Introduction to MATLAB Programming

Prof. Alberto Paccanaro
McCrea Room 120
http://www.paccanarolab.org

Material in these slides follows the content in Chapter 3 of:
Matlab: a practical introduction to programming and problem solving
by Stormy Attaway, 3rd edition, Elsevier Inc.

Slides adapted from Matlab, S. Attaway, 3rd Ed., Elsevier Inc.

Roadmap

- Scripts
  - Documentation
  - Input
  - Output
  - Simple plots
  - File I/O

- Functions
  - Input & output arguments
  - Scope

Scripts

- Scripts are files in MATLAB that contain a sequence of MATLAB instructions, implementing an algorithm
- Scripts are interpreted, and are stored in M-files (files with the extension .m)
- To create a script, click on “New Script” under the HOME tab; this opens the Editor
- Once a script has been created and saved, it is executed by entering its name at the prompt
- the **type** command can be used to display a script in the Command Window

Documentation

- Scripts should always be ***documented*** using ***comments***
- Comments are used to describe what the script does, and how it accomplishes its task
- Comments are ignored by MATLAB
- Comments are anything from a % to the end of that line
- In particular, the first comment line in a script is called the "H1 line"; it is what is displayed with **help**

Input

- The **input** function does two things: ***prompts*** the user, and reads in a value
- General form for reading in a number:

```
variablename = input('prompt string')
```

- General form for reading a character or string:

```
variablename = input('prompt string', 's')
```

- Must have separate **input** functions for every value to be read in

Output

- There are two basic output functions:
  - **disp**, which is a quick way to display things
  - **fprintf**, which allows formatting

- The **fprintf** function uses ***format strings*** which include ***place holders***; these have ***conversion characters***:

  %d integers

  %f floats (real numbers)

  %c single characters

  %s strings

- **%#.#x** where # is an integer and x is the conversion character, specifies a field width and the number of decimal places

- **%.#x** where # is an integer and x is the conversion character, specifies just the number of decimal places (or characters in a string)

- Example:

```
fprintf('The first element in the array is %3.4f, the string is …
                   %s. ', a(1,1), str)
```

- **Check the documentation for fprintf for details on formatting outputs**

Formatting Output

- Other formatting:

  \n newline character

  \t tab character

  left justify with '**-**' e.g. %-5d

  to print one slash: ****

  to print one single quote: **' '** (two single quotes)

- Printing vectors and matrices: usually easier with **disp**

Script with I/O Example

- The Target Heart Rate (THR) for a relatively active person is given by

  THR = (220-A) * 0.6   where A is the person's age in years

- We want a script that will prompt for the age, then calculate and print the THR. Executing the script would look like this:

```
>> thrscript
Please enter your age in years: 33
For a person 33 years old,
the target heart rate is 112.2.
>>
```

Example Solution

thrscript.m

```
% Calculates a person's target heart rate

age = input('Please enter your age in years: ');
thr = (220-age) * 0.6;
fprintf('For a person %d years old,\n', age)
fprintf('the target heart rate is %.1f \n', thr)
```

Note that the output is suppressed from both assignment statements. The format of the output is controlled by the **fprintf** statements.

Simple Plots

- Simple plots of data points can be created using **plot**
- To start, create variables to store the data (can store one or more point but must be the same length);

  plot(x,y)   or   plot(y)

  (if x is to be 1,2,3,etc. it can be omitted)
- The default is that the individual points are plotted with straight line segments between them, but other options can be specified in an additional argument which is a string
- options can include color (e.g. 'b' for blue, 'g' for green, 'k' for black, 'r' for red, etc.)
- can include ***plot symbols*** or ***markers*** (e.g. 'o' for circle, '+', '*')
- can also include ***line types*** (e.g. '--' for dashed)
- For example, **plot(x,y, 'g*--')**

Labeling the Plot

- By default, there are no labels on the axes or title on the plot
- Pass the desired strings to these functions:
  - **xlabel('string')**
  - **ylabel('string')**
  - **title('string')**

- The axes are created by default by using the minimum and maximum values in the x and y data vectors. To specify different ranges for the axes, use the **axis** function:
  - **axis([xmin xmax ymin ymax])**

Other Plot Functions

- **clf** clears the figure window
- **figure** creates a new figure window (can # e.g. figure(2))
- **hold** is a toggle; keeps the current graph in the figure window
- **legend** displays strings in a legend
- **grid** displays grid lines

File I/O: load and save

- There are 3 modes or operations on files:
  - read from
  - write to (assumes from the beginning)
  - append to (writing to, but starting at the end)

- There are simple file I/O commands for saving a matrix to a file and also reading from a file into a matrix: **save** and **load**

- If what is desired is to read or write something other than a matrix, lower level file I/O functions must be used (later class)

load and **save**

- To write the contents of a matrix variable to a file:

  save ‘filename’ matrixvariablename

  save ‘filename’ matrixvariablename –ascii

  save ‘filename’ matrixvariablename –ascii –append

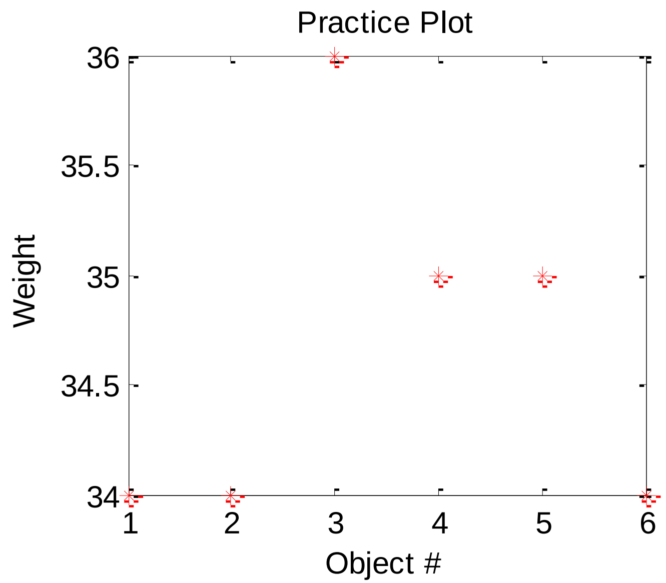
- To read from a file into a matrix variable:

  load ‘filename.ext’

  - Note: this will create a matrix variable named “filename” (same as the name of the file but not including the extension on the file name)
  - This can only be used if the file has the same number of values on every line in the file; every line is read into a row in the matrix variable

Example using **load** and **plot**

- A file ‘objweights.dat’ stores weights of some objects all in one line, e.g.   33.5 34.42 35.9   35.1 34.99 34
- We want a script that will read from this file, round the weights, and plot the rounded weights with red *’s:



Example Solution

Note that **load** creates a row vector variable named *objweights*

```
load ‘objweights.dat’
y = round(objweights);
x = 1:length(y);  % Not necessary
plot(x,y, 'r*')
xlabel('Object #')
ylabel('Weight')
title('Practice Plot')
```

User-Defined Functions

User-Defined Functions are functions that you write

- There are several kinds; for now we will focus on the kind of function that calculates and returns one value
- You write what is called the function definition (which is saved in an M-file)
- Then, using the function works just like using a built-in function:
  - you *call* it by giving the function name and passing *argument(s)* to it in parentheses
  - that sends *control* to the function which uses the argument(s) to calculate the result
  - which is then *returned*

General Form of Function Definition

- The function definition would be in a file fnname.m:

```
function outarg = fnname(input arguments)
% Block comment
Statements here; eventually:
outarg = some value;
end
```

- The definition includes:
  - the function header (the first line)
  - the function body (everything else)

Function header

- The header of the function includes several things:

  function outarg = fnname(input arguments)

- The header always starts with the reserved word “function”
- Next is the name of an output argument, followed by the assignment operator
- The function name “fnname” should be the same as the name of the m-file in which this is stored
- The input arguments correspond one-to-one with the values that are passed to the function when called
- **Variables are *local* to the function**

Function Example

- For example, a function that *calculates and returns the area of a circle*
  - There would be one input argument: the radius
  - There would be one output argument: the area
  - In an M-file called calcarea.m:

```
function area = calcarea(rad)
% This function calculates the area of a circle
area = pi * rad * rad;
end
```

- Function name same as the M-file name
- Putting a value in the output argument is how the function returns the value; in this case, with an assignment statement

Calling the Function

This function could be called in several ways:

```
>> calcarea(4)
```

This would store the result in the default variable ans

```
>> myarea = calcarea(9)
```

This would store the result in the variable *myarea*

```
>> disp(calcarea(5))
```

This would display the result, but it would not be stored for later use

Passing arrays to functions

- Because the * operator was used instead of .*,

  area = pi * rad * rad;

  arrays could not be passed to this function as it is

- To fix that, change to the array multiplication operator .*

```
function area = calcarea(rad)
% This function calculates the area of a circle
area = pi * rad .* rad;
end
```

- Now a vector of radii could be passed to the input argument *rad*

General Form of Simple Program

fn.m

```
function out = fn(in)
out = value based on in;
end
```

script.m

- Get input
- Call fn to calculate result
- Print result

Example Program

- The volume of a hollow sphere is given by

  $4/3\ \pi\ (R_o^{\ 3} - R_i^{\ 3})$

  where R_o is the outer radius and R_i is the inner radius

- We want a script that will prompt the user for the radii, call a function that will calculate the volume, and print the result.

Example Solution

```
% This script calculates the volume of a hollow sphere

inner = input('Enter the inner radius: ');
outer = input('Enter the outer radius: ');

volume = vol_hol_sphere(inner, outer);

fprintf('The volume is %.2f\n', volume)
```

vol_hol_sphere.m

```
function hollvol = vol_hol_sphere(inner, outer)

% Calculates the volume of a hollow sphere

hollvol = 4/3 * pi * (outer^3 - inner^3);

end
```

Introduction to scope

- The scope of variables is where they are valid
- The Command Window uses a workspace called the base workspace
- Scripts also use the base workspace
- This means that variables created in the Command Window can be used in a script and vice versa
- Functions have their own workspaces – so local variables in functions, input arguments, and output arguments only exist while the function is executing