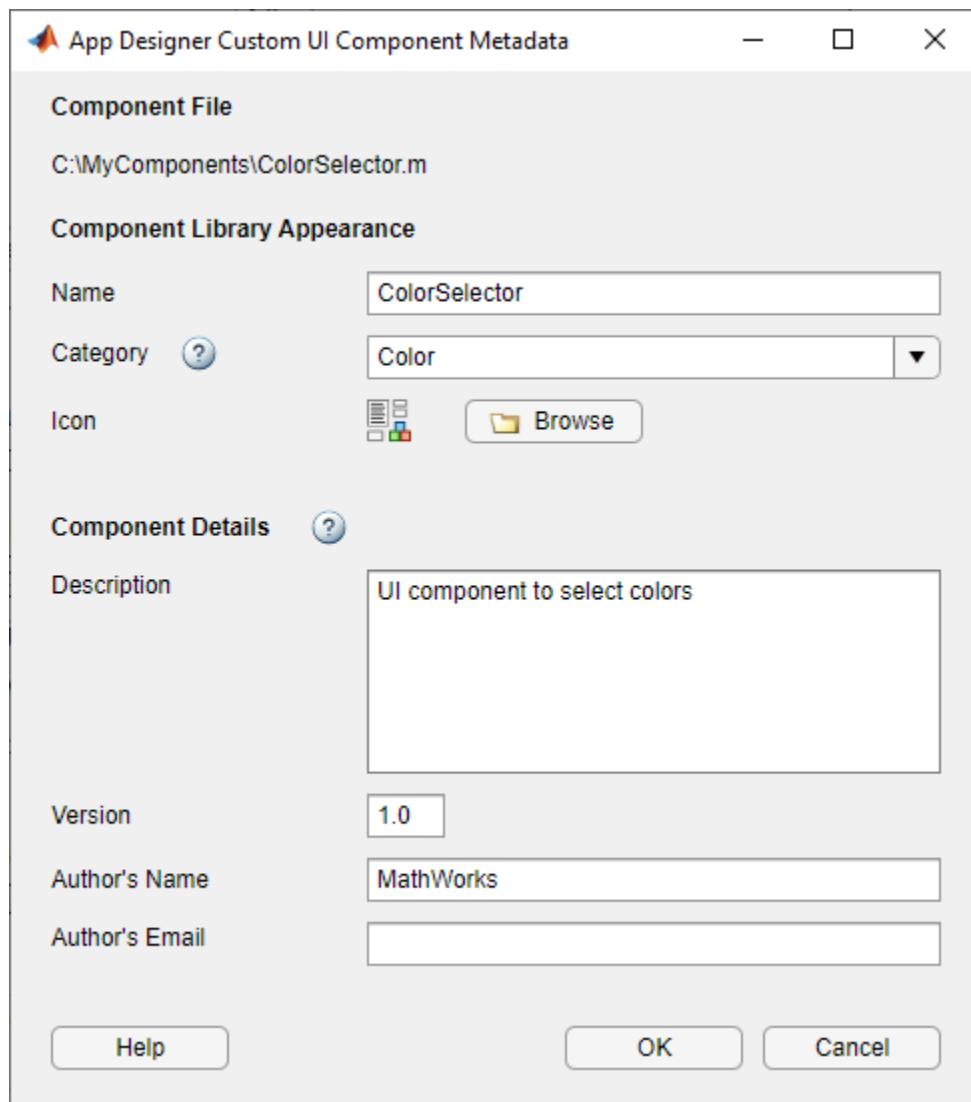
Reconfigure a previously configured UI component class when:

- You want to change existing UI component metadata and update how the component is displayed in the App Designer **Component Library**.
- You have made changes to the UI component position or layout in your class definition.

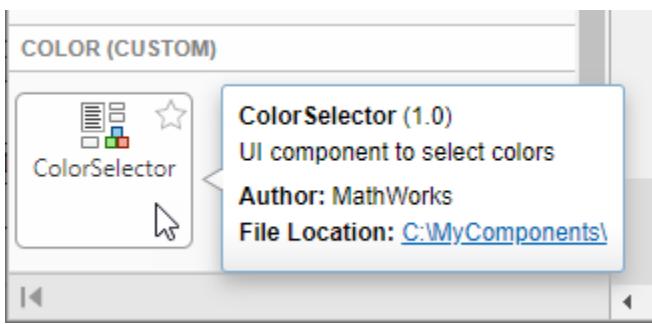
To reconfigure your UI component, call the `appdesigner.customcomponent.configureMetadata` function by passing it the path to your UI component class file. The function opens the App Designer Custom Component Metadata dialog with the existing metadata prepopulated.

```
appdesigner.customcomponent.configureMetadata('C:\MyComponents\ColorSelector.m');
```

Update the metadata by changing the category to **Color** and setting the author's name to **MathWorks**, then select **OK**.



Go back to App Designer. The component now appears in the **Color** section of the **Component Library**. Hover on the component. The author's name now appears.



Remove UI Component From App Designer

To remove a custom UI component from App Designer, use the `appdesigner.customcomponent.removeMetadata` function.

Call the function by passing it the path to your component class file. The function removes the metadata for the UI component from the `appDesigner.json` file inside the `resources` folder, and removes the component from the App Designer **Component Library**.

```
appdesigner.customcomponent.removeMetadata('C:\MyComponents\ColorSelector.m');
```

After you remove the App Designer metadata for a custom UI component, any App Designer apps that use it do not load correctly. To continue editing apps that use the UI component, reconfigure it using the `appdesigner.customcomponent.configureMetadata` function.

Share Configured UI Component

After configuring a UI component, you can share the component for others to use in App Designer. You can either share the relevant files directly or package the component as a toolbox. In either case, you must also share the generated `resources` folder.

Share UI Component Files Directly

To share a configured UI component directly with a user, create and share a folder with these contents:

- The UI component class file
- The generated `resources` folder

Instruct the user you are sharing the UI component with to add the shared folder to the MATLAB path.

Package UI Component as a Toolbox

Package your UI component as a toolbox by following the steps in “Create and Share Toolboxes”. Make sure the folder you package as a toolbox has these contents:

- The UI component class file
- The generated `resources` folder

You can share the resulting `.mltbx` file directly with your users. To install it, they must double-click the `.mltbx` file in the MATLAB **Current Folder** browser.

Alternatively, you can share your UI component as an add-on by uploading the `.mltbx` 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, select **Add-Ons** .
- 2** Find the add-on by browsing through available categories on the left side of the Add-On Explorer window. Alternatively, 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.

See Also

Functions

`appdesigner | appdesigner.customcomponent.configureMetadata |
appdesigner.customcomponent.removeMetadata`

Classes

`matlab.ui.componentcontainer.ComponentContainer`

Related Examples

- “Custom UI Component Development Overview” on page 15-2

Customize Properties of HTML UI Component

To extend your custom UI component using third-party visualizations or widgets, create a component class that contains an HTML UI component. Use the HTML UI component to customize the component appearance and to interface with third-party libraries, and use the component class to define component properties and callbacks that the user can set.

Class Construction Overview

To create a custom UI component class that uses an HTML UI component, there are two files that you must create.

- UI component class file — In this file, you define your component class. You specify its properties, its property values, the events it listens for, and the callback functions it triggers. Its properties must include one that will contain the HTML UI component.
- HTML source file — In this file, you configure and update the visual appearance of the UI component, listen for user interactions, and pass the information that an interaction has occurred to the UI component class.

Your code must communicate changes to property values and user interactions across these two files.

Enable Response to Property Updates

Since the UI component class file defines the properties that users can set, but the HTML source file controls the visual style of the component, these two files need to communicate about property updates.

In the UI component class file, configure the properties of your UI component. Specify the properties that users can set by defining them as public properties in a `properties` block. In the `update` method of your class, store the values of the public properties as fields in a `struct` in the `Data` property of your HTML UI component. This gives the HTML source file access to these property values.

In the HTML source file, use the property values to update the appearance of the HTML UI component. To do so, in the `setup` function inside of a `<script>` tag, access the values of the fields in `Data` and use them to modify the style properties of your HTML elements.

Enable Response to User Interactions

Users define component callback functions in MATLAB, but these callbacks often listen for a response to an action performed on an HTML element defined in the HTML source file. So these two files also need to communicate about user interactions.

In the UI component class file, first create the callback properties of your UI component. Create an `events` block with the `HasCallbackProperty`. When you define an event in this block, MATLAB creates an associated public callback property for the UI component. For example, if you create an event named `ButtonPushed`, this will automatically create a public property for your class named `ButtonPushedFcn`.

To execute a user-defined callback function associated with a user interaction, your code must first recognize when the user interaction has occurred. In the UI component class file, give the HTML UI component a way to do this. In the `setup` method, set the `Data` property of the HTML UI component to a `struct` with a field to store information about whether the interaction has occurred. Because

the class file and the HTML source file share this property and its value, the HTML source file can update the value to communicate the user interaction status to the UI component class. To accomplish this, in the HTML source file, in the `setup` function inside of a `<script>` tag, create an event listener that listens for the user interaction. When the listener detects the interaction, update the `Data` property of the HTML UI component.

After the UI component class file receives the information that a user interaction has occurred, it must then trigger the event associated with the interaction. Create a class method to do this. In the class method, first call the built-in `notify` method to trigger the event you defined. This executes the user-defined callback function associated with the event. Then, set the `Data` property of the HTML component to wait for another interaction. In the `setup` method of the UI component class file, set the `DataChangedFcn` property of the HTML component to the class method you defined. The HTML UI component executes this method automatically whenever the `Data` property changes. Therefore, after the HTML source file updates the `Data` property to communicate that the interaction has occurred, this method executes the appropriate callback.

RoundButton Class Implementation

This example demonstrates a typical structure for writing a custom UI component class that uses an HTML UI component. The class creates a button with a custom rounded style. It allows users to specify the button color, text, text color, and response on click.

To define your UI component class, create two files in the same folder on the MATLAB path:

- `RoundButton.m` — UI component class definition
- `RoundButton.html` — HTML source file

RoundButton.m Class Definition

RoundButton class	Discussion
<pre>classdef RoundButton < matlab.ui.componentcontainer.ComponentContainer</pre>	Create a custom UI component named RoundButton by defining a subclass of the <code>matlab.ui.componentcontainer.ComponentContainer</code> class.
<pre>properties Color {mustBeMember(Color, ... {'white','blue','red','green','yellow'})} = 'white' FontColor {mustBeMember(FontColor, ... {'black','white'})} = 'black' Text (1,:) char = 'Button'; end</pre>	Define the <code>Color</code> , <code>FontColor</code> , and <code>Text</code> public properties for your <code>RoundButton</code> class. These are properties that the user can set when creating a <code>RoundButton</code> instance. For more information on defining properties, see "Manage Properties of Custom UI Components" on page 15-9.
<pre>properties (Access = private, Transient) HTMLComponent matlab.ui.control.HTMLComponent; end</pre>	Define a <code>HTMLComponent</code> private property to hold the HTML UI component.
<pre>events (HasCallbackProperty, NotifyAccess) % Generate a ButtonPushedFcn callback ButtonPushed end</pre>	Define a <code>ButtonPushed</code> event in an events block. Specify the <code>HasCallbackProperty</code> for the events block to automatically generate a <code>ButtonPushedFcn</code> public property for the class.
<pre>methods (Access=protected)</pre>	Create a methods block.

RoundButton class	Discussion
<pre> function setup(obj) % Set the initial position of this component obj.Position = [100 100 80 40]; % Create the HTML component obj.HTMLComponent = uihtml(obj); obj.HTMLComponent.Position = [1 1 obj.Position(3,4)]; obj.HTMLComponent.HTMLSource = fullfile(pwd,'button.html'); obj.HTMLComponent.Data = struct('Clicked' false); obj.HTMLComponent.DataChangedFcn = @(s,e)notifyClick(obj); end </pre>	<p>Define the <code>setup</code> method for your class. Within the method, set the initial position of your component relative to its parent container.</p> <p>Then, create an HTML component by calling the <code>uihtml</code> function. Set the following properties for your HTML component:</p> <ul style="list-style-type: none"> • Position — the position of the HTML component relative to the position of the custom UI component. • HTMLSource — the source file that contains the HTML markup for the HTML component. • Data — a <code>struct</code> with a <code>Clicked</code> field with value <code>false</code>. Code in the HTML source file sets this field to <code>true</code> when the user clicks the HTML component. • DataChangedFcn — an anonymous function that calls a class method named <code>notifyClick</code>. This function runs when the Data property of the HTML component changes.
<pre> function update(obj) % Update the HTML component data obj.HTMLComponent.Data.Color = obj.Color; obj.HTMLComponent.Data.FontColor = obj.FontColor; obj.HTMLComponent.Data.Text = obj.Text; obj.HTMLComponent.Position = [1 1 obj.Position(3,4)]; end </pre>	<p>Define the <code>update</code> method for your class. Within the method, store the values of the <code>Color</code>, <code>FontColor</code>, and <code>Text</code> properties as fields in the <code>Data</code> property of the HTML component. This enables you to update the attributes of the HTML button element, and lets the HTML component listen for when these properties are changed.</p>
<pre> function notifyClick(obj) if obj.HTMLComponent.Data.Clicked obj.HTMLComponent.Data.Clicked = false; drawnow notify(obj, 'ButtonPushed'); end end </pre>	<p>Define the function that runs when the <code>Data</code> property changes, which is called <code>notifyClick</code>. The function first checks to see if the <code>Clicked</code> field of the HTML component data is <code>true</code>. If so, the function sets the <code>Clicked</code> data field to <code>false</code> to await the next button click. The function also fires the <code>ButtonPushed</code> event, which in turn executes the user-defined <code>ButtonPushedFcn</code> property.</p>
<pre> end end </pre>	<p>Close the methods block and the class definition.</p>

RoundButton.html Source Definition

HTML Source	Discussion
<pre> <!DOCTYPE html> <html> <head> </pre>	<p>Open the <code><html></code> tag and the <code><head></code> tag.</p>

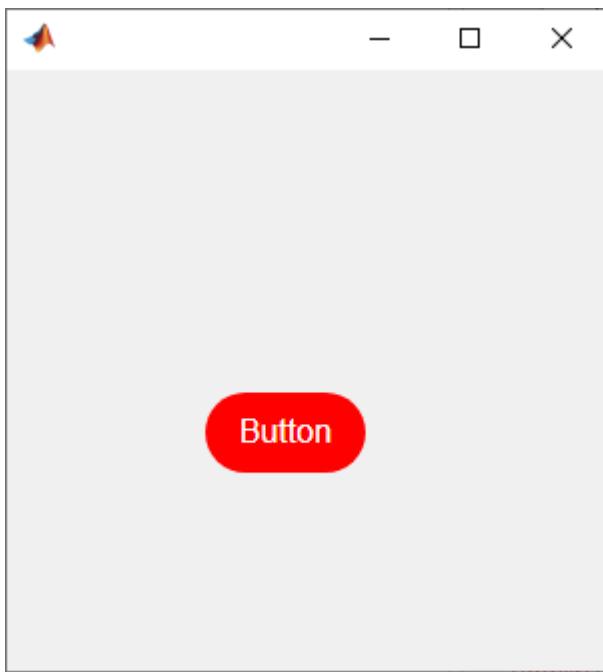
HTML Source	Discussion
<pre data-bbox="241 287 861 853"><style> html, body { height: 100%; text-align: center; } button { width: 100%; height: 100%; border-radius: 2em; font-size: 1em; cursor: pointer; border: none; } button:focus { outline: 0; } </style></pre>	<p>Define the style for the HTML content using CSS markup:</p> <ul style="list-style-type: none"> Set the height of the HTML body to scale to fill the entire container in which it is displayed. Define the relative size of the button within the document body, the radius of the button edges, the font size, the cursor style when pointing to the button, and the button border style.
<pre data-bbox="241 853 861 994"><script type="text/javascript"> function setup(htmlComponent) {</pre>	<p>Write a <code>setup</code> function inside of a <code><script></code> tag to connect your JavaScript object, called <code>htmlComponent</code>, to the HTML UI component you created in MATLAB.</p>
<pre data-bbox="241 994 861 1227"> htmlComponent.addEventListener("DataChanged", function(event) { buttonElement = document.getElementById("roundButton"); buttonElement.style.backgroundColor = htmlComponent.Data.BackgroundColor; buttonElement.innerHTML = htmlComponent.Data.Text; buttonElement.style.color = htmlComponent.Data.FontColor; }); }</pre>	<p>Add an event listener to the <code>htmlComponent</code> JavaScript object. This event listener listens for any change in the <code>Data</code> property of the <code>htmlComponent</code> MATLAB object, and then updates the attributes of the HTML button element in accordance with the <code>RoundButton</code> property values.</p>
<pre data-bbox="241 1227 861 1586"> button = document.getElementById("roundButton"); button.addEventListener("click", function(event) { htmlComponent.Data.Clicked = true; htmlComponent.Data = htmlComponent.Data; }); }</pre>	<p>Add an event listener to the HTML button. This event listener listens for the button element to be clicked. When a user clicks the button, the function:</p> <ul style="list-style-type: none"> Updates the <code>Clicked</code> field of the HTML component data to <code>true</code> Explicitly set the <code>Data</code> property of the HTML component. This notifies the MATLAB HTML component object to execute the <code>DataChangedFcn</code> callback.
<pre data-bbox="241 1586 861 1691"> } </script> </head></pre>	<p>Close the <code>setup</code> function and the <code><script></code> and <code><head></code> tags.</p>
<pre data-bbox="241 1691 861 1797"><body> <button id="roundButton"></button>
 </body></pre>	<p>Create a button element in the body of the HTML document.</p>
<pre data-bbox="241 1797 861 1831"></html></pre>	<p>Close the <code><html></code> tag.</p>

Create a RoundButton Instance

After creating and saving `RoundButton.m` and `RoundButton.html`, create an instance of the `RoundButton` class in a UI figure.

Specify the `Color`, `FontColor`, and the `ButtonPushedFcn` callback properties as name-value arguments.

```
fig = uifigure('Position',[200 200 300 300]);
btn = RoundButton(fig, ...
    'Color','red',...
    'FontColor','white',...
    'ButtonPushedFcn',@(o,e)disp('Clicked'));
```



Click the button. The Command Window displays `Clicked`.

See Also

Classes

`matlab.ui.componentcontainer.ComponentContainer`

Functions

`uifigure` | `uihtml`

Related Examples

- “Custom UI Component Development Overview” on page 15-2
- “Create HTML File That Can Trigger or Respond to Data Changes” on page 5-23

Create UIs with GUIDE

What Is GUIDE?

GUIDE: Getting Started

In this section...

"UI Layout" on page 16-2

"UI Programming" on page 16-2

Note The GUIDE environment will be removed in a future release. After GUIDE is removed, existing GUIDE apps will continue to run in MATLAB but they will not be editable in GUIDE.

To continue editing an existing GUIDE app, see "GUIDE Migration Strategies" on page 4-5 for information on how to help maintain compatibility of the app with future MATLAB releases. To create new apps interactively, "Develop Apps Using App Designer" instead.

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.

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 17-2
- “GUIDE Options” on page 17-8

GUIDE Preferences

In this section...

- "Set Preferences" on page 17-2
- "Confirmation Preferences" on page 17-2
- "Backward Compatibility Preference" on page 17-4
- "All Other Preferences" on page 17-4

Note The GUIDE environment will be removed in a future release. After GUIDE is removed, existing GUIDE apps will continue to run in MATLAB but they will not be editable in GUIDE.

To continue editing an existing GUIDE app, see "GUIDE Migration Strategies" on page 4-5 for information on how to help maintain compatibility of the app with future MATLAB releases. To create new apps interactively, "Develop Apps Using App Designer" instead.

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.

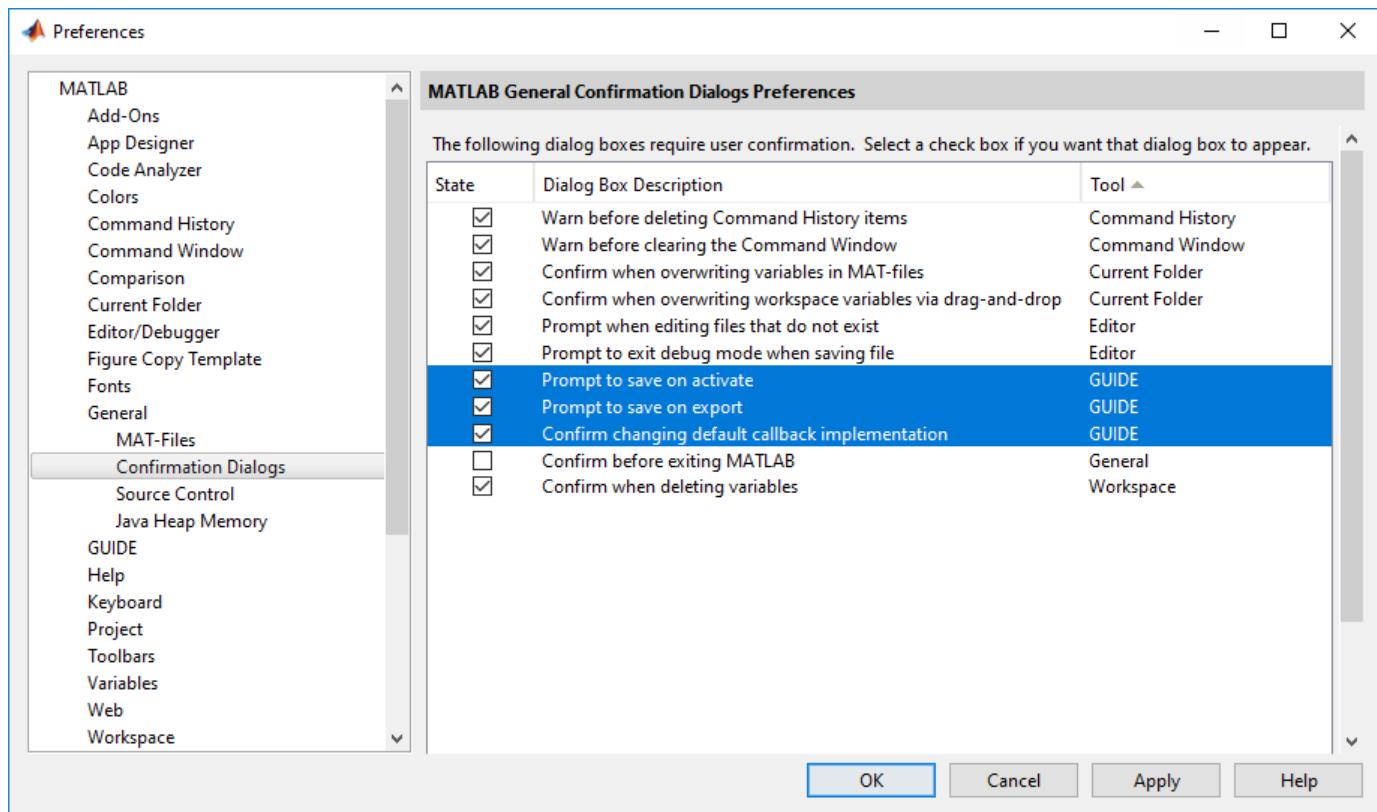
The preferences are in different locations within the Preferences dialog box:

Confirmation Preferences

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

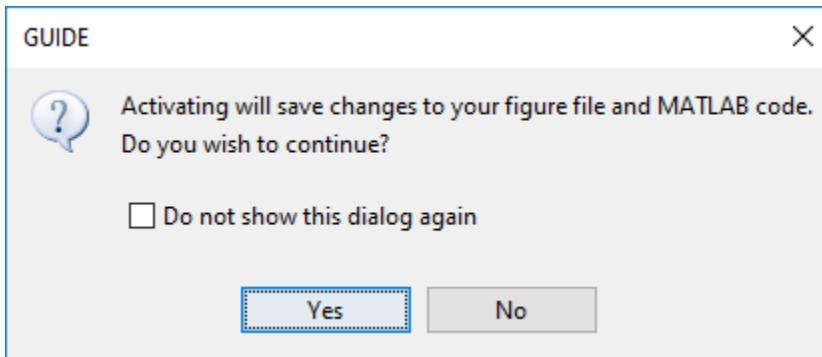
- Activate a UI from GUIDE.
- Export a UI from GUIDE.
- Change a callback signature generated by GUIDE.

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.



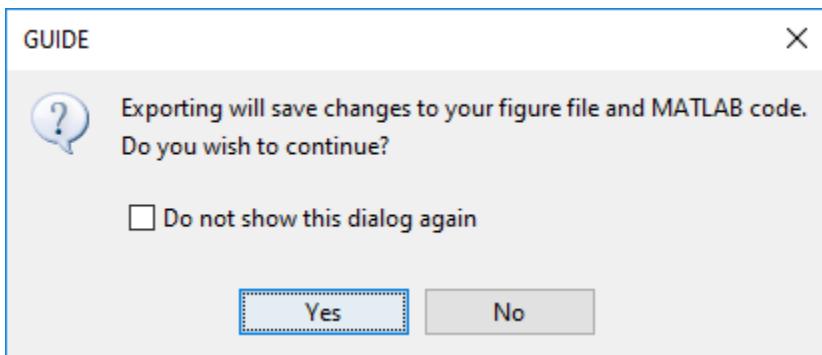
Prompt to Save on Activate

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 to MATLAB-file**, 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

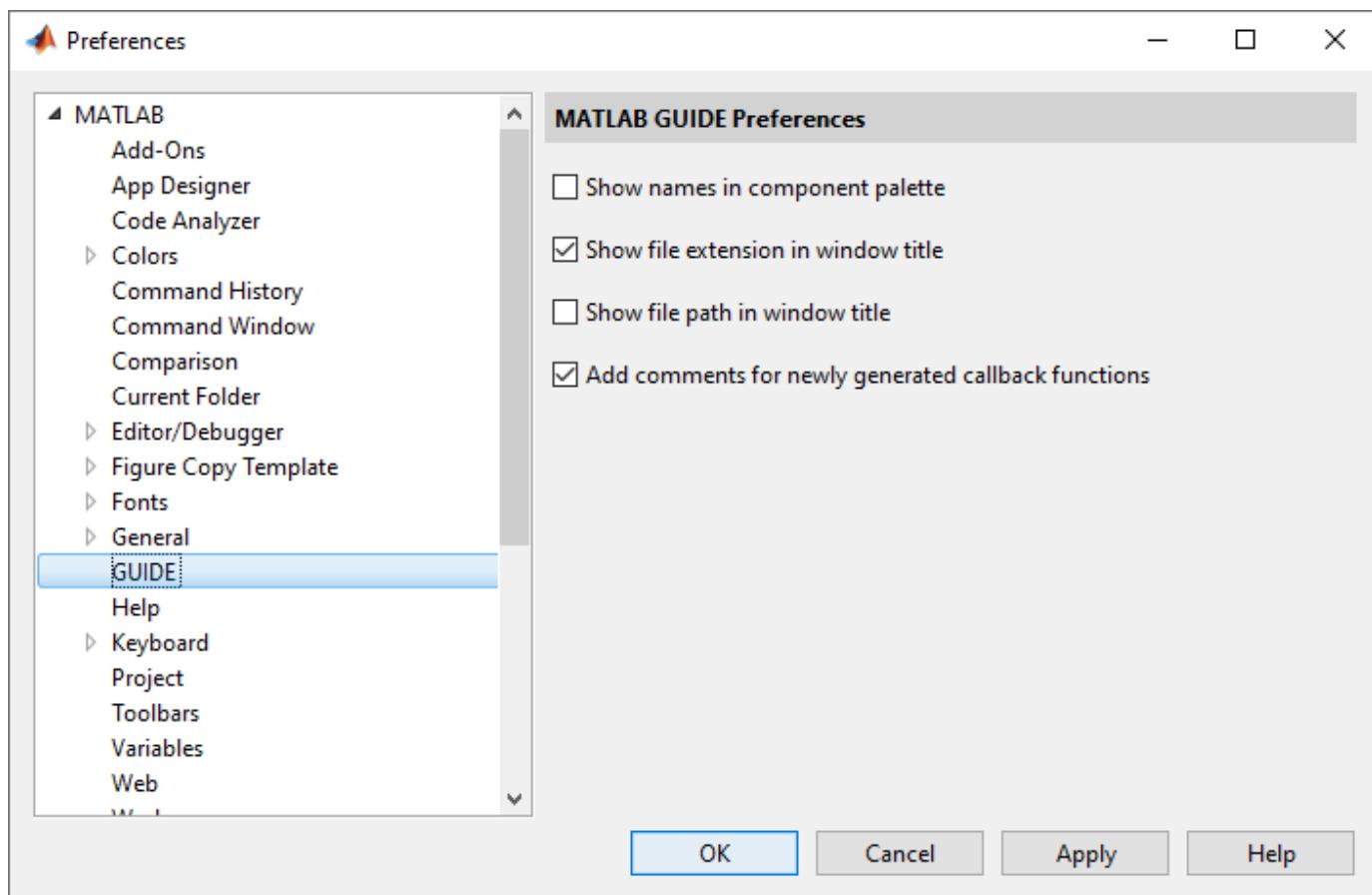
UI FIG-files created or modified with MATLAB 7.0 or a later version are not automatically compatible with Version 6.5 and earlier versions. GUIDE automatically generates FIG-files, which are binary files that contain the UI layout information.

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.

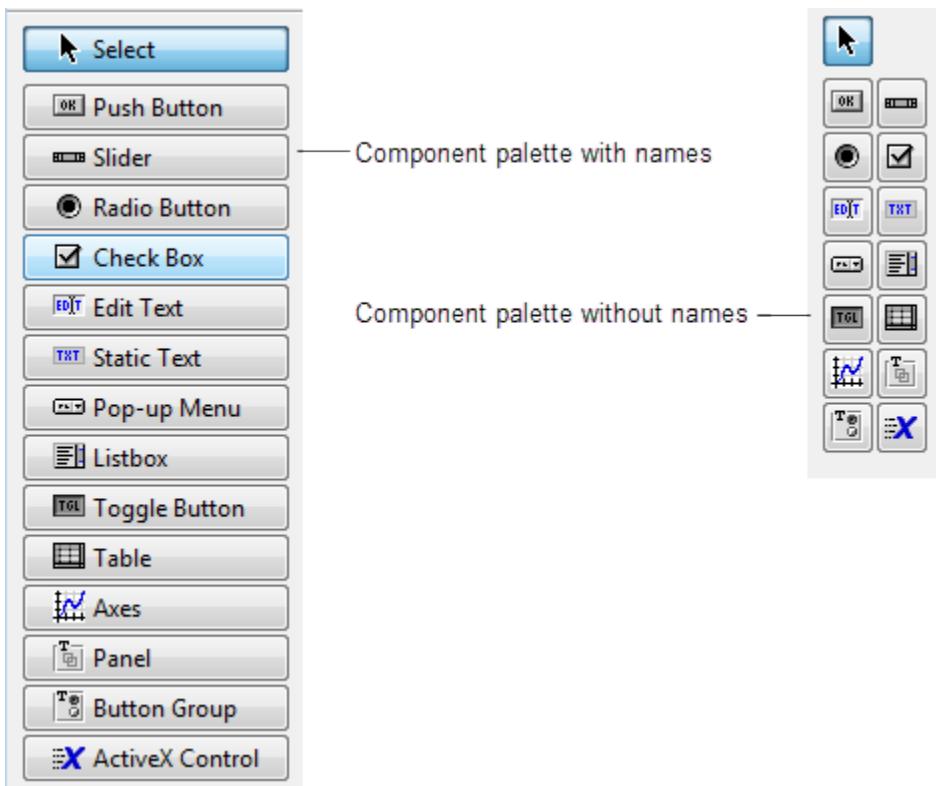


The following topics describe the preferences in this dialog:

- “Show Names in Component Palette” on page 17-5
- “Show File Extension in Window Title” on page 17-6
- “Show File Path in Window Title” on page 17-6
- “Add Comments for Newly Generated Callback Functions” on page 17-6

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 19-2 for more information about callbacks and about the arguments described in the preceding comments.

See Also

Related Examples

- “GUIDE Options” on page 17-8
- “GUIDE Migration Strategies” on page 4-5

GUIDE Options

In this section...

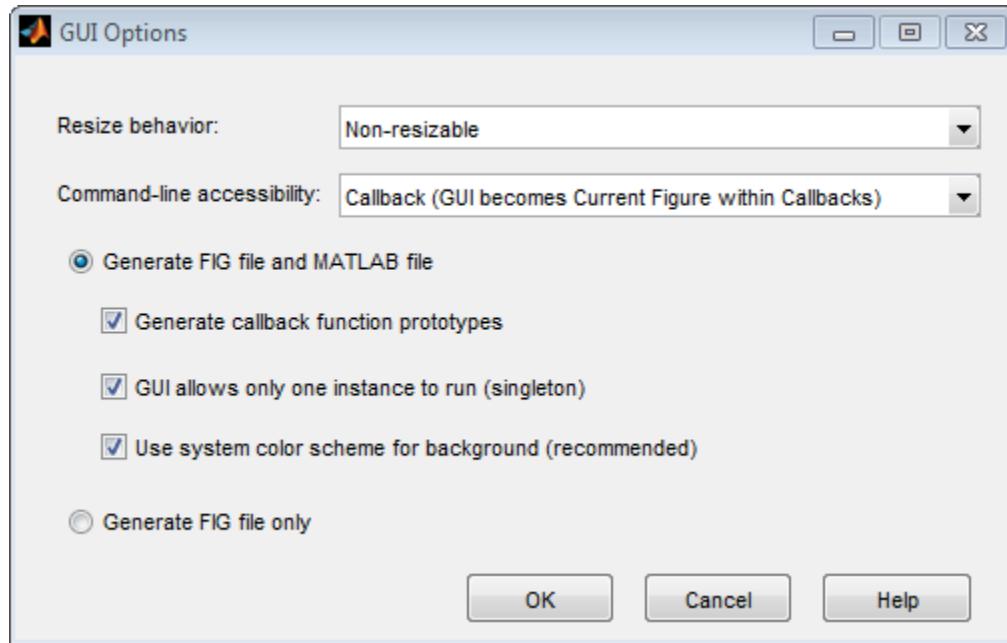
- “The GUI Options Dialog Box” on page 17-8
- “Resize Behavior” on page 17-8
- “Command-Line Accessibility” on page 17-9
- “Generate FIG-File and MATLAB File” on page 17-10
- “Generate FIG-File Only” on page 17-11

Note The GUIDE environment will be removed in a future release. After GUIDE is removed, existing GUIDE apps will continue to run in MATLAB but they will not be editable in GUIDE.

To continue editing an existing GUIDE app, see “GUIDE Migration Strategies” on page 4-5 for information on how to help maintain compatibility of the app with future MATLAB releases. To create new apps interactively, “Develop Apps Using App Designer” instead.

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 17-10
- “GUI Allows Only One Instance to Run (Singleton)” on page 17-10
- “Use System Color Scheme for Background” on page 17-10

See “Files Generated by GUIDE” on page 3-16 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 18-40.

See “Write Callbacks in GUIDE” on page 19-2 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 19-2

See Also

Related Examples

- “GUIDE Preferences” on page 17-2

Lay Out a UI Using GUIDE

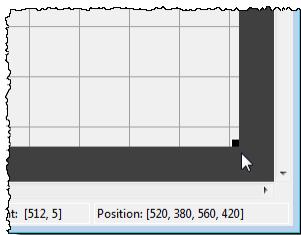
- “Set the UI Window Size in GUIDE” on page 18-2
- “Add Components to the GUIDE Layout Area” on page 18-4
- “Create Menus for GUIDE Apps” on page 18-40

Set the UI Window Size in GUIDE

Note The GUIDE environment will be removed in a future release. After GUIDE is removed, existing GUIDE apps will continue to run in MATLAB but they will not be editable in GUIDE.

To continue editing an existing GUIDE app, see “GUIDE Migration Strategies” on page 4-5 for information on how to help maintain compatibility of the app with future MATLAB releases. To create new apps interactively, “Develop Apps Using App Designer” instead.

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.

Setting the **Units** property to **characters** (nonresizable UIs) or **normalized** (resizable UIs) gives the UI a more consistent appearance across platforms.

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 3-2
- “GUIDE Options” on page 17-8

Add Components to the GUIDE Layout Area

In this section...

- “Place Components” on page 18-4
- “User Interface Controls” on page 18-8
- “Panels and Button Groups” on page 18-22
- “Axes” on page 18-26
- “Table” on page 18-29
- “Resize GUIDE UI Components” on page 18-37

Note The GUIDE environment will be removed in a future release. After GUIDE is removed, existing GUIDE apps will continue to run in MATLAB but they will not be editable in GUIDE.

To continue editing an existing GUIDE app, see “GUIDE Migration Strategies” on page 4-5 for information on how to help maintain compatibility of the app with future MATLAB releases. To create new apps interactively, “Develop Apps Using App Designer” instead.

Place Components

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

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.

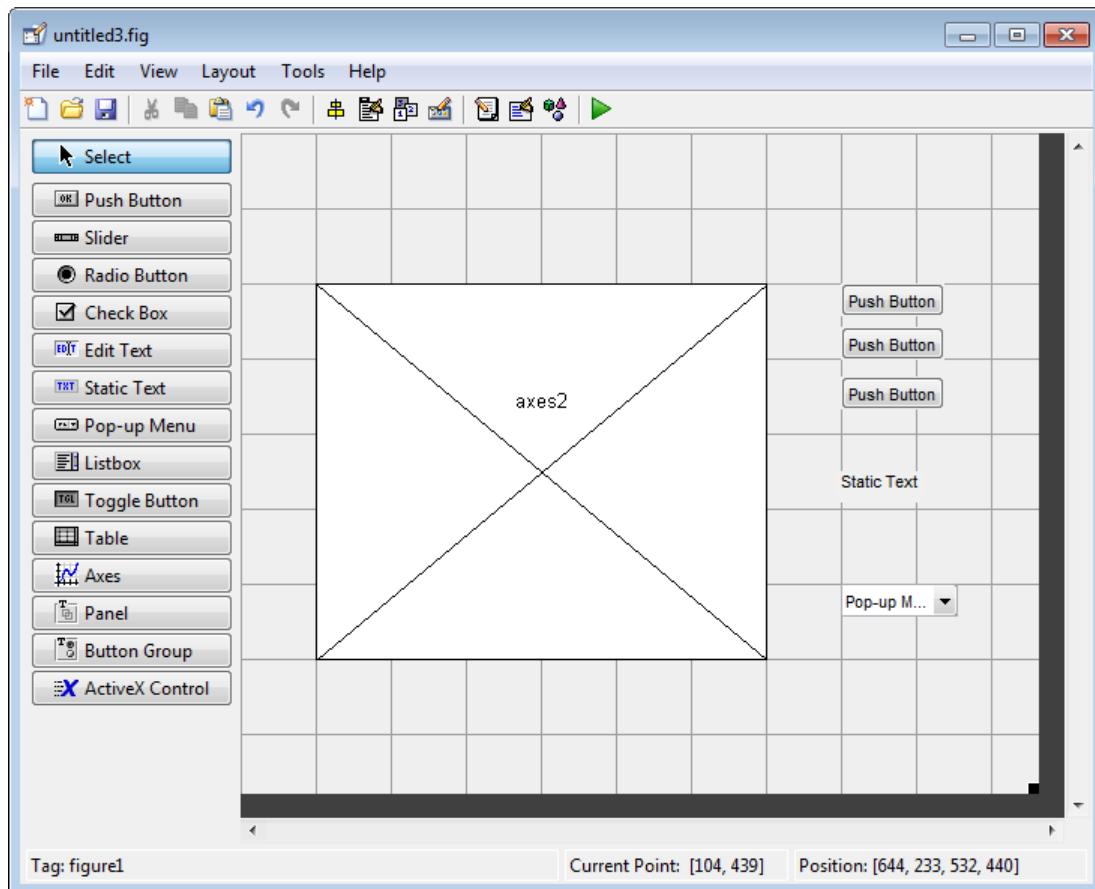
The components listed in the following table have additional considerations; read more about them in the sections described there.

If You Are Adding...	Then...
Panels or button groups	See “Add a Component to a Panel or Button Group” on page 18-6.

If You Are Adding...	Then...
Menus	See “Create Menus for GUIDE Apps” on page 18-40

- 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 18-7 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 18-8
 - “Panels and Button Groups” on page 18-22
 - “Axes” on page 18-26
 - “Table” on page 18-29

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.

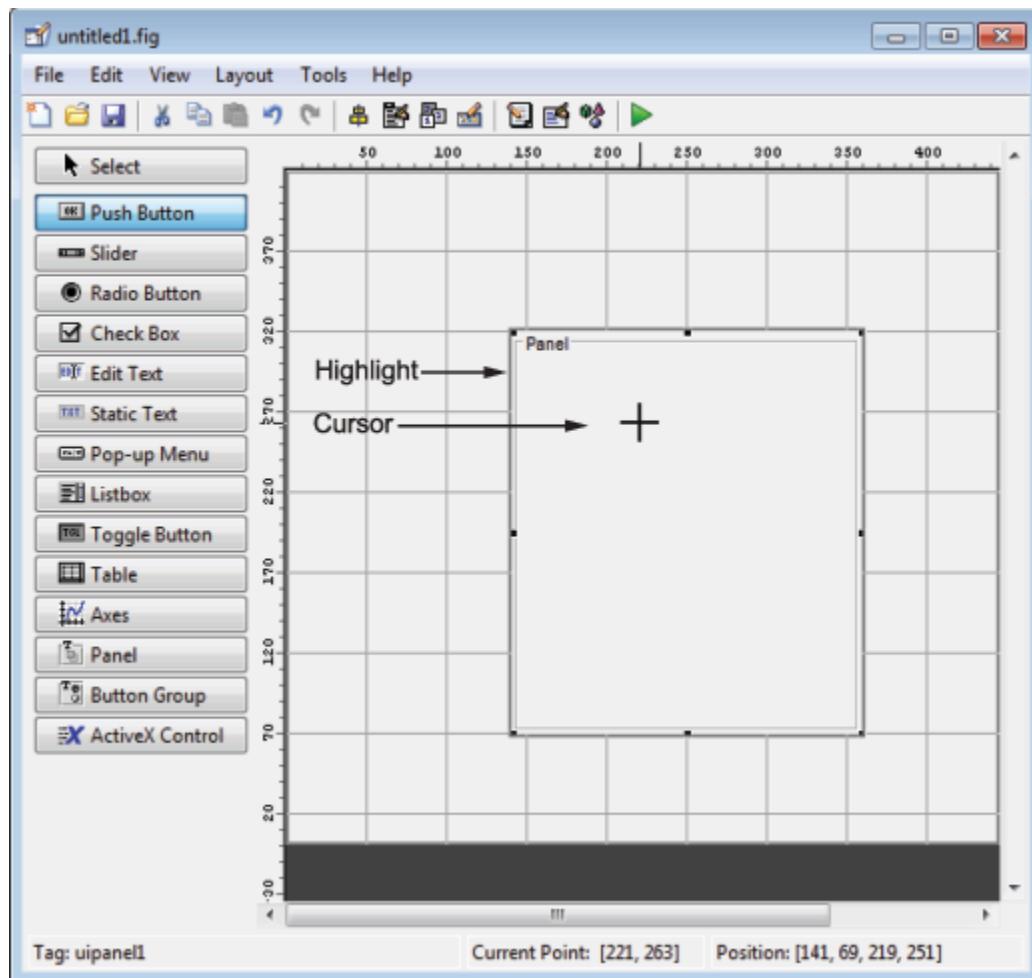
Here is how to interpret the coordinates in the status bar and rulers:

- 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.



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 18-7 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 18-22 for more information on how to incorporate panels and button groups into a UI.

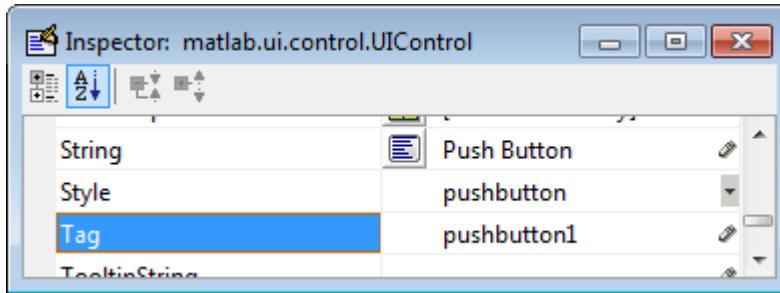
Assign an Identifier to Each Component

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 18-8
- “Push Button” on page 18-9
- “Slider” on page 18-10
- “Radio Button” on page 18-12
- “Check Box” on page 18-13
- “Edit Text” on page 18-14
- “Static Text” on page 18-15
- “Pop-Up Menu” on page 18-16
- “List Box” on page 18-18
- “Toggle Button” on page 18-20

Commonly Used Properties

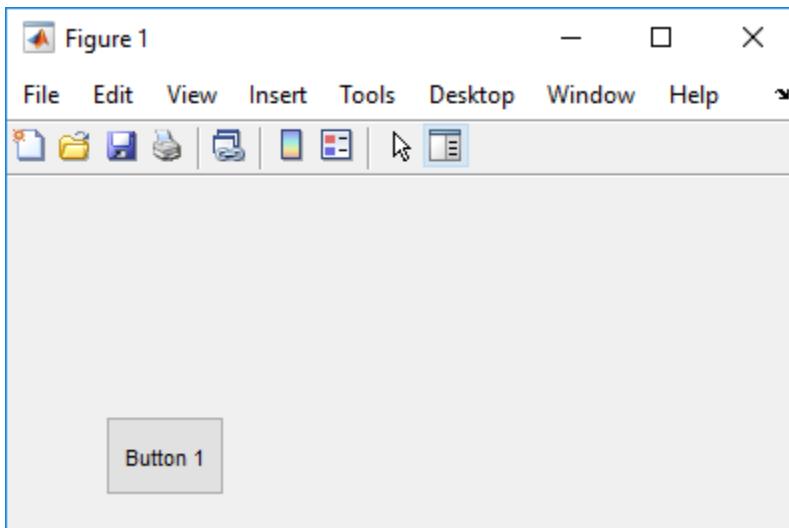
The most commonly used properties needed to describe a user interface control are shown in the following table. Instructions for a particular control may also list properties that are specific to that control.

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.
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.

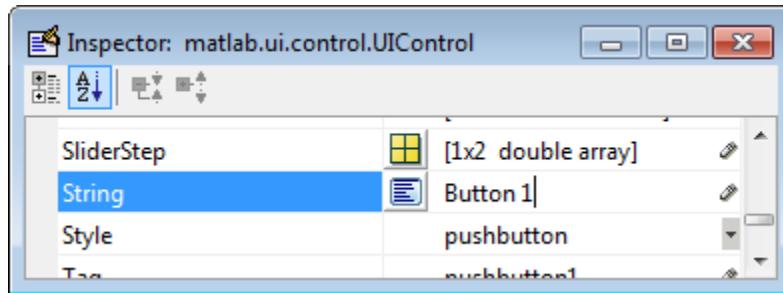
For a complete list of properties and for more information about the properties listed in the table, see [Uicontrol](#).

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);
```

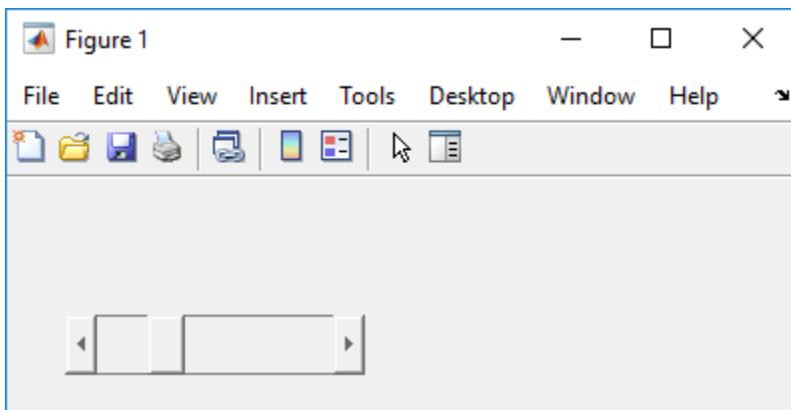
where `pushbutton1` is the push button's **Tag** property.



See `ind2rgb` for information on converting a matrix `X` and corresponding colormap, i.e., an (`X`, `MAP`) image, to RGB (truecolor) format.

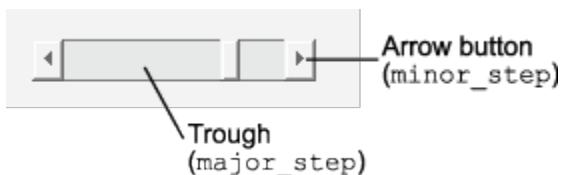
Slider

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`.



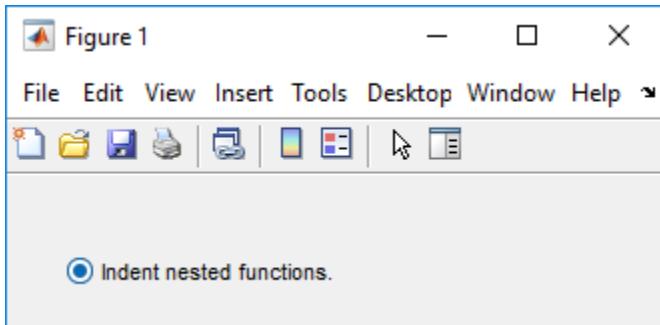
- If you want to set the location or size of the component to an exact value, then modify its `Position` property.

The slider component provides no text description or data entry capability. Use a “Static Text” on page 18-15 component to label the slider. Use an “Edit Text” on page 18-14 component to enable a user to input a value to apply to the slider.

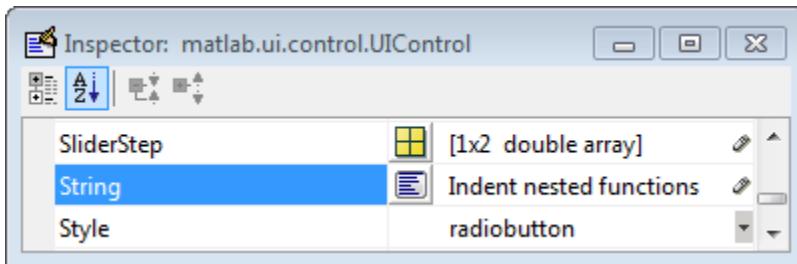
On Mac platforms, the height of a horizontal slider is constrained. If the height you set in the position vector exceeds this constraint, the displayed height of the slider is the maximum allowed. The height element of the position vector is not changed.

RadioButton

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.

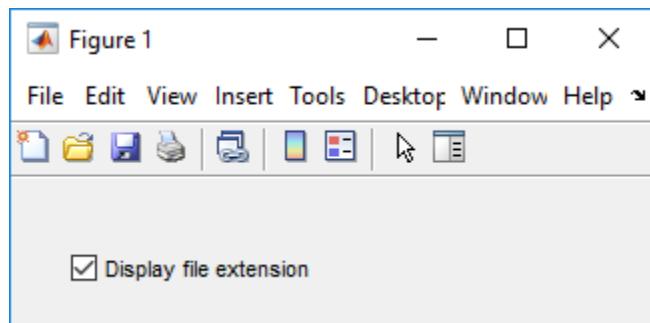
- 
- Create the radio button with the button selected by setting its **Value** property to the value of its **Max** property (default is 1). Set **Value** to **Min** (default is 0) to leave the radio button unselected. Correspondingly, when the user selects the radio button, the software sets **Value**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

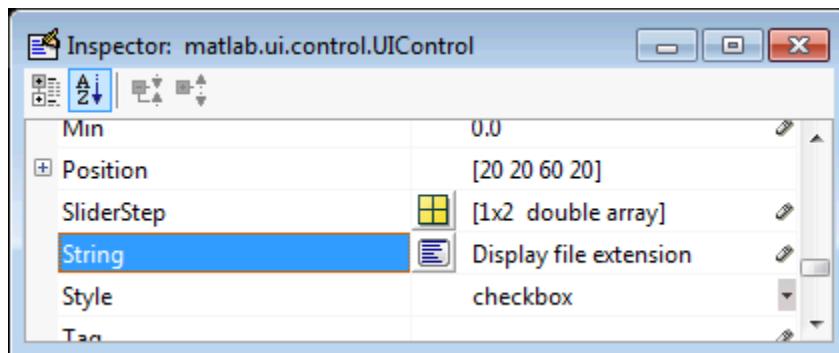
To manage exclusive selection of radio buttons and toggle buttons, put them in a button group. See “Button Group” on page 18-24 for more information.

Check Box

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

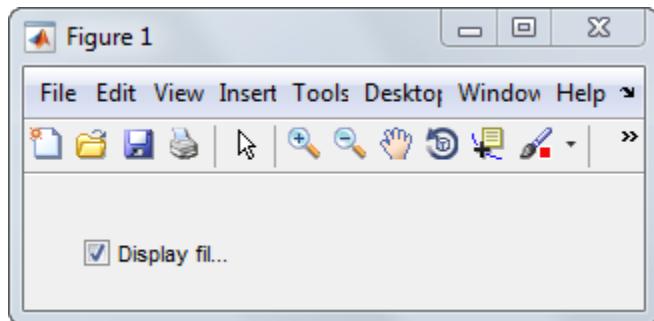


- Specify the check box label by setting the **String** property to the desired label, in this case, **Display file extension**.



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**.

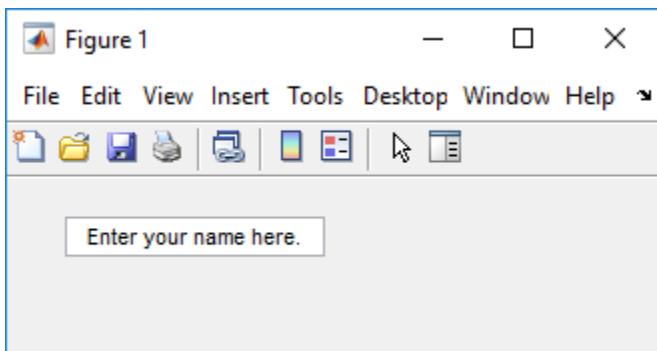
The check box accommodates only a single line of text. If you specify a component width that is too small to accommodate the specified **String** property value, MATLAB software truncates the value with an ellipsis.



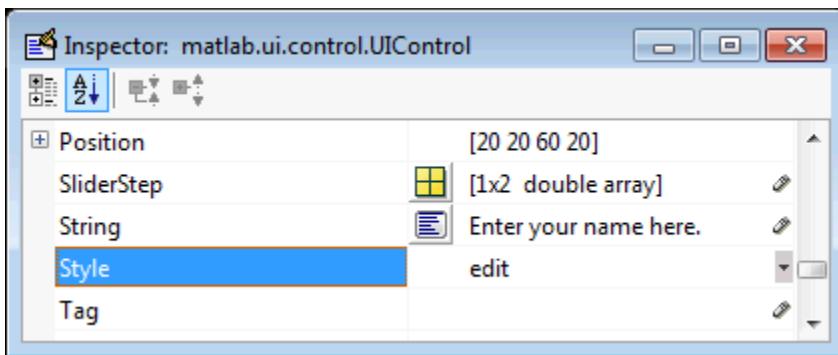
- Create the check box with the box checked by setting the **Value** property to the value of the **Max** property (default is 1). Set **Value** to **Min** (default is 0) to leave the box unchecked. Correspondingly, when the user clicks the check box, the software sets **Value** to **Max** when the user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

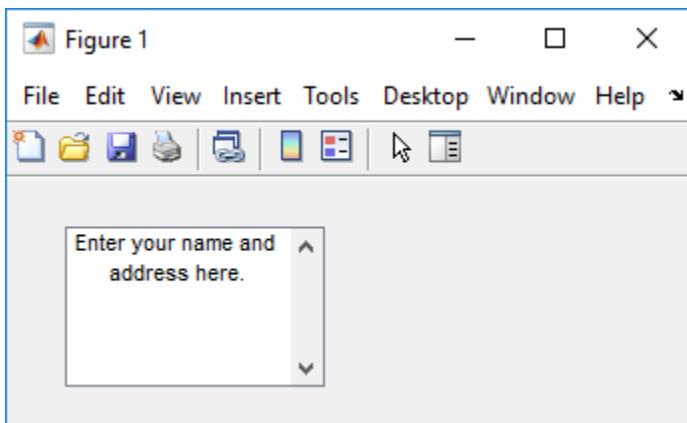


- Specify the text to be displayed when the edit text component is created by setting the **String** property to the desired value, in this case, **Enter your name here**.

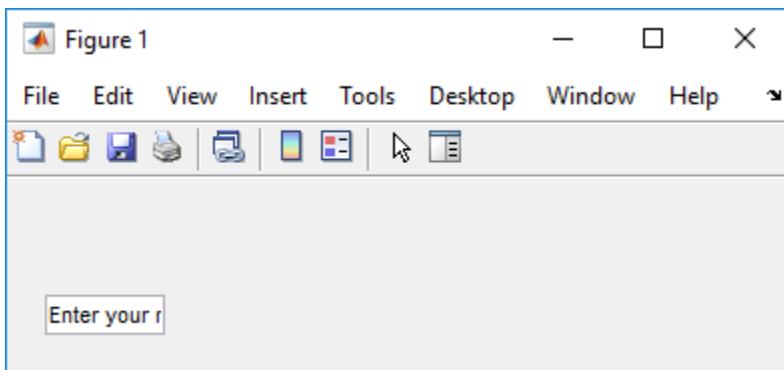


To display the & character in a label, use two & characters. The words **remove**, **default**, and **factory** (case sensitive) are reserved. To use one of these as a label, prepend a backslash character (\). For example, \remove yields **remove**.

- 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.

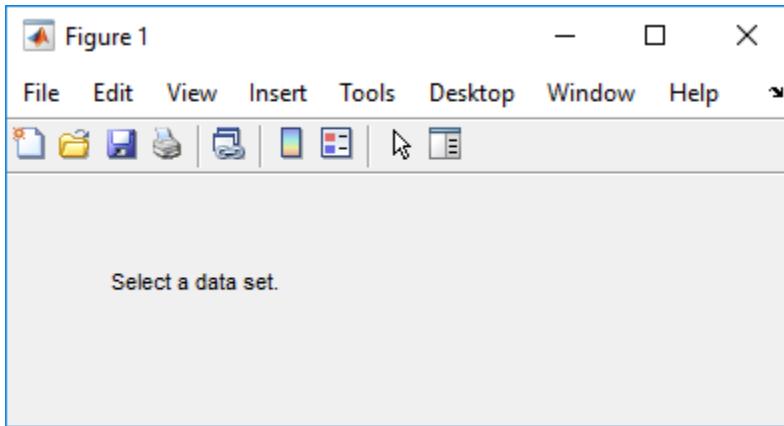


- If you want to set the position or size of the component to an exact value, then modify its **Position** property.
- You specify the text font to display in the edit box by typing the name of a font residing on your system into the **FontName** entry in the Property Inspector. On Microsoft® Windows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

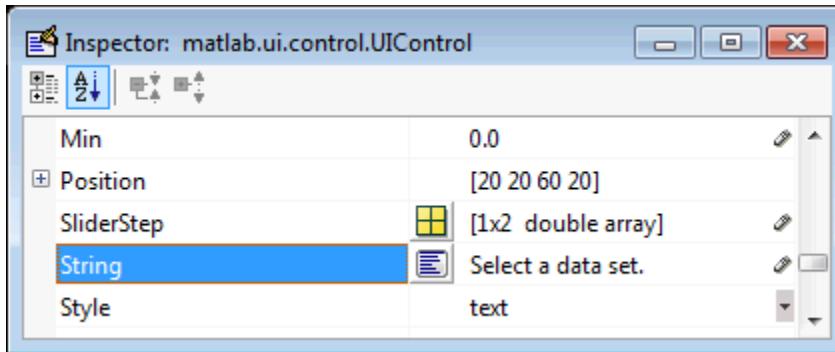
Tip To find out what fonts are available, type **uisetfont** at the MATLAB prompt; a dialog displays containing a list box from which you can select and preview available fonts. When you select a font, its name and other characteristics are returned in a structure, from which you can copy the **FontName** 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:

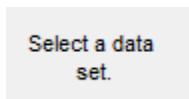


- Specify the text that appears in the component by setting the component **String** property to the desired text, in this case **Select a data set.**



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**.

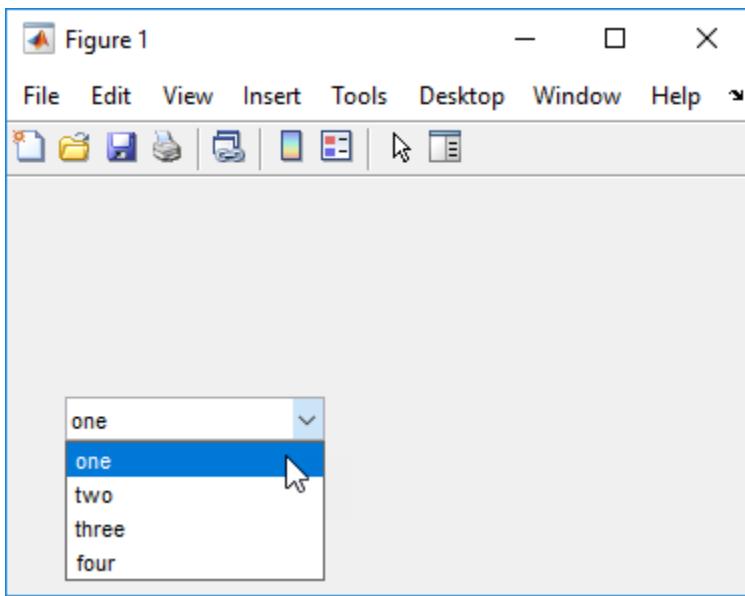
If your component is not wide enough to accommodate the specified value, MATLAB wraps the displayed text.



- 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 18-14.

Pop-Up Menu

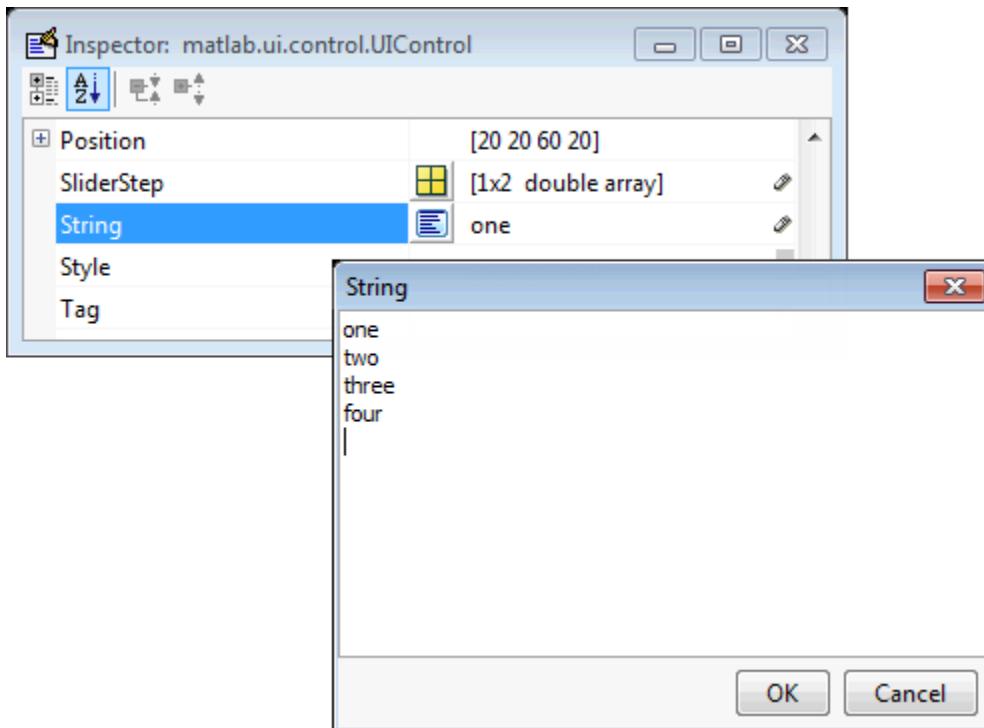
To create a pop-up menu (also known as a drop-down menu or combo box) with items **one**, **two**, **three**, and **four**, as shown in this figure:



- Specify the pop-up menu items to be displayed by setting the **String** property to the desired items. Click the



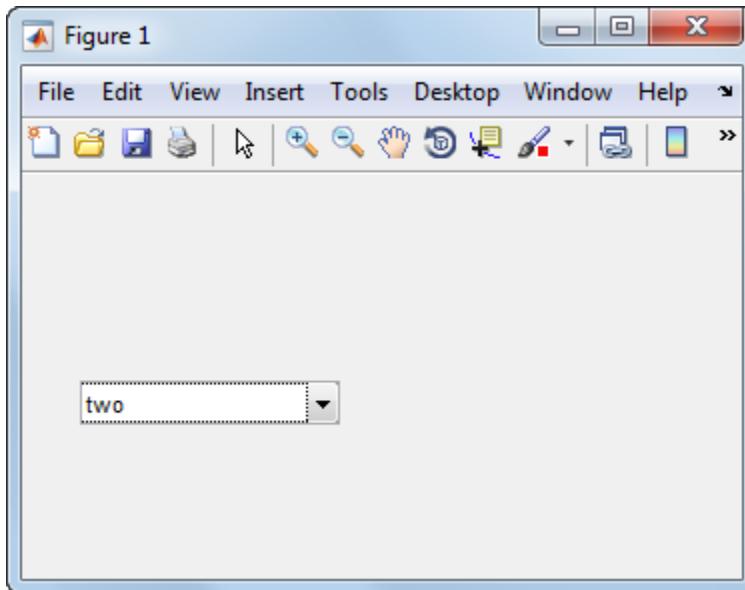
button to the right of the property name to open the Property Inspector editor.



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.

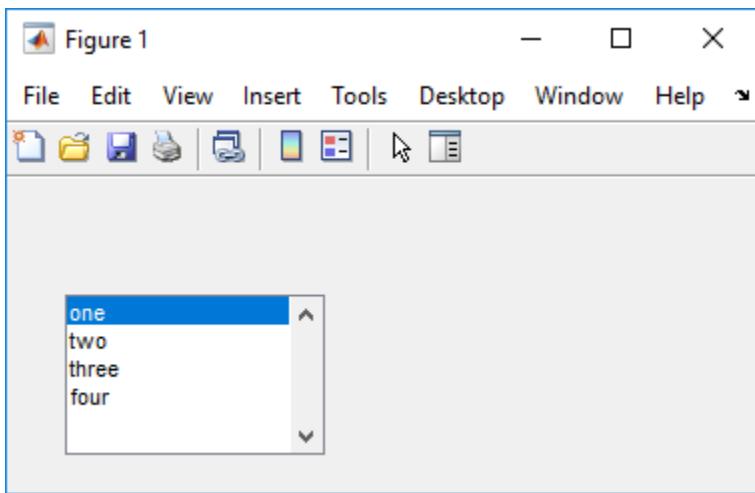
- To select an item when the component is created, set **Value** to a scalar that indicates the index of the selected list item, where 1 corresponds to the first item in the list. If you set **Value** to 2, the menu looks like this when it is created:



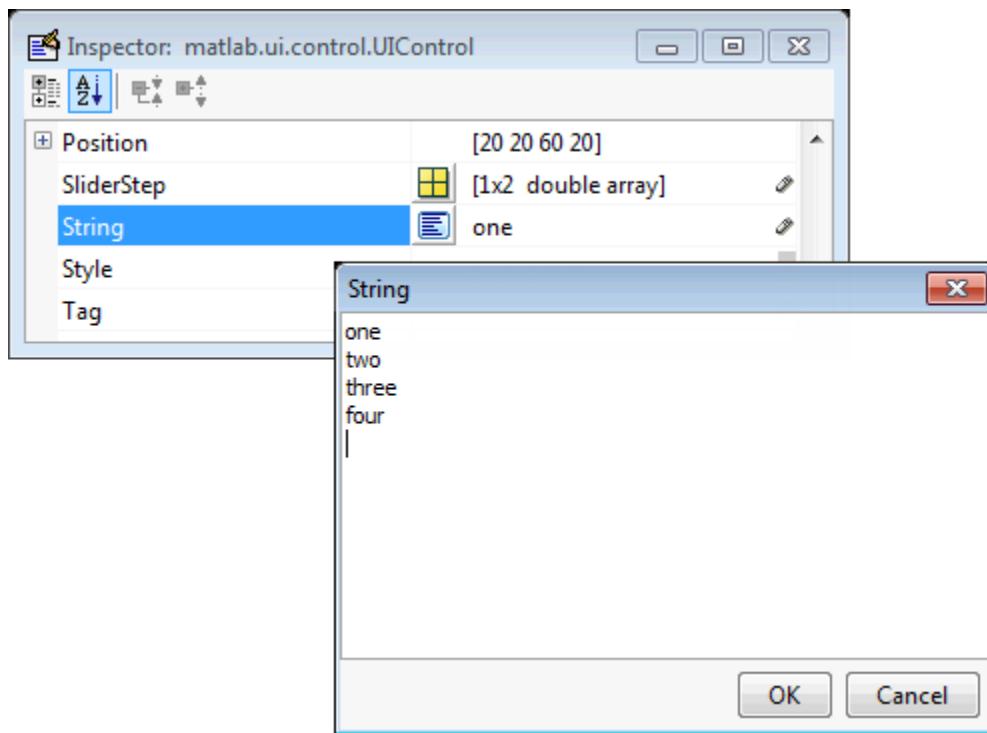
- If you want to set the position and size of the component to exact values, then modify its **Position** property. The height of a pop-up menu is determined by the font size. The height you set in the position vector is ignored.
- The pop-up menu does not let you add a label. Use a "Static Text" on page 18-15 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:



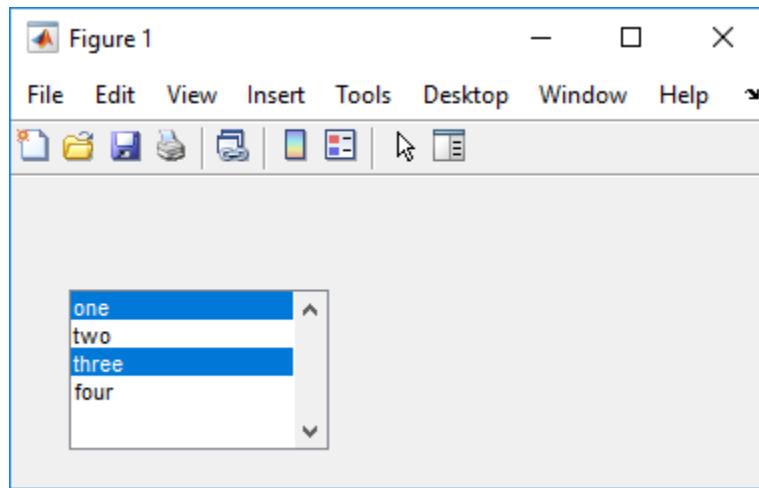
- Specify the list of items to be displayed by setting the **String** property to the desired list. Use the Property Inspector editor to enter the list. You can open the editor by clicking the  button to the right of the property name.



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.

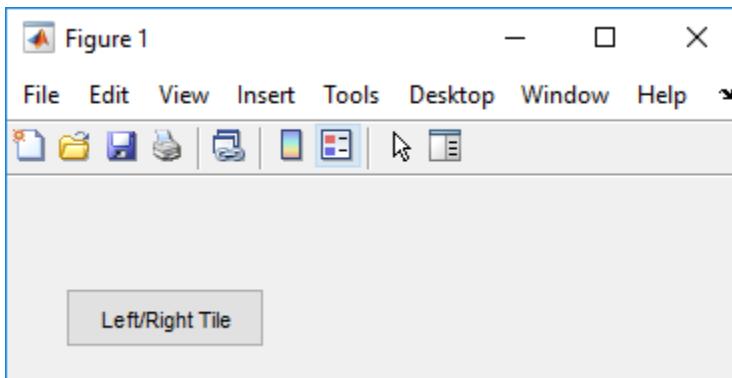


To enable selection of more than one item, you must specify the **Max** and **Min** properties so that their difference is greater than 1. For example, **Max = 2**, **Min = 0**. **Max** default is 1, **Min** default is 0.

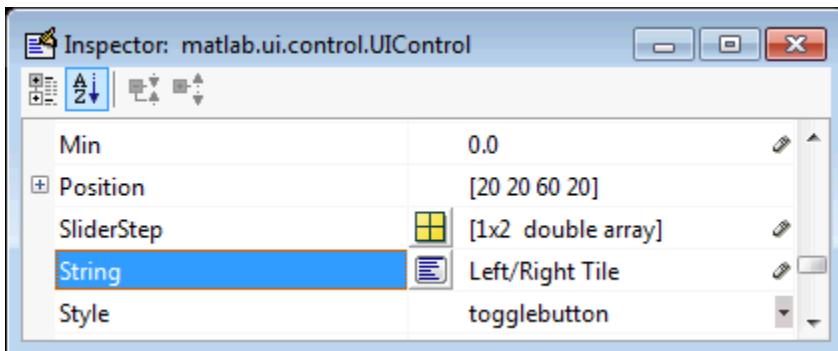
- If you want no initial selection, set the **Max** and **Min** properties to enable multiple selection, i.e., **Max - Min > 1**, and then set the **Value** property to an empty matrix **[]**.
- If the list box is not large enough to display all list entries, you can set the **ListBoxTop** property to the index of the item you want to appear at the top when the component is created.
- If you want to set the position or size of the component to an exact value, then modify its **Position** property.
- The list box does not provide for a label. Use a “Static Text” on page 18-15 component to label the list box.

Toggle Button

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

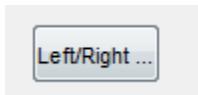


- Specify the toggle button label by setting its **String** property to the desired label, in this case, **Left/Right Tile**.

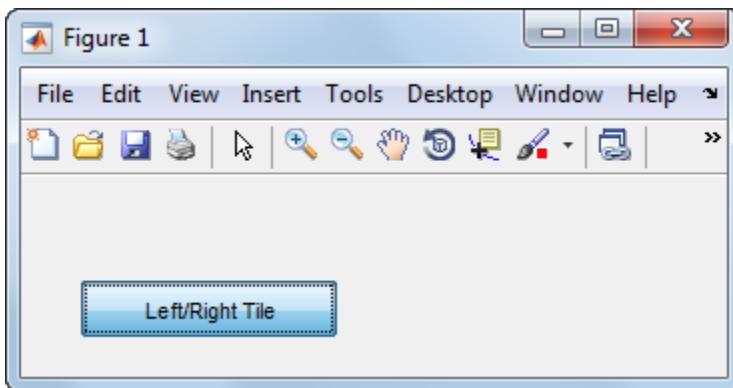


To display the & character in a label, use two & characters. The words **remove**, **default**, and **factory** (case sensitive) are reserved. To use one of these as a label, prepend a backslash character (\). For example, \remove yields **remove**.

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.



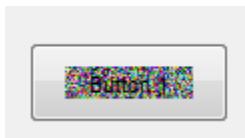
- Create the toggle button with the button selected (depressed) by setting its **Value** property to the value of its **Max** property (default is 1). Set **Value** to **Min** (default is 0) to leave the toggle button unselected (raised). Correspondingly, when the user selects the toggle button, MATLAB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



To manage exclusive selection of radio buttons and toggle buttons, put them in a button group. See “Button Group” on page 18-24 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 18-22
- “Panel” on page 18-23
- “Button Group” on page 18-24

Commonly Used Properties

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

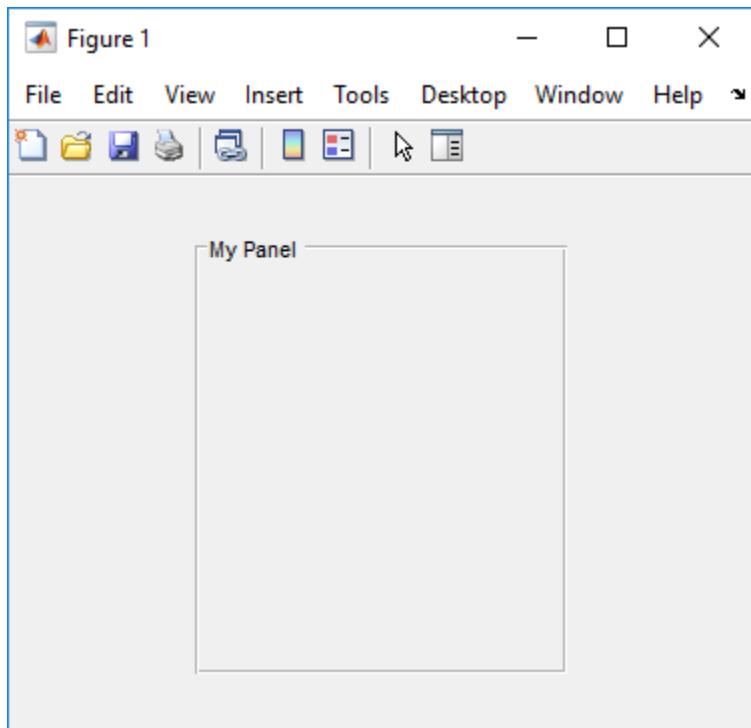
Property	Values	Description
Position	4-element vector: [distance from left, distance from bottom, width, height].	Size of the component and its location relative to its parent.
Title	Character vector (for example, 'Start').	Component label.
TitlePosition	lefttop, center top, righttop, leftbottom, centerbottom, 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

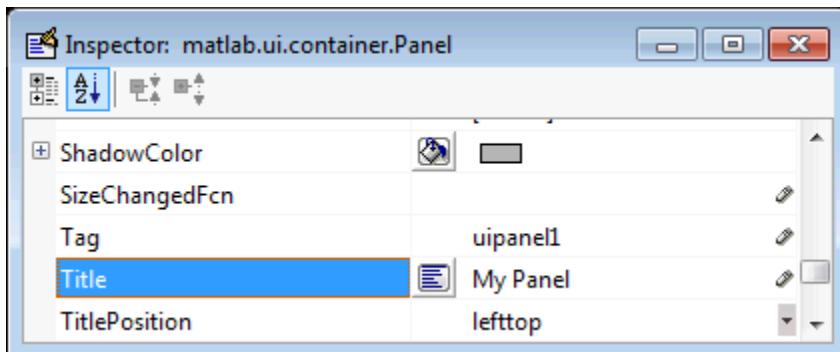
For a complete list of properties and for more information about the properties listed in the table, see the **Uipanel** and **Uibuttongroup**.

Panel

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

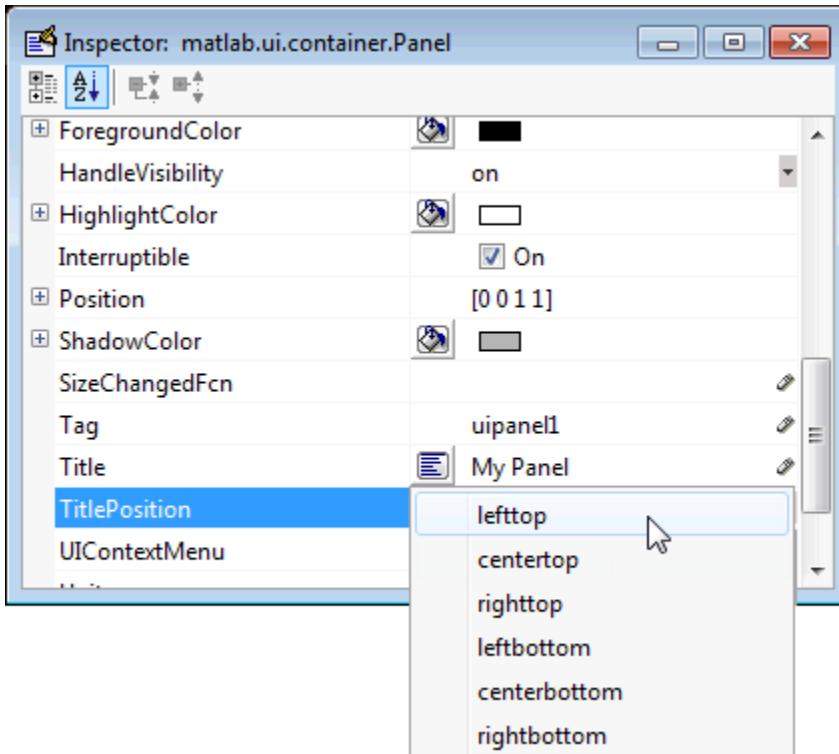


- Specify the panel title by setting the **Title** property to the desired value, in this case **My Panel**.



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**.

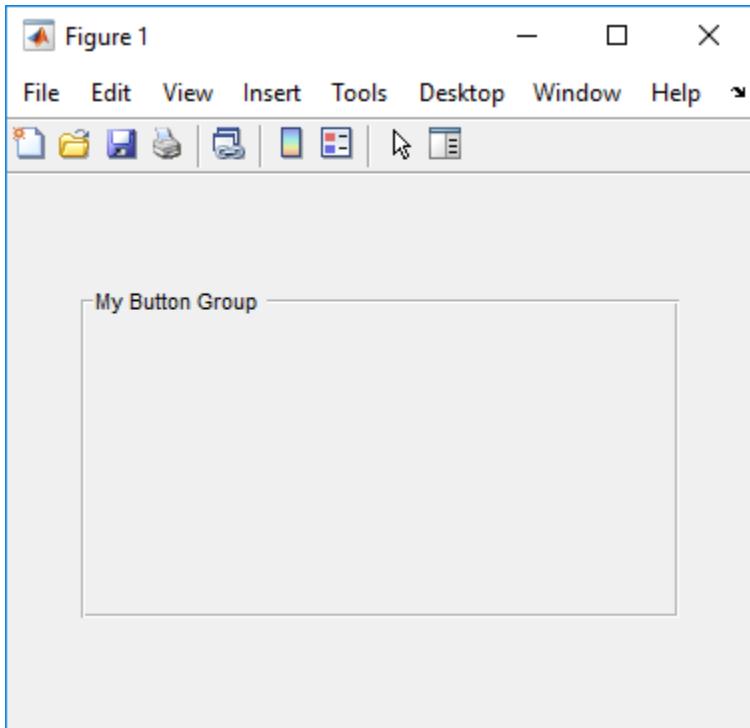
- Specify the location of the panel title by selecting one of the available **TitlePosition** property values from the pop-up menu, in this case **lefttop**. You can position the title at the left, middle, or right of the top or bottom of the panel.



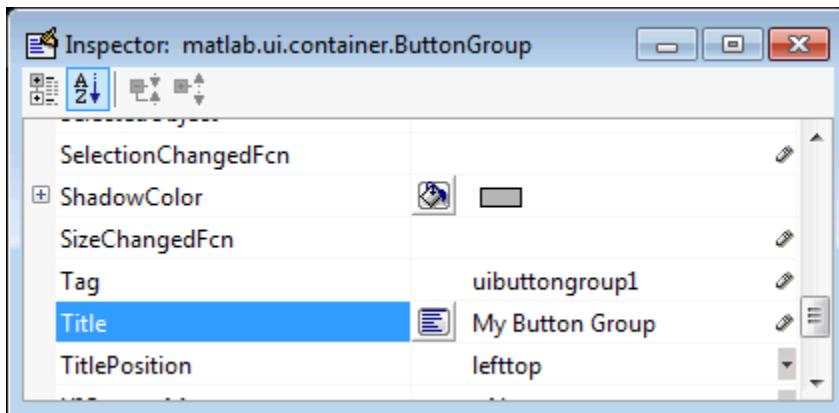
- If you want to set the position or size of the panel to an exact value, then modify its **Position** property.

Button Group

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

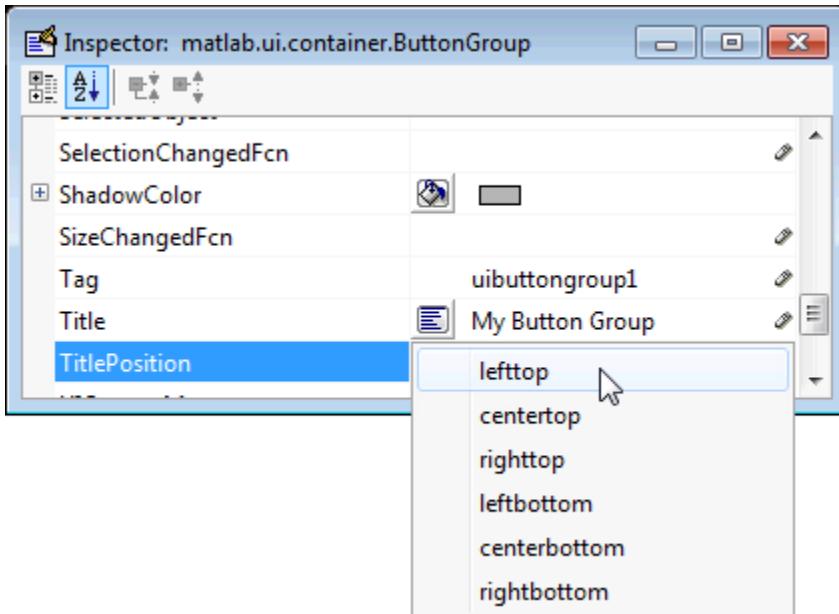


- Specify the button group title by setting the **Title** property to the desired value, in this case **My Button Group**.



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 18-26
- “Create Axes” on page 18-27

Commonly Used Properties

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

Property	Values	Description
NextPlot	<code>add</code> , <code>replace</code> , <code>replacechildren</code> . Default is <code>replace</code>	Specifies whether plotting adds graphics, replaces graphics and resets axes properties to default, or replaces graphics only.

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.
Units	normalized, centimeters, characters, inches, pixels, points. Default is normalized.	Units of measurement used to interpret position vector

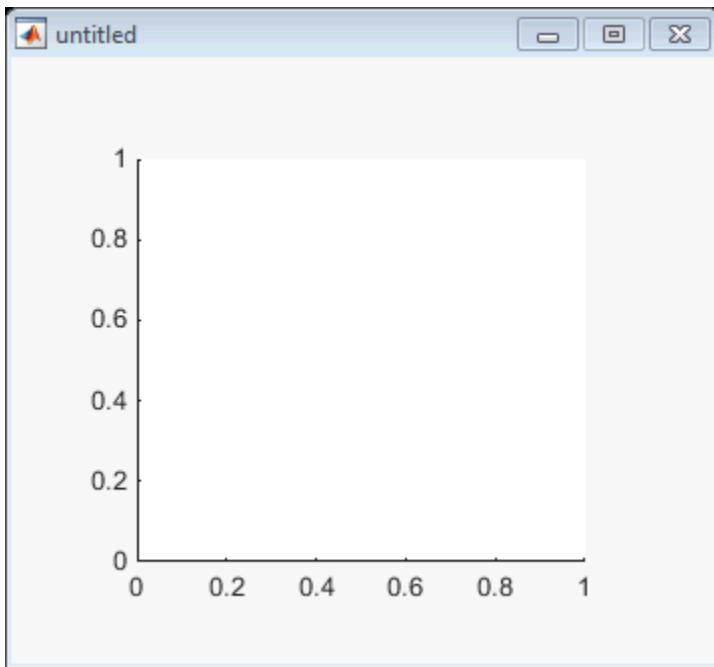
For a complete list of properties and for more information about the properties listed in the table, see [Axes](#).

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 18-27 for information about setting the `NextPlot` property.

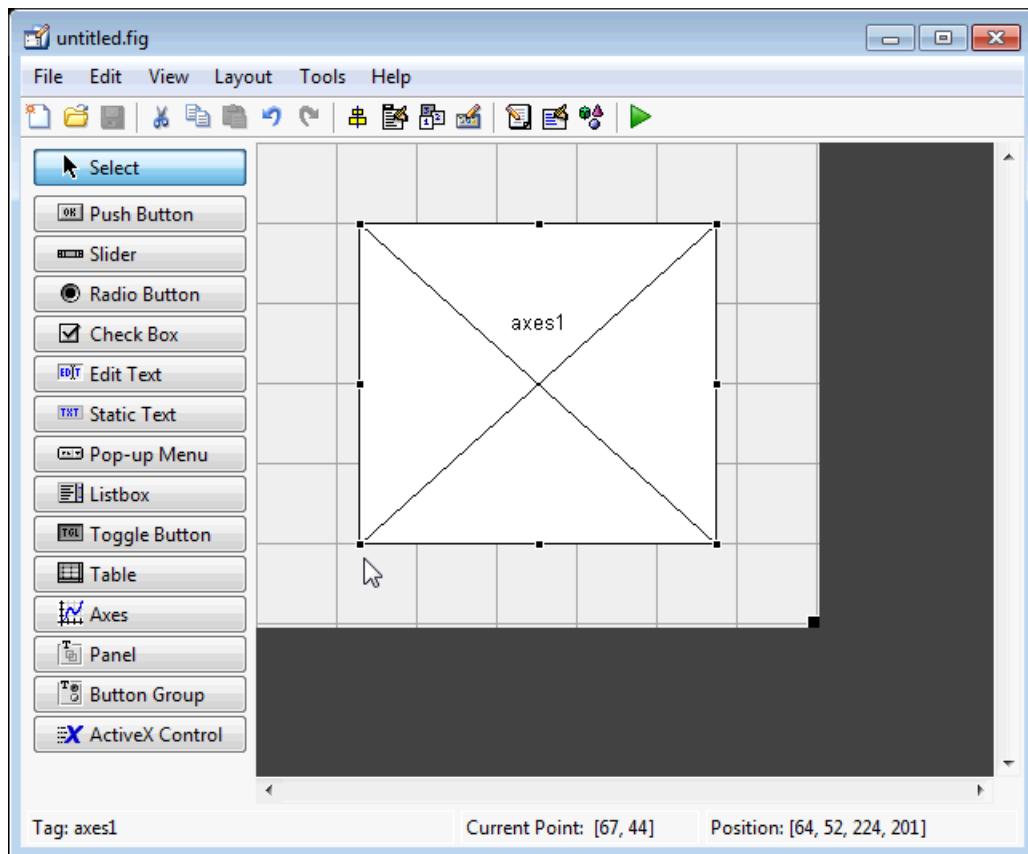
Create Axes

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.



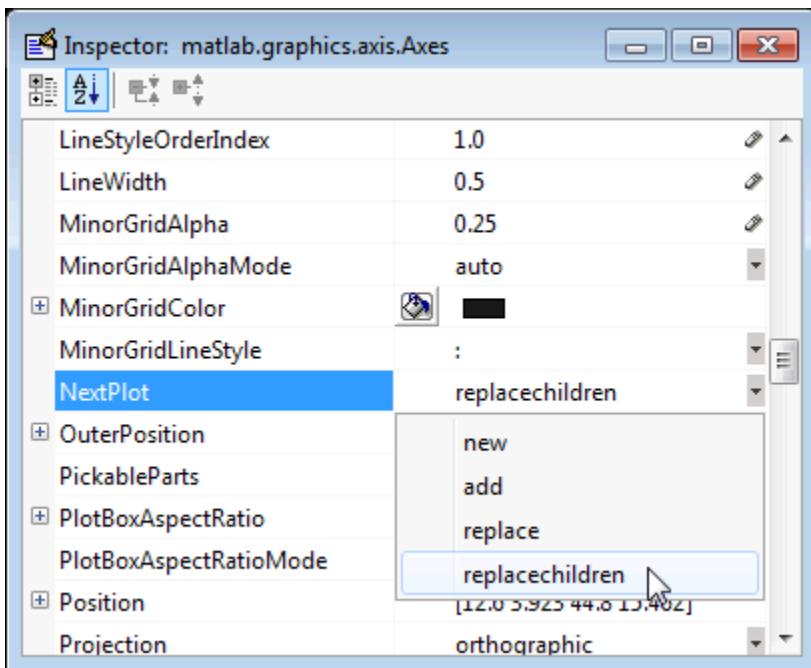
- Use the `title`, `xlabel`, `ylabel`, `zlabel`, and `text` functions in the code file to label an axes component. For example,

```
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

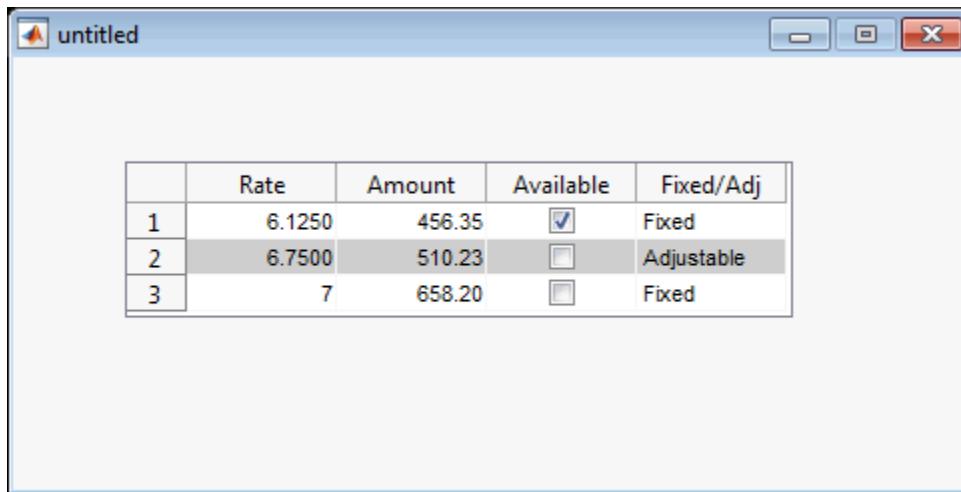
The most commonly used properties of a table component are listed in the table below. These are grouped in the order they appear in the Table Property Editor. Please refer to [uitable](#) documentation for detail of all the table properties:

Group	Property	Values	Description
Column	ColumnName	1-by- <i>n</i> 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- <i>n</i> cell array or 'auto'	Width of each column in pixels; individual column widths can also be set to 'auto'
	ColumnEditable	logical 1-by- <i>n</i> matrix scalar logical value empty matrix ([])	Determines data in a column as editable
Row	RowName	1-by- <i>n</i> cell array of character vectors	Row header label names

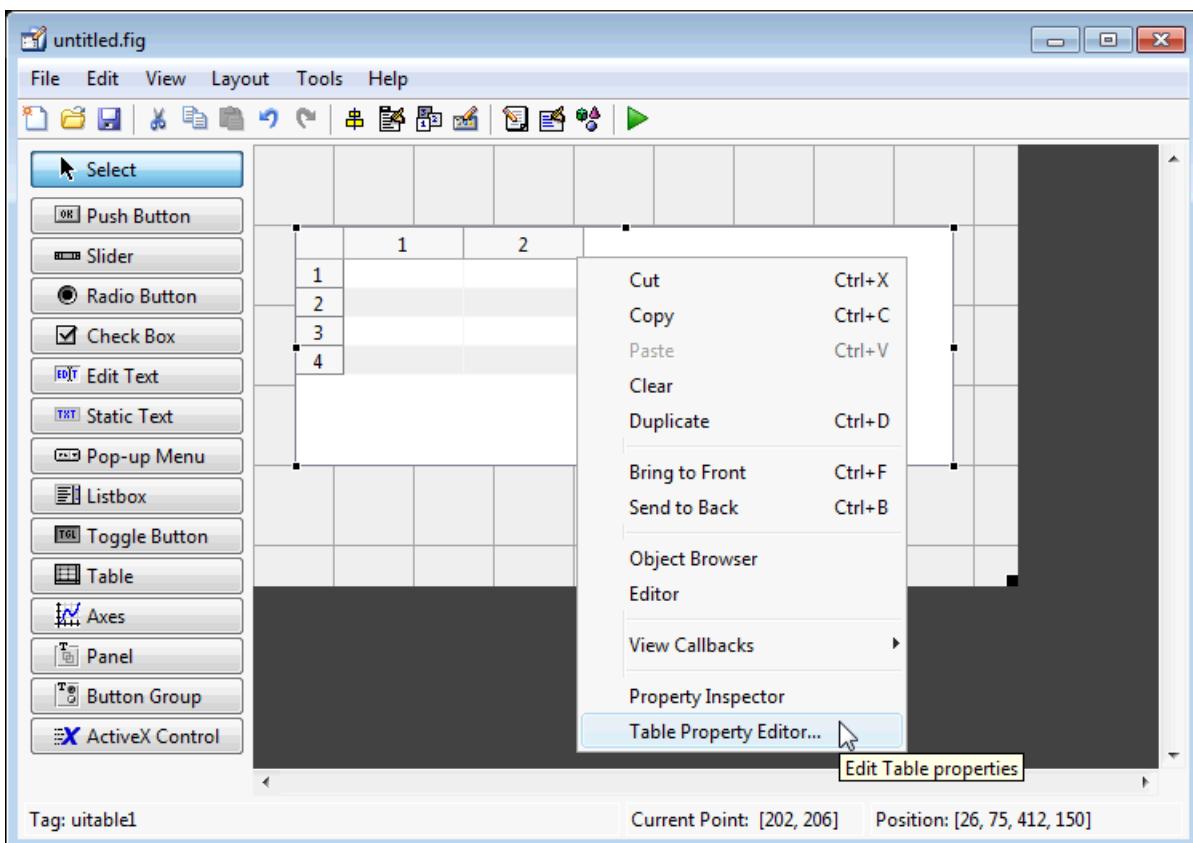
Group	Property	Values	Description
Color	BackgroundColor	<i>n</i> -by-3 matrix of RGB triples	Background color of cells
	RowStriping	{on} off	Color striping of table rows
Data	Data	Matrix or cell array of numeric, logical, or character data	Table data.

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

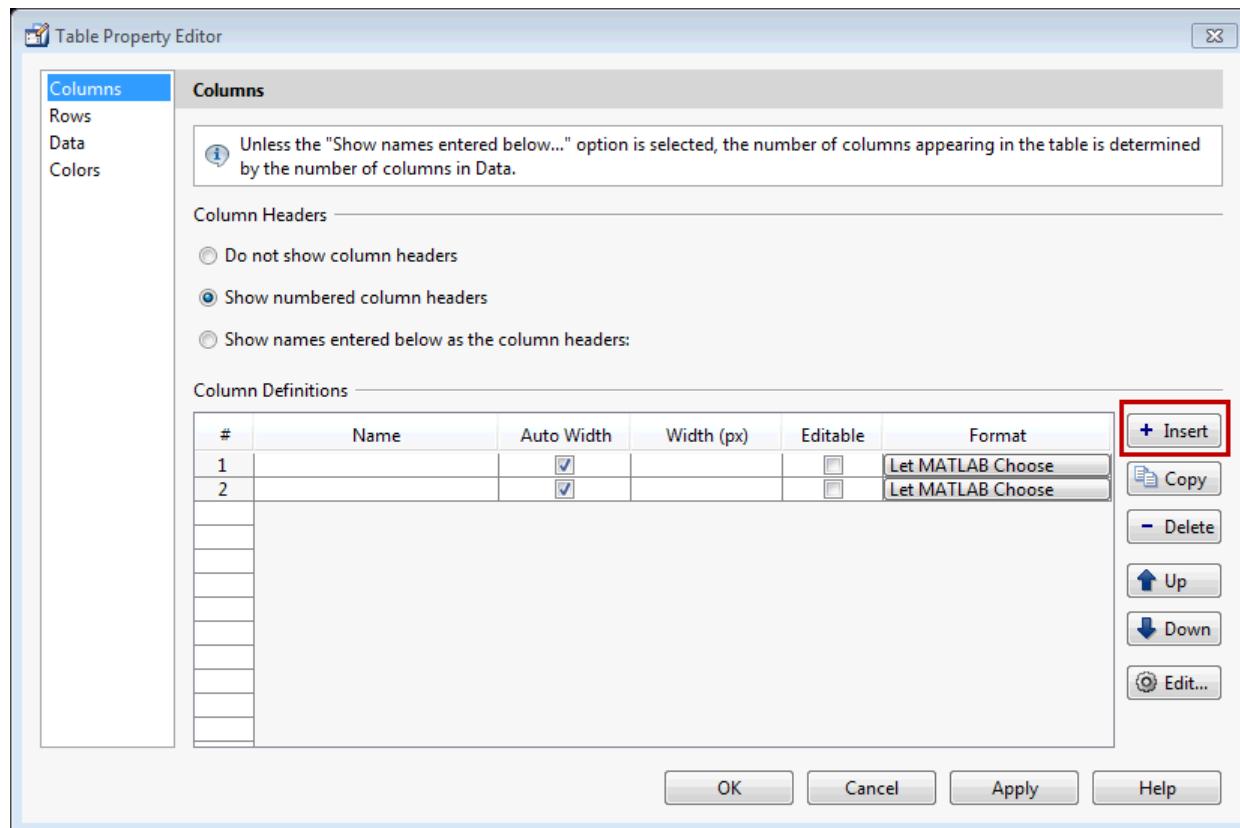
Button	Purpose	Accelerator Keys	
		Windows	Macintosh
Up	Moves selected entry up one row	Ctrl+uparrow	Cmd+uparrow
Down	Moves selected entry down one row	Ctrl+downarrow	Cmd+downarrow

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

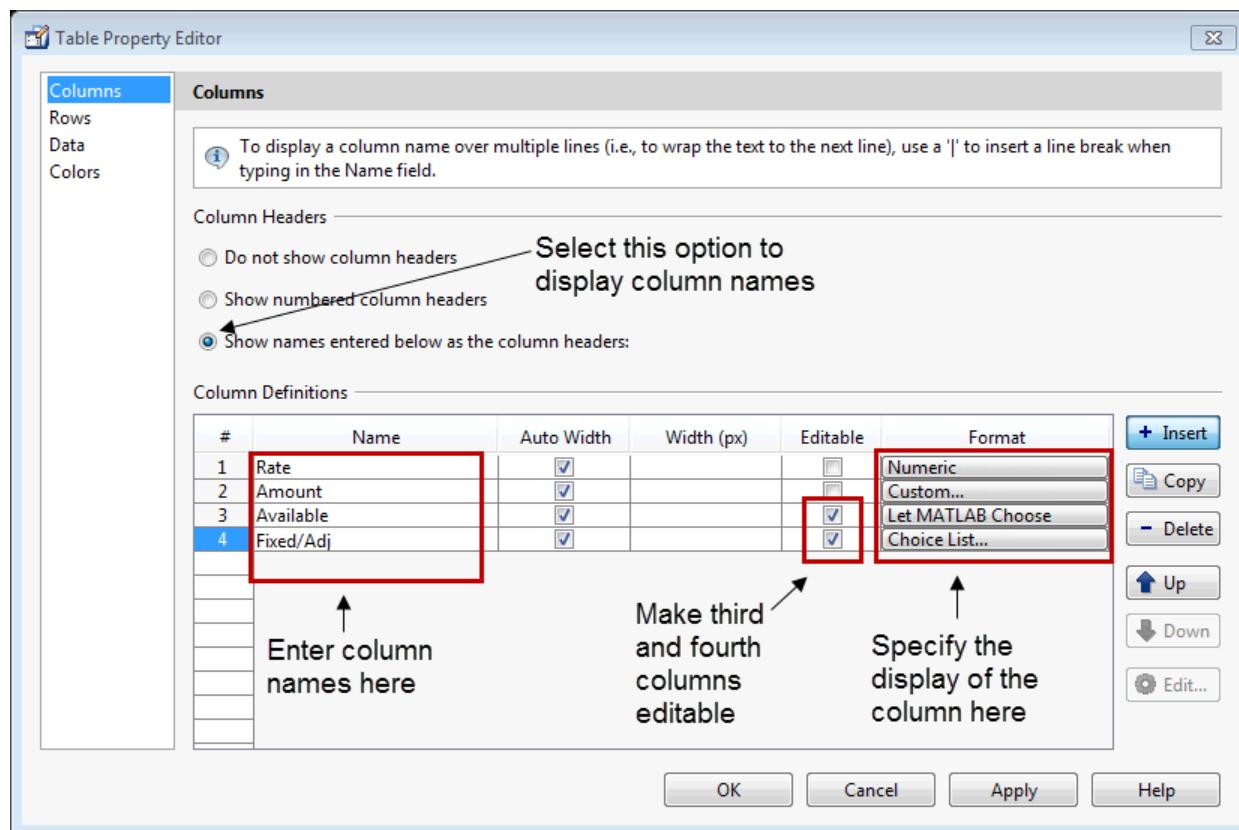
To save changes to the table you make in the Table Property Editor, click **OK**, or click **Apply** commit changes and keep on using the Table Property Editor.

Set Column Properties

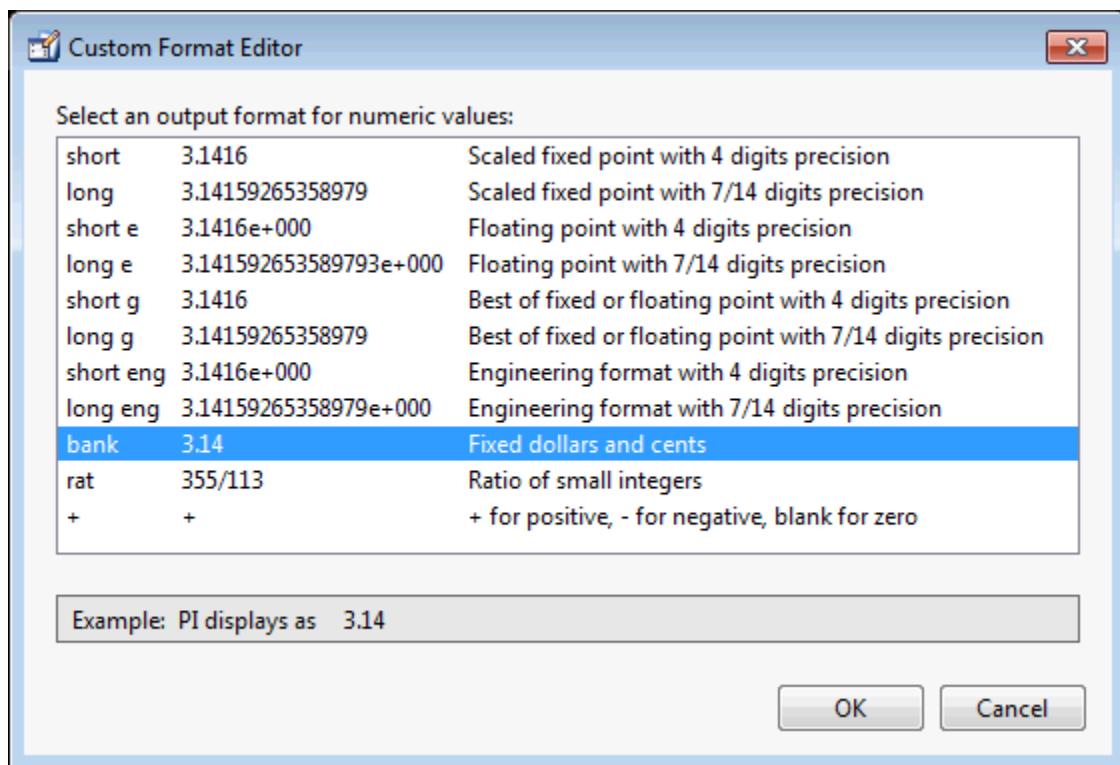
Click **Insert** to add two more columns.



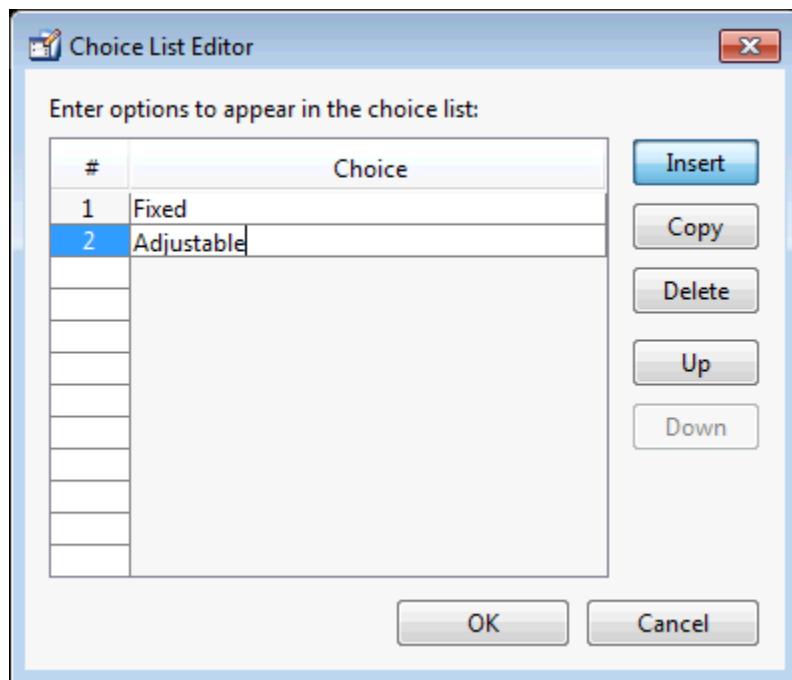
Select **Show names entered below as the column headers** and set the **ColumnName** by entering Rate, Amount, Available, and Fixed/Adj in **Name** group. for the Available and Fixed/Adj columns set the **ColumnEditable** property to on. Lastly set the **ColumnFormat** for the four columns



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



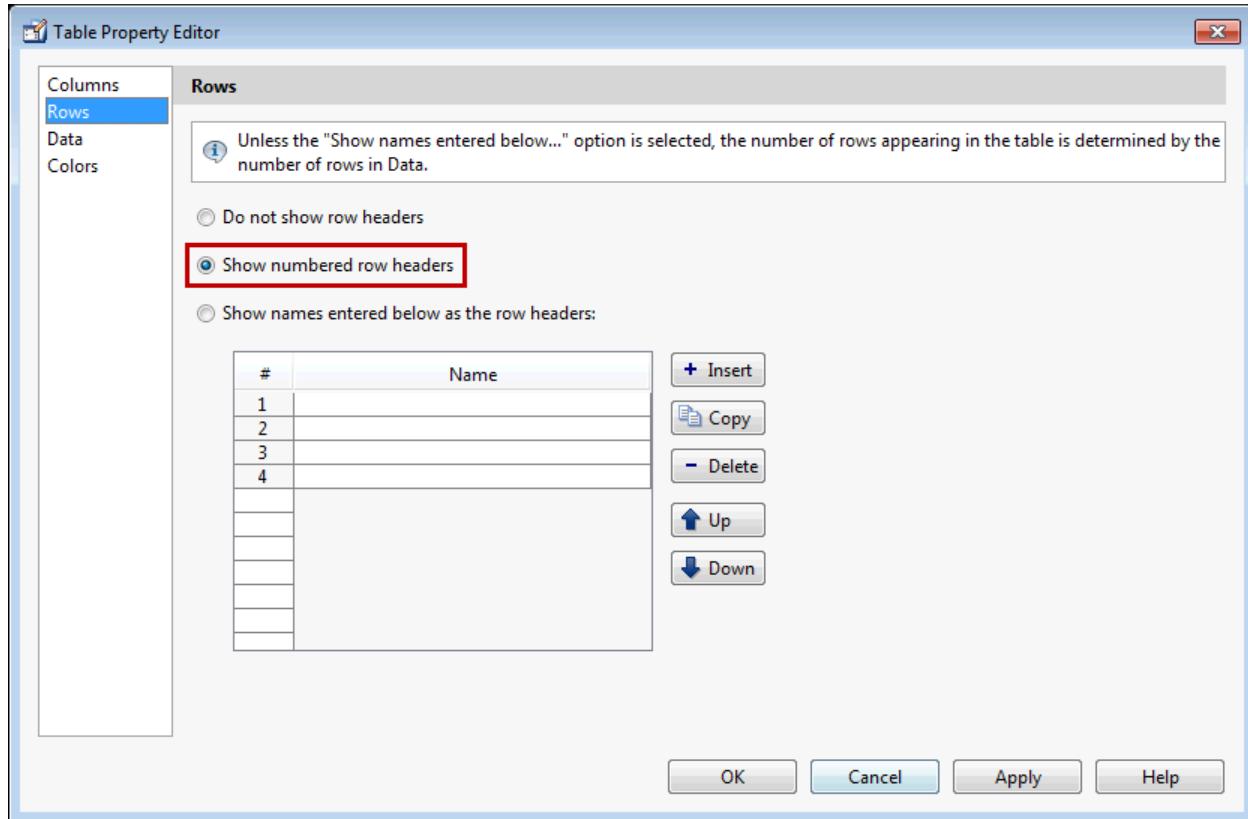
Leave the Available column at the default value. This allows MATLAB to chose based on the value of the Data property of the table. For the Fixed/Adj column select Choice List to create a pop-up menu. In the Choice List Editor, click **Insert** to add a second choice and type Fixed and Adjustable as the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

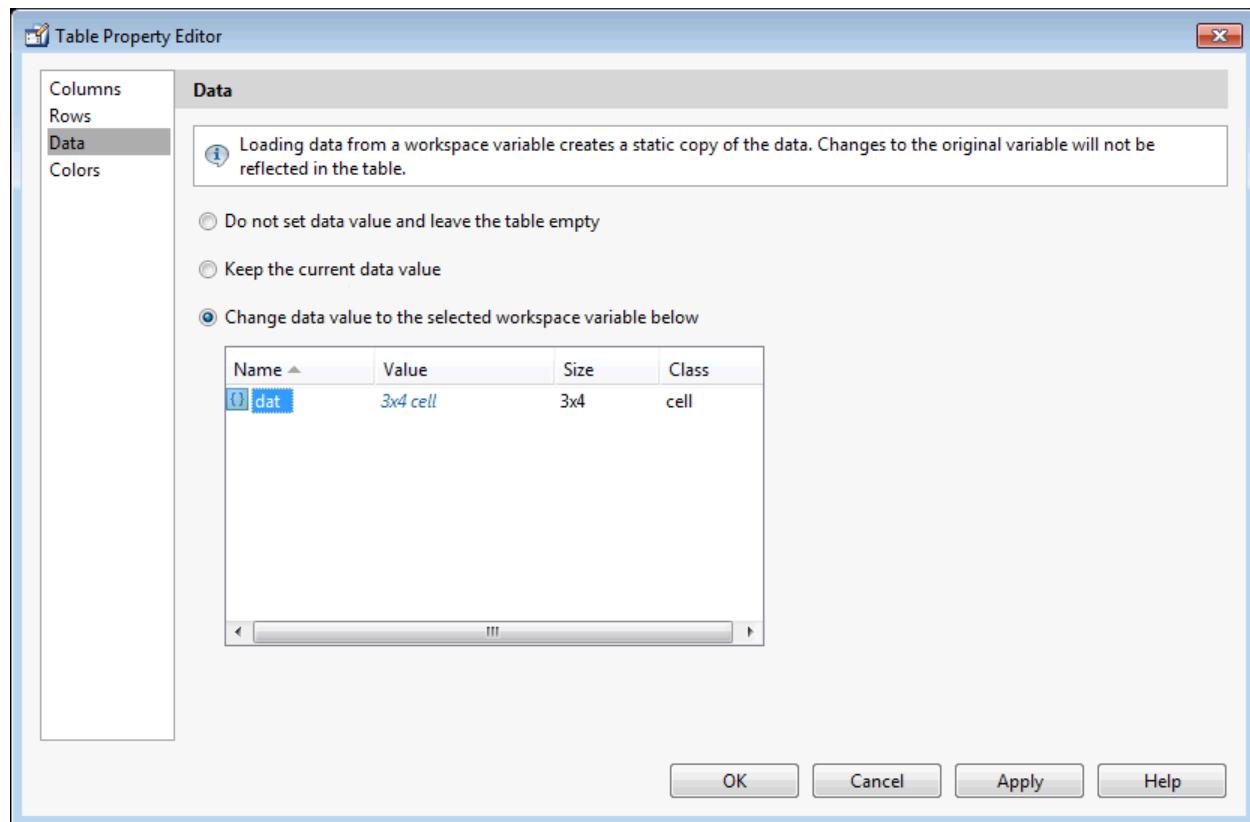


Set Data Properties

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:

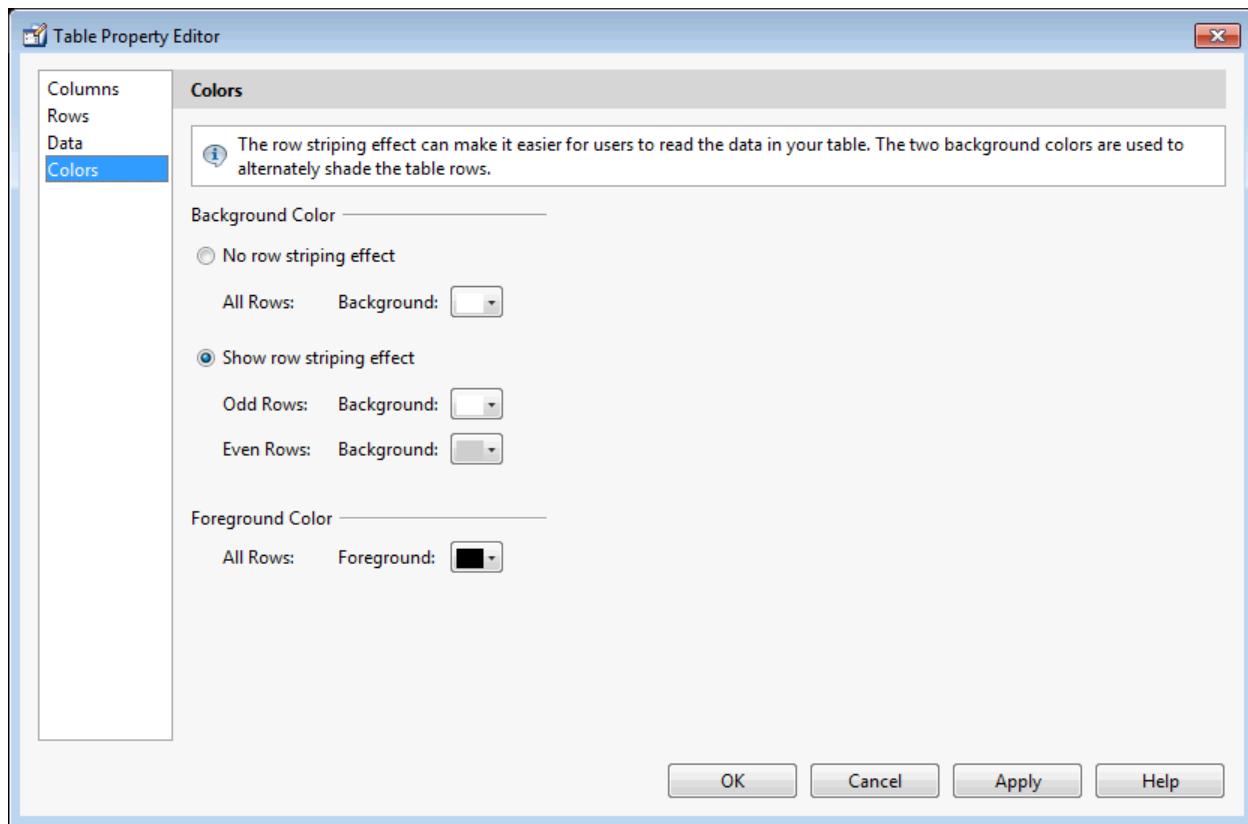
```
dat = {6.125, 456.3457, true, 'Fixed';...
6.75, 510.2342, false, 'Adjustable';...
7, 658.2, false, 'Fixed'};
```

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 suitable properties to the table via the Property Inspector.

Resize GUIDE UI Components

You can resize components in one of the following ways:

- “Drag a Corner of the Component” on page 18-37
- “Set the Component's Position Property” on page 18-37

Drag a Corner of the Component

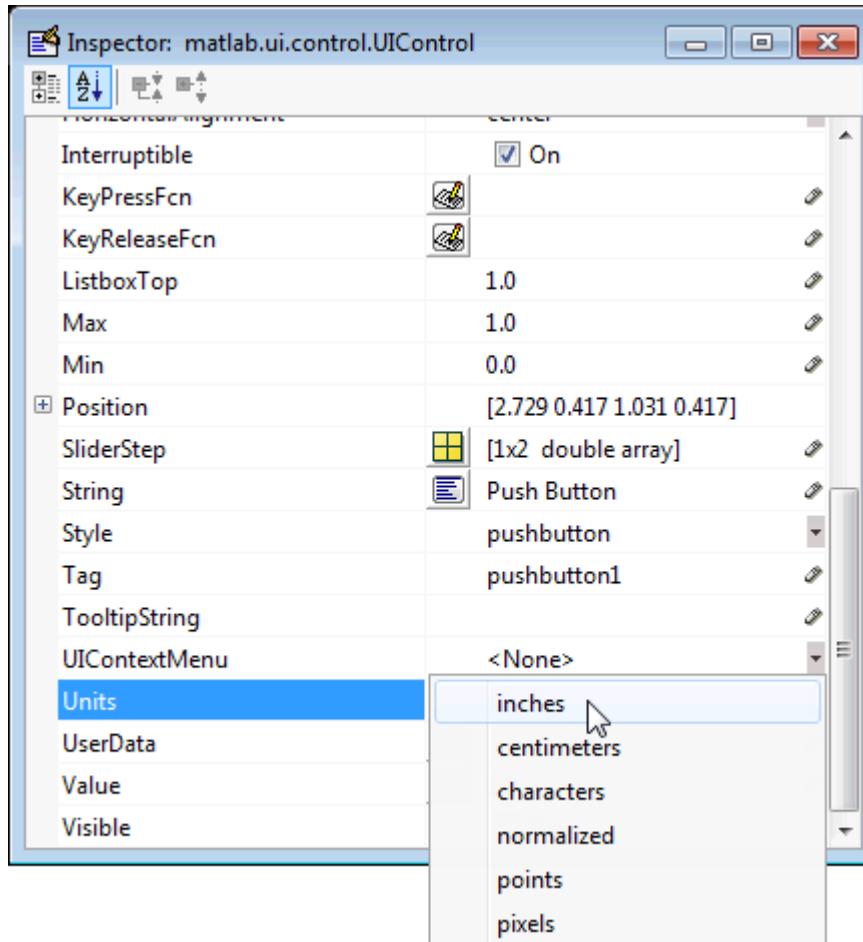
Select the component you want to resize. Click one of the corner handles and drag it until the component is the desired size.



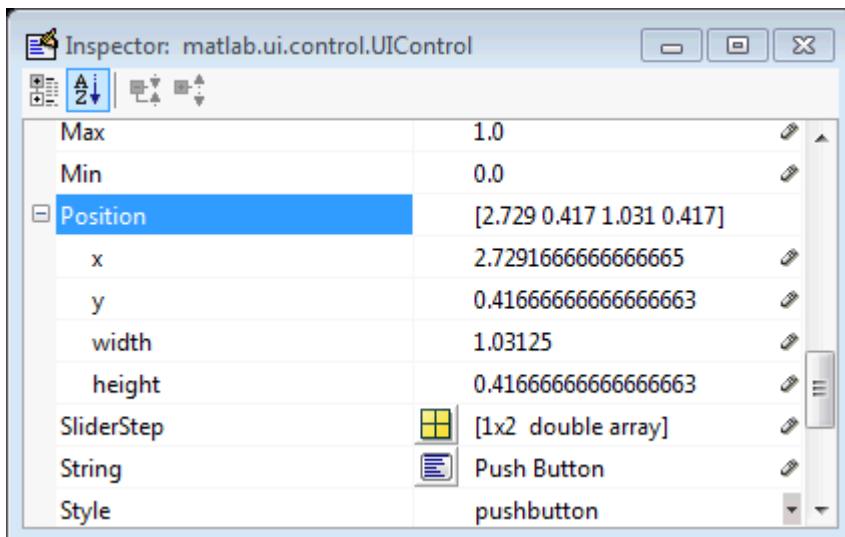
Set the Component's Position Property

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.

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 3-2
- “Write Callbacks in GUIDE” on page 19-2
- “Callbacks for Specific Components” on page 19-14
- “Lay Out Apps in App Designer Design View” on page 6-2
- “App Building Components” on page 5-2

Create Menus for GUIDE Apps

In this section...

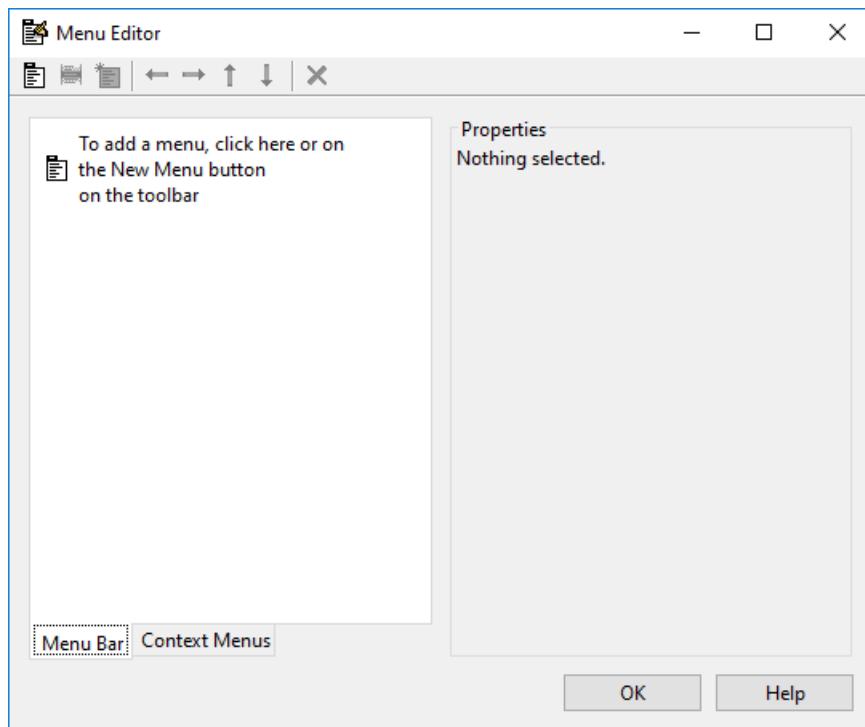
[“Menus for the Menu Bar” on page 18-40](#)

[“Context Menus” on page 18-47](#)

Note The GUIDE environment will be removed in a future release. After GUIDE is removed, existing GUIDE apps will continue to run in MATLAB but they will not be editable in GUIDE.

To continue editing an existing GUIDE app, see “GUIDE Migration Strategies” on page 4-5 for information on how to help maintain compatibility of the app with future MATLAB releases. To create new apps interactively, “Develop Apps Using App Designer” instead.

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 18-41](#)
- [“Add Standard Menus to the Menu Bar” on page 18-42](#)
- [“Create a Menu” on page 18-42](#)
- [“Add Items to a Menu” on page 18-43](#)

- “Additional Drop-Down Menus” on page 18-45
- “Cascading Menus” on page 18-45

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.

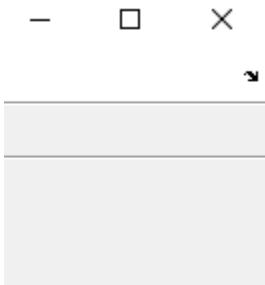


Figure windows with a standard menu bar also have a **Desktop** menu from which the user can dock and undock them.

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 `WindowState` 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 `WindowState` is no longer set to 'docked'.
- If you undock a UI window created with `WindowState` '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.

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

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

Add Standard Menus to the Menu Bar

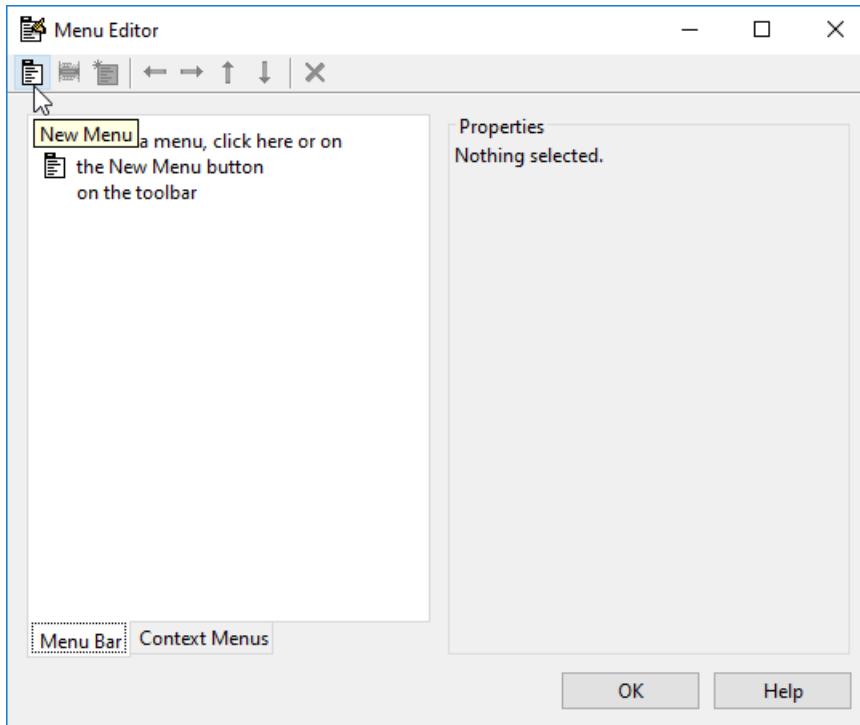
The figure `MenuBar` property controls whether your UI displays the MATLAB standard menus on the menu bar. GUIDE initially sets the value of `MenuBar` to `none`. If you want your UI to display the MATLAB standard menus, use the Property Inspector to set `MenuBar` to `figure`.

- 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`'.

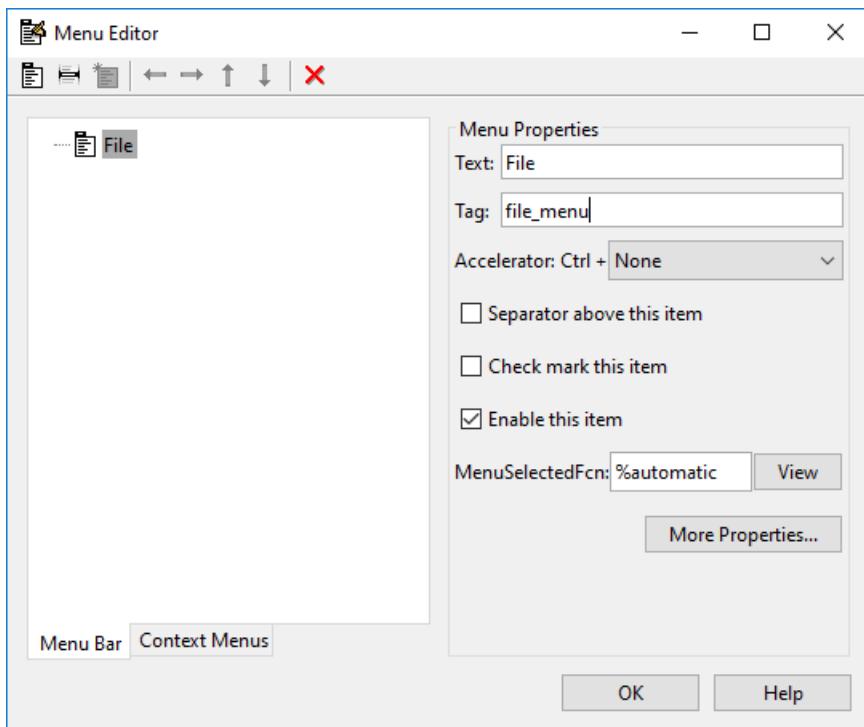
Create a Menu

- 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.



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.

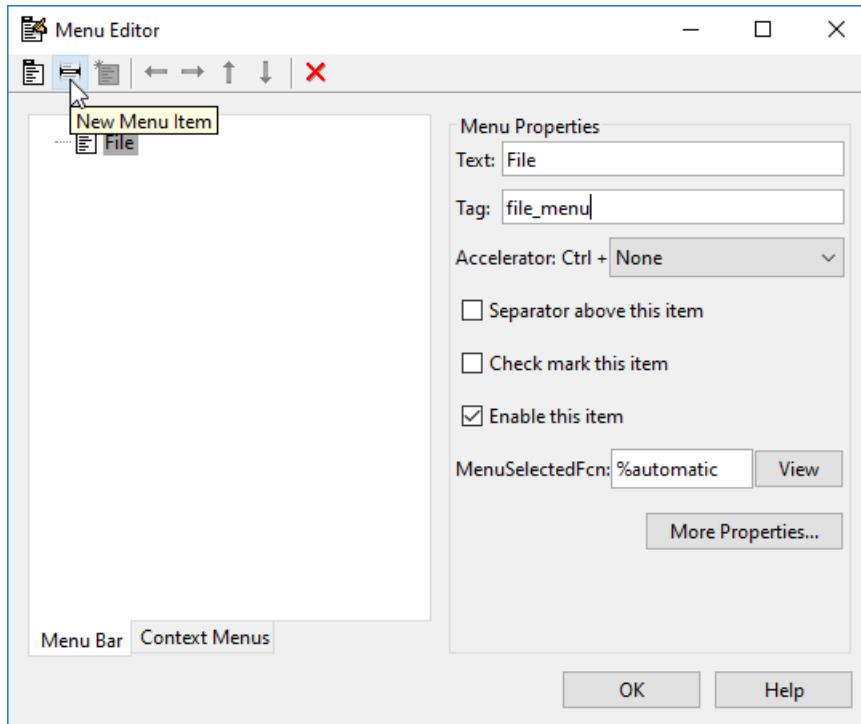
Text is a text label for the menu item. To display the & character in a label, use two & characters. The words **remove**, **default**, and **factory** (case sensitive) are reserved. To use one of these as labels, prepend a backslash character (\). For example, \remove yields **remove**.

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.

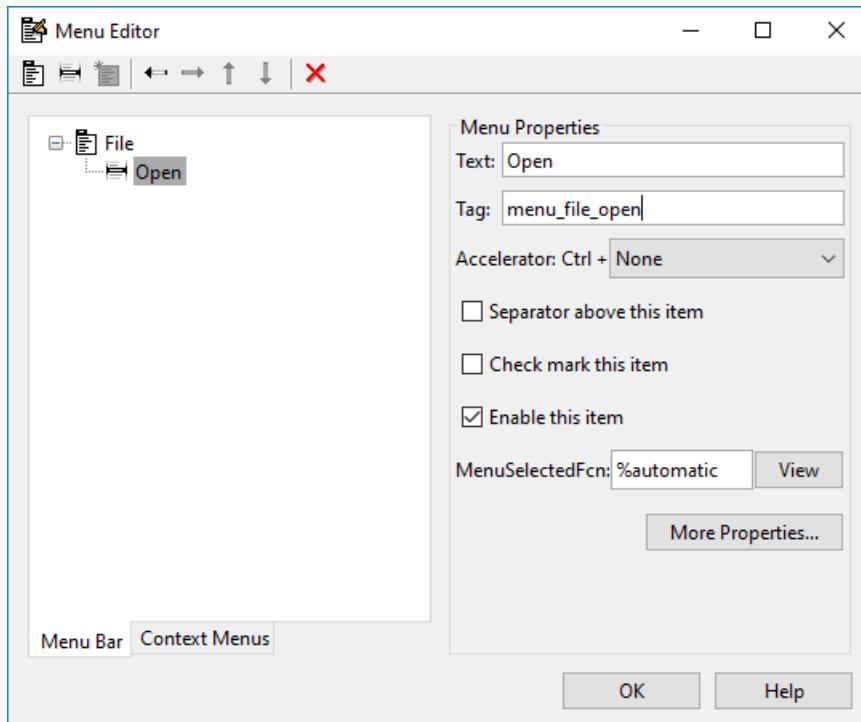
Add Items to a Menu

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.



- 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 18-48.
- 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 19-21 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.

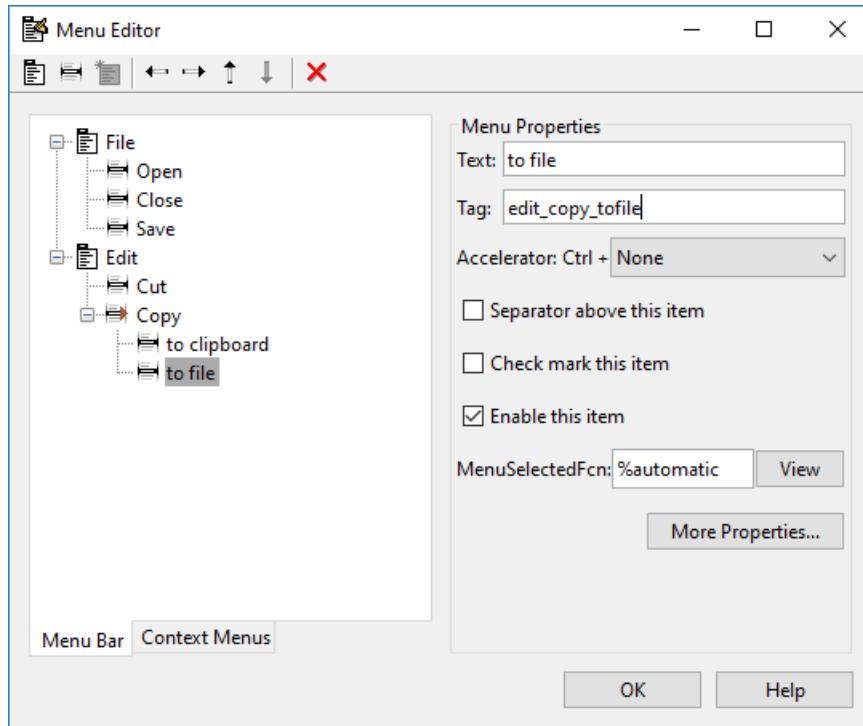
See “Menu Item” on page 19-21 and “How to Update a Menu Item Check” on page 19-23 for programming information and basic examples.

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.

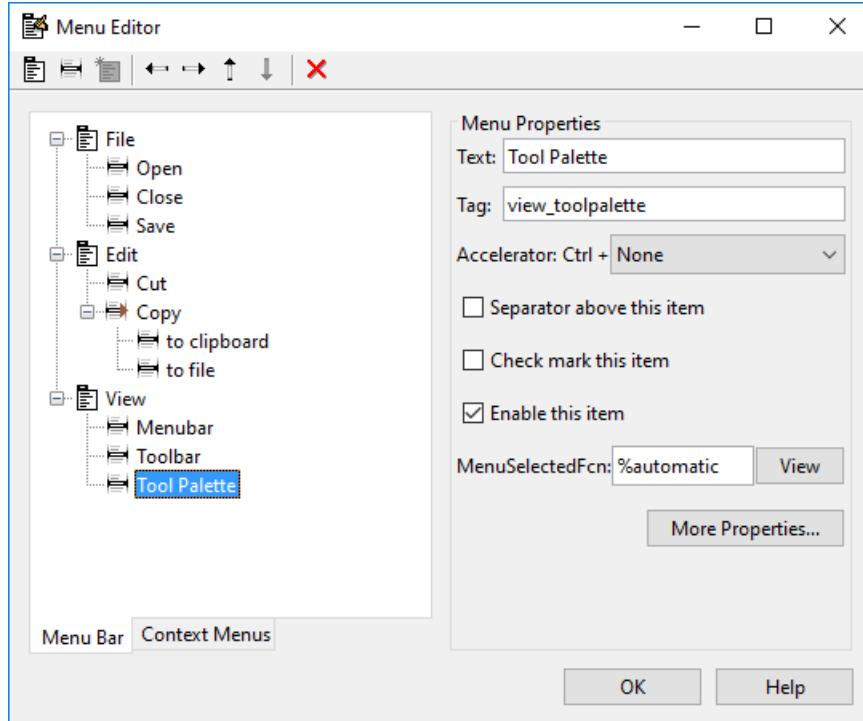
Cascading Menus

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.

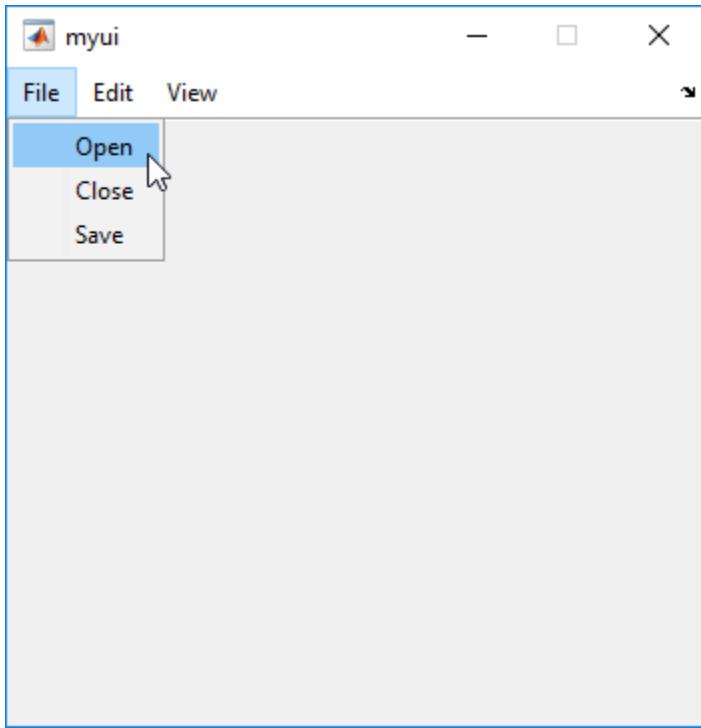


See “Menu Item” on page 19-21 for information about programming menu items.

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:

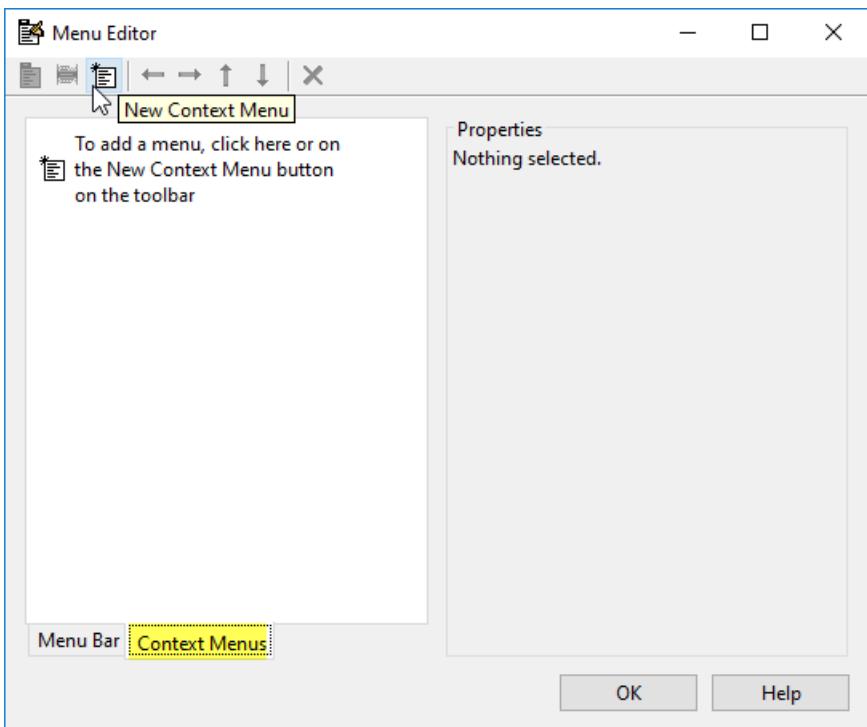
- 1 “Create the Parent Menu” on page 18-47
- 2 “Add Items to the Context Menu” on page 18-48
- 3 “Associate the Context Menu with an Object” on page 18-51

See “Menus for the Menu Bar” on page 18-40 for information about defining menus in general. See “Menu Item” on page 19-21 for information about defining local callback functions for your menus.

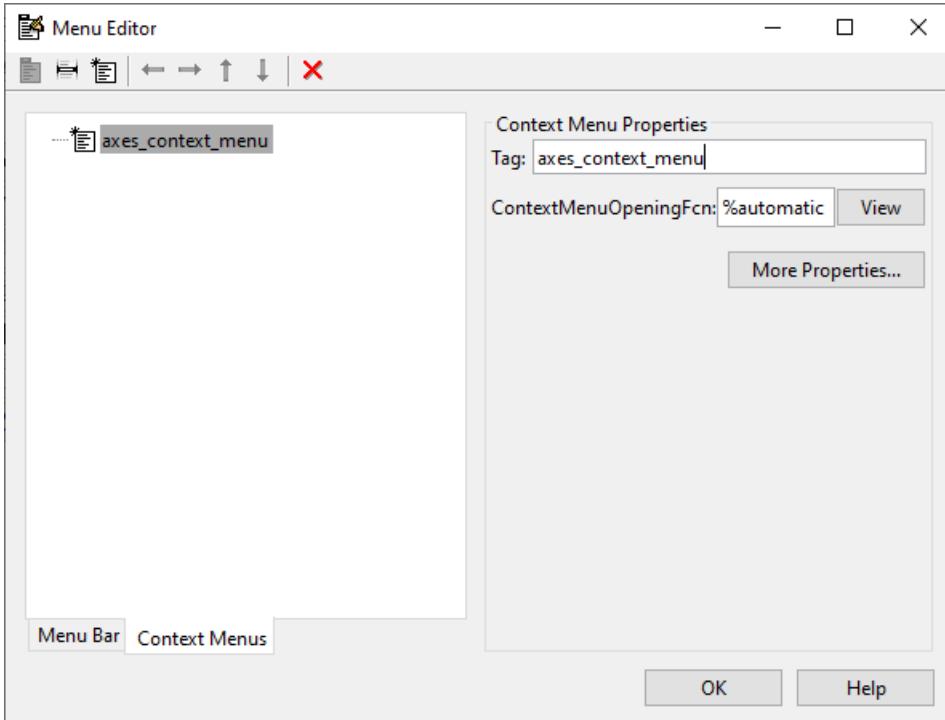
Create the Parent Menu

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.



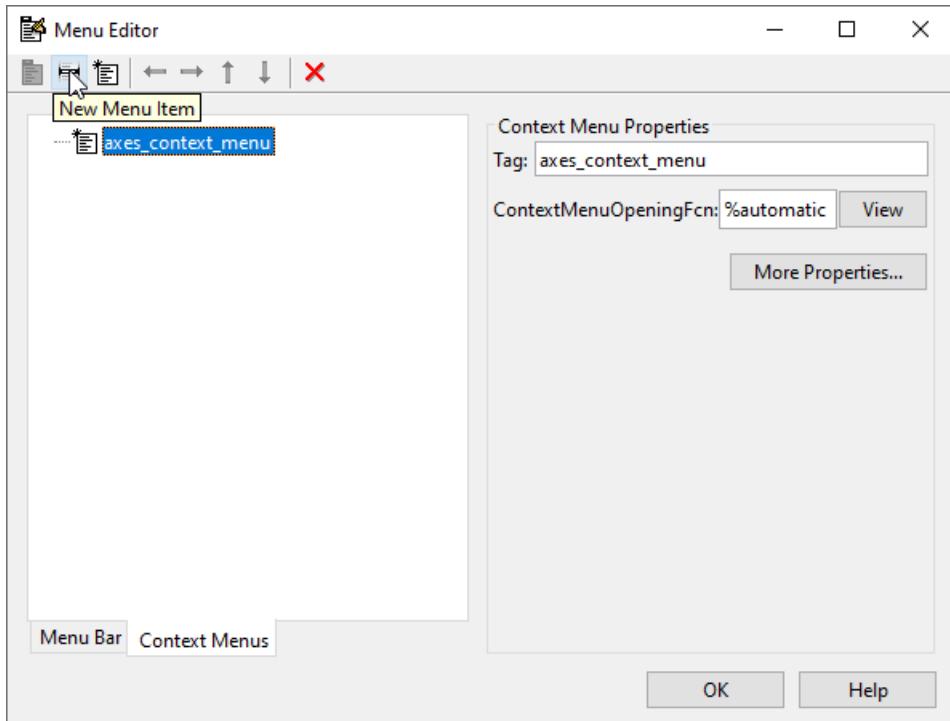
- 2 Select the menu, and in the **Tag** field type the context menu tag (axes_context_menu in this example).



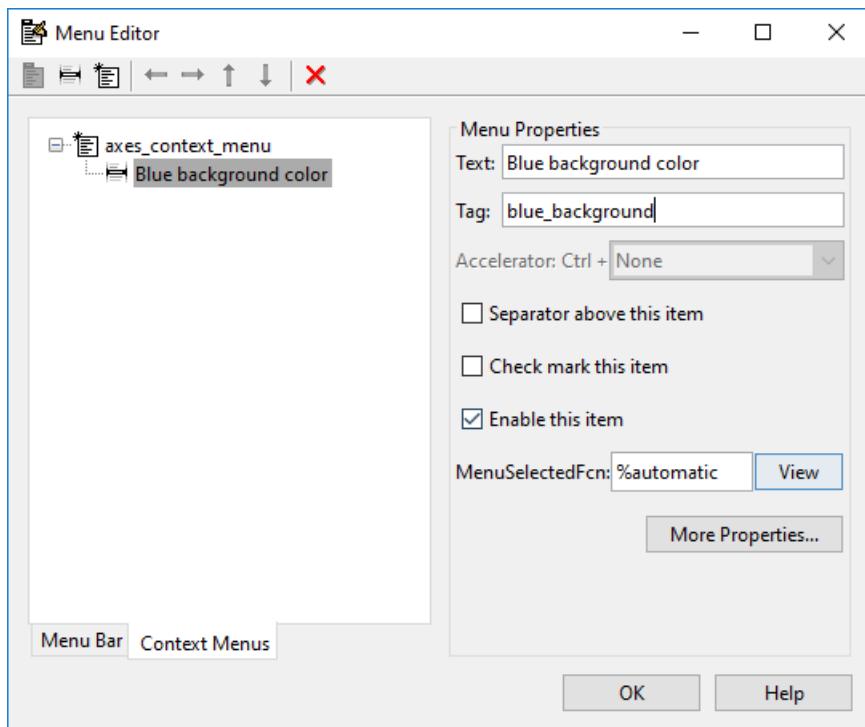
Add Items to the Context Menu

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 18-48. See "How to Update a Menu Item Check" on page 19-23 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 19-21 for more information about specifying this field and for programming menu items.

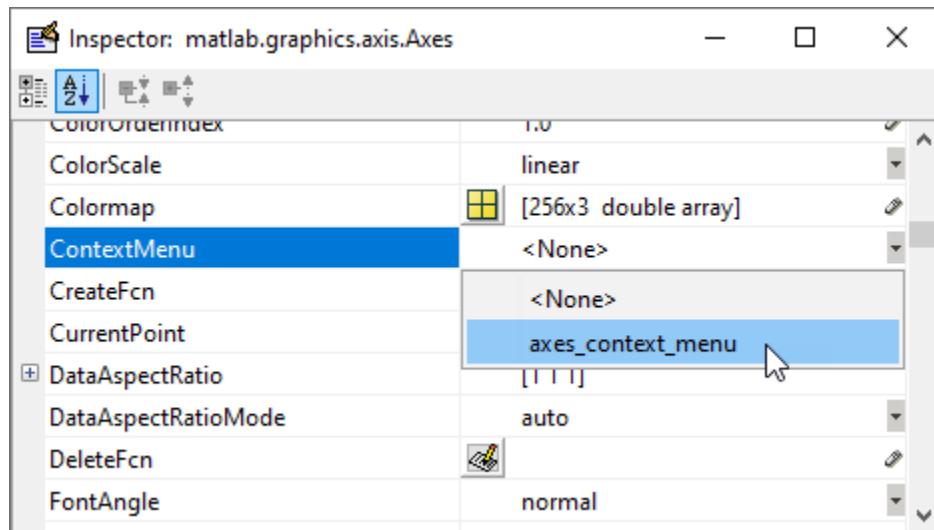
The **View** button displays the callback, if there is one, in an editor. If you have not yet saved the UI, GUIDE prompts you to save it.

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

Associate the Context Menu with an Object

- 1 In the Layout Editor, select the object for which you are defining the context menu.
- 2 Use the Property Inspector to set this object's **ContextMenu** property to the name of the desired context menu.

The following figure shows the **ContextMenu** 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 19-21 for information on defining the syntax.

See “How to Update a Menu Item Check” on page 19-23 for programming information and basic examples.

See Also

Related Examples

- “Write Callbacks in GUIDE” on page 19-2
- “Callbacks for Specific Components” on page 19-14
- “App Building Components” on page 5-2

Programming a GUIDE App

- “Write Callbacks in GUIDE” on page 19-2
- “Callbacks for Specific Components” on page 19-14
- “Examples of GUIDE Apps” on page 19-27

Write Callbacks in GUIDE

In this section...

- “Callbacks for Different User Actions” on page 19-2
- “GUIDE-Generated Callback Functions and Property Values” on page 19-4
- “GUIDE Callback Syntax” on page 19-4
- “Share Data Among GUIDE Callbacks” on page 19-5
- “GUIDE Example: Share Slider Data Using guidata” on page 19-10
- “GUIDE Example: Share Data Between Two Apps” on page 19-10
- “GUIDE Example: Share Data Among Three Apps” on page 19-11
- “Renaming and Removing GUIDE-Generated Callbacks” on page 19-13

Note The GUIDE environment will be removed in a future release. After GUIDE is removed, existing GUIDE apps will continue to run in MATLAB but they will not be editable in GUIDE.

To continue editing an existing GUIDE app, see “GUIDE Migration Strategies” on page 4-5 for information on how to help maintain compatibility of the app with future MATLAB releases. To create new apps interactively, “Develop Apps Using App Designer” instead.

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
ButtonDownFcn	End user presses a mouse button while the pointer is on the component or figure.	axes, figure, uibuttongroup, uicontrol, uipanel, uitable,
Callback	End user triggers the component. For example: selecting a menu item, moving a slider, or pressing a push button.	uicontextmenu, uicontrol, uimenu
CellEditCallback	End user edits a value in a table whose cells are editable.	uitable
CellSelectionCallback	End user selects cells in a table.	uitable

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, uitable
DeleteFcn	Callback executes just before MATLAB deletes the figure.	axes, figure, uibuttongroup, uicontextmenu, uicontrol, uimenu, uipushtool, uipanel, uitable, uitoggletool, uitable
KeyPressFcn	End user presses a keyboard key while the pointer is on the object.	figure, uicontrol, uipanel, uipushtool, uitable, uitable
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
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 19-4.

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
WindowKeyPressFcn WindowKeyReleaseFcn WindowScrollWheel	figure
KeyPressFcn	figure, uicontrol, uitable
KeyReleaseFcn	figure, uicontrol, uitable
SelectionChangedFcn	uibuttongroup
CellEditCallback CellSelectionCallback	uitable

Share Data Among GUIDE Callbacks

To create controls, menus, and graphics objects in your app that are interdependent, you must explicitly share data with the parts of your app that need to access the component.

Method	Description	Requirements and Trade-Offs
Share UserData	<p>Get or set property values directly through the component object.</p> <p>All UI components have a <code>UserData</code> property that can store any MATLAB data.</p>	<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.
Share Application Data	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.
Use <code>guidata</code>	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.

Share UserData in GUIDE Apps

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.

To set up a GUIDE app for sharing slider data with the `UserData` property, perform these steps:

- 1 In the Command Window, type `guide` to open a new blank GUI.
- 2 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**.

- 3 Select the push button tool from the component palette at the left side of the Layout Editor and drag it into the layout area.
- 4 Select the slider tool from the component palette at the left side of the Layout Editor and drag it into the layout area.
- 5 Select **File > Save**. Save the UI as `myslider.fig`. MATLAB opens the code file in the Editor.
- 6 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.

- 7 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.

- 8 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.

- 9 Save your code by pressing **Save** in the Editor Toolbar.

Share Application Data in GUIDE Apps

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 one variable, you can use `setappdata` to store multiple variables.

To set up a GUIDE app for sharing application data, perform these steps:

- 1 In the Command Window, type `guide` to open a new blank GUI.
- 2 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**.
- 3 Select the push button tool from the component palette at the left side of the Layout Editor and drag it into the layout area.
- 4 Select the slider tool from the component palette at the left side of the Layout Editor and drag it into the layout area.
- 5 Select **File > Save**. Save the UI as `myslider.fig`. MATLAB opens the code file in the Editor.
- 6 Set the initial value of the application data in the opening function, `myslider_OpeningFcn`. This function executes just before the UI is visible to users. In `myslider_OpeningFcn`, insert these commands immediately after the command, `handles.output = hObject`.

```
setappdata(handles.figure1, 'slidervalue', 0);
setappdata(handles.figure1, 'difference', 1);
```

After you add the commands, `myslider_OpeningFcn` looks like this.

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

% Choose default command line output for junk
handles.output = hObject;
setappdata(handles.figure1, 'slidervalue', 0);
setappdata(handles.figure1, 'difference', 1);

% Update handles structure
guidata(hObject, handles);

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

Notice that `handles` is an input argument to `myslider_OpeningFcn`. The `handles` variable is a structure that contains all the components in the UI. Each field in this structure corresponds to a separate component. Each field name matches the `Tag` property of the corresponding component. In this case, `handles.figure1` is the figure object. Thus, `setappdata` can use this figure object to store the data.

- 7 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`.

- 8 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`.

- 9 Save your code by pressing **Save** in the Editor Toolstrip.

Use guidata to Store and Share Data 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.

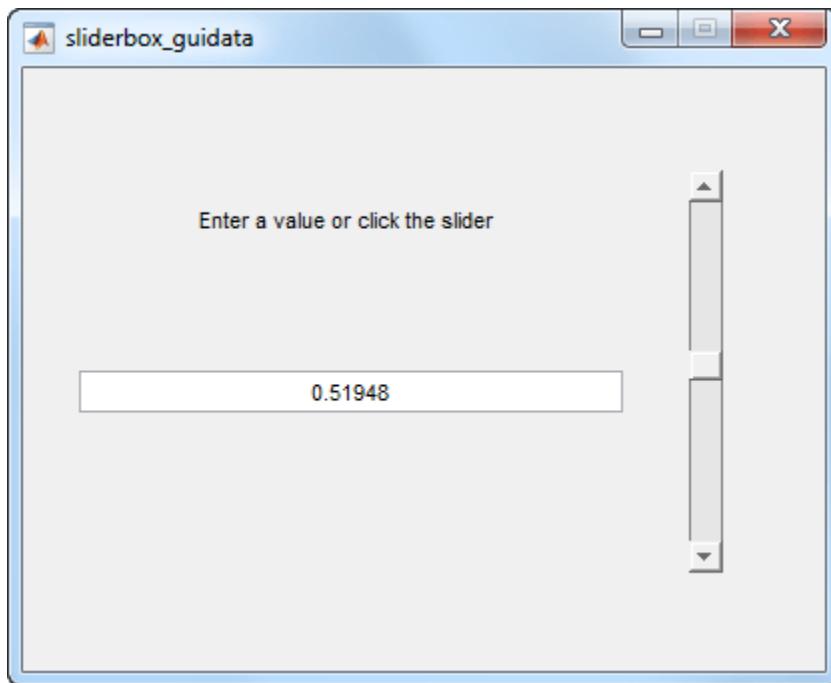
To modify your data in a callback function, modify the `handles` structure, and then store it using the `guidata` function. This slider callback function shows how to modify and store the `handles` structure in a GUIDE callback function.

```
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
```

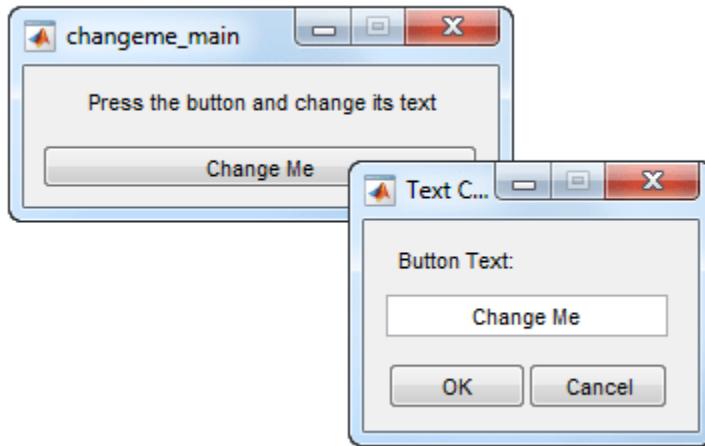
GUIDE Example: Share Slider Data Using guidata

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:

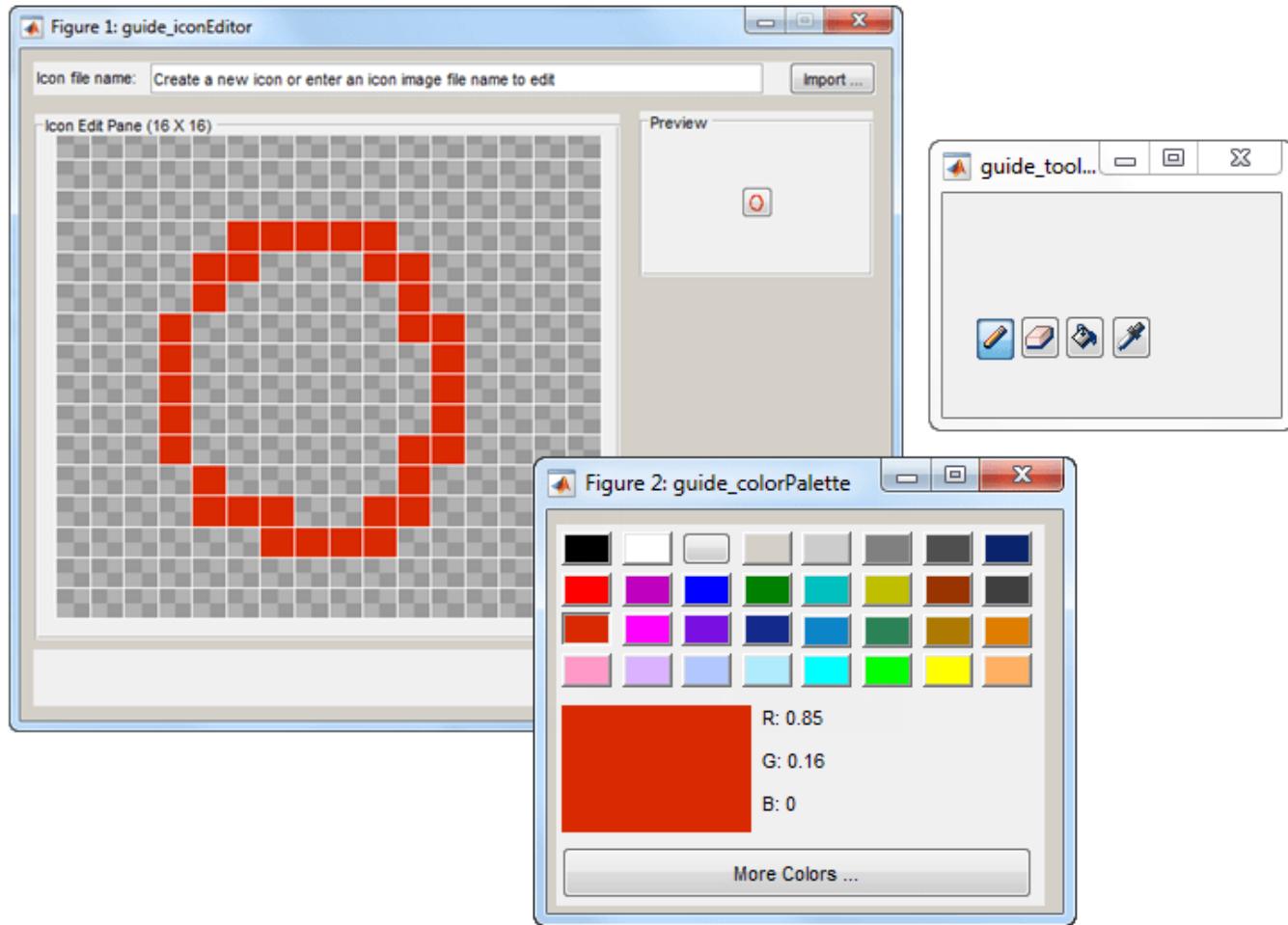
```
changeme_dialog('changeme_main', handles.figure)
```

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.

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 19-14
- “Anonymous Functions”
- “Share Data Among Callbacks” on page 12-2
- “Write Callbacks in App Designer” on page 7-15

Callbacks for Specific Components

Note The GUIDE environment will be removed in a future release. After GUIDE is removed, existing GUIDE apps will continue to run in MATLAB but they will not be editable in GUIDE.

To continue editing an existing GUIDE app, see “GUIDE Migration Strategies” on page 4-5 for information on how to help maintain compatibility of the app with future MATLAB releases. To create new apps interactively, “Develop Apps Using App Designer” instead.

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. For general information about coding callbacks, see “Write Callbacks in GUIDE” on page 19-2 or “Write Callbacks for Apps Created Programmatically” on page 11-2.

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.

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 19-20 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.

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 20-6 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 CreateFns

% 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.

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**.

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
```

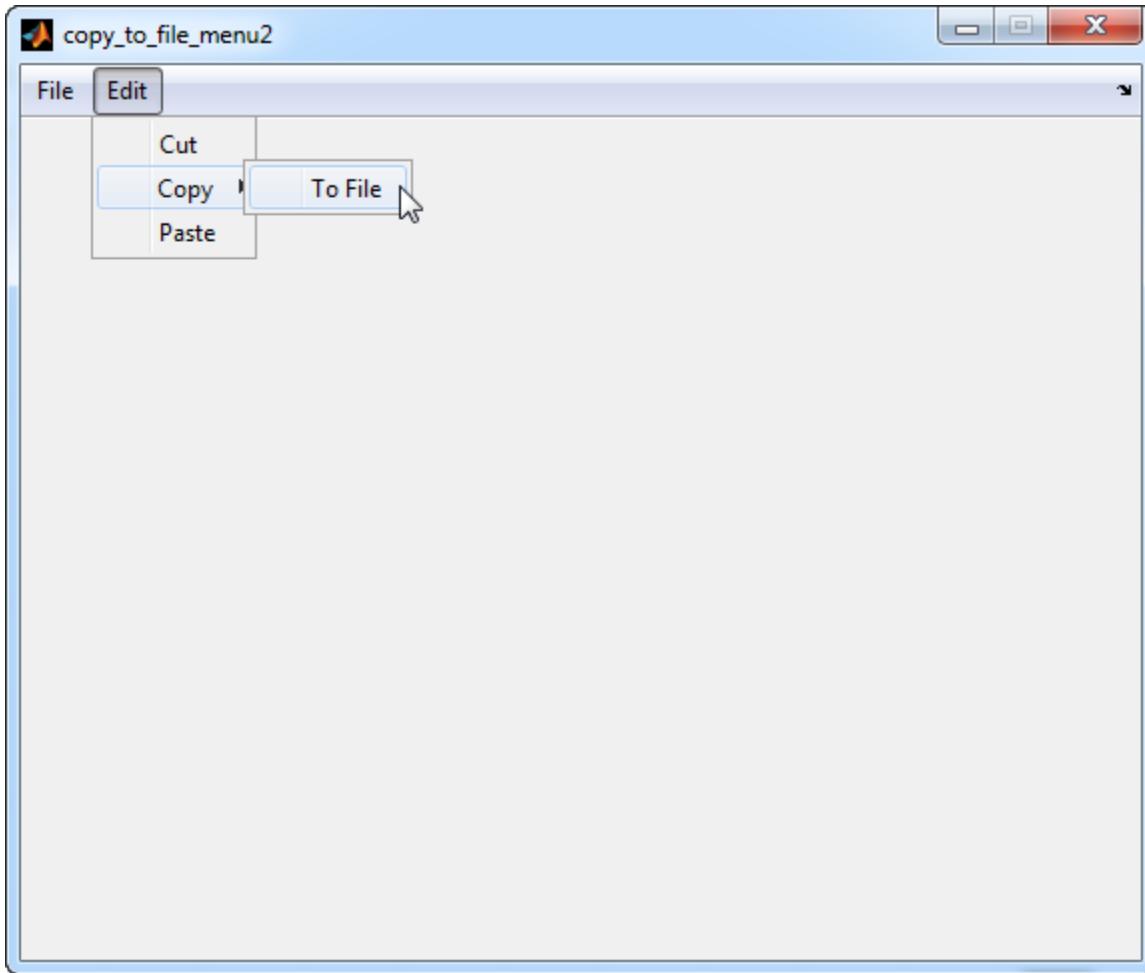
```
% EventName: string 'SelectionChanged' (read only)
% OldValue: handle of the previously selected object or empty
% NewValue: handle of the currently selected object
% handles structure with handles and user data (see GUIDATA)
switch get(eventdata.NewValue,'Tag') % Get Tag of selected object.
    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 18-40 for more information about creating menu items in GUIDE.

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.

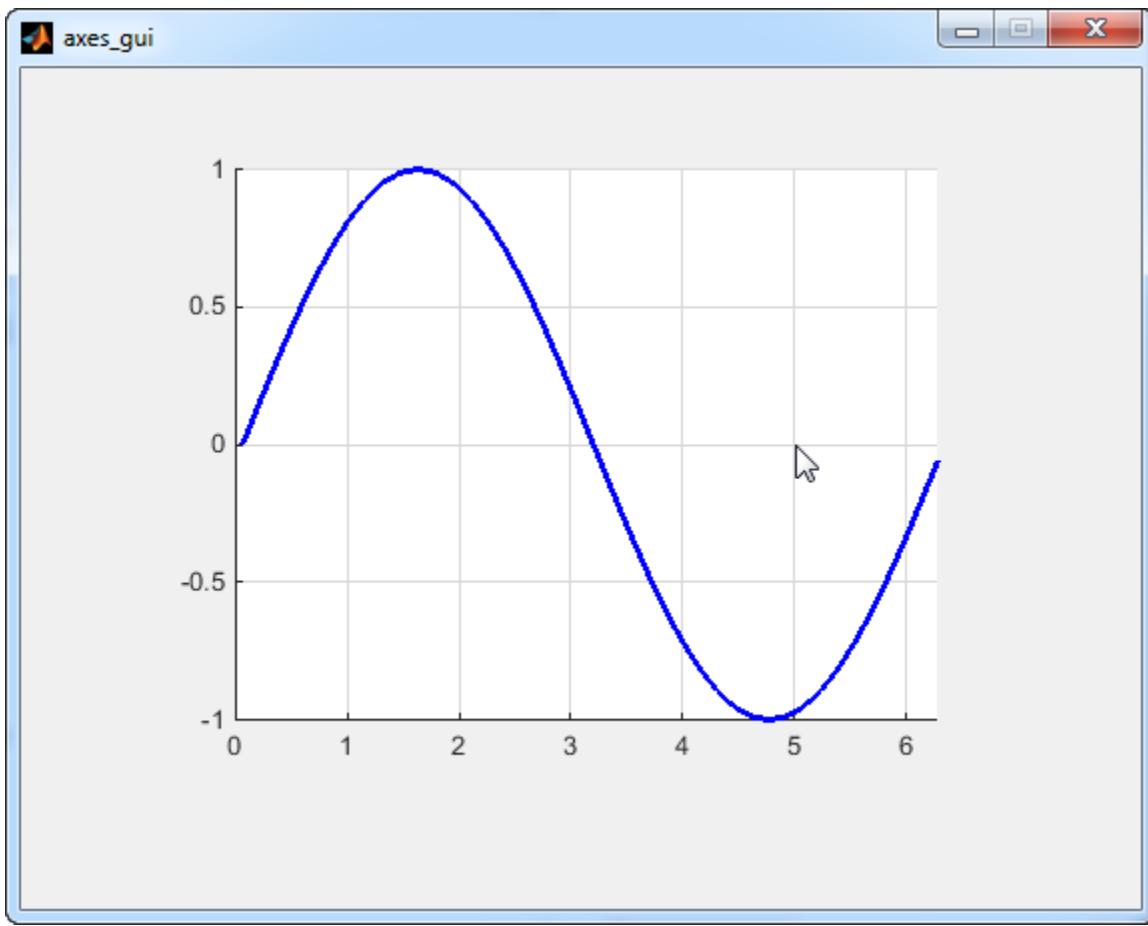
This code is an example of the table callback function, `CellEditCallback`. Associate this function with the table `CellEditCallback` property to make it execute when the end user edits a cell in the table.

```
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.

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 19-2
- “Write Callbacks for Apps Created Programmatically” on page 11-2

- “Write Callbacks in App Designer” on page 7-15

Examples of GUIDE Apps

Note The GUIDE environment will be removed in a future release. After GUIDE is removed, existing GUIDE apps will continue to run in MATLAB but they will not be editable in GUIDE.

To continue editing an existing GUIDE app, see “GUIDE Migration Strategies” on page 4-5 for information on how to help maintain compatibility of the app with future MATLAB releases. To create new apps interactively, “Develop Apps Using App Designer” instead.

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.

- “GUIDE App With Parameters for Displaying Plots” on page 20-2
- “Interactive List Box App in GUIDE” on page 20-6
- “Automatically Refresh Plot in a GUIDE App” on page 20-9

Examples of GUIDE UIs

- “GUIDE App With Parameters for Displaying Plots” on page 20-2
- “Interactive List Box App in GUIDE” on page 20-6
- “Automatically Refresh Plot in a GUIDE App” on page 20-9

GUIDE App With Parameters for Displaying Plots

Note The GUIDE environment will be removed in a future release. After GUIDE is removed, existing GUIDE apps will continue to run in MATLAB but they will not be editable in GUIDE.

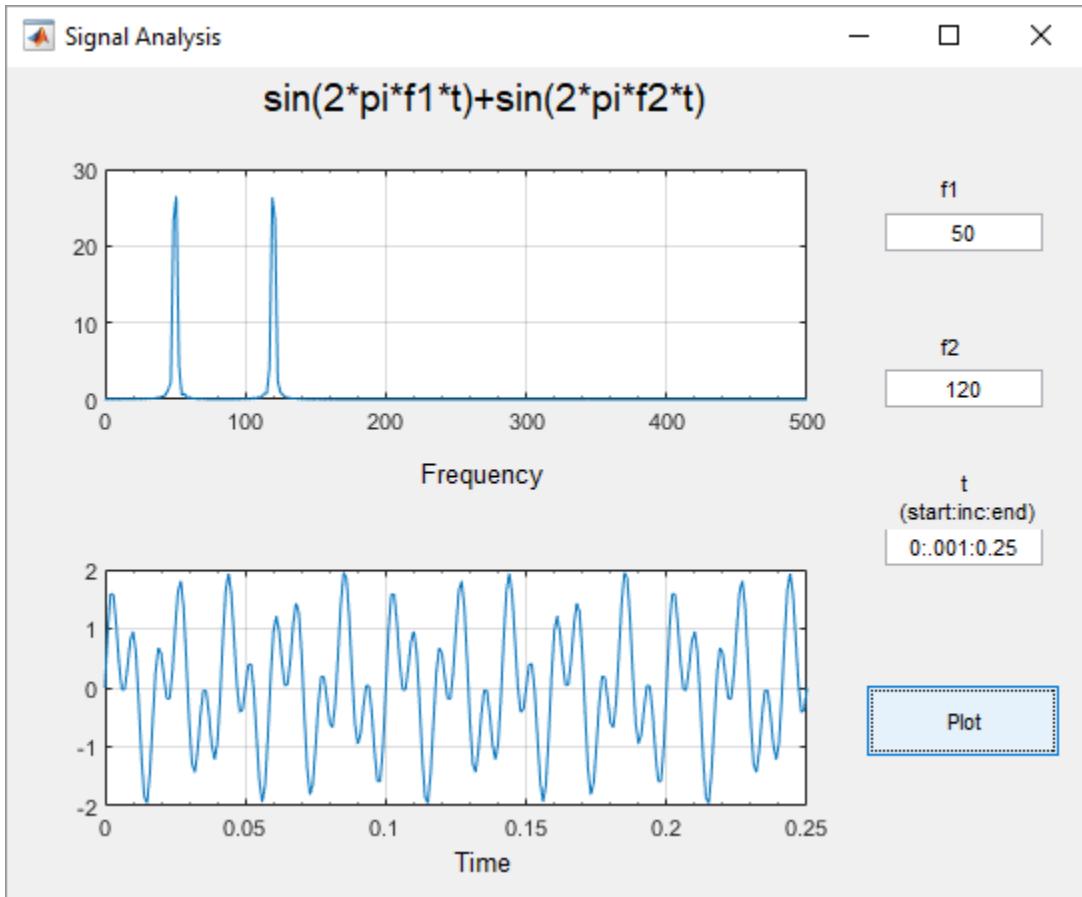
To continue editing an existing GUIDE app, see “GUIDE Migration Strategies” on page 4-5 for information on how to help maintain compatibility of the app with future MATLAB releases. To create new apps interactively, “Develop Apps Using App Designer” instead.

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');
end

```

```
    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(handles.frequency_axes,'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 3-2
- “Write Callbacks in GUIDE” on page 19-2
- “Share Data Among Callbacks” on page 12-2

Interactive List Box App in GUIDE

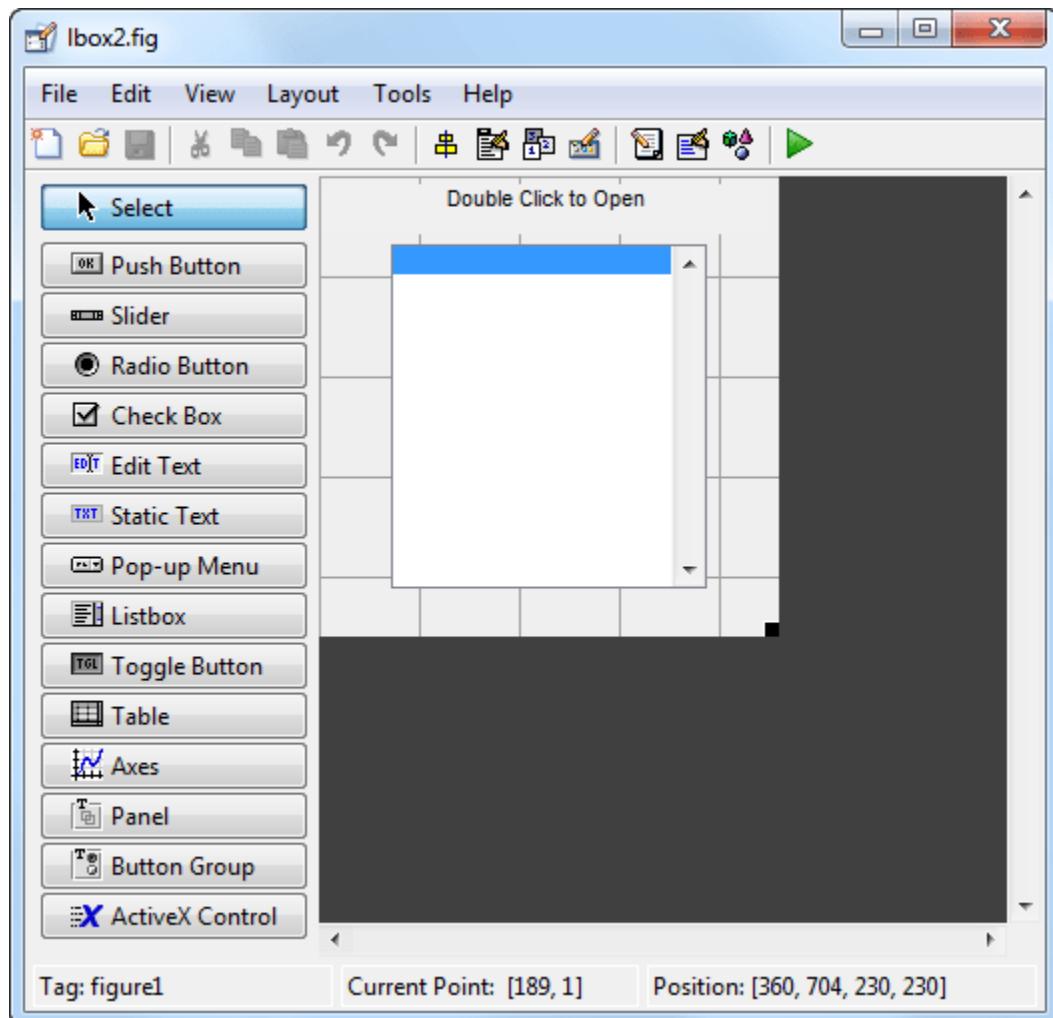
Note The GUIDE environment will be removed in a future release. After GUIDE is removed, existing GUIDE apps will continue to run in MATLAB but they will not be editable in GUIDE.

To continue editing an existing GUIDE app, see “GUIDE Migration Strategies” on page 4-5 for information on how to help maintain compatibility of the app with future MATLAB releases. To create new apps interactively, “Develop Apps Using App Designer” instead.

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.

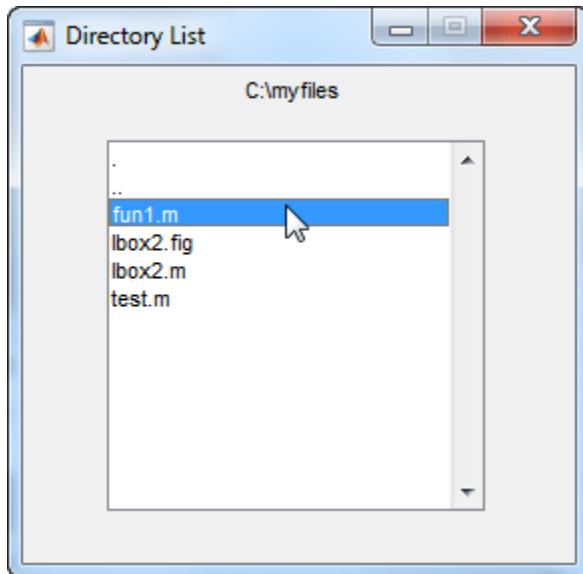
Open and Run The Example

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.

Ibox2_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  
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  
end
```

See Also

Related Examples

- “Create a Simple App Using GUIDE” on page 3-2
- “Write Callbacks in GUIDE” on page 19-2
- “Share Data Among Callbacks” on page 12-2

Automatically Refresh Plot in a GUIDE App

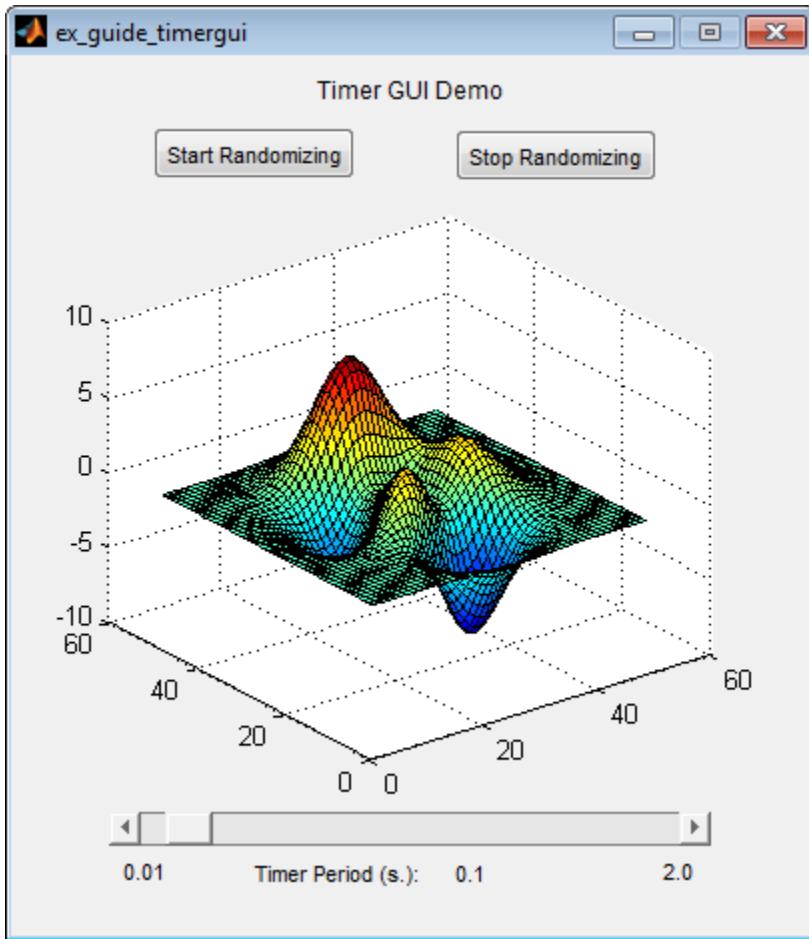
Note The GUIDE environment will be removed in a future release. After GUIDE is removed, existing GUIDE apps will continue to run in MATLAB but they will not be editable in GUIDE.

To continue editing an existing GUIDE app, see “GUIDE Migration Strategies” on page 4-5 for information on how to help maintain compatibility of the app with future MATLAB releases. To create new apps interactively, “Develop Apps Using App Designer” instead.

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 19-2

App Packaging

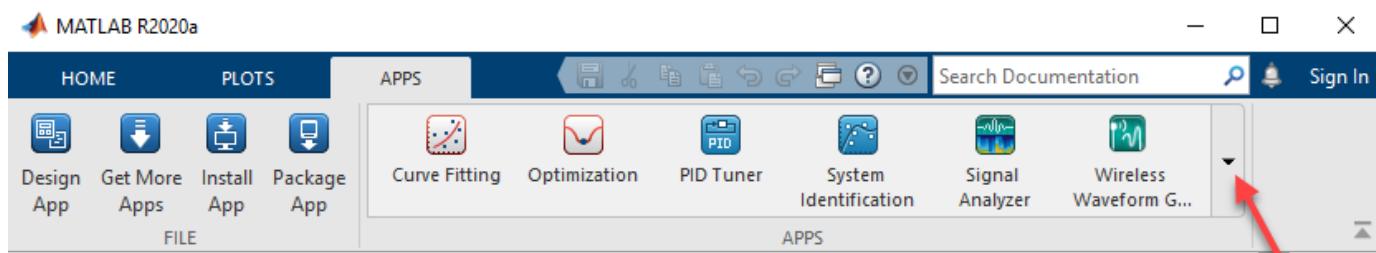
Packaging GUIs as Apps

- “Get and Create Apps” on page 21-2
- “Package Apps From the MATLAB Toolstrip” on page 21-5
- “Package Apps in App Designer” on page 21-7
- “Modify Apps” on page 21-9
- “Ways to Share Apps” on page 21-10
- “MATLAB App Installer File — `mlappinstall`” on page 21-14
- “App Packaging Dependency Analysis” on page 21-15

Get and Create Apps

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 design your own apps using the App Designer development environment. 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

App Designer is the recommended environment for building apps in MATLAB. 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 21-5
- “Modify Apps” on page 21-9
- “Ways to Share Apps” on page 21-10

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 “App Packaging Dependency Analysis” on page 21-15.

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 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 from File”.

See Also

Related Examples

- “Modify Apps” on page 21-9
- “Ways to Share Apps” on page 21-10
- “MATLAB App Installer File — `mlappinstall`” on page 21-14
- “App Packaging Dependency Analysis” on page 21-15

Package Apps in App Designer

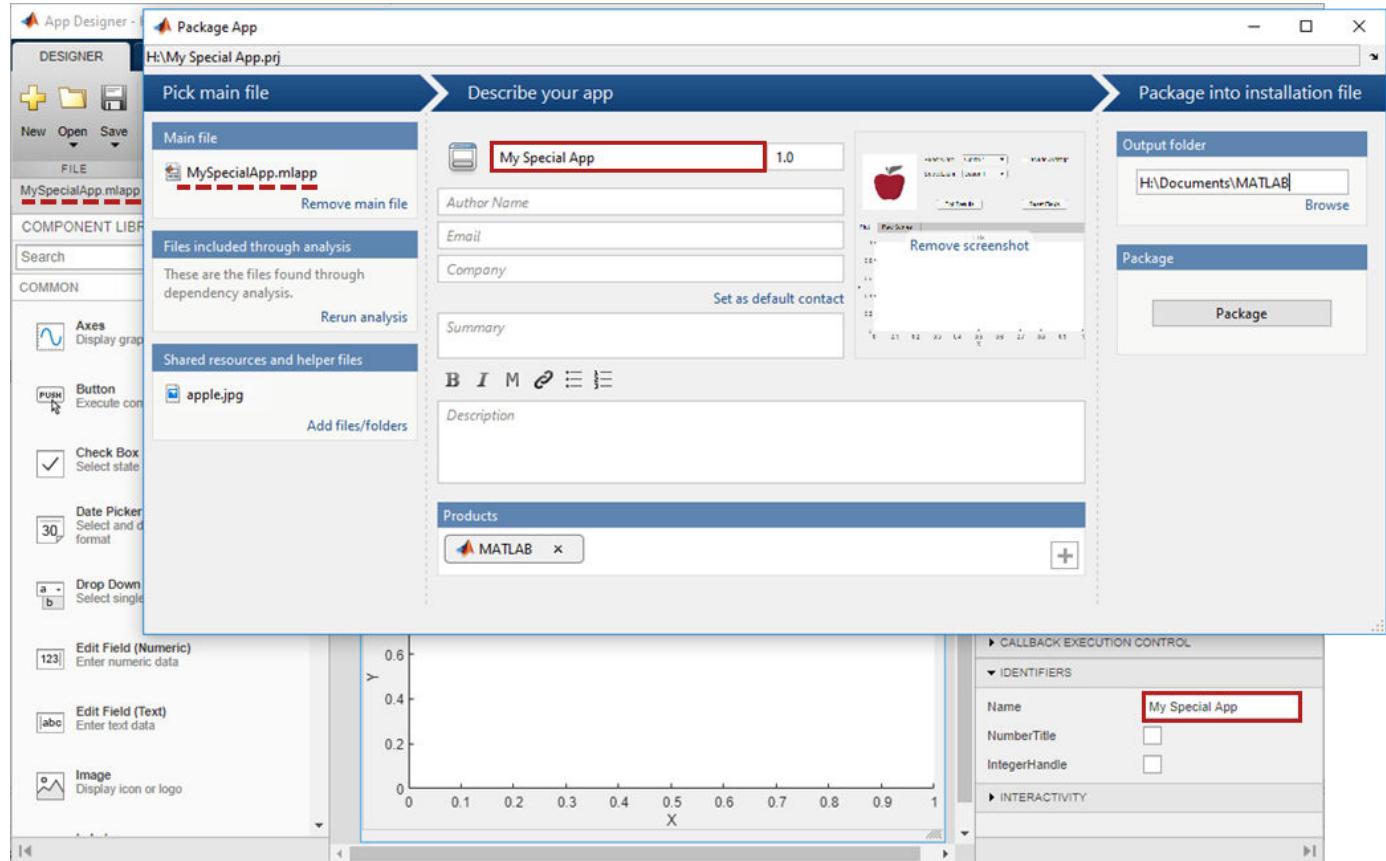
After creating an app in App Designer, you can package it into a single installer file that you can easily share with others. The underlying functionality for packaging apps in App Designer is the same as the functionality that underlies the **Add-Ons > Package App** option in the MATLAB Toolstrip.

- 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 name 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 location where the installation file will be saved.
 - 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 21-5
- “Ways to Share Apps” on page 21-10
- “MATLAB App Installer File — `mlappinstall`” on page 21-14
- “App Packaging Dependency Analysis” on page 21-15

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 21-10
- “MATLAB App Installer File — `mlappinstall`” on page 21-14
- “App Packaging Dependency Analysis” on page 21-15

Ways to Share Apps

There are several ways to share your apps.

- “Share MATLAB Files Directly” on page 21-10 — 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 21-12 — 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 21-13 — 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 21-13 — 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/matlab-runtime.html>.

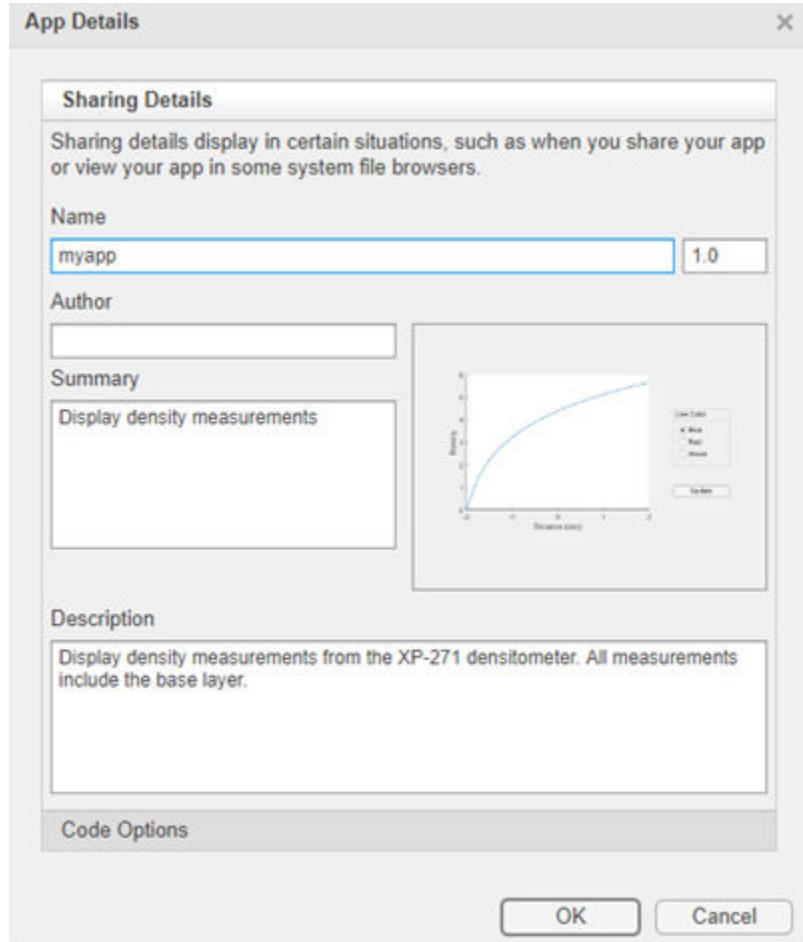
Share MATLAB Files Directly

If you created your app in GUIDE, share the `.fig` file, the `.m` file, and all other dependent files with your users.

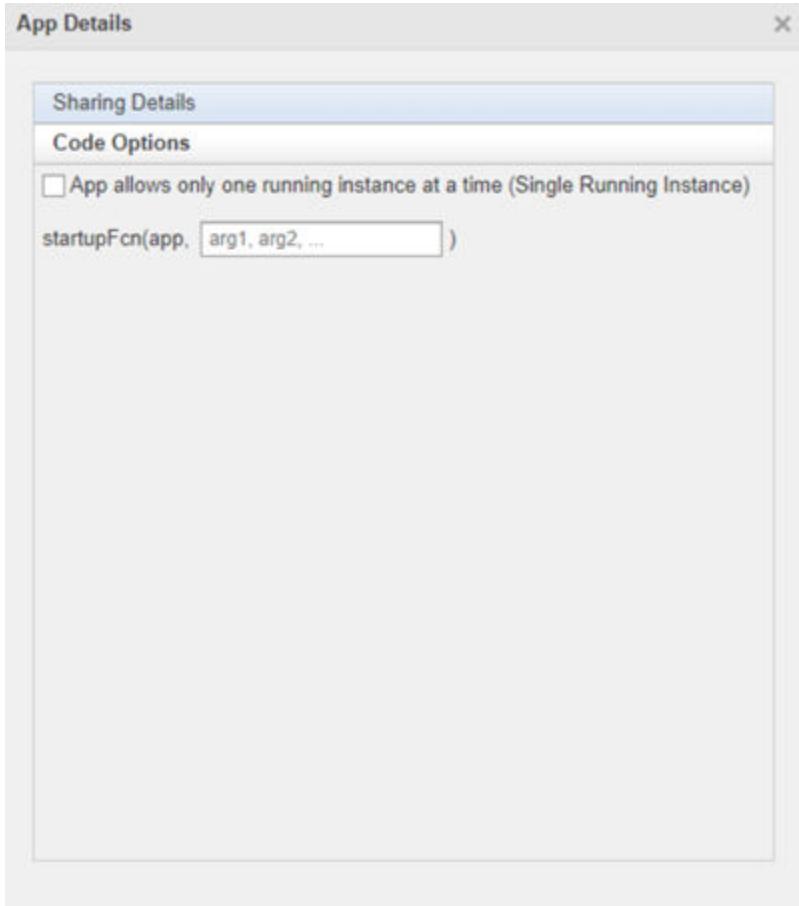
If you created your app programmatically, share all `.m` files and other dependent files with your users.

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, version, author, summary, and description by clicking **App Details**  in the **Designer** tab of the App Designer toolstrip. 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.



To specify input arguments and whether your app can run multiple instances at a time or only a single instance, expand the **Code Options** section and select from the available options.



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 21-7 or “Package Apps From the MATLAB Toolstrip” on page 21-5. 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. (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. In addition, certain functionality is not supported in deployed web apps. For more information, see “Limitations and Unsupported Functionality” (MATLAB Compiler).

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 Toolbar, 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

- “Get and Create Apps” on page 21-2
- “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 21-5

App Packaging 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 is not guaranteed 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`, `xlsinfo`, `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`

