14:440:127– Introduction to Computers for Engineers

Notes for Lecture 07

Rutgers University, Spring 2010 Instructor- Blase E. Ur

1 Animation

Taking the power of loops together with Matlab's *plot* function, we can create animations. The idea behind animated graphs is that we'll graph something, pause, graph something slightly different, pause, graph something slightly different, and so on. In essence, you'll write a loop that plots slightly different things in each iteration.

There are two very useful functions you'll use when creating animations. One is *pause( )*, which pauses for a specified number of seconds i.e. *pause(0.5)* stops the program for 0.5 seconds before continuing.

The other useful function is *axis([xmin xmax ymin ymax])*, which specifies a particular axis for a graph/plot. This allows you to slowly add more data to your plot, but have a consistent axis. Using the *axis* command, which should almost always directly follow *plot*, the axes won't change, making it seem like only the graph is moving.

One example of animation is to graph a single point over time. For t = 0 through 20, let's graph $x(t) = t * cos(t)$ and $y(t) = 5 * sin(t)$. To do this, we'll create a *for loop* for t running from 0 to 20, moving up in small increments so that it looks like the point is moving continuously. We'll display it as an X rather than as a single point so that we can actually see it. We'll define our axes based on the theoretical minima and maxima of the function within our specified domain. Here's one way of animating this function:

```
% X floating through space
for t=0:0.1:20   % choose a small interval
    x=t*cos(t);
    y=5*sin(t);
    pause(0.05) % can come anywhere in the loop
    plot(x,y,'X')
    axis([-20 20 -5 5])  % must follow plot
end
```

It might be easier to precompute all of the points we'll display and store them as vectors, and then just loop through these vectors. Let's rewrite the above loop using this new method. Note that we can also improve upon our method of determining the axes since we've precomputed all points that will be displayed:

```
% X floating through space
t = 0:0.1:20;
x=t.*cos(t);  % don't forget dot operations
y=5*sin(t);
for j = 1:length(x)
    pause(0.05) % can come anywhere in the loop
    plot(x(j),y(j),'X')
    axis([min(x) max(x) min(y) max(y)])
end
```

Another use of animation is to show motion over time by leaving a "trail" of where the X has been. Let's retain our method of precomputing all of the points we'll want to show. Now, rather than just looping through and displaying a single point at a time, we'll keep displaying more points each time we go through the loop. Note that we'll use 1:j to signify "the first j points in the vector."

```
% X floating through space, leaving a trail
t = 0:0.1:20;
x=t.*cos(t);  % don't forget dot operations
y=5*sin(t);
for j = 1:length(x)
    pause(0.05) % can come anywhere in the loop
    plot(x(1:j),y(1:j),x(j),y(j),'X')
    axis([min(x) max(x) min(y) max(y)])
end
```

Notice the complicated *plot* command we use. This will plot the first j points of the x and y vectors normally, and also plot just the j'th point as an X.

As you've come to expect in programming, there's yet another way to accomplish a similar goal in Matlab by plotting a single X at a time, but using *hold on* to keep all of the X's on screen:

```
% A trail of X's
for t=0:0.01:20
    x=t*cos(t);
    y=5*sin(t);
    pause(0.005)
    plot(x,y,'X')
    axis([-20 20 -5 5])
    hold on
end
```

2 User-Defined Functions

As you saw in our early lectures, you can type in something like *sin(0.32)*. What this does is calculate the sine of 0.32. To be more technical, *sin* is the name of a function. You pass it a single input value or *argument*, 0.32 in this case. It returns some value (output), 0.3146 in this case.

In Matlab, you're not just constrained to using so-called *built-in functions* such as *sin*. You can write your own, which we'll call *user-defined functions*.

2.1 Creating a User-Defined Function

To write your own function called *myFirstFunction*, you just need to create an m-file, with a few complications:

- You MUST save your code as an m-file, using the file name *myFirstFunction.m*. Of course, change the name to match the desired name of your function.
- The first line of this m-file needs to be as follows:
  ```
  function OUTPUT = NAME(INPUT)
  ```
- *NAME* should be replaced by the name of your function. In this example, it would be *myFirstFunction*
- *OUTPUT* should be replaced by the name of some variable that you'll use in your function. After Matlab runs all of the code in your function, it will look up the final value of your output variable and return that as the result of your function. Do you want to include more than one output? No problem! Create a vector of output variables, which means you should replace *OUTPUT* with something like: *[out1 out2 out3]*. Note that **unless the person calling the function explicitly requests multiple outputs, such as with [a b] = myFirstFunction(in), only the first output will be returned**.
- *INPUT* is a comma delimited list of the input variables. Let's say you typed *(in1,in2)* as your list of input variables (notice that these are in parentheses, and separated by a comma– they're not a vector). Then, if someone typed myFirstFunction(5,10) in Matlab to execute your function, *in1* would be set equal to 5, and *in2* would be set equal to 10.

- Following your first line, just type Matlab code that implements your function (performs any calculations you need). Don't forget that by the end of the last line of this code, your output variables need to have been set.
- If you want to create a function with no outputs, just put the empty vector *[ ]* in place of the outputs. For a function without inputs, you can simply have *NAME( )*.

2.2 Simple Functions

Let's say you wanted to create a function called *doubler*, which doubles some number x when you type *doubler(x)*. You can type the following code, which you MUST save as *doubler.m*

```
function y = doubler(x)
y = 2*x;
```

Now, once you've saved this file as *doubler.m*, you can type something like *doubler(22)* in the Matlab workspace or in other m-files, and your function will execute as if it were a built-in function. That statement would evaluate to 44.

Most of the time when you write a function, you'll want to have every (or almost every) line end with a semi-colon; otherwise, all of your intermediary calculations will be displayed on screen every time you call that function!

Now, let's see an example where we need to have more than 1 input and more than 1 output. This example will take two numbers and calculate both the sum and product.:

```
function [s p] = sumAndProduct(x,y)
s = x+y;
p = x*y;
```

Thus, if I typed sumAndProduct(5,3) in Matlab, I would get *ans = 8*. If I typed [a b] = sumAndProduct(5,3) in Matlab, I would get *a = 8 b = 15*. What if I wanted to always return both answers? In that case, only have one output variable, but let that output variable be a vector:

```
function [s] = sumAndProduct(x,y)
s(1) = x+y;
s(2) = x*y;
```

2.3 Help

Recall that when you type *help sin*, or the equivalent for any other function, instructions are displayed. For the functions you write, you can also create help functions! Immediately following the official "first line" of your function (*function output=name(input)*, create comments. These comments will be displayed when you type *help functionName*. Once you start writing Matlab code after those comments, any further comments will not be displayed i.e.

```
function [s p] = sumAndProduct(x,y)
% here is my help function
% this and the previous line are displayed
s = x+y;
% however, this line is not
p = x*y;
```

2.4 Determining the number of input or output arguments

Sometimes, you'll want to write a function that behaves differently depending on the number of input arguments (values passed to the function). Inside of a function's m-file, the variable *nargin* will contain the number of input arguments actually passed to the function. You can use this to customize a function based on that number, as follows:

```
function o = mult(x,y,z)
if(nargin==1)
    o=x;
elseif(nargin==2)
    o=x*y;
else
    o=x*y*z;
end
```

Note that in the first line, you need to give variable names for the greatest number of possible arguments. If you forget to use if statements containing nargin and instead try to refer to any of those undefined arguments, you'll get an error and Matlab will stop executing that function.

Similarly, the variable *nargout* contains the number of output arguments requested. For instance, if the user calls the max function by typing *a = max(M)*, nargout would be equal to 1 inside *max*. However, if the user instead calls the max function by typing *[a b] = max(M)*, nargout would be equal to 2 inside *max*. Therefore, you can use *nargout* to vary the output of your function based on the number of output elements requested. For example, the *size* function does this. (Compare the output when

2.5 Variable Scope

- What would happen if you used the same variable names in both a function your write and your Matlab workspace/other m files? Would this cause a problem? NO!
- Can you access variables you've defined in your workspace inside your function code, or vice versa? NO (except in cases we'll address momentarily).

The reason for these "NO" answers is the scope of the variables. The *scope* of a variable refers to the parts of your program (i.e. different m-files, the workspace) that can access that variable. In Matlab, the scope of variables used in functions is just that function itself. After you've run a function, the variables used in that function essentially disappear, and you generally can't access those variables from the Workspace. Similarly, variables from the Workspace or other m-files can't be accessed inside a function. Thus, if you need some particular value inside a function, you need to include that value within the function's input arguments.

Recall that m-files that aren't functions can indeed access variables from the workspace, and any variables created in the m-file will be accessible back in the workspace. You can think of m-files that aren't function definitions as equivalent to just typing those lines into the workspace.

2.5.1 Global Variables

Of course, the full truth is slightly more complicated. You can create what are called *global variables*. These can be accessed (and changed) from multiple locations. Creating a global variable in Matlab is a bit more complicated than creating global variables in other languages. Everywhere you lpan to use this global variable, you must include the line *global x*, where x is the name of the variable. Assuming that you've included that line in each relevant location (every relevant m-file, the workspace, etc.), you can use x in any of those locations, and any changes made to x will affect every single one of those locations.

3 Anonymous Functions

Sometimes, you'll want to write a one-line function that you'll need for a few minutes, but you won't want to go through the trouble of creating that function in a separate m-file. In these cases, you can use what is called an anonymous function. *Anonymous Functions* let you quickly, in the middle of a Matlab program (or in the workspace), create a one-line function. Here's the syntax:

```
NAME = @(INPUT)  STATEMENT
```

You'll notice that there's no output listed. Instead, the result of executing the one STATEMENT listed is returned as the output. Also, note that the NAME of the function is technically a "function handle." We'll learn

about function handles later in this lecture.

As an example anonymous function, let's say you had to calculate a number of logarithms base 8 in the middle of Matlab code you're writing. To define a function *log8* as the logarithm base 8, you could write the following anonymous function anywhere in your code (prior to using the log8 function):

```
log8 = @(x)  log(x)/log(8)
```

From then on, you can use *log8()* in your code.

4 Subfunctions

Inside a function's m-file, below the function itself, you can include complete definitions for other functions. These are called *subfunctions* and are ONLY available to the main function. Here's an example where we create a subfunction:

In the following example, I create a function that returns, true or false, whether a number is a palindrome. Inside my *ispalindrome* function, I create a subfunction called *reverseit*. I'd save this whole file as *ispalindrome.m*. In my Matlab command window, I could certainly type *ispalindrome(1551)* and get the answer 1, or *ispalindrome(1552)* and get the answer 0. However, if I tried to type *reverseit('1501')* in my command window, I'd get an error about *reverseit* being an undefined function or variable. Why? Subfunctions aren't available to anything but their calling function.

```
function isit = ispalindrome(number)
%%% this is the main function
numberAsString = num2str(number);
isit=0;
if(strcmp(numberAsString,reverseit(numberAsString)))
     isit=1;
end

function xr = reverseit(x)
%%% this is a subfunction that reverses a vector/string
len = length(x);
xr = char(size(x)); % creates empty string
for z = 1:len
    xr(z) = x(len+1-z);
end
```

Note that I used a function called *num2str* in this example. *num2str* takes a number as its input and returns that number as a string. This conversion is very useful since a string is a vector of single characters; in essence, we are taking a number and returning a vector of its digits, where each digit is a single number (as a string). If you'd like to get a vector of each digit of a number N as numbers, note that the following method works for splitting an integer into a vector of its digits. (The -48 is a result of the ASCII chart for storing numbers, which we'll learn about later in the class. We'll similarly learn about *uint8* later in the course):

```
if(fix(x)==x)
     xdigits = uint8(num2str(x)) - 48
end
```

5 Feval and Function Handles

Interestingly, there's a function whose sole purpose is to call other functions. This function is called *feval* (function evaluate). The first input argument is the handle to the function. A function handle is actually a data type (in contrast to a character or an integer) that allows you to call a function indirectly. If you've already defined a function *userDefined* in its own m-file, or if it's built-in, the way to get a function handle is be preceding the name of the function by the ampersand:

```
feval(@sin,pi/2)
```

However, if you create an anonymous function, the variable storing that function is already a function handle, so you **cannot** include an ampersand:

```
divideby2 = @(x) x/2;
feval(divideby2,5)
```

6 Error Checking

As you write your own functions, it's good practice to make sure that your functions function as intended even for 'bad inputs.' For instance, if we return to the *doubler* function that we created at the beginning of this lecture and try *doubler('cow')*, we're surprised to see that it seems to return a vector of 3 numbers, although (until we learn the ASCII chart later in the course), these numbers seem kind of random.

Instead of allowing our function to return an answer that makes no sense, we should instead return an error message. The *error* command in Matlab lets us define our own error messages, halting the program upon execution:

```
function y = doubler(x)
if(isnumeric(x))  % isnumeric returns 1 only if x is a number
     y = 2*x;
else
     error('x is not numeric.  doubler(x) only works when x is a number')
end
```

7 Set Functions

There exist a number of functions that are very useful if you're using Matlab vectors to represent sets: *unique, intersect, union,* and *setdiff.*

unique(S) returns one of each unique element of S, sorted in ascending order. In other words, it removes all duplicate elements in the set S. *intersect(S1,S2)* returns all elements, sorted in ascending order, that are in BOTH S1 and S2. In contrast, *union(S1,S2)* returns all elements, sorted in ascending order, that are in EITHER S1 or S2. Finally, *setdiff(S1,S2)* returns all elements of S1, sorted in ascending order, that are NOT in S2. Of course, note that the order of the inputs to *setdiff* is very important since *setdiff(S1,S2)* is not generally equal to *setdiff(S2,S1).*

```
>> A = [10 54 5 12 13 12 8];
>> B = [21 46 12 19 8 4 4 2];

>> unique(A)
ans =     5     8    10    12    13    54

>> intersect(A,B)
ans =     8    12

>> union(A,B)
ans =     2     4     5     8    10    12    13    19    21    46    54

>> setdiff(A,B)
ans =     5    10    13    54

>> setdiff(B,A)
ans =     2     4    19    21    46
```

8 Example Functions

Here are some example functions. Many of them involve loops, so that you can get a bit more practice with *for and while loops*:

8.1 1: Temperature Converter

We'll first write a function that, given as input a temperature in Celsius, returns that temperature in Fahrenheit.

```
function f = CelsiusToFahrenheit(c)
f = 9/5*c+32;
```

Remember to save this file as *CelsiusToFahrenheit.m* or else the function won't work. Also note that you would call this function by typing something like *CelsiusToFahrenheit(7)*.

8.2 2: Temperature Converter, Anonymously

Since the previous example was a one line function, we could have written it as an anonymous function in the middle of another program:

```
CelsiusToFahrenheit = @(c) 9/5*c+32;
```

8.3 3: Temperature Converter Expanded

Now let's write a slightly more advanced function that accepts input in either Celsius or Fahrenheit (which you must specify), and then returns the result converted to the other system.

```
function converted = TempConverter(temp,system)
% Converts Fahrenheit to Celsius, or vice versa
% TempConverter accepts two arguments
%      temperature, and system ('F' or 'C')
% TempConverter(15,'C') converts 15 C to Fahrenheit
% TempConverter(32,'F') converts 32 F to Celsius
converted=-inf;
if(system=='F')  %input fahrenheit
    converted = (temp-32)*5/9;
elseif(system=='C')   %input celsius
    converted = 9/5*temp+32;
end
```

Remember to save this file as *TempConverter.m* or else the function won't work. Note that you would call this function by typing something like *TempConverter(25,'C')*. Additionally, if you type *help TempConverter*, the big block of comments following the first line of the function will be displayed.

8.4 4: MyUnique

To keep with the theme of today's lecture, let's write our own version of the *unique* function from the previous section using loops:

```
% method one
function onlyunique = MyUnique1(V)
onlyunique = [];
for x = 1:length(V)
     if(~sum(onlyunique==V(x)))
         onlyunique(end+1) = V(x);
     end
end
onlyunique = sort(onlyunique)
```

```
% method two
function Vunique = MyUnique2(V)
repeats = zeros(1,length(V));
for x = 1:length(V)
     if(sum(V(1:(x-1)) == V(x)))
         repeats(x) = 1;
     end
end
Vunique = sort(V(repeats==0))
```

8.5 5: MyUnion

In Matlab, the *union* function returns the union of two sets: all of the values in each, without repetitions. Here's our function that does the same:

```
function [u] = myunion(a,b)
u = unique(a);
b = unique(b);
for j = 1:length(b)
    if(sum(u==b(j))==0)
        u(end+1) = b(j);
    end
end
u = sort(u);
```

8.6 6: MyIntersect

In Matlab, the *intersect* function returns the intersection of two sets: all of the unique values that are in both sets.

```
function [inter] = myintersect(a,b)
a = unique(a);
b = unique(b);
inter = [];
for j = 1:length(a)
    if(sum(b==a(j)))  % if a(j) is in b
        inter(end+1) = a(j);
    end
end
```

8.7 7: MySetDiff

In Matlab, *setdiff(A,B)* returns the set of unique elements that are in A, but not in B. Here's our function:

```
function [lonely] = mysetdiff(a,b)
a = unique(a);
b = unique(b);
lonely = [];
for j = 1:length(a)
    if(sum(b==a(j))==0)  % if a(j) is not in b
        lonely(end+1) = a(j);
    end
end
```

8.8 8: IsPalindrome

A palindrome is a word, phrase, or number that is the same forwards and backwards. Let's write a function that tests whether an input is a palindrome. Note that we'll use some functions you haven't seen before; look them up if their purpose is not obvious from the name. Also recall that *num2str* converts a number to a string, which lets us use fliplr to reverse it. fliplr(53) is 53, but fliplr('53') is '35'.

```
function isit = ispalindrome(X)
% ispalindrome(X) returns 1 if X is a palindrome, 0 otherwise
% X can be a number, string, or string containing spaces.
if(isnumeric(X))  % X is a number
    isit = strcmp(num2str(X),fliplr(num2str(X)));
elseif(ischar(X))  % X is a string
    X = lower(X);  % convert to lower case
    X(X==' ') = [];  % removes spaces.  Spaces become the empty vector
    isit = strcmp(X,fliplr(X));
end
```

8.9 9: Largest Prime Factor

Let's create a function that returns the largest prime factor of its input, X.

```
function [f] = largestfactor(X)
f = 2;
while(X>1)
   if(rem(X,f)==0)  % if we can divide evenly
       X = X/f;   % divide out that factor
   else
       f = f+1;  % let's move on to the next number
                 % note that after 2, we could only look at odd numbers
   end
end
```

What if we wanted to return a vector of all the factors if they request a second output? Just edit the code above:

```
function [f thelist] = largestfactor(X)
f = 2;
list = [];
while(X>1)
   if(rem(C,f)==0)  % if we can divide evenly
       C = C/f;   % divide out that factor
       list(end+1) = f;  % add that factor to the end of the list
   else
       f = f+1;  % let's move on to the next number
                 % note that after 2, we could only look at odd numbers
   end
end
```

What if the user tried to factor a negative number, or a vector of numbers, or a non integer? Let's begin error checking:

```
function [f thelist] = largestfactor(X)
% LARGESTFACTOR  Largest prime factors, all prime factors
% [a] = largestfactor(X) returns the largest prime factor of X
% [a b] = largestfactor(X) returns a = the largest prime factor of X
%                              and b = a vector of all prime factors of X
if(length(X)>1)
    error('Error.  Input X must be scalar.');
elseif(X<=0)
    error('Error.  Input X must be positive');
elseif(X~=fix(X))
    error('Error.  Input X must be an integer');
end
f = 2;
list = [];
while(X>1)
   if(rem(X,f)==0)
       X = X/f;
       list(end+1) = f;
   else
       f = f+1;
   end
end
```

8.9.1 10: Max Function that Finds Locations

Now, let's create our own max function that will return the maximum value if called using a command like *a = ourmax(M)*, but can also return the location if called using *[c d] = ourmax(M)*

```
function [maxval loc] = ourmax(M)
z = size(M);
if(z(1)==1) % convert row vectors to column vectors
            % we can now treat them like matrices
            % and use the same process for both vectors and matrices
   M=M';
   z = size(M);
end
if(length(M)==0)  % if empty vector
    maxval=[ ];
    loc = [ ];
else
    for c=1:z(2)  % calculate max in each column
        biggest=-inf;
        for r=1:z(1)
            if(M(r,c)>biggest) % found bigger
                biggest=M(r,c);
                temploc=r; % temporarily store row of max
            end
        end
        maxval(c)=biggest; % max in each column
        loc(c)=temploc; % which row that max is in
    end
end
```