

**WEEK 11**



**WRITING YOUR OWN  
m-FILES**

# WHAT IS THE PROBLEM?



- There are an unlimited number of applications that can be implemented with MATLAB
- MATLAB's own functions, as rich as they are, will never be able to cover all such cases
- What is needed is the ability to write your own functions and call them up as if they came with MATLAB

# SCRIPTS AND FUNCTIONS



- To perform this task, MATLAB provides two possibilities
  - *scripts*
  - *functions*
- *scripts* automate long sequences of commands
- *functions* implement user-specific algorithms


# SCRIPTS

- We want to plot *sombrero* by a single command
- Open up a new m-file and type
  - `[X,Y]=meshgrid(-8:.5:8)`
  - `R=sqrt(X.^2+Y.^2);`
  - `Z=sin(R)./R;`
  - `surf(X,Y,Z);grid`
- Save the file under *sombrero.m*
- Just type *sombrero* and watch

# FUNCTIONS

- Functions are more flexible than scripts.
- Functions can be customized by accepting user defined arguments
- Variables inside *functions* are entirely local and do not appear in your workspace

# SYNTAX FOR FUNCTIONS

- 
- A .m file defining a *function* has the following make up
    - The first line begins with *function*
    - Commented lines at the top of the file are typed back in response to the *help* command issued by the user
    - No *end* statement
  - When saved as a.m file, *function* takes on the name of the call statement. For example if inside an m-file you type
    - `function m=biggest(x)`
    - Saving the m file will automatically name it `biggest.m`

# INPUT/OUTPUT ARGUMENTS



- A function may have any number of input or output arguments. It may also have no input and no output arguments
- Examples:
  - `[mean,variance]=function stat(x);%find mean and standard deviation of data stored in x. This is an example of one input and two output arguments`
  - `m=average(x); input x, output m`

# GENERATING A RECTANGULAR PULSE

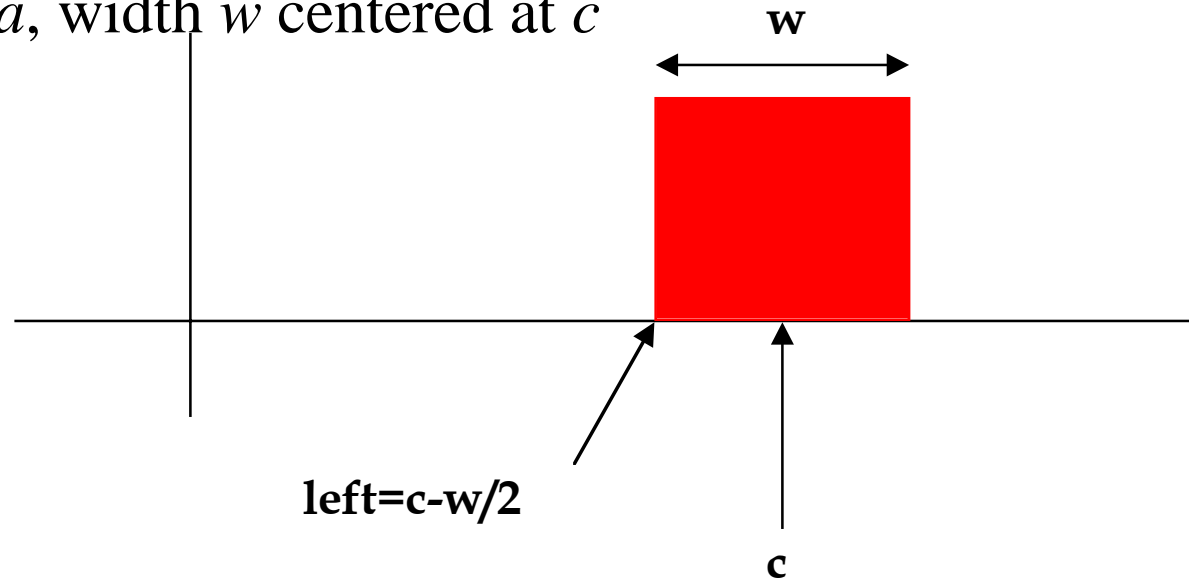


- Let's say we want to generate a rectangular pulse of unit height and width 10
- Here is how the function will look like
  - function `p=rect`
  - %generate a square pulse
  - `p=[zeros(1,10),ones(1,10),zeros(1,10)];`
- Now, save this function and do the following
- In the command window type
  - `x=rect;`
  - `stem(x);grid`

# CONTROLLING HEIGHT, WIDTH AND POSITION



- This time we want to generate a specific pulse of amplitude  $a$ , width  $w$  centered at  $c$



# PASSING ON PARAMETERS



- The revised function now looks like
  - `function p=rect(a,w,c)`
  - `%amplitude(a), width(w) and center(c)`
  - `left=c-w/2;`
  - `right=left;`
  - `p=[zeros(1,left),a*ones(1,w),zeros(1,right)];`
- Call this function with a command like
  - `x=rect(2,10,20);`
  - `stem(x)`

# ECHO, INPUT, KEYBOARD AND PAUSE



- *echo* types back commands as an m-file is run
- *input* takes input from keyboard
  - *n=input('enter number of iterations')*
- *keyboard* suspends execution and returns control to the command line
- *pause* stops execution until a key is pressed

# ECHO

- In case you want to see the commands as they are being executed inside a script, use echo as follows

```
echo on;%turns on echo
```

```
[x,y]=meshgrid(-5:1:5);
```

```
r=sqrt(x.^2+y.^2);
```

```
z=sin(r)./r;
```

```
echo off;%turns echo off from here on.
```

```
mesh(x,y,z)
```

# INPUT

- Sometimes the program needs user defined input at runtime

```
range=input('enter range for x and y')
```

```
[x,y]=meshgrid(-range:1:range);
```

```
r=sqrt(x.^2+y.^2);
```

```
z=sin(r)./r;
```

```
mesh(x,y,z)
```

# KEYBOARD



- When placed in an m-file, *keyboard* suspends execution and returns control to the user. Execution resumes when return is typed (you actually have to type r-e-t-u-r-n). Keyboard is used for debugging.


# PAUSE



- Placement of *pause* anywhere in an m-file suspends execution. This is a useful tool to see how far the program goes before crashing.
- *pause* is different from *keyboard*. *pause* does not return control to the user
- Pressing return key will resume execution

# ENFORCING CONDITIONS USING

*if*

- 
- *if* in MATLAB, like other programming languages, tests a condition. There is a difference, however. This is the way *if* works in MATLAB

```
if expression
statements
else
statements
end
```


- Here is how it works: if *expression* is all nonzero, the following statements execute. Expression following *if* is a logical statement consisting of 0's and 1's

# EXAMPLE USING *if*



- Want to compare two arrays. If any of their entries are equal, a flag must be set
  - $a=[1\ 3\ 6\ 8]$ ;  $\longrightarrow$   $a = b$  is  $[0\ 1\ 0\ 0]$
  - $b=[2\ 3\ 8\ 9]$ ;
  - if  $a = b$
  - $c=1$ ;
  - else
  - $c=0$
  - end
  - else

# TEXT STRINGS

- 
- Text strings are entered into MATLAB surrounded by single quotes. This a useful feature when users want to pass on alphanumeric options to a function.
  - Strings are generated by enclosing them inside quotations
    - `s='Hello'`
  - If you type `s`, you get `s=Hello`
  - `s` is now a 1x5 vector

# STRING COMPARISON



- The command *strcmp* compares a user specified string with a number of expected options
- *strcmp*(s1,s2) returns 1 if strings s1 and s2 are the same. Returns a zero if they are not.
- For example,
  - $a = \text{strcmp}(\text{'john'}, \text{'john'}) \longrightarrow a = 1$
  - $a = \text{strcmp}(\text{'john'}, \text{'jill'}) \longrightarrow a = 0$

## USING *if* WITH *strcmp*

- *strcmp* returns a zero or one so it is a perfect match for use with *if*. In the following, *name* field is compared with *john* or *jill* and appropriate action is taken. Remember, *strcmp* returns a 1 or a zero depending on whether a match is detected

- *if strcmp(name,'john')*

do this

- *elseif strcmp(name,'jill')*

do something else

- *end*

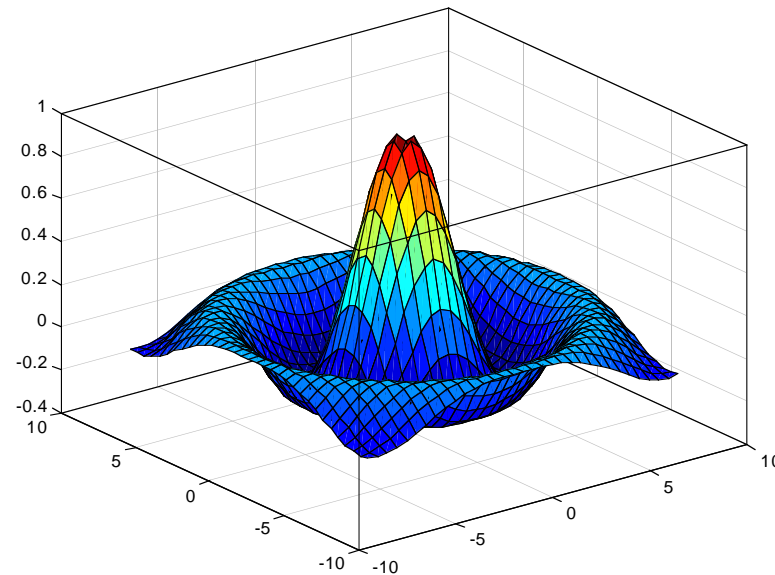
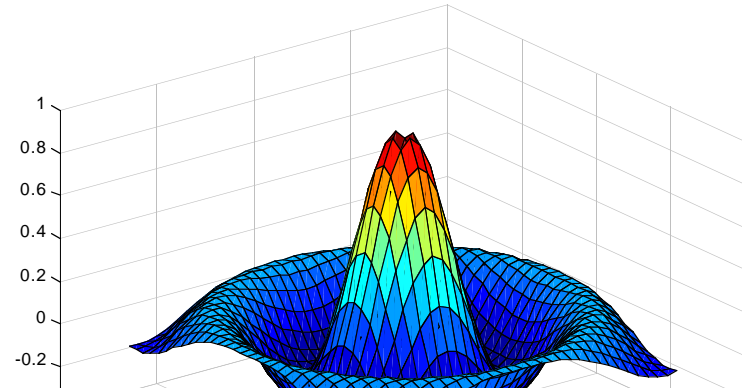
# ENCLOSING A PLOT IN A BOX



- Want to write a function to plot  $z = \sin(\sqrt{x^2 + y^2})$  over the  $(-4:2:4)$  range. Name the function *jello.m*.
- *jello* should accept an input string *box* in the form of *jello(box)*.
- usage
  - **box='on'**
  - **jello(box);%** encloses the entire plot inside cube
- If *box* is set to off, then no enclosure should appear

# ACTUAL CODE

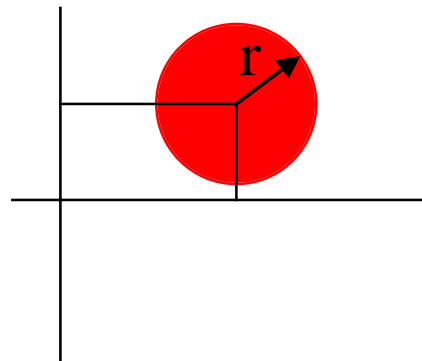
```
■ function z=jello(box);  
■ %plot Jello over [-4 4] range  
■ [x,y]=meshgrid(-4:.2:4);  
■ r=sqrt(x.^2+y.^2);  
■ z=sin(r);  
■ if strcmp(box,'on')  
■ surf(x,y,z);grid  
■ set(gca,'box','on')  
■ else  
■ surf(x,y,z);grid  
■ set(gca,'box','off')  
■ end
```



# HANDS-ON PRACTICE-1



- Write a function that draws a circle with specified radius at specified location on the x-y plane. Example usage is
  - `circle(r, c)`
- $r$  is a scalar, indicating the radius of the circle.  $c$  is a two element array indicating the  $(x,y)$  position of the center of the circle



# HANDS-ON PRACTICE-2



- Do the previous problem with this added feature.
  - `circle(r,c,linecolor)`
  - `linecolor` is a string specified by the user from among the following options: `black`, `red` and `blue`. Circle should then be plotted with the specified color
  - Example call
  - `circle(.5,[1 2], 'red')`
- Since there are more than two options, you need to use `ELSEIF` instead of `ELSE`. Use help on *if* to see its use

# HOMework



- Write a function *cap* that limits any function to a specified percentage of its maximum height and plots it with a user specified colormap stored in the string *usermap*
- For example, if I have my 3D function stored in *z*, then
  - $z_{\text{capped}} = \text{cap}(z, 0.6, \text{usermap})$
  - cuts off the function to 60% of its peak value then plots it with *usermap* colors. Valid usermaps are *hot*, *gray* and *hsv*. Make sure to check your work