GUI with Multiple Axes

On this page…

About the Multiple Axes Example
View and Run the Multiple Axes GUI
Designing the GUI
Plot Push Button Callback
Validating User Input as Numbers

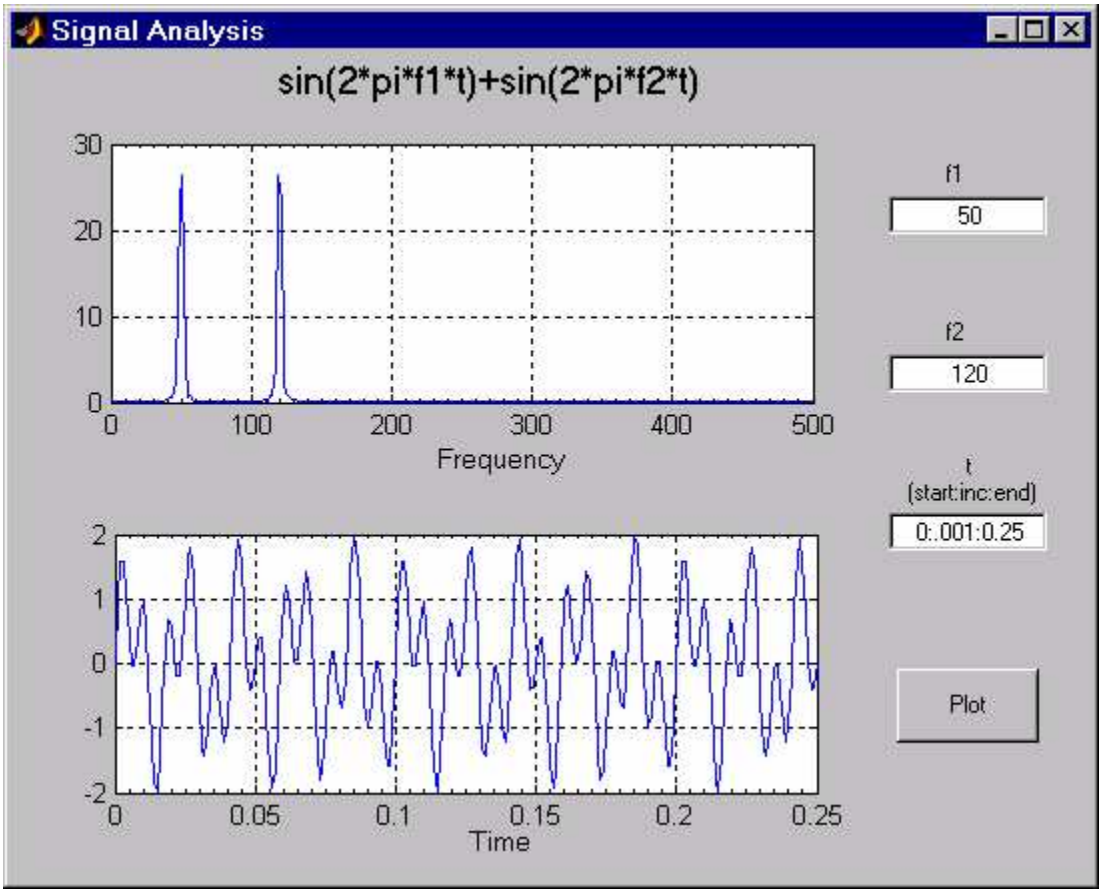
About the Multiple Axes Example

This example creates a GUI that plots data that it derives from three parameters entered by the user. The parameters define a time- and frequency-varying signal. One of the GUI's two axes displays the data in the time domain and the other displays it in the frequency domain.

GUI-building techniques illustrated in this example include:

- Controlling which axes object is the target for plotting commands.
- Using edit text controls to read numeric input and MATLAB expressions.
- Converting user inputs from strings to numbers and validating the result.
- Restoring focus to an edit text box when user input fails validation.

When you first open the Signal Analysis GUI, it looks as shown in the following figure. It evaluates the expression printed at the top of the figure using the parameters `f1`, `f2`, and `t` that the user enters. The upper line graph displays a Fourier transform of the computed signal displayed in the lower line graph.



Note You can create a more advanced GUI that also displays time and frequency plots by following the GUIDE

example GUI to Interactively Explore Data in a Table.

Back to Top

View and Run the Multiple Axes GUI

If you are reading this in the MATLAB Help browser, you can access the example FIG-file and code file by clicking the following links. If you are reading this on the Web or in PDF form, go to the corresponding section in the MATLAB Help Browser to use the links.

If you intend to modify the layout or code of this GUI example, first save copies of its code file and FIG-file to your current folder. (You need write access to your current folder to do this.) Click the following links to copy the example files to your current folder and open them.

1. Click here to copy the files to your current folder.
2. Type `guide two_axes` or click here to open the GUI in GUIDE.
3. Type `edit two_axes` or click here to open the GUI code file in the Editor.

You can view the properties of any component by double-clicking the component in the Layout Editor to open the Property Inspector for it. You can modify either the figure, the code, or both. Then you can save the GUI in your current folder using **File** > **Save as** from GUIDE. This saves both files, allowing you to rename them, if you choose.

> **Note** Only rename GUIDE GUIs from within GUIDE. Renaming GUIDE files from a folder window or the command line prevents them from operating properly until you restore their original names.

If you just want to run the GUI or inspect it in GUIDE, follow these steps:

1. Click here to add the example files to the MATLAB path (only for the current session).
2. Click here to run the `two_axes` GUI
3. Click here to display the GUI in the GUIDE Layout Editor (read only)..
4. Click here to display the GUI code file in the MATLAB Editor (read only).

> **Note** Do not save GUI files to the `examples` folder where you found them or you will overwrite the original files. If you want to save GUI files, use **File** > **Save as** from GUIDE, which saves both the GUI FIG-file and the GUI code file.

Back to Top

Designing the GUI

This GUI plots two graphs that depict three input values:

- Frequency one (`f1`)
- Frequency two (`f2`)
- A time vector (`t`)

When the user clicks the **Plot** button, the GUI puts these values into a MATLAB expression that is the sum of two sine functions:

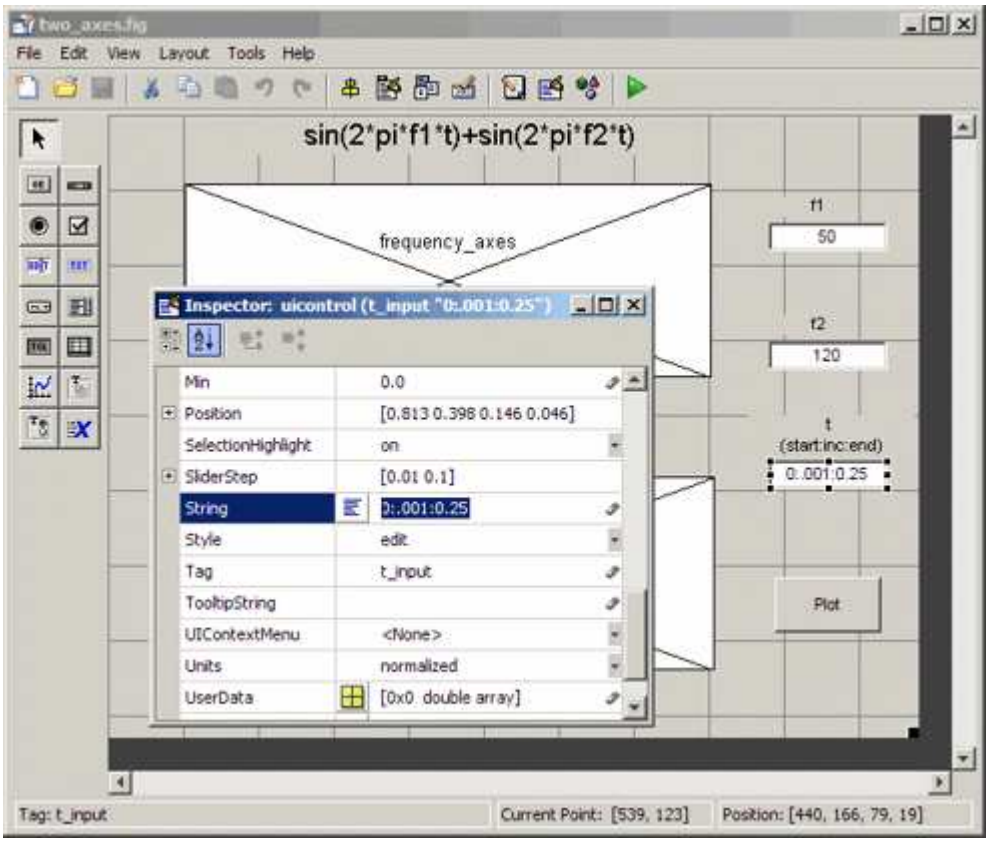
```
x = sin(2*pi*f1*t) + sin(2*pi*f2*t)
```

The GUI then calculates the FFT (fast Fourier transform) of `x` and plots the data in the frequency domain and the time domain in separate axes.

Specifying Default Values for the Inputs

The GUI provides default values for the three inputs. This enables users to click the **Plot** button and see a result as soon they run the GUI. The defaults also indicate typical values that the user might enter.

To create the default values, set the `String` property of the edit text. The following figure shows the value set for the time vector.

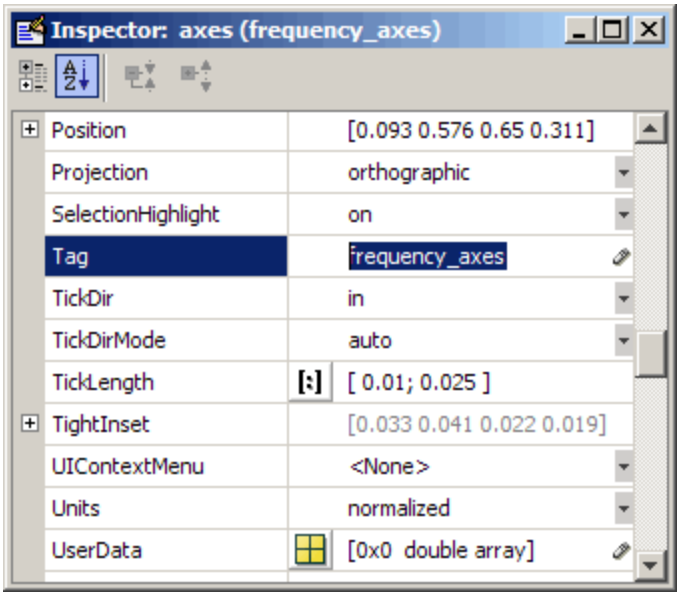


Identifying the Axes

Since there are two axes in this GUI, you must specify which one you want to target when plotting data. Use the `handles` structure, which contains the handles of all components in the GUI, to identify that axes. The `handles` structure is a variable that GUIDE passes as an argument to all component callbacks:

```
component_callback(hObject, eventdata, handles)
```

The structure contains handles for all GUI components. You access the handles using field names that GUIDE derives from the components' `Tag` property. To make code more readable (and to make it easier to remember) this example sets the `Tag` property to descriptive names. The following graphic shows how to set the upper axes `Tag` to `'frequency_axes'` in the Property Inspector.



Altering the `Tag` causes GUIDE to set the field name for the frequency plot axes to `frequency_axes` in the `handles` structure. Within the `plot_button_Callback`, you access that axes' handle with `handles.frequency_axes`. You use the handle as the first argument to `plot` to ensure that the graph is displayed in the correct axes, as follows:

```
plot(handles.frequency_axes,f,m(1:257))
```

Likewise, the `Tag` of the time axes is set to `time_axes`, and the call to `plot` uses it as follows:

```
plot(handles.time_axes,t,x)
```

For more information, see handles Structure. For the details of how to use the handle to specify the target axes, see Plot Push Button Callback.

GUI Option Settings

GUIDE has a set of preferences called GUI Options, available from the **Tools** menu. Two GUI Options settings are particularly important for this GUI:

- Resize behavior: **Proportional**
- Command-line accessibility: **Callback**

Proportional Resize Behavior. Selecting **Proportional** as the resize behavior enables users to resize the GUI to better view the plots. Using this option setting, when you resize the GUI, everything expands or shrinks proportionately except text.

Callback Accessibility of Object Handles. When GUIs include axes, their handles should be visible from other objects' callbacks. This enables you to use plotting commands like you would on the command line. **Callback** is the default setting for command-line accessibility.

For more information, see GUI Options.

Back to Top

Plot Push Button Callback

This GUI uses only the **Plot** button callback. You do not need to code callbacks for the edit text components unless you want to validate their inputs. When a user clicks the **Plot** button, the callback performs three basic tasks: it gets user input from the edit text components, calculates data, and creates the two plots.

Getting User Input

The three edit text boxes are where the user enters values for the two frequencies and the time vector. The first task for the callback is to read these values. This involves:

- Reading the current values in the three edit text boxes using the `handles` structure to access the edit text handles.
- Converting the two frequency values (`f1` and `f2`) from strings to doubles using `str2double`.
- Evaluating the time string using `eval` to produce a vector `t`, which the callback used to evaluate the mathematical expression.

The following code shows how the callback obtains the input:

```
% Get user input from GUI
f1 = str2double(get(handles.f1_input,'String'));
f2 = str2double(get(handles.f2_input,'String'));
t = eval(get(handles.t_input,'String'));
```

The **Plot** button callback avoid generating errors due to receiving improper inputs. To make sure that the inputs `f1`, `f2`, and `t` can be used in computations, the edit text callbacks test the values as soon as the user enters them. To see how this is done, see Validating User Input as Numbers .

Calculating Data

After constructing the string input parameters to numeric form and assigning them to local variables, the next step is to calculate data for the two graphs. The `plot_button_Callback` computes the time domain data using an expression of sines:

```
x = sin(2*pi*f1*t) + sin(2*pi*f2*t);
```

The callback computes the frequency domain data as the Fourier transform of the time domain data:

```
y = fft(x,512);
```

For an explanation of this computation, see the `fft` function.

Plotting the Data

The final task for the `plot_button_Callback` is to generate two plots. This involves:

- Targeting plots to the appropriate axes. For example, this code directs a graph to the time axes:

  ```
  plot(handles.time_axes,t,x)
  ```

- Providing the appropriate data to the `plot` function
- Turning on the axes grid, which the `plot` function automatically turns off

Performing the last step is necessary because many plotting functions (including `plot`) clear the axes and reset properties before creating the graph. This meansthat you cannot use the Property Inspector to set the `XMinorTick`, `YMinorTick`, and grid properties in this example, because they are reset when the callback executes `plot`.

When looking at the following code listing, note how the `handles` structure provides access to the handle of the axes, when needed.

Plot Button Code Listing

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

% Get user input from GUI
f1 = str2double(get(handles.f1_input,'String'));
f2 = str2double(get(handles.f2_input,'String'));
t = eval(get(handles.t_input,'String'));

% Calculate data
x = sin(2*pi*f1*t) + sin(2*pi*f2*t);
y = fft(x,512);
m = y.*conj(y)/512;
f = 1000*(0:256)/512;

% Create frequency plot in proper axes
plot(handles.frequency_axes,f,m(1:257))
set(handles.frequency_axes,'XMinorTick','on')
grid on

% Create time plot in proper axes
plot(handles.time_axes,t,x)
set(handles.time_axes,'XMinorTick','on')
grid on
```

Back to Top

Validating User Input as Numbers

GUI users type parameters into three edit text boxes as strings of text. If they type an inappropriate number or something that is not a number, the graphs can fail to inform or even to generate. Preventing bad inputs from being processed is an important function of almost any GUI that performs computations. In this GUI, it is important that:

- All three inputs are positive or negative real numbers.
- The `t` (time) input is a vector that increases monotonically and is not too long to legibly display.

You can make a GUI respond in a variety of ways to inappropriate inputs. These include:

- Clearing an invalid input, forcing the user to enter another one.

- Replacing an invalid input with its last previous valid value or its default value.
- Disabling controls that initiate processing of the input.
- Displaying an error alert or playing a sound.

You can combine these actions, as appropriate. In this example, each of the edit text control callbacks validates its own input. If the input fails validation, the callback disables the **Plot** button, changes its `String` to indicate the type of problem encountered, and restores focus to the edit text control, highlighting the erroneous input. As soon as the user re-enters a value that is acceptable, the **Plot** button is enabled with its `String` set back to `'Plot'`. This approach prevents plotting errors and avoids the need for an error dialog.

Validating the `f1` and `f2` inputs is not difficult. These inputs must be real scalar numbers that can be positive or negative. The `str2double` function handles most cases, returning `NaN` (Not a Number) for nonnumeric or nonscalar string expressions. An additional test using the `isreal` function makes sure that the user has not entered a complex number, such as `'4+2i'`. The `f1_input_Callback` contains the following code to validate user input for `f1` :

```
f1 = str2double(get(hObject,'String'));
if isnan(f1) || ~isreal(f1)
    % isdouble returns NaN for non-numbers and f1 cannot be complex
    % Disable the Plot button and change its string to say why
    set(handles.plot_button,'String','Cannot plot f1')
    set(handles.plot_button,'Enable','off')
    % Give the edit text box focus so user can correct the error
    uicontrol(hObject)
else
    % Enable the Plot button with its original name
    set(handles.plot_button,'String','Plot')
    set(handles.plot_button,'Enable','on')
end
```

Similar code validates the `f2` input.

The time vector input, `t`, is more complicated to validate. As the `str2double` function does not operate on vectors, the `eval` function is called to convert the input string into a MATLAB expression. Because a user can type many things that `eval` cannot handle, the first task is to make sure that `eval` succeeded. The `t_input_Callback` uses `try` and `catch` blocks to do the following:

- Call `eval` with the `t_input` string inside the `try` block.
- If `eval` succeeds, perform additional tests within the `try` block.
- If `eval` generates an error, pass control to the `catch` block.
- In that block, the callback disables the **Plot** button and changes its `String` to `'Cannot plot t'`.

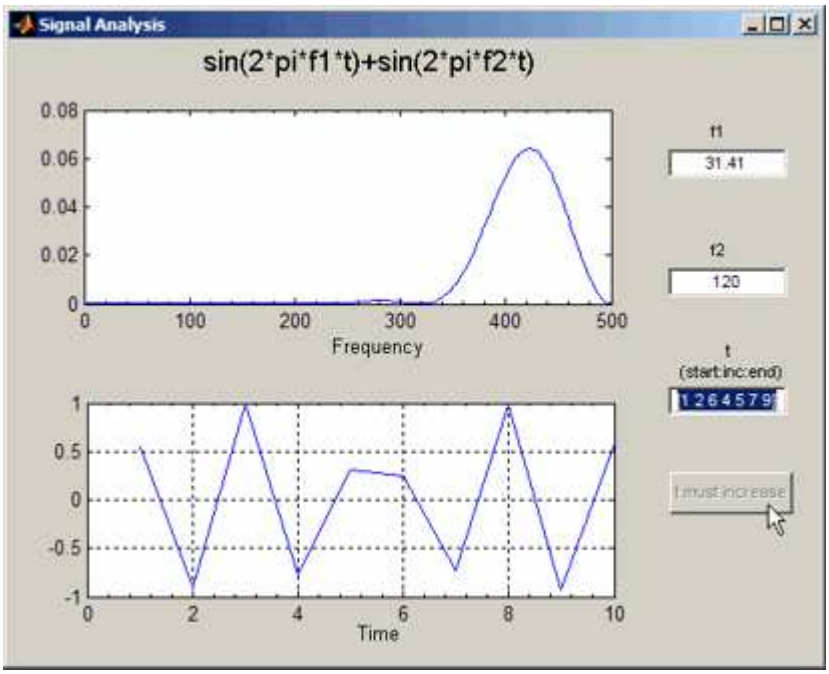
The remaining code in the `try` block makes sure that the variable `t` returned from `eval` is a monotonically increasing vector of numbers with no more than 1000 elements. If `t` passes all these tests, the callback enables **Plot** button and sets its `String` to `'Plot'`. If it fails any of the tests, the callback disables the **Plot** button and changes its `String` to an appropriate short message. Here are the `try` and `catch` blocks from the callback:

```
% Disable the Plot button ... until proven innocent
set(handles.plot_button,'Enable','off')
try
    t = eval(get(handles.t_input,'String'));
    if ~isnumeric(t)
        % t is not a number
        set(handles.plot_button,'String','t is not numeric')
    elseif length(t) < 2
        % t is not a vector
        set(handles.plot_button,'String','t must be vector')
    elseif length(t) > 1000
        % t is too long a vector to plot clearly
        set(handles.plot_button,'String','t is too long')
    elseif min(diff(t)) < 0
        % t is not monotonically increasing
        set(handles.plot_button,'String','t must increase')
    else
        % All OK; Enable the Plot button with its original name
        set(handles.plot_button,'String','Plot')
```

```
            set(handles.plot_button,'Enable','on')
            return
        end
        % Found an input error other than a bad expression
        % Give the edit text box focus so user can correct the error
        uicontrol(hObject)
    catch EM
        % Cannot evaluate expression user typed
        set(handles.plot_button,'String','Cannot plot t')
        % Give the edit text box focus so user can correct the error
        uicontrol(hObject)
end
```

The edit text callbacks execute when the user enters text in an edit box and presses **Return** or clicks elsewhere in the GUI. Even if the user immediately clicks the **Plot** button, the edit text callback executes before the **plot** button callback activates. When a callback receives invalid input, it disables the **Plot** button, preventing its callback from running. Finally, it restores focus to itself, selecting the text that did not validate so that the user can re-enter a value.

As an example, here is the GUI's response to input of a time vector, `[1 2 6 4 5 7 9]`, that does not monotonically increase.



In this figure, the two plots reflect the last successful set of inputs, `f1 = 31.41`, `f2 = 120`, and `t = [1 2 3 4 5 7 9]`. The time vector `[1 2 6 4 5 7 9]` appears highlighted so that the user can enter a new value. The highlighting results from executing the command `uicontrol(hObject)` in the above listing.

Back to Top

Examples of GUIDE GUIs GUI for Animating a 3-D View