Chapter 15

Graphical User Interfaces (GUIs) for Scientific Models

15.1 Introduction

The design of Graphical User Interfaces (GUIs) is usually regarded as an art for programming-for-the-consumer. Middle managers and fast-food workers need GUIs because they are computer illiterates, so the conventional wisdom goes, but surely scientific programmers know their own codes, and are beyond pull-down menus and radio buttons.

To a degree, the conventional wisdom is correct. There is little point in developing a bullet-proof, super-reliable interface for a research code that will be used for a couple of years by a couple of postdocs or students and then discarded.

However, there a couple of reasons why GUIs are useful in science. The first is that a computer program is a dialogue between itself and its programmers and users. Comment lines are the traditional way for the program to "talk". Modern GUIs provide additional modes of communication. In particular, dialog boxes allow a program to bombard users with information even while running.

A second reason is that GUIs can simplify tasks such as specifying initial parameters and interactively analyzing output. Without a GUI, one can edit an initialization file and store output numbers in files for analysis by a stand-alone graphics program. However, this is awkward and time-consuming; it easy to push a button or edit a labeled text box than to search among many lines of code for that one line that needs to be modified. If a scientific task needs to be done over and over again, but never exactly the same way twice, a simple GUI may be a worthwhile investment.

A third reason is that writing a GUI can force the programmer to think: What's important? What parameters are needed to completely specify the problem? In theory, one should always think about these questions anyway, GUI or not. In practice, it is easy to omit details that become crucial only later. Modern structured languages like C and Pascal require the programmer to make a lot of declarations in part to force concrete, specific thinking instead of vague, making-it-up-as-you-go-along coding.

A fourth reason is that over the lifetime of a research code, the programmer can forget much of the fine structure of a program. When I was an undergraduate, Joel Primack, who was my mentor for a research project, stressed the importance of writing everything down at the time. "After six months, you'll read your manuscripts and programs as if they had been written by someone else. So you have to document what you're doing as if you were writing for someone else." It is obviously important to pro-

vide written documentation and lots of comments for computer programs; the reader you write for is your own Future Self, and he or she is an amnesiac. GUIs can help to minimize the defects of memory by making it possible to rerun a program even when the programmer has forgotten the

A fifth reason for GUIs is that science has become much more collaborative. More and more, codes are community models or group models. More and more, the usefulness of a code is being measured by its ability to be shared. GUIs tremendously improved the "collaborativity" of a program.

15.2 Taxonomy

GUI elements can be classified by who initiates the action:

1. Program-initiated GUIS:
   (i) Help, Warning and Error Dialog Boxes
   (ii) Message (Info) Boxes
   (iii) Input, Question, and List Dialog Boxes

2. User-initiated GUIS:
   (i) Display and/or Plot
   (ii) Pause
   (iii) Exit Loop
   (iv) Reassign-While-Running

These GUI elements can be further classified by their aims and actions:

1. Informational:
   (i) Help, Warning and Error Dialog Boxes
   (ii) Message (Info) Boxes
   (iii) Display and/or Plot

2. Interactive Control:
   (i) Pause
   (ii) Exit Loop
   (iii) Reassign-Variables-While-Running

3. Assigment:
   (i) Specification of Cases
   (ii) Modify Default Values at Initialization
   (iii) Reassign-Variables-While-Running

In what follows, we shall describe how each of these different elements can be useful, giving illustrations in Matlab.

15.3 Dialog Boxes

A dialog box is a GUI function that pops up a box full of text on the screen. The program is suspended until the "OK" button is closed.

Dialog boxes need to be used sparingly because they do suspend the program. However, warning or information messages that are merely printed on the screen while the program continues can be easily missed. If the program has made a serious and perhaps fatal error, the program *should* suspend until the user had evaluated the problem.

15.3.1 Information Dialog Boxes

Matlab provides four easy-to-use dialog boxes that require no response except clicking on the "OK" button:

1. Help: **helpdlg('Message string','Title-of-box string')**
2. Warning: **warndlg('Message string','Title-of-box string')**
3. Error: **errordlg('Message string','Title-of-box string','replace')**
4. Message (alias "Info"): **msgbox('Message string','Title-of-box string','IconData','IconCMap')**

In all cases, all arguments except the Ḿessage string' are optional. The third argument of the error box forces it to replace the default error box, which does not happen otherwise.

Each of the first three species of dialog boxes displays an icon that signifies its function: a face plus comic-strip-like dialog balloon for the "help" box, a triangle with an exclamation point inside for the "warning" box, and a hand help up as if to say "Stop!" for the "error" dialog box. The universality of these icons is an illustration of good graphic design: the triangle-with-exclamation-point is employed to warn Macintosh users that an old file is about to be written, and Photoshop users that an RGB-specified color is out of the "gamut" of colors that can be printed by the usual CMYK inks.

By default, the message box displays no icon. However, an optional third argument can be used to specify one of the standard icons, or one can use the optional third and fourth arguments to specify the image data of a custom icon plus the accompanying color map.

In general, use the "help" box to provide information only when the run is not in peril such as "Sarah, I haven't finished the hydrology subroutine yet, but you can run this to model a "dry" atmosphere". A "warning" box is appropriate when the program has identified a difficulty that may or may not make the user terminate the run after reading the dialog box. An "error" box should be displayed only when the problem is so serious that the coder thinks that the user should almost certainly abort the run; the purpose of the box is to explain *why* the run has to be stopped.

The message box is used to supply information of a non-threatening nature. If a code is to be shared by other users, it may be very helpful to pop up an initial message to explain (i) what the code does and (ii) how the user can access various GUI menus and input elements to control the program. For example, **msgbox({ Hi, Fred! You can control the program as follows; The menu allows you to select any of the five published case; Button 'Do your own' will pop up an input GUI screen; Modify the parameters as you see fit.; Good luck! John})** .

In Matlab, **msgbox** will automatically textwrap a string argument. Alternatively, one make the message string into a cell array to force a line break after each element of the array: **'Message String' = {'line one'; 'line two'; 'line three'}** .

15.3.2 Requester (Input) Dialog Boxes

It is possible to write more sophisticated dialog boxes that not only give information, but also request it. Matlab, for example, provides three species of dialog boxes that request input from the user:

1. input dialog box **inputdlg**
2. list dialog box **listdlg**

3. question dialog box **questdlg**

The **inputdlg** box requests the user to insert one or more lines of text. The allowed syntax is

- **replystring=inputdlg('Prompt')**
- **replystring=inputdlg('Prompt','Box Title')**
- **replystring=inputdlg('Prompt','Box Title',number_of_lines)**
- **replystring=inputdlg('Prompt','Box Title',number_of_lines,'Default String or Cell Array')**

where the **replystring** is always either a string or a cell array of strings. The **replystring** is of the same type as **'Prompt'**: if **'Prompt'** is a cell array with four elements, then **replystring** is a cell array of four elements also. One can access the second string of the array by **Replystring{2}** , and convert the string to a floating point number using **eval(string)**.

The integer **number_of_lines** specifies the number of lines that are to be input for each string **Prompt**. If **number_of_lines=2** and **'Prompt'** is a cell array of three strings, then a dialog box will appear with three text boxes each of two lines to hold the user input.

Of course, one can request input from the command line, without using a dialog box, by using the Matlab command **input**. The dialog box allows for more complicated string input.

Indeed, the input dialog box can be used to replace the more sophisticated initialization GUI described in the next section. The goal is to present the user with a screen full of default values for various parameters with the freedom to change as few or as many as desired and then close the dialog box. For example, suppose the parameters are the number of grid points in the x and y directions, which are integers, and the viscosity coefficient, which is a floating point number.

The dialog box shown in Table 15.1 will pop up an input dialog box with three white rectangles, each one line tall, which will contain the default values. The cell array **Prompt** is used to specify the labels for each input line. The cell array **DefaultAns** contains the default parameter values, which must be specified as *strings*. The user can click on some or all of the white rectangles to edit them. The default can be deleted through the backspace or delete key and the new value inserted. When editing is done, one merely clicks "OK" and the dialog box will disappear. The output of the dialog box is the cell array **Answer**. Its elements can be converted to numbers, either integer or floating point, by using the **eval** function.

In this example, there are three lines, but the input dialog box allows an *arbitrary* number of lines [as many will fit on the screen!]. One is free to change all the default values, or none, or some. One can also changes one's mind in the middle of editing, and re-edit lines that have already been changed. The **Answer** cell array is returned only when the "OK" button is clicked.

This input-with-defaults dialog box is so simple and powerful that it is highly recommended. The only limitation is that the input lines are stacked vertically with about thirty spaces reserved for each input, more than one would ever need even to specify a full precision, sixteen decimal place number. In the next section, we will describe a more sophisticated GUI that allows the user to pack more input boxes on a screen at the cost of more fiddling.

Table 15.1: Initialization GUI: Input Dialog Box

```
Prompt = 'no. x grid points=';'no. y grid points=';'viscosity=';       % labels for each input line
Title='initialization dialog box' ;       % title to be printed at top of dialog box
DefaultAns='25';'33';'1.378' ;       % Default values; user can change some, all or none
LineNo=1;       % Number of lines in each input box, always one to insert a number
Answer = inputdlg(Prompt,Title,LineNo,DefaultAns);       % The dialog box appears
% The user can click on a given white box with a default number and type
% to change it. When all editing of defaults is over, simply click "OK"
NX=eval(Answer1) ;       % eval converts the cell array (string) to a number
NY=eval(Answer2);
nu=eval(Answer3);       % Floating point and integers are converted by identical means.
```

A list box does not request keyboard input, but instead offers a list of options, and asks the user to choose one (or click "cancel"). The syntax is **[Selection, OK] = listdlg(*'PromptString'*,TitleofBox,*'ListString'*,S)**. **'PromptString',TitleofBox** are optional; **TitleofBox** is a string that serves as a title or prompt for the dialog box, such as 'Choose a Case'. **S** is a cell array of strings that label the choices, such as **S={'Choice One';'Selection Two'; 'Option Three'}**. (There are a lot of additional options; see the Matlab help for this function.) The strings in italics are names of promperties (i. e., *'ListString'* and *'PromptString'* and must not be changed.) If one hits "Cancel" when the list box appears, **OK=0**; if one selects something, then **OK=1** and **Selection=[]**, the empty set. (More good design; the main program can check **OK** before attempting to respond to the selections.) **Selection** is an integer which is the number of the selection — if the list offers four possibilities, **Selection** will be 1, 2, 3, or 4 — if only one selection is made. In the case of multiple selections, which can be done by Shift-clicking on a Macintosh, then **Selection** will be a vector of integers.

The "question dialog box" **questdlg** is a variation on the list box in which (i) the number of choices is precisely three and (ii) the choices are offered as pushbuttons instead of text-to-be-clicked-and-highlighted. In contrast to the list box, which returns an integer, a question dialog box returns a string. The syntax is **ButtonName = questdlg('Question','TitleofBox', 'StringforFirstChoice',... 'StringforSecondChoice', StringforThirdChoice','StringforDefaultChoice')**. The 'TitleofBox' string will appear as the title of the box. The 'Question' string will appear as a prompt inside the box, followed by a question mark. Clicking the second button will make the string **ButtonName = 'StringforSecondChoice'**.

15.4 Push-Button Menu

When a program can run in a number of standard modes or execute a number of standard cases, the user needs to be able to specify which mode or case. Of course, the user could edit the program file, but this requires accessing a main program with dozens or even hundreds of lines to modify only a single line.

It is far better to have the program do it by GUI by displaying a pop-up menu of pushbuttons, each labeled with one of the modes or cases. In Matlab, the syntax is

integer_flag = menu('Title of the menu','Case One','Case Two','Case Three')

There is no restriction on the number of pushbuttons. When a button is pushed, the variable **integer_flag** is set equal to one if the first choice was pushed, two if the second choice was pushed and so on. This integer can be used in an **if** statement to switch to execute the user-chosen option.

In scientific studies, it is common to run a hundred different cases or parameter combinations during the "exploratory" phase of the study. Most of these cases were

run in haste and only partially analyzed, and in any event are far too numerous for any journal. The PUBLISHED version of a numerical study will normally discuss only four or five cases in detail. One can archive the published cases in a single main program with a popup menu. It is then easy to rerun these cases for further analysis, perhaps as part of a follow-up study years later, even if the programmer has graduated.

15.5 Example: An Initialization GUI, Such as That for a Fluid Mechanics Code

A time-dependent modelling code requires the specification of a large number of parameters. Some of these are discrete variables, such as the number of grid points in the spatial coordinates, the number of time steps, the strength of the viscosity coefficient and so on. Others are continuous functions. All can be specified through either by statements in the code or by user input through a GUI.

A drawback of requester dialog boxes without defaults is that ALL INPUT DATA must be COMPLETELY SPECIFIED each time the code is run. If a code contains 30 parameters, and one only changes one or two from run to run, the need to reenter the other twenty-eight parameters each time is a ghastly time-waster. In the previous section, we showed that an input dialog box with defaults can avoid this problem. In this section, we will describe a different GUI, employing individual static and editable text boxes, that can allow the user to view all the parameters, and then SELECTIVELY change only those that needed to be modified from the defaults. The text box GUI is more complicated to build, but gives the user more freedom, and in principle allows a larger number of input parameters to be displayed and edited on a single screen.

One technical detail is that the initialization GUI must be informed when the user is satisfied. In Matlab, the easiest way to do this is to include (i) a function that creates a figure with all the initialization GUI elements and (ii) the statement **waitfor(gcf)**, which pauses the program until the current figure disappears. One can therefore resume execution of the program merely by closely the figure window that displays the initialization GUI.

The GUI needs two species of elements: “static” text boxes, which display the labels identifying and explaining each parameter and “editable” text boxes. The editable boxes initially display the default values of each parameter, which should be assigned before any of the GUI boxes are called. The user can then change any or all of the text in the editable boxes. The result of the change will be passed back as a string variable.

However, a string variable is not the desired answer; we want to execute an assignment statement that uses the the editable text, converted into a number, as the right-hand side. One could add a bunch of conditional statements in the main program to convert each string into the proper assignment, but there are two problems: (i) the GUI *pauses* the main script so that it can’t execute anything and (ii) even if the main program is still alive, adding several statements to the main program for each assignment from the function would mix the code of the function and main program, which is poor programming practice. (The code that does the work of a function should be *isolated* in the function, not contribute 80 lines to the main program, too.)

Fortunately, it is possible to work around these problems by using a “callback”.

15.6 Callbacks

In Matlab, when a GUI object is activated, it supplies an executable statement to the main program which is called the "callback" of the GUI object. If the callback string is **'NX=5'**, for example, then this assignment statement will be executed in the main program. The callback string is not limited to a single statement, however; it can be a whole list of statements separated by commas or semicolons.

One annoying complication is that in the call to the GUI object, the sequence of statements must be defined as a string. This means that the callback string must begin and end with an apostrophe. However, strings-within-the-string must be bounded by double apostrophes so that the interpreter won't be fooled into stopping prematurely before reading the entire callback.

User-dependent data can be included in the callback by using the **eval** function:

```
NXstring_= uicontrol(qcf,...
      'Style','edit',...
    'Units','normalized',...
'Position',[0.11 0.85 0.1 0.1],...
    'String',int2str(NX),...
 'Callback','NX=eval(get(NXstring_,"String"))' );
```

The **uicontrol** is the generic Matlab name for almost any GUI object that isn't a menu. The **'Style'** property specifies the type of GUI, in this case, an editable text box. The **'Units'** and **'Position'** control the placement of the editable text box on the figure window. When it first appears, the **'String'** property of an editable text box displays the default value of the string. The default value of **NX**, which is defined by a standard assignment such as **NX=20**, is converted from an integer to a string by the built-in Matlab function **int2str**. When the figure window first appears, this integer-converted-to-string appears in the editable text box. When the text in the box is changed by moving the cursor into the box, deleting the existing numbers and then typing in new ones from the keyboard, the **'String'** property is updated to its new value. This triggers an immediate updating of the **'Callback'**, which implies that the assignment statement is executed in the main workspace. The right-hand side of the assignment statement is computed by the **eval** function, which takes a string as its argument and then executes the string as if it were characters typed directly into the command window. The string for **eval** comes from executing a **get** command, which first finds the graphic object whose handle is **NXstring_** and then returns the 'String' property of that object.

One complication is that the callback must use **get** nested inside the **eval** to use the editable text as the right-hand side of the assignment statement. However, the real subtlety is that the **eval** and **get** functions are executed in the workspace of the main script. If the **uicontrol** statement is in a function, then the **get** function has no direct way to find the handle to the editable text box, **NXstring_**, which is defined only in the subroutine.

There are several ways to get around this problem. In our sample program, the string variable is declared **global** by including the statement **global NXstring_** in both the function code and the main program. The main program can then find the editable text box's handle to evaluate its updated **'String'** property.

Although this is probably the simplest and most effective strategy for an initialization window, which is onscreen just once and then disappears, most Matlab books including the manuals suggest instead storing information in the **'UserData'** property of the figure window. The "global variable" strategy will fail if (i) the user types a **clear** statement in the Matlab command window or (ii) if the user makes a second call to

the function that contains the **uicontrol** statement. Neither of these difficulties can arise for an initialization window since the main program is suspended (except for the execution of GUI callbacks).

The "UserData" storage-of-GUI-handles is more robust than the global variable method for very complicated GUIs in which the GUI objects are persistent while the main program is running. Even if a **clear global** statement is executed, the figure keeps its **'UserData'** intact until the figure window is closed. However, this method of storing handles is inappropriate for GUIs that are called only once. The reason is that the UserData strategy requires that one call the GUI-creating function twice, once to initialize the GUI-object and store the handle in the UserData, and the second time to retrieve the UserData.

15.7 Variation: Input Slider

For a jazzier initialization screen, one can replace an editable text box by a slider. This is much more complicated because to be legible and flexbile, the slider needs four GUI elements:

1. Static text box to list the minimum of the slider.
2. Static text box to list the upper limit of the slider range.
3. Editable text box to
   (i) show the value of the slider as it changes.
   (ii) bypass the slider to allow a precise numerical value to be typed in.
4. Slider itself.

Because there are four elements, and also because of the mutual dependence of the editable text box and the slider on each other, the code is much more complicated than for a simple editable text box. Why bother?

The answer is: no compelling reason. However, once one has a prototype of a slider to work with, such as the Matlab code in the two tables here, it takes only a couple of minutes to edit the prototype to make a new slider to control a new variable. Sliders have become popular because it is easier, at least for some users, to click or drag the slider than to type in a precise numerical value in a text box.

Sliders are *never* necessary, however, so the reader who is put off by the intricacy of slider coding can simply ignore sliders.

Table 15.2: Initialization GUI

MAIN PROGRAM

```
global NXstring_; % This global statement allows the editable
% text box to find NXstring_ when executing callback.
NX=GUI_INIT
waitfor(gcf); % suspends program until the figure is CLOSED.
```

INITIALIZATION FUNCTION

```
function NX=GUI_INIT;
% There are five statements:
% (i) global declaration
% (ii) creation of figure
% (iii) default assignment
% (iv) define static text box
% (v) define editable text box
% Sixth statement creates a pushbutton to CLOSE
% the window and return control to the main program
global NXstring_; % Zeroth, create a figure and store its figure handle
fh=figure;
% First, assign a DEFAULT value to NX
NX=20;
% The output of the function is ALWAYS assigned to this default. If the editable text box
% is changed,then NX will change in the main program, but only because the text box callback
% will cause NX= new value to be executed in the main workspace.
% Second, create a STATIC TEXT BOX to label the editable text box.
NXlabel= uicontrol(fh,...
'Style','text',...
'Units','normalized',...
'Position',[0.01 0.85 0.1 0.1],...
'String','no. x grid pts.=');
% Create EDITABLE TEXT BOX; enter numbers in box to change NX
% Default for NX is the string entered after the first 'STRING'
% in the NXstring_ call
NXstring_= uicontrol(fh,...
'Style','edit',...
'Units','normalized',...
'Position',[0.11 0.85 0.1 0.1],...
'String',int2str(NX),...
'Callback','NX=eval(get(NXstring_,"String"))' );
% Editable textbox returns the box-handle NXstring_
% In the callback, "get" looks up the string property of
% the object whose handle is NXstring_.
% eval(s) takes the string 's' as its argument, and
% executes the string as a matlab statement; in this context,
% it converts the string to a number.
% The number which is output by eval is then assigned to NX
closefighandle= uicontrol(gcf,...
'Style','pushbutton',...
'Units','normalized',...
'Position',[0.9 0.1 0.1 0.1],...
'String','Close',... 'CallBack','close');
```

Table 15.3: Matlab Main Program and First Function for a SLIDER

```
global NYsliderhandle_ slidervalue_; % This global statement allows the editable
            % text box to find NYsliderhandle_ when executing callback.
% The value of the slider is returned as the global variable slidervalue_
% when the figure window is closed.
GUI_sliderfather
waitfor(gcf); % suspends program until the figure is CLOSED.
NY=slidervalue_
```

```
function GUI_sliderfather;
global slidervalue_
% The obvious technique of setting slidervalue=GUI_sliderfather doesn't work unfortunately.
%The output argument slidervalue is assigned when GUI_sliderfather is first executed, that is,
% when the figure window is created. Even though slidervalue_ is
% changed by later slider movements, the output value of
% GUI_sliderfather, is not updated when the window is closed.

figurehandle=figure;

% Second step: define minimum and maximum values for the slider.
slidermin=1; slidermax=100;

% Next make the slider appear
GUI_sliderB('initialize',slidermin,slidermax)
disp('Slidervalue in GUI_sliderfather just below')
slidervalue=slidervalue_

closefighandle= uicontrol(figurehandle,...
'Style','pushbutton',...
'Units','normalized',...
'Position',[0.9 0.1 0.1 0.1],...
'String','Close',...
'CallBack','slidervalue=slidervalue_, close');
```

Table 15.4: Second Function for a Slider, Called by Function in the Previous Table

FUNCTION GUI_SLIDERB

```
function GUI_sliderB(command_str,slidermin,slidermax)
% The value of the slider is returned as the global variable slidervalue_
% when the figure window is closed.
% Input: command_str is always the string 'initialize'
% slidermin, slidermax are the maximum and minimum values of the slider
% The default value for the slider is the average of slidermin, slidermax.
% The calling routine MUST create a figure window before GUI_sliderB is called.
% WARNING: this subroutine sets values in the 'UserData' of the parent figure, gcf
global slidervalue_
% The 'value' property of the slider uicontrol whose handle is
% h_sldr is used to store the numerical value chosen in the slider.
% If GUI_sliderB is called with a command_str other than 'initialize',
% it means it has been called recursively by itself, and the next
% block of lines should be executed.
if  strcmp(command_str,'initialize')
handles = get(gcf,'userdata'); h_sldr = handles(1); h_min = handles(2);
h_max = handles(3); h_val = handles(4);
slidermin=handles(5); % get minimum and maximum slider values
slidermax=handles(6); % from userdata on non-initialization calls
slidervalue_= get(h_sldr,'value')
end
if strcmp(command_str,'initialize') % BEGINNING of INITIALIZATION
h_frame = uicontrol(gcf,'style','frame','Units','normalized',...
'position',[0.01 0.01 0.4 0.4]);
h_sldr = uicontrol(gcf,'callback','GUI_sliderB("Slider Moved");',...
'style','slider','min',slidermin,'max',slidermax,'Units','normalized',...
'position',[0.1 0.02 0.2 0.17]);
set(h_sldr,'Value',0.5*(slidermin+slidermax));
% h_min, h_max are handles to two static text boxes that show
% the minimum and maximum values of slider, defined by slidermin, slidermax
h_min = uicontrol(gcf,'style','text','string',num2str(get(h_sldr,'min')),...
'Units','normalized',... 'position',[0.02 0.2 0.09 0.09]);
h_max = uicontrol(gcf,'style','text','string',num2str(get(h_sldr,'max')),...
'Units','normalized','position',[0.25 0.2 0.09 0.09]);
% h_val is the handle for an editable text box. The value it
% displays is slaved by the callback to the 'Value' property
% of the slider. However, if the text box is edited on-screen,
% this update will also change the value of slidervalue_
h_val = uicontrol(gcf,'callback','GU_sliderB("Change Value");',...
'style','edit','string',num2str(get(h_sldr,'value')),...
'Units','normalized','position',[0.13 0.2 0.1 0.1]);
% Next two lines insert the four figure handles plus the
% values of slidermin, slidermax into the 'Userdata' of the parent figure
handles = [h_sldr h_min h_max h_val slidermin slidermax];
set(gcf,'userdata',handles);

slidervalue_= get(h_sldr,'value')
% END of INITIALIZATION

elseif strcmp(command_str,'Change Value')

user_value = str2num(get(h_val,'string'));

% The if statement sets user_value equal to the default if
% user_value has not been previously defined (so its length is zero)
if  length(user_value); user_value = (get(h_sldr,'max')+get(h_sldr,'min'))/2;
% Next two lines ensure that user_value cannot exceed slidermin and slidermax.
user_value = min([user_value get(h_sldr,'max')]);
user_value = max([user_value get(h_sldr,'min')]);

set(h_sldr,'value',user_value); % set slider's 'Value' property equal to user_value
set(h_val,'string',num2str(get(h_sldr,'value'))); % Set editable box's property equal to user_value, too.
slidervalue_= get(h_sldr,'value')
elseif strcmp(command_str,'Slider Moved')
set(h_val,'string',num2str(get(h_sldr,'value'))); slidervalue_= get(h_sldr,'value')
end
```

15.8 Example Two: Post-Run Analysis ("Postvisualization")

Programs often do complicated analysis, using many different species of graphs, after the completion of the calculation. It is rather tedious to type in all the subroutine calls, graph titles, axis limits, and so on to make nice graphs. If the graphs are somewhat standardized, then it may be helpful to associate some menus with an illustration program.

Table 15.5 lists Matlab code for plotting a function $y(x)$ and putting up some menus that allow the user to change the type of graph and the marker symbols very rapidly by clicking menus. When the **uimenu** command is used without a callback but specifying a figure handle as its first argument, **uimenu** creates a top-level menu that appears at either the top of the screen (Macintosh) or the top of the plotting figure window (other platforms) with a text label specified by its **'Label'** property. When **uimenu** is called with a first argument that is the handle of a top-level menu item, the result is a submenu that appears when the top-level menu is dragged down. If the **uimenu** has a callback, it will be executed.

For a post-run analysis, it is not necessary to include a **waitfor** command in the main program to suspend execution of the main program. Instead, the main program can be allowed to terminate. The menus and the plotting window remain alive, but the user can make further modifications to the graphs by typing command lines. Thus, the predictable tasks have been automated and can be done by menus, but the user is still free to do the unpredictable by typing on the keyboard.

One generic problem with Matlab GUIs is that the callback strings can become very complicated and hard to read and debug. To avoid this, the function in Table 15.5 calls another function in its callbacks. This dependent function in the callbacks is listed in Table 15.6. The Matlab manuals and third-party texts consider it very sound practice to define new functions to serve as callbacks.

Table 15.5: Matlab Menu Interface for Post-Run Graphics

```
function dummy=GUIgraphoptions(x,y)
global GUIgraphoptionsPlotType GUIgraphoptionsMarkerType;
% The callback assignment is always executed in the
% base workspace rather than that of this function.
% We must include a global statement in both this function
% and in the calling program so that the
% main program can pass the variables back to this function.
% The callback that actually does the plot was done by
% a function, GUIgraphoptions2.m, because trying to
% put all the statements with conditions in the callback
% itself was a mess.
% This figure returns control to the command line
% but the figure and its menus remain active.
% One can modify the figure by typing commands from the keyboard.
if size(x)  = size(y), Hd_error = errordlg('Argument of GUIgraphoptions is zero',...
'Error Dialog Box Example'); end figurehandle=figure(1);
Hd_help = helpdlg...
({'Graph of (x,y) will be plotted or updated (replotted)'; ...
'only when the Display Plot menu item is activated'},...
'GUIgraphoptions.m Dialog Box');
% Default values
GUIgraphoptionsPlotType=1;
GUIgraphoptionsMarkerType=1;
handle_menuone=uimenu(figurehandle,'Label','Plot Type');
handle_menutwo=uimenu(figurehandle,'Label','Marker Type');
handle_menuthree=uimenu(figurehandle,'Label','Display Plot');
menuitem11= uimenu(handle_menuone,'Label','Line Graph',...
'Callback','GUIgraphoptionsPlotType=1');
menuitem12= uimenu(handle_menuone,'Label','Stem Plot',...
'Callback','GUIgraphoptionsPlotType=2');
menuitem21= uimenu(handle_menutwo,'Label','Open Disks',...
'Callback','GUIgraphoptionsMarkerType=1');
menuitem22= uimenu(handle_menutwo,'Label','x',...
'Callback','GUIgraphoptionsMarkerType=2');
% The next menu item has a callback that calls a function
% to actually display the plot
menuitem31= uimenu(handle_menuthree,'Label','Display Plot',...
'Callback','GUIgraphoptions2(x,y)');
```

Table 15.6: Matlab Menu Interface: Dependent Function

```
function GUIgraphoptions2(x,y)
global GUIgraphoptionsPlotType GUIgraphoptionsMarkerType;
if GUIgraphoptionsPlotType==1 & GUIgraphoptionsMarkerType==1 plot(x,y,'k-o'); end % if
if GUIgraphoptionsPlotType==1 & GUIgraphoptionsMarkerType==2 plot(x,y,'k-x'); end % if
if GUIgraphoptionsPlotType==2 & GUIgraphoptionsMarkerType==1 stem(x,y,'k-o'); end % if
if GUIgraphoptionsPlotType==2 & GUIgraphoptionsMarkerType==2 stem(x,y,'k-x'); end % if
```

15.9 Interactive Control Through Menubar (Pulldown) Menus and Computational Steering

Pushbutton menus to initialize a calculation are easy to write because the menu is active only at the very beginning and the user is allowed to only make one choice. However, Matlab also supports menus attached to the menubar of a figure or the screen. These are much more powerful in the sense that they allow the user to interactively alter the run in the middle of execution.

The **uimenu** command can be used to add menus both at top level, and at sublevels. On the Macintosh, top level menu items are added to the usual top-of-the-screen menu bar. (It is only possible to have a single top level menu; there is a bug on the Mac such that selecting any uimenu-added top level item simultaneously causes all the uimenu-top level items to execute). On Unix and Windows machines, the top level menu items appear on top of the figure window.

On any platform, one can add submenu items by using the handle of the top level **uimenu** command as the first argument for the **uimenu** command that creates the submenu.

The advantage of a menubar menu is that the user can select the item at any time during the course of a run. This makes it possible — in the middle of execution, and without stopping the code — to

- Display or plot variables
- Pause execution
- Exit a loop
- Change variables
- Load from or save to a file

A sample Matlab code is shown in Table 15.7. The loop factors a matrix and does two plots during each iteration merely as standins for the number-crunching of a typical code; in numerical applications, the loop would be over timestep or iteration number. While a long run is unfolding, it is common to watch graphs on the screen, and want more. Menus make it possible to call up additional printed output or additional graphs to confirm that the run is failing or to shed new light on interesting phenomena in the middle of execution. If a Newton iteration is failing to converge, or converging very slowly, one can use a pulldown menu to change the approximate root, and restart the iteration from a different number. Similarly, one can save interesting immediate results by pulling up a save-to-file dialog box.

Table 15.7: Menus-to-Interrupt a Loop

```
figure_handle=figure;
h_uimenu=uimenu(figure_handle,'Label','PauseLoop')
% One can display or plot variables in the middle of the loop through the next two sub-menus
h1=uimenu(h_uimenu,'Label','Display Egad!','Callback','display(["Egad!= ",num2str(Egad)])')
h2=uimenu(h_uimenu,'Label','Plot a Row','Callback',['plot(A(1:n));' 'title' 'drawnow;' 'pause'])

% One can change a variable in the middle of the loop through either of the next two sub-menus
h3=uimenu(h_uimenu,'Label','Input Smoker','Callback',...
                    'Smokercell=inputdlg("Smoker=","Smoker inputdlg");Smoker=eval(Smokercell1)')
h4=uimenu(h_uimenu,'Label','Assign Boomer=3','Callback','Boomer=3')

% One can pause the loop
h5=uimenu(h_uimenu,'Label','Pause','Callback','display("paused");pause')

% One may also stop the loop without stopping execution of statements after the end of the loop.
% The callback changes the value of a flag, which triggers an IF statement to skip the guts of the loop.
h6=uimenu(h_uimenu,'Label','Eject from Loop','Callback','myflag=0')
for j=1:jmax
if myflag==1
B=inv(A); disp(['j=',int2str(j)]); slurpy=mesh(A); drawnow; plot(A(5,1:4)); drawnow; iter=iter+1
end end
```

15.10 Summary

GUIs can be useful in scientific applications, especially for *shared* codes. The ease of coding a GUI varies tremendously with the active lifetime of the GUI object. In terms of increasing difficulty, the three categories are

1. "Individual dialog boxes and pushbuttons"
2. "Persistent-while-paused"
3. "Persistent throughout execution of the main program"

Dialog boxes and pushbutton menus, last just long enough for the user to make a single choice. These are easy to write. "Persistent-while-paused" objects, such as window that displays editable text boxes that control initialization parameters, suspend the main program while the user can perform one or many inputs. The program resumes only after the window disappears, which minimizes side effects. GUIs that persist while the main program is running or while other GUI functions are called are very tricky to write because the main program and various GUIs, all simultaneously executing and/or displaying windows, can easily clobber each either.

The best way to code a GUI is to copy a prototype, such as the Matlab examples in the table, and then lightly edit it to fit the user's needs. Patrick Marchand's *Graphics and GUIs with Matlab*, now in its second edition, is also a good source of sample code and ideas. A borrowed template insulates the user from most of the complexities of GUI-writing; the downside is that the user doesn't understand the borrowed template very well. However, because the purpose of the GUI is merely to *report* or *assign* variables, there is little danger that the GUI will lead to errors.

Mathworks, Inc., now provides a built-in GUI-maker called **guide**. Marchand's book describes an alternative he dubbed **guimaker**. The reason that GUI-writing applications exist is that combining multiple GUIs on a single screen requires a lot of trial-and-error. One difficulty is interaction between different GUIS. Another is that each GUI has a specified size and position, and specifying new position and sizes until the boxes are aligned and do not overlap is very tedious. Unfortunately, neither **guide** nor **guimaker**

is particularly easy to use. The best strategy is therefore to steal or build a template, and recycle it endlessly.

Persistent GUIs with mouse-draggable objects and lots of different GUI objects on the screen are the ultimate in user-friendliness. They are also sufficiently complicated to write that they can consume vast amounts of programming time without materially advancing the science. Indeed, sometimes GUI-programming can be a way of evading the science when an investigation is blocked, an algorithm is unstable or error-prone, or a project lacks clear goals.

A good scientist should use simple GUIs a lot and complicated GUIs not at all.