14:440:127– Introduction to Computers for Engineers

Notes for Lecture 09

Rutgers University, Spring 2010 Instructor- Blase E. Ur

1 Menus

Rather than using the *input* statement, you can use Matlab to create a graphical menu, allowing the user to click on their selection. The example below creates a menu and then uses *switch case* to identify which option was chosen. Note that you must set the *menu* command equal to a variable, just as you did with the *input* statement. The first input argument to the menu function is the text displayed on the top of the menu. All of the subsequent arguments are the choices, which are strings separated by commas. The *menu* function returns an integer indicating which option was chosen.

```
food = menu('Welcome to White Castle, may I take your order?',
               'Cheeseburger','Chicken Rings','Mr. Pibb and Red Vines','Ok');
switch food
    case 1
        disp('Good call on the cheeseburger.')
    case 2
        disp('Chicken comes in ring form?')
    case 3
        disp('Crazy delicious!')
    case 4
        disp('Yo momma''s so fat, she went to White Castle...')
        disp('...looked at the menu... and said OK');
end
```

2 GUIs- Graphical User Interfaces

To make a GUI (graphical user interface) in Matlab, type *guide* in the workspace. The GUI editor will pop up. Choose "Blank GUI". You should see a grid, along with a bunch of buttons on the left. You can click and drag these buttons onto the grid, and you'll be making your interface. First, you design the aesthetics of your interface. Then, click on the green "play button" to the top right of the GUIDE editor and you'll program all of the logic behind the interface.

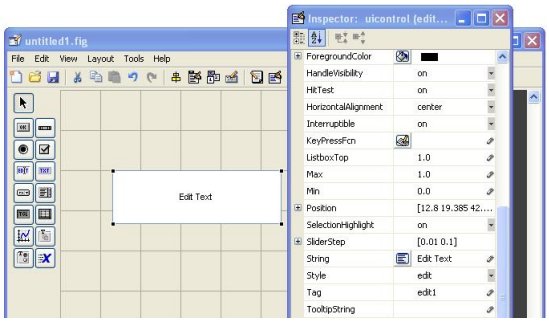
After programming the logic behind the interface, you'll see that you've saved both a *.fig* and a *.m* file. You need **both** of these files to make your GUI work! Also, if you want to edit the interface for your GUI after programming part of it, type *guide* in the workspace and choose the "open existing GUI" tab.

It is extremely important to note that we've made a paradigm shift in our approach to computer programming when working with GUIs. Previously, our programs more or less ran from top to bottom, in order. With GUIs, we are basically doing things in an event-driven fashion. We have a function for each 'event,' such as a button being pressed or a user entering text. When that event happens, we run the code in the associated function. Therefore, you never really know in what order the functions will execute.

Note that GUIs aren't covered in your book. Instead, you should check out the following web tutorial, from which I took part of this lecture (and followed much of the same terminology): www.blinkdagger.com/matlab/matlab-gui-graphical-user-interface-tutorial-for-beginners

2.1 Properties of Objects

Each time you drag another button/text/thing into your interface when using *GUIDE*, you create a new "object." An object can be a clickable button, an editable text box, a static (unchanging) text box, a graph, radio buttons, or all sorts of fun, exciting, and glamorous things. Each "object" has a number of properties that define its appearance. You can change these properties by double clicking an object. A window like the following will open:



You'll notice that the names of the properties are on the left, and the values are on the right. You can use this property inspector to change the appearance of the properties. Here are some of the most important properties:

- **Tag** is the *name* of the object. The **Tag** thus allows you to refer to the object in your Matlab code, retrieving and changing the properties of the object. If the **tag** of your object is "edit1," then the variable *handles.edit1* will reference that object. Yes, you always precede the **Tag** by "*handles.*," which we'll discuss in a future lecture.
- **String** is another important property. For either type of text box, the **String** property is the text that's displayed. Similarly, for a clickable button, the **String** is the text displayed on the button.
- For checkboxes, the most useful property is **Value**, which will usually contain 1 if the box is checked, and 0 if it is not.
- There are many other properties, many of which are self-explanatory.

2.2 The GUI m-File

After you create your interface with *guide*, you'll end up with an m-file for your GUI that likely looks much different than m-files you've seen before. This single m-file likely contains dozens of functions. When you run your interface, Matlab doesn't go through this m-file sequentially. Rather, as functions are "activated" or "called" based on actions in the GUI, Matlab will execute the code in that subfunction only. Here are some to look out for. Let's assume I saved my gui as "blase.m" (and thus "blase.fig"):

- `blase_OpeningFcn` is a function that executes when the GUI is first opened. Let's say that every time the GUI was opened, you wanted a song to play. You would put commands to play a song in this function.
- Assume you've created a clickable button with the tag *pushbutton1*. There will be a function `pushbutton1_Callback` that executes each time the user clicks on the corresponding button. Let's say you had a sound file that played the sound "ouch," and each time someone clicked the button, you wanted the "ouch" sound to play. You'd put commands for playing the sound into this function.

2.3 Functions for Programming a GUI

Inside a GUI's m-file, there are two commands you'll use extensively. One is *get*, and the other is *set*. As you may guess, *get* retrieves the value of a certain property, and *set* changes the value of those properties. Often, you'll want to get or set the properties for a GUI element. To do this, you'll need to use the structure called "handles", which allows you to refer to that GUI element.

For instance, let's say you had a static (unchanging) text box whose *tag* is "static1". If you wanted to set its "FontWeight" property, you could say:

```
set(handles.static1, 'FontWeight', 'bold');
```

Notice that you always say *handles.TAG*, replacing "TAG" with the name of that element. Handles is a structure,

which is why you use the period. Notice also that "FontWeight", the name of a property, is a string.

Copying the online tutorial referenced earlier, I made a program called "adding.m", which had two editable text boxes (which I tagged *edit1* and *edit2*) and a button (which I tagged *pushbutton1*). Note that the function `pushbutton1_Callback` is run whenever the button tagged *pushbutton1* is clicked. In my example, I want to add the numbers in *edit1* and *edit2* when the button is clicked, and then to display the example in *text2* (a static, non-editable text box). Below, you can find my code that did this. Note that we needed to convert from strings to numbers and back again– you can't add strings!

```
function pushbutton1_Callback(hObject, eventdata, handles)
a = get(handles.edit1,'String');
b = get(handles.edit2,'String');
c = str2num(a) + str2num(b);
set(handles.text2,'String',num2str(c));
```

There are a handful of other fairly useful functions when working with GUIs. *ginput* is a function that allows you to get the coordinates of a mouse clicking on a plot, etc. To get one input, you'd use *[x y] = ginput(1)*. To instead keep allowing the user to click until they hit return, instead use *[x y] = ginput()*. The coordinates that are returned are based on the axes of the graph.

When working with GUIs, you may want a user to click somewhere in the GUI. You can use the function *waitforbuttonpress*. The program pauses until they either click the mouse or hit a key on the keyboard. This function returns 0 when the mouse is clicked, and 1 if the keyboard is clicked.

If they clicked on a GUI with their mouse, the variable *gco* will contain the handle to the last object they clicked on. Remember set/get? The first argument to those was the handle to an object (i.e. handles.item1), so you can just replace that first argument with *gco* to refer to the last object that was clicked.

3 Example: The Russian Reversal

```
function varargout = soviet(varargin)
gui_Singleton = 1;
gui_State = struct('gui_Name',       mfilename, ...
                   'gui_Singleton',  gui_Singleton, ...
                   'gui_OpeningFcn', @soviet_OpeningFcn, ...
                   'gui_OutputFcn',  @soviet_OutputFcn, ...
                   'gui_LayoutFcn',  [] , ...
                   'gui_Callback',   []);
if nargin && ischar(varargin{1})
    gui_State.gui_Callback = str2func(varargin{1});
end
if nargout
    [varargout{1:nargout}] = gui_mainfcn(gui_State, varargin{:});
else
    gui_mainfcn(gui_State, varargin{:});
end
```

```
% --- Executes just before soviet is made visible.
function soviet_OpeningFcn(hObject, eventdata, handles, varargin)
handles.output = hObject;
guidata(hObject, handles);
backgroundImage = importdata('flag_ussr.jpg');
axes(handles.axes1);  %place image onto the axes
image(backgroundImage);  %remove the axis tick marks
axis off
set(handles.edit1,'BackgroundColor',[1 0 0]);
set(handles.edit1,'ForegroundColor',[1 1 1]);
set(handles.text1,'BackgroundColor',[1 0 0]);
set(handles.text1,'ForegroundColor',[1 1 1]);
set(handles.text2,'BackgroundColor',[1 0 0]);
set(handles.text2,'ForegroundColor',[1 0 0]);
set(handles.text4,'BackgroundColor',[1 0 0]);
set(handles.text4,'ForegroundColor',[1 0 0]);
```

```
% --- Outputs from this function are returned to the command line.
function varargout = soviet_OutputFcn(hObject, eventdata, handles)
varargout{1} = handles.output;
```

```

function edit1_Callback(hObject, eventdata, handles)
```

```
% --- Executes during object creation, after setting all properties.
function edit1_CreateFcn(hObject, eventdata, handles)
if ispc && isequal(get(hObject,'BackgroundColor'), get(0,'defaultUicontrolBackgroundColor'))
    set(hObject,'BackgroundColor','white');
end
```

```
function pushbutton1_Callback(hObject, eventdata, handles)
america = get(handles.edit1,'String');
l = find(america==' ');
first = america(1:l(1)-1);
second = america(l(1)+1:l(2)-1);
third = america(l(2)+1:end);
disp(get(handles.edit1,'ForegroundColor'))
secondmod = second;
russia = [third ' ' secondmod ' ' first];
set(handles.text4,'String',russia);
set(handles.edit1,'ForegroundColor',[1 0 0]);
z = rand(1,100);
set(handles.edit1,'BackgroundColor',[1 0 0]);
set(handles.edit1,'ForegroundColor',[1 0 0]);
set(handles.text1,'BackgroundColor',[1 0 0]);
set(handles.text1,'ForegroundColor',[1 0 0]);
set(handles.text2,'BackgroundColor',[1 0 0]);
set(handles.text2,'ForegroundColor',[1 1 1]);
set(handles.text4,'BackgroundColor',[1 0 0]);
set(handles.text4,'ForegroundColor',[1 1 1]);
for x = 1:98
    set(handles.text4,'ForegroundColor',[z(x) z(x+1) z(x+2)]);
    pause(.1)
end
set(handles.edit1,'BackgroundColor',[1 0 0]);
set(handles.edit1,'ForegroundColor',[1 1 1]);
set(handles.text1,'BackgroundColor',[1 0 0]);
set(handles.text1,'ForegroundColor',[1 1 1]);
set(handles.text2,'BackgroundColor',[1 0 0]);
set(handles.text2,'ForegroundColor',[1 0 0]);
set(handles.text4,'BackgroundColor',[1 0 0]);
set(handles.text4,'ForegroundColor',[1 0 0]);
```

4 Lo-Fi Prototyping

In class, we showed a demo of lo-fidelity prototyping. The most important take-away point is that it's important for engineers to do initial tests of the usability of their designs. Pencil and paper works perfectly fine!

5 Numerical vs. Symbolic Math

In Matlab, Mathematica, and other computer programs that 'do' math, it's essential to differentiate between *numerical* and *symbolic* methods. The most basic explanation is that *numerical* methods deal with estimates and approximations. If you refer to a variable X, Matlab will look up what number is stored in the variable X.

In contrast, *symbolic* methods deal with exact answers. If you refer to a symbol X, it will treat that X as simply the symbol X, which can stand for anything. It won't look up what's stored in any variable since X is simply the symbol X.

It may seem that it's best to always use symbolic computations. However, in engineering, we often want numbers at the end of working on a problem. Of course, we can often use symbolic methods and then substitute for the symbols at the last minute. The real reason is that it's not always mathematically possible to get exact answers in a reasonable amount of time, or at all; sometimes, estimates must suffice!

In this lecture, we'll first look at numerical methods (estimates), and then at symbolic methods. Note that we're just touching the surface of numerical methods. Chemical Engineering, Mechanical Engineering, and Biomedical

Engineering all require that you take a class where numerical methods are discussed extensively. 'Matlab 2', if you will.

6 Numerical- Solving Equations

6.1 One variable- fzero

You can use numerical methods to solve equations in one variable using the *fzero* function. First, you must rewrite the equation to have zero on one side. i.e. to solve `cos(x) = sin(x)`, you would actually want to find where `cos(x) - sin(x)` equals 0. The *fzero* function takes two inputs: 1) a function handle, and 2) a "guess" for the correct value. This is because the numerical method estimates the answer by simply "looking around" for an input value that results in a function output that's approximately 0. Thus, for different "guess" values, you may get different answers if multiple values of x cause the expression to equal 0, or even no answer at all! Not getting an answer doesn't necessarily mean there's no solution to the equation. It may just mean you're looking in the wrong place as the *fzero* function will effectively give up if it doesn't seem to find values heading towards zero!

```
cs = @(x) cos(x) - sin(x);

fzero(cs,2)
    ans = 0.7854

fzero(cs,7)
    ans = 7.0686
```

6.2 Multiple variables- fsolve

Matlab can solve nonlinear systems of equations (i.e. where you're solving for multiple variables), but it's a much harder process. Let's first take a look at such a system:

$$2x - y = e^{-x} \tag{1}$$

$$x + 2y = e^{-y} \tag{2}$$

The first thing to notice in this system of two equations is that there are two variables. Great. That means we can, in theory, find values for x and y. The second salient feature of these equations is that they're non-linear. In other words, you can't write them as a linear combination $Ax + By + C = 0$ because of the stubborn exponential terms. Therefore, we can't use left-division to solve these as we could for a linear system of equations.

To solve this nonlinear system of equations, let's first write these two equations as a single user-defined function. Matlab requires us to use a trick here. Matlab likes to have functions with one input when using *fsolve*, so we'll write a function with one input, N, where N(1) represents our x value and N(2) represents our y value. We also need to write out output as one vector with two output values, where the first element of the output is the first equation written as an expression that equals 0, and the second element of the output is the second equation written as an expression that equals 0. In other words, rewrite the equations with all the terms on one side of the equals sign. Here's our function:

```
function out = functionof2variables(N)
out = [2*N(1) - N(2) - exp(-N(1));  N(1) + 2*N(2) - exp(-N(2))]

%%%% recall that N(1) is x, N(2) is y, and these two equations are solved for 0
```

Now, you can call the *fsolve* function, giving it the function handle of the user-defined function we just created, and also initial guesses for the variables. Here, our first guess is that x and y are both 5. The *fsolve* function works best (and may ONLY work) if you give it good guesses. (i.e. try this example with -500 as your initial guess for each. SOLVE FAIL.)

```
[N fval] = fsolve(@functionof2variables,[5 5])
```

When Matlab gives us the two outputs we requested, N will contain approximate values for x and y that solve our system of nonlinear equations, and *fval* is the value of our user-defined function *functionof2variables* at the solution Matlab found. Note that since we rewrote our system of equations so that each equation would equal 0, we want *fval* to be as close to 0 as possible:

```
%%% %our output
N =
    0.4255    0.1976

fval =
  1.0e-010 *
   -0.2989
   -0.0472
```

7 Numerical- Calculus

Matlab has very good built-in methods to estimate solutions to calculus problems. Keep in mind that these estimates need to be taken with a grain of salt– you need to understand how they work and verify that the estimate you're trying to do (and the answer that you get) make sense! In general, for data you collect as an engineer, you will use numerical methods to estimate the solutions to calculus problems based on that data. In contrast, if you know the exact equation for which you want to find an integral or derivative, use symbolic math.

7.1 Numerical Derivatives

Given some set of x points and y points, how do you estimate the slope (first derivative) of these values? Well, recall that the derivative $\frac{dy}{dx}$ is just the change in y over the change in x: $\frac{\Delta y}{\Delta x}$.

There's a Matlab function called *diff* that accepts ONE argument– a single vector. It then calculates the difference between each element. Thus, if we take the *diff(y) ./ diff(x)*, we find the change in y divided by the change in x for each point, thus estimating the derivative:

```
x=0:.1:10;
y = x.^2;
slope = diff(y)./diff(x);
```

Notice that slope is now a vector of the slopes at all of the points corresponding to the x and y vectors. How can we find out what the slope is when x = 3.4? Well, simply look at points just above and just below 3.4 for x and f(x), and calculate the change in slope:

```
x = [3.4-.0001  3.4+.0001];
y = x.^2;
slope = diff(y)./diff(x)
```

7.2 Numerical Integrals

There are many numerical techniques for integration, starting with the trapezoid rule. You can calculate an estimated definite integral using the trapezoid rule by sending the *trapz* function the vector of x points and the vector of y points, as with plot:

```
x = 0:.1:10;
y = x.^2;
Z = trapz(x,y);
disp(Z)
```

However, a better method is to use *quad* or *quadl*, which use Simpson and Lobatto quadrature (more accurate methods to estimate the integral). Notice that, rather than giving these functions the vectors of x and y points, you give them a function (i.e. as a function handle from an anonymous or user-defined function), along with the lower and upper bounds for the integral. In most cases, *quad* and *quadl* give you the same answer; it's only in special cases (which you'll learn about in later numerical methods classes) when you need to decide which to use.

```
y = @(x) x.^2;
Z = quad(y,0,1);
disp(Z)
```

Use anonymous functions here when possible since they're short and sweet. However, user-defined functions are necessary when your function is more complicated. In a situation like *quad*, where a function handle is required, you must precede the name of user-defined functions (or Matlab's built-in functions) by the `@` symbol, which creates a "handle" to that function. In other words, you're telling Matlab in the example below not to look up the variable h to see what number it contains, but rather that there's a function named h that will be an input to the *quad* function. Recall that you DON'T do this for an anonymous function:

```
%%% in an m-file called h.m, create:
function out = h(x)
out = 2*x - 3;

%%% Then, in the workspace, type
Z2 = quad(@h,0,5);   %% don't forget the @
disp(Z2)
```

8 Symbolic Methods

In the previous section, we saw numerical methods that estimate the answers to calculus problems. This was useful whenever we had a data set we collected and needed to perform calc operations on that data, or when the function describing the data is fairly complicated.

However, if you are dealing with a relatively simple equations, you can use symbolic math, which deals with variables as 'symbols,' allowing you to get the exact answers rather than estimates. For instance, taking the derivative of x^2 would return $2x$ rather than an estimate at some particular point. In essence, you can think of the numerical methods as estimating a definite integral, whereas symbolic methods can either evaluate an indefinite integral OR give an exact answer for a definite integral.

8.1 Intro to Symbolic Math

You can specify which 'variables' should be considered symbolic by typing something like *syms x y*, which will make x and y symbolic variables. From then on, whenever you refer to x or y, Matlab will treat them as a symbol rather than as a variable.

```
syms x;
2*x^3
    ans =  2*x^3
```

8.2 Making Symbolic Math All Neat and Pretty

There are functions to help you work with symbolic variables, including *collect*, *expand*, *factor*, and *simplify*, among others. You may be able to guess what these do by the names (collect like terms, expand powers, factor an expression, or put the expression into its simplest possible form). For instance, let's see how collect and factor work:

```
syms x;
collect(x^2 - 3*x^2 + 5 - 2*x + 32*x)
    ans =  -2*x^2+5+30*x
factor(x^3-1)
     ans =  (x-1)*(x^2+x+1)
```

8.3 Converting symbols to numbers

As engineers, you might get to the point where symbols are no longer useful. You need to figure out what numbers these symbols represent. To do this, use the *subs* command, which is short for 'substitute'. In the following example,

we'll store a symbolic expression in the variable c. Then, we'll tell Matlab that in the expression stored in the variable c, it should substitute a 1 everywhere there was an x:

```
syms x;
c = x^3 - 32;
subs(c,x,1)
    ans =
       -31
```

You can also do this for multiple variables at a time. *subs* still expects 3 input arguments, so you still give it the variable that stores the equation/expression in which you'd like to perform the substitution, then a cell array of all the symbols for which you'd like to substitute, and then a vector of the values that will be substituted in for the respective symbols:

```
syms x y
c = 3*x^2 - 5*y;
subs(c,{x,y},[1 2])
     ans = -7
     % note that we made the substitution x = 1, and y = 2
```

9 Symbolic- Solving Equations

You can also solve equations symbolically using the *solve* function, which works for single equations or systems of equations. Note that, in contrast to the *fzero* fuinction, you can enter equations, not just expressions that should equal 0:

```
syms x;
solve('x^2 + 5*x -3 = 11','x')
     ans =  -7  2
```

Note that in this example, we need to put the symbol x in quotes. What gives? Well, because our equation contains an equals sign, Matlab would get confused and not realize that we're writing an equation. It would think we're trying to set a variable. Therefore, the whole equation must be stored as a string. Let's try another example:

$$cos(a * x) = 3/6 \tag{3}$$

```
solve('cos(a*x) = 3/6')
    ans =
        -pi/(3*a)
        pi/(3*a)
```

In general, *solve* also handles non-linear systems. Let's reprise our earlier example from our discussion of *fsolve* and use *symbolic methods* to find a solution. Note that we get the example returned to us as something called a *structure*, which we'll discuss next week. For now, just know that $c.x$ and $c.y$ give us the x and y components of the structure c.

$$2x - y = e^{-x} \tag{4}$$

$$x + 2y = e^{-y} \tag{5}$$

```
c = solve('2*x - y = exp(-x)', 'x + 2*y = exp(-y)');
c.x
     ans = 0.42551406154010883454714967781887142
c.y
     ans = 0.19759432948542880834848668074929059
```

10 Symbolic- Calculus

10.1 Limits

To find the answer of a limit for a symbolic equation, you can use the *limit* function, specifying the equation and the location at which the limit is taken:

```
syms x;
limit(1/x,inf)
     ans = 0
```

10.2 Derivatives

The function to find symbolic derivatives has the same name as the one for finding numerical derivatives: *diff*. Matlab looks at the type of input you give it to determine whether to do a symbolic or numerical derivative:

```
syms x;
diff(2*x^2 + 5*x - 382)
    ans = 4*x+5
```

10.3 Integrals

For symbolic integrals, there's an even easier function than for numeric integrals: *int*. Commonly, you'll either specify 1 or 3 input arguments:

- $int(E)$ calculates the symbolic, indefinite integral of that expression.
- $int(E, a, b)$ calculates the definite integral (the answer) on the interval [a,b]

```
syms x;
int(3*x^2 + 4)
    ans =  x^3 + 4*x

int(3*x^2 + 4, 0, 5)
    ans = 145
```