Graphical User Interfaces

A graphical user interface (GUI) is the link between a software package and the user. In general, it is composed of a set of commands or menus, objects and instruments such as buttons, by means of which the user establishes a communication with the program. The GUI eases the tasks of inputting data and displaying output data.

Creation of a GUI with the Tool GUIDE

MATLAB has a tool to develop GUIs in an easy and quick way. This tool is called **G**raphical **U**ser **I**nterface **D**evelopment **E**nvironment and is better known by its acronym `GUIDE`. This tool can create a GUI empty window, add buttons and menus to our GUI, and windows to enter data and plot functions, as well as the access to the objects callbacks. When we create a GUI with `GUIDE`, two files are created: a fig-file which is the graphical interface and an m-file which contains the functions, the description for the GUI parts, and the callback.

A **callback** is defined as the action that implements an object of the GUI when the user clicks on it or uses it. For example, when the user clicks on a button in a GUI, a program containing the instructions and tasks to be realized is executed. This program is called the callback. A callback is coded in the m-language.

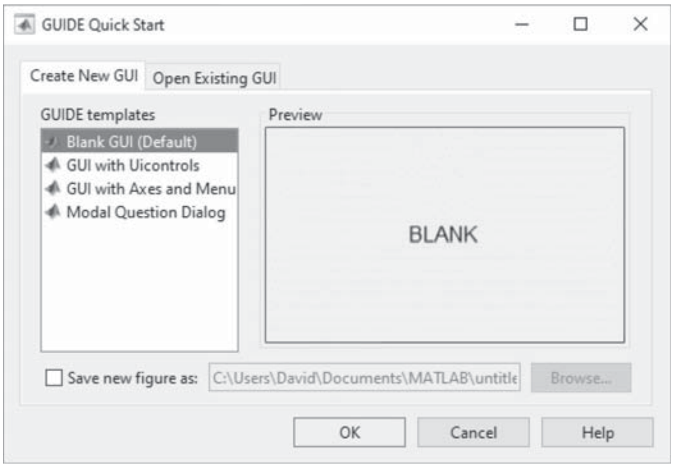

FIGURE 8.1: `GUIDE` Quick Start window.

Starting `GUIDE`

GUIDE can be started from the MATLAB menu with `New`→ `App`→ `GUIDE`. It can also be started by typing **`guide`** in the `Command Window`. Any of these choices opens the `GUIDE Quick Start` window shown in Figure 8.1. Here we can start a new GUI with an empty blank GUI where we can add and arrange the object for the GUI. We can also start a new GUI with `uicontrols`, with a set of axes and a menu already in the GUI, and finally, a GUI with question dialog buttons. Alternatively, we can open a GUI previously started in the tab `Open Existing GUI`. If we select `Blank GUI` we get the work window of Figure 8.2. In this work window we see at the left a set of buttons or objects that can be used in the GUI. (To see the object names in the buttons, in the `File` menu select `File` → `Preferences ...` → `GUIDE` and select the option `Show names in component palette`.) Each button has a function which is described by the button's name and it is self describing.

On the upper part we see the toolbar. It contains icons to create a new GUI or figure, to open an existing GUI, and to save the GUI. It also has icons to copy, paste, and cut parts of the GUI, as well as to undo and redo actions on the GUI objects. In addition, the toolbar has icons to align the objects in the GUI, another icon for the `Editor` and for the `Property Inspector`, to display the browser, and to execute the GUI. Some properties are further explained in Table 8.1.

Properties of Objects in a GUI

Each object that we place in the GUI has properties that can be edited with the `Property Inspector`. For example, for a `Push Button` Figure 8.3 shows the `Property Inspector` with some of the properties of this button. Some of the most common properties are shown in Table 8.2.

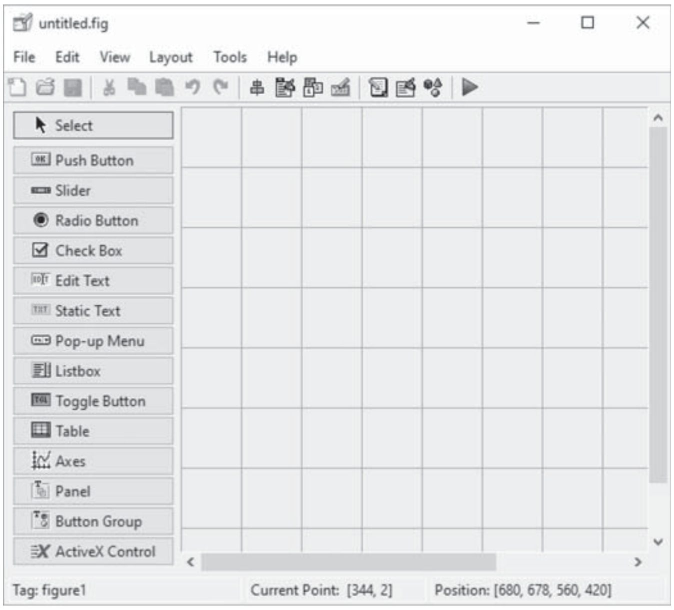

FIGURE 8.2: A blank GUI.

TABLE 8.1: Important icons in the `GUIDE` toolbar.

Property icon name	Description
Property inspector	It refers to the properties of each object in the GUI. They include color, name, tag, value, and the callback among others.
Align Objects	It aligns the objects in the work window.
Toolbar editor	It creates a toolbar in the GUI.
M-file editor	It opens the MATLAB editor to edit the callbacks.

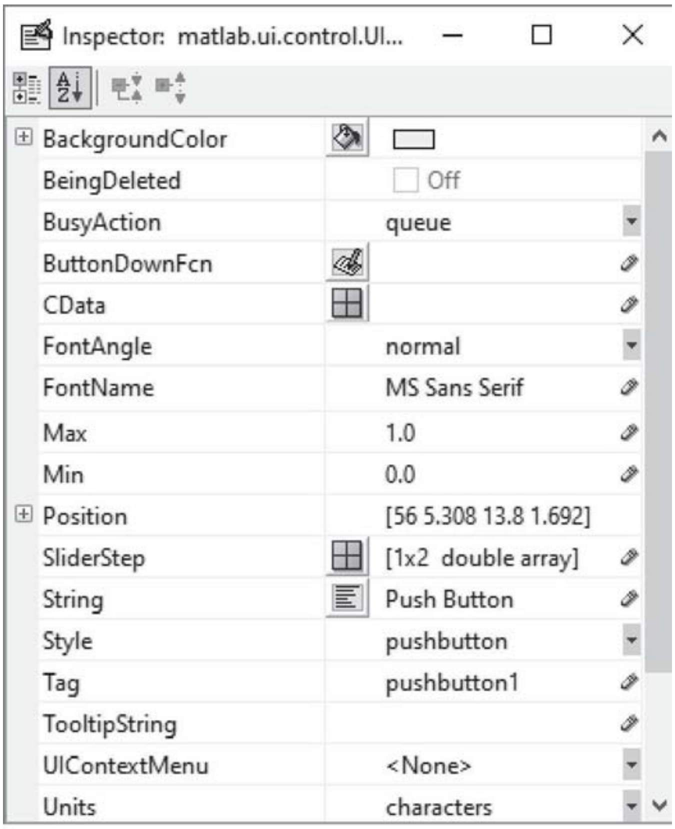

FIGURE 8.3: Property inspector.

A Simple GUI

We show with a plotting example the procedure to create a GUI. Let us suppose that we wish to create a GUI that plots a user defined function. Thus, we need a text box to enter the function, two text boxes to enter the initial and final points in the plot, a set of axes to plot the function, and a button to run the GUI. Additionally, we can add a button to close the GUI after we finish.

1. The first step is to open a blank GUI work window.
2. We add the required objects.
   - We start with two push buttons. A button to plot the function and another one to close the GUI.
   - Three `Edit Text` boxes, a text box to enter the function to be plotted and two text boxes for the x-axis limits.
   - A set of axes.
   - Five `Static Texts` for labels. The GUI is shown in Figure 8.4.

TABLE 8.2: Most used properties for objects in a GUI.

Property	Description
Background color	Changes the background color of the object.
Callback	Set of instructions to be executed by the object.
Enable	Activates the object.
String	Mostly used in the cases of buttons, edit text boxes, and static text boxes. It contains the text displayed in the object.
Tag	It identifies the object.

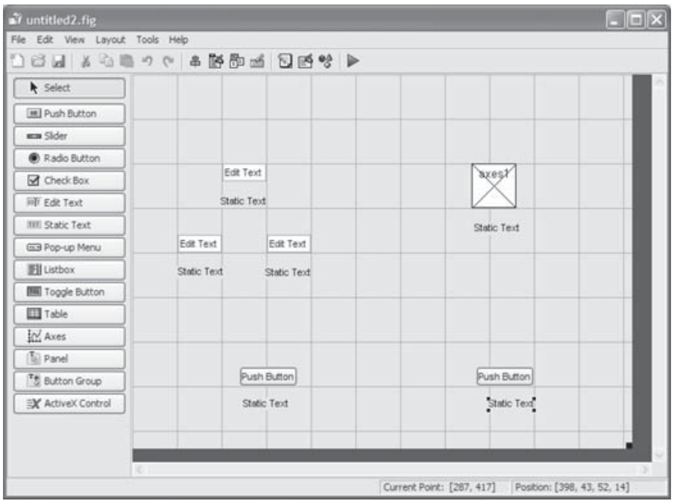

FIGURE 8.4: GUI with required objects.

3. We can stretch each of the objects to the desired final size. We also change the `String` property of each object as we can see in Figure 8.5. To change the `String` property, we double click on each one of the objects to open the `Property inspector` and change the `String` in the `Static texts`. The `Font weight` property is changed to bold. For each of the remaining elements we clear the `Strings`.

4. For the `Edit Text` box we change the `Tag` property to `The_function`. This is the variable name for the function to be plotted. For the remaining `Edit Text` boxes we change the tags to `Initial_x` and `Final_x`.

5. We change the `Tag` properties of the push buttons to `Plot_function` and `CloseGUI`.

6. We save the GUI as `plotter.fig`.

7. We now need to edit the callbacks.

8. First we edit the callback of the `Close` button. We only need to add the instruction:

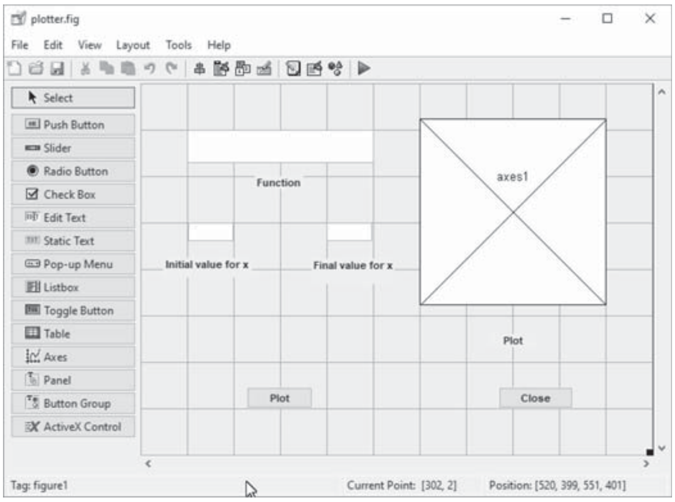

FIGURE 8.5: GUI with objects stretched to its final size and final labels.

```
close(gcbf)
```

which indicates to close the figure where the object is embedded. In this case the figure, that is the GUI, where the push button is located. To edit the callback we select this push button and click on the mouse right hand button to open the menu shown where we select `View callbacks` → `Callback`. This opens the m-file editor in the portion corresponding to this push button. The m-file is then as follows:

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

9. We now edit the callback for the button `Plot`. In this callback we have to read the function we wish to plot. We also read the lower and upper limits for the variable `x` which we declare as a symbolic variable with `syms x`. Finally, we plot the function in the set of axes in the GUI. When we read data from a GUI, this data is read as a string. Thus, at some point we have to put the data in the correct format, for example an integer variable, a boolean variable, and so on. First, we open the callback for the button `Plot` by right clicking on the push button and selecting `View callbacks` → `Callback`. This opens the m-file editor. To read the information in the strings and make it amenable for calculations we

use the instruction `eval` to convert the string value we have read to a numeric value. For example, for the lower limit of `x`

```
lower_x_value = eval(get(handles.Initial_x, 'string'));
```

The instruction `get` fetches the string variable which is located in the object with the handle `Initial_x` and the instruction `eval` converts it to a real variable. A similar instruction applies for the upper `x` limit and for the function to be plotted. That is,

```
lower_x_value = eval( get( handles.Initial_x, 'string'));
upper_x_value = str2num(get( handles.Final_x, 'string'));
y = eval(get( handles.The_function, 'string'));
```

We are using `eval` and `str2num` to convert from string to a number. They both accomplish the same task.

10. Now, we are ready to get the plot. We now get the `x`-axis points with

```
xx = [lower_x_value:0.2:upper_x_value];
```

11. The function to be plotted is now in the variable `y` which is a string. To be able to accept `x` as a symbolic variable, we add the instruction `syms x`. This makes the variable `y` a symbolic variable.

12. To evaluate the function `y` at the set of points `xx` we substitute the variable `x` with the vector `xx` with

```
yb = subs(y, x, xx);
```

13. Now, we plot the vector `yb` with

```
plot(xx, yb)
```

14. Finally, we add a grid with

```
grid on
```

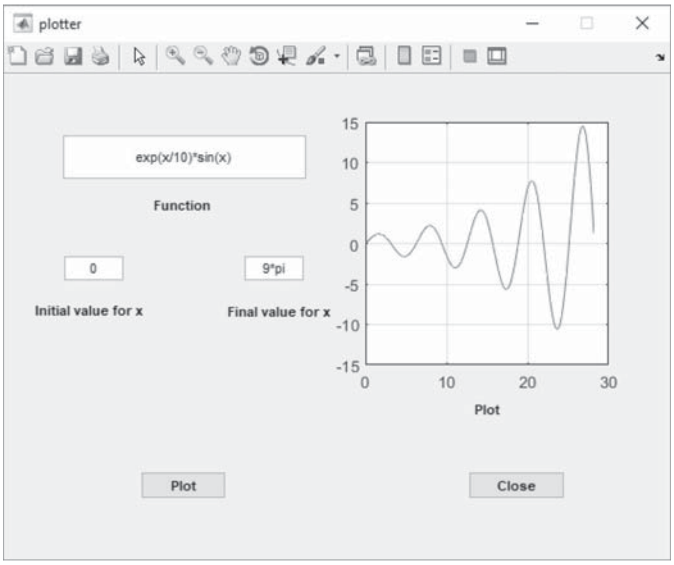

FIGURE 8.6: Plot of function $e^{(x/10)}\sin(x)$ from 0 to 9π.

15. The final callback for the push button `Plot` is

```
%  -- Executes on button press in Plot_function.
function Plot_function_Callback(hObject, eventdata, handles)
% hObject handle to Plot_function (see GCBO)
% eventdata reserved - to be defined in a future version of
MATLAB
% handles structure with handles and user data (see GUIDATA)
syms x
lower_x_value = eval( get( handles.Initial_x, 'string'));
upper_x_value = str2num( get( handles.Final_x, 'string'));
xx = [lower_x_value:0.2:upper_x_value];
y = eval(get( handles.The_function, 'string'));
yb = subs(y, x, xx);
plot(xx, yb)
grid on
```

16. To finish the GUI we add a toolbar. From the `Tools` menu select `Toolbar Editor` .... When it opens change the icons as desired. For this example we only click the `OK` button to display the default toolbar. Once we are finished we save the current GUI as `plotter`.

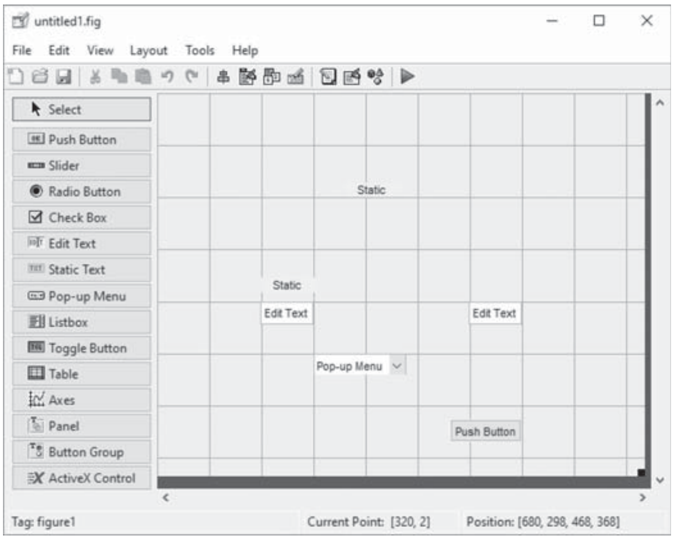

FIGURE 8.7: Initial layout for the GUI.

Note that the complete GUI is composed of the interface that is saved in `plotter.fig`, that is a figure file, and the m-file `plotter.m` that contains the callbacks for the objects in the GUI.

Now, to run the GUI either we type `plotter` in the MATLAB command window or click on the `Run` icon in the GUI work window. Figure 8.6 shows a plot for the function `exp(x/10)*sin(x)`.

Examples

This section presents two examples introducing some of the other objects for the GUIs. The first example shows the pulldown menu for a GUI that converts temperature given in Fahrenheit degrees to Celsius and vice versa. The second example is a GUI for the calculation of put and call options using the Black-Scholes function from the Financial Derivatives toolbox.

Example Temperature conversion

Temperature conversion among the different scales, Celsius, Fahrenheit, and Kelvin, is possible if we know the conversion equations. They are available in any physics textbook and they are:

```
F = 1.8*C + 32
K = C + 273.15
C = (F - 32)*5/9
K = (F - 32)*5/9 + 273.15
C = K - 273.15
F = 1.8*(K - 273.15) + 32
```

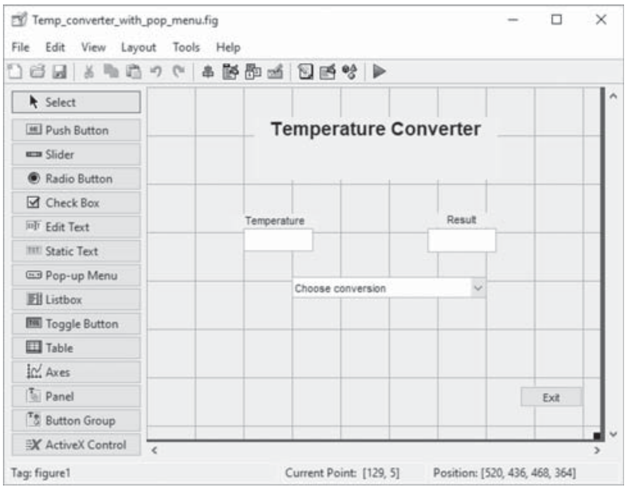

FIGURE 8.8: GUI layout with strings and sizes changed.

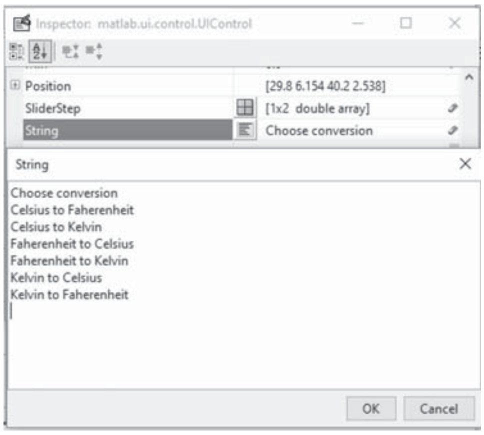

FIGURE 8.9: String for the pop-up menu. Click on the icon to the right of `String`.

Now we create a GUI that implements these conversion equations. The initial GUI layout is shown in Figure 8.7. We now implement the instructions for each object. We use the regular instructions to close the GUI and to read data. To choose the conversion we use a `Pop-up Menu` and for the input temperature data we use an `Edit Text` box. We have changed the strings and size of the objects for each of the GUI components so that they look as shown in Figure 8.8. The `Static Text` to the right of the `Edit text` box for the result has a blank string. The string for the `Pop-up Menu` is set by clicking on the string property at the `Property Inspector` for the menu. This opens the `String` window for the `Pop-up Menu` where we add the information shown in Figure 8.9. We now change the tags for each of the components, as shown in Table 8.3.

TABLE 8.3: Tag names.

GUI Component	Tag
Pop-up menu	`temp_conv`
Edit box below the text `Temperature`	`input_temp`
Edit box below the text `Result`	`result`
Push button	`closeGUI`
Static text for `Result`	`degrees`

We save the GUI with the name `Temp_converter.m`. We design the GUI in such a way that when the user chooses the conversion with the `Pop-up Menu`, the conversion takes place and the result is written in the result box. The callback we need to edit is the one corresponding to the `Pop-up Menu`. First we need to read in the temperature from the `Edit Text` box. We do this with

```
temp = eval(get(handles.input_temp, 'string'))
```

Note that the value of temperature stored in the `Edit Text` box is stored in the variable `temp`. Now, we add the instructions to read the conversion from the `Pop-up Menu`. The variable `val` indicates which conversion we implement with:

```
val = get(hObject, 'Value');
switch val
   case 2
      % Celsius to Fahrenheit
      resultt = temp*1.8 + 32;
      set(handles.degrees, 'string', 'Fahrenheit')
   case 3
      % Celsius to Kelvin
      resultt = temp + 273.15;
      set(handles.degrees, 'string', 'Kelvin')
   case 4
      % Fahrenheit to Celsius
      resultt = (temp - 32)*5/9;
      set(handles.degrees, 'string', 'Celsius')
   case 5
      % Fahrenheit to Kelvin
      resultt = (temp - 32)*5/9 + 273.15;
      set(handles.degrees, 'string', 'Kelvin')
   case 6
      % Kelvin to Celsius
      resultt = temp - 273.15;
      set(handles.degrees, 'string', 'Celsius')
```

```
    case 7
       % Kelvin to Fahrenheit
       resultt = (temp - 273.15)*1.8 + 32;
       set(handles.degrees, 'string', 'Fahrenheit')
end
```

Finally, we write the result to the edit text box result:

```
set(handles.result, 'string', resultt)
```

The complete callback is listed now:

```
% Executes on selection change in temp_conv.
%
function temp_conv_Callback(hObject, eventdata, handles)
% hObject handle to temp_conv (see GCBO).
% handles structure with handles and user data (see GUIDATA).
% Hints: contents = get(hObject, 'String') returns temp_conv
%  contents as cell array.
% contents get(hObject,'Value') returns selected item
% from temp_conv.
temp = eval(get(handles.input_temp,'string'));
val = get(hObject, 'Value');

switch val
    case 2
       % Celsius to Fahrenheit
       resultt = temp*1.8 + 32;
       set(handles.degrees, 'string', 'Fahrenheit')
    case 3
       % Celsius to Kelvin
       resultt = temp + 273.15;
       set(handles.degrees, 'string', 'Kelvin')
    case 4
       % Fahrenheit to Celsius
       resultt = (temp - 32)*5/9;
       set(handles.degrees, 'string', 'Celsius')
    case 5
       % Fahrenheit to Kelvin
       resultt = (temp - 32)*5/9 + 273.15;
       set(handles.degrees, 'string', 'Kelvin')
    case 6
       % Kelvin to Celsius
       resultt = temp - 273.15;
       set(handles.degrees, 'string', 'Celsius')
```

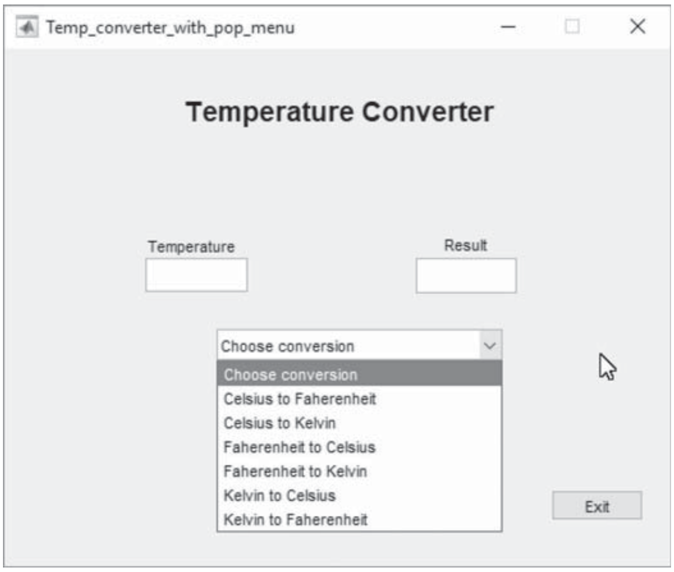

FIGURE 8.10: A run for the temperature conversion GUI.

```
    case 7
      % Kelvin to Fahrenheit
      resultt = (temp - 273.15)*1.8 + 32;
  end
  set(handles.degrees, 'string', 'Fahrenheit')
```

and a run is shown in Figure 8.10.

Example Solution of the Black-Scholes equation.
we show how to calculate call and put options for European options. There we show that we have to solve the Black-Scholes differential equation:

$$\frac{\partial f}{\partial t} + rS\frac{\partial f}{\partial S} + \frac{1}{2}\sigma^2 S^2 \frac{\partial^2 f}{\partial S^2} = rf$$

whose solution is given by:

1. For the call option

$$c = S_0 N(d_1) K e^{-rT} N(d_2)$$

2. For the put option

$$p = K e^{-rT} N(-d_2) - S_0 N(-d_1)$$

where

$$d_1 = \frac{\ln(\frac{S_0}{K}) + (\frac{r+\sigma^2}{2})T}{\sigma\sqrt{T}}$$

$$d_2 = \frac{\ln(\frac{S_0}{K}) + (\frac{r-\sigma^2}{2})T}{\sigma\sqrt{T}} = d_1 - \sigma\sqrt{T}$$

Here $N(x)$ is the cumulative probability distribution function for a variable that is normally distributed with a mean of zero and a standard deviation equal to 1, S_0 is the stock price at time zero, and K is the strike price. The function $N(x)$ is integrated into MATLAB as `normcdf(x)`.

In this example, we construct a GUI that has as input the stock price S_0, the strike price K, the maturity time T, the interest rate variation r, and the volatility σ. We use the Black-Scholes function from the Financial Derivatives toolbox that has the format:

```
[call, put] = blsprice(price, strike, rate, time, volatility)
```

For example, for the data: stock price `S0 = 42`, strike price `K = 40`, interest rate `r = 10%`, maturity time `T = 6 months = 0.5` years, and a volatility σ `= 20%`, we have:

```
[call, put] = blsprice (42, 40, 0.1, 0.5, 0.2)
```

which gives the results for the call and put options as:

```
call = 4.7594          put = 0.8086
```

The layout for the GUI to carry out this computation is shown in Figure 8.11. To this layout we change the strings for each `Static Text`, each `Edit Text`, and the `Push Buttons` as shown in Figure 8.12. Then, we change the tags for the `Edit Text` boxes with the first word in the name of the `Static Text` box next to each of them. That is, the `Edit Text` next to `Stock price` has the tag equal to Stock, and so on. For the `Static Text` boxes we also give the tag name in the same way. Then the top `Static Text` box next to `call` option we make the tag equal to `call` and the other one has the tag equal to `put`. For the `Push Buttons` we give the tags `Calculate` and `Close`.

We save the GUI as `BlackScholes`. Now we edit the callback for the button `Close` as in the previous example. In the callback for this `Push Button` we add:

```
close(gcbf)
```

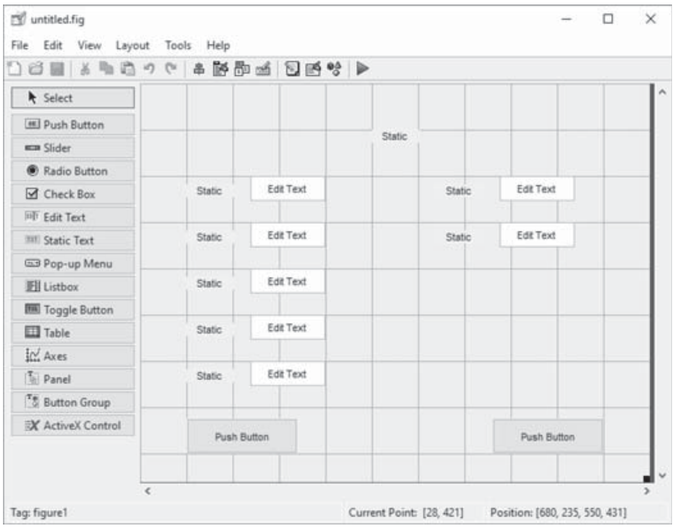

FIGURE 8.11: GUI layout.

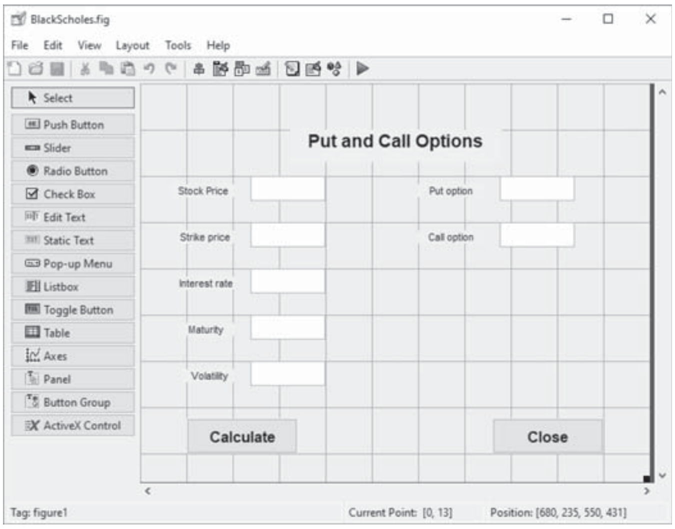

FIGURE 8.12: Final GUI layout.

The next step is to execute the Black-Scholes equation in the callback for the button `Calculate`. First we read the data from the `Edit Text` boxes and then execute the instruction `blsprice`. Finally, we write the results to the empty `Static Text` boxes. The callback for the push button `Calculate` is now described:

1. First we read in the variables for the Black-Scholes instruction `blsprice`. We do this with the instructions `eval` and `get`. As we explained above, `get` reads the string from the `Edit Text` and `eval` converts the string to a numerical value assigned to the variable `stock`. To read the variable from the `Edit Text` box Stock we use then:

```
stock = eval( get(handles.Stock, 'string'));
```

We read the five variables with:

```
stock = eval( get(handles.Stock, 'string'));
strike = eval( get(handles.Strike, 'string'));
int = eval( get(handles.Interest, 'string'));
mat = eval( get(handles.Maturity, 'string'));
vol = eval( get(handles.Volatility, 'string'));
```

2. Now, we make the calculation with the `blsprice` solution with:

```
[call, put] = blsprice(stock, strike, int, mat, vol)
```

3. We write the results to the empty `Static` text boxes with

```
set(handles.call, 'string', call)
set(handles.put, 'string', put)
```

The complete callback for the push button `Calculate` is:

```
% Executes on button press in Calculate.
%
function Calculate Callback(hObject, eventdata, handles)
% hObject handle to Calculate (see GCBO)
%
% eventdata reserved - to be defined in a future version
% of MATLAB
%
% handles structure with handles and user data (see GUIDATA)
stock = eval( get(handles.Stock, 'string'));
strike = eval( get(handles.Strike, 'string'));
int = eval( get(handles.Interest, 'string'));
mat = eval( get(handles.Maturity, 'string'));
vol = eval( get(handles.Volatility, 'string'));
[call, put] = blsprice(stock, strike, int, mat, vol);
set(handles.call, 'string', call);
set(handles.put, 'string', put);
```

Now we execute the GUI by clicking on the `Run` icon. Then we enter the values required and the results are shown in Figure 8.13.

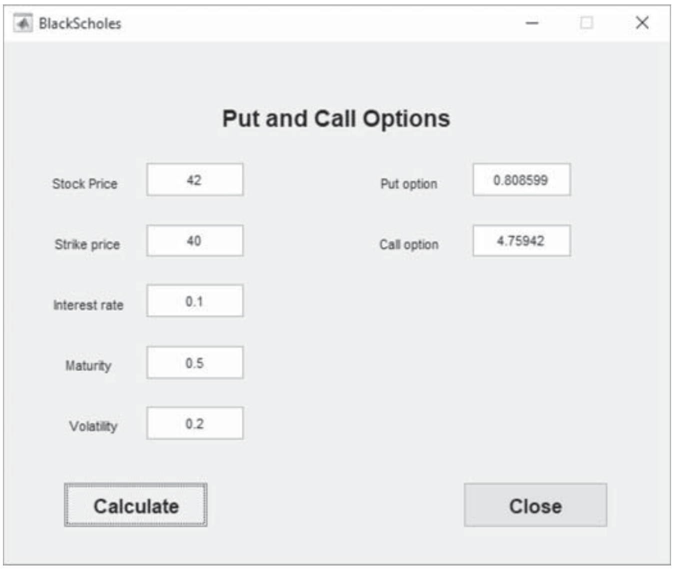

FIGURE 8.13: Final GUI with data.

Deployment of MATLAB Graphical User Interfaces

MATLAB allows deployment of MATLAB files so they can be distributed and used by other users without having a MATLAB license. These deployed files can be used from EXCEL, .NET, Java, or as stand-alone executable files. MATLAB has a tool called `Deployment tool` that guides us in the making of executable files.

In order to run an application outside of MATLAB, the end user needs to install the **MATLAB Compiler Runtime** also known as `MCR`. This is a set of functions that the executable generated uses to run. Thus, it is compulsory to install it. It can also be downloaded free of charge from the Mathworks web page "www.mathwoks.com".

There are three steps that have to be followed to obtain an executable file from a GUI which is composed of m-files and fig-files. These are:

1. Create a project.

2. Add the files.

3. Build the executable file and pack the project.

Now we describe each one of the steps:

1. We start the `Deployment Tool` by selecting the `APPS` tab in the main MATLAB window. There we select the `APPLICATION DEPLOYMENT` set and click on the `Application Compiler` icon. We can also type `deploytool` at

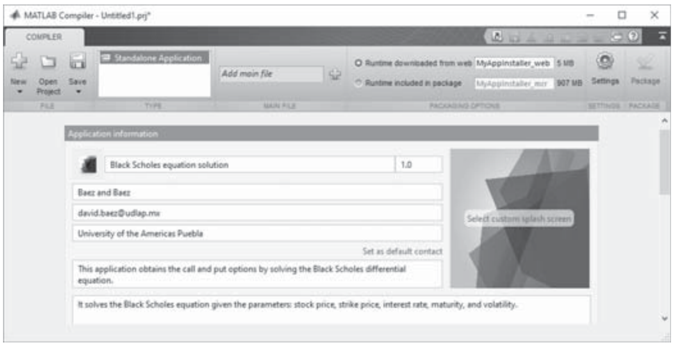

FIGURE 8.14: Deployment tool.

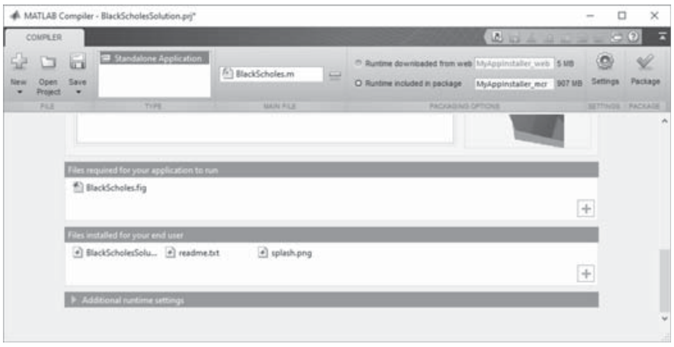

FIGURE 8.15: Loading the files to be deployed.

the `Command Window`. This opens the deployment menu. Then, we select the icon for `New Project` and which type of executable we wish to make.

2. We fill out the information requested such as application name (`BlackScholesSolution`) and additional details about it as shown in Figure 8.14. We save the project as `BlackScholesSolution.prj`. Now we add the files we need in our project by clicking on the plus sign next to the message `Add main file`. We work with the interface developed in Example 8.2. We start the process by adding the m-file `BlackScholes.m`. The file `BlackScholes.fig` and every other file needed are automatically loaded. We also check the radio button to include the `MCR` in the package if the user does not have it already (see Figure 8.15).

3. Once we have the required m-files, we proceed to build the project. We do this by clicking on the ✓mark located in the top right corner of the deployment window. When the process starts, the window shown in Figure 8.16 is opened. It is related to the deploying steps and it goes from `Creating the binaries`, to `Packing`, and `Archiving`.

4. When the process is finished, we have a set of folders with the required executable file and the `MCR` packed and ready for installation. The parent folder

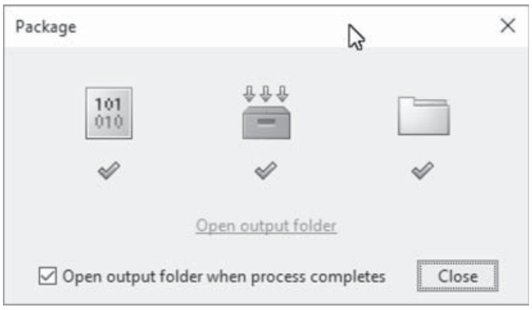

FIGURE 8.16: Creating the executable file to be deployed.

is the `BlackScholesSolution` folder and it contains three folders as follows:

- `for_redistribution`.

  It contains the file `MyAppInstaller_mcr` which installs the `MATLAB Compiler Runtime` (MCR).

- `for_redistribution_files_only`.

  It contains the executable file `BlackScholesSolution.exe`, an icon, a picture that is displayed before opening the application, and a readme.txt document with installation instructions.

- `for_testing`.

  It contains the executable file `BlackScholesSolution.exe`, a readme.txt file with installation information, the image shown when opening the application, and two other files related to the deployment process.

The folders we need to distribute are the first two: `for_redistribution` and `for_redistribution_files_only`.

5. Now we execute the file `BlackScholesSolution.exe` to open the window requesting the input data. A run produces the same window corresponding to the application shown in Figure 8.13. As we see, it is very easy to deploy MATLAB programs to use in a computer that does not have a MATLAB license.

Concluding Remarks

In this chapter we have presented the techniques to create graphical user interfaces, known as GUIs, that eases the process to execute a MATLAB program. We have presented three examples which are representative of typical GUIs. We have also presented the techniques to deploy GUIs and to create executable files that can be run in a platform without a MATLAB license.