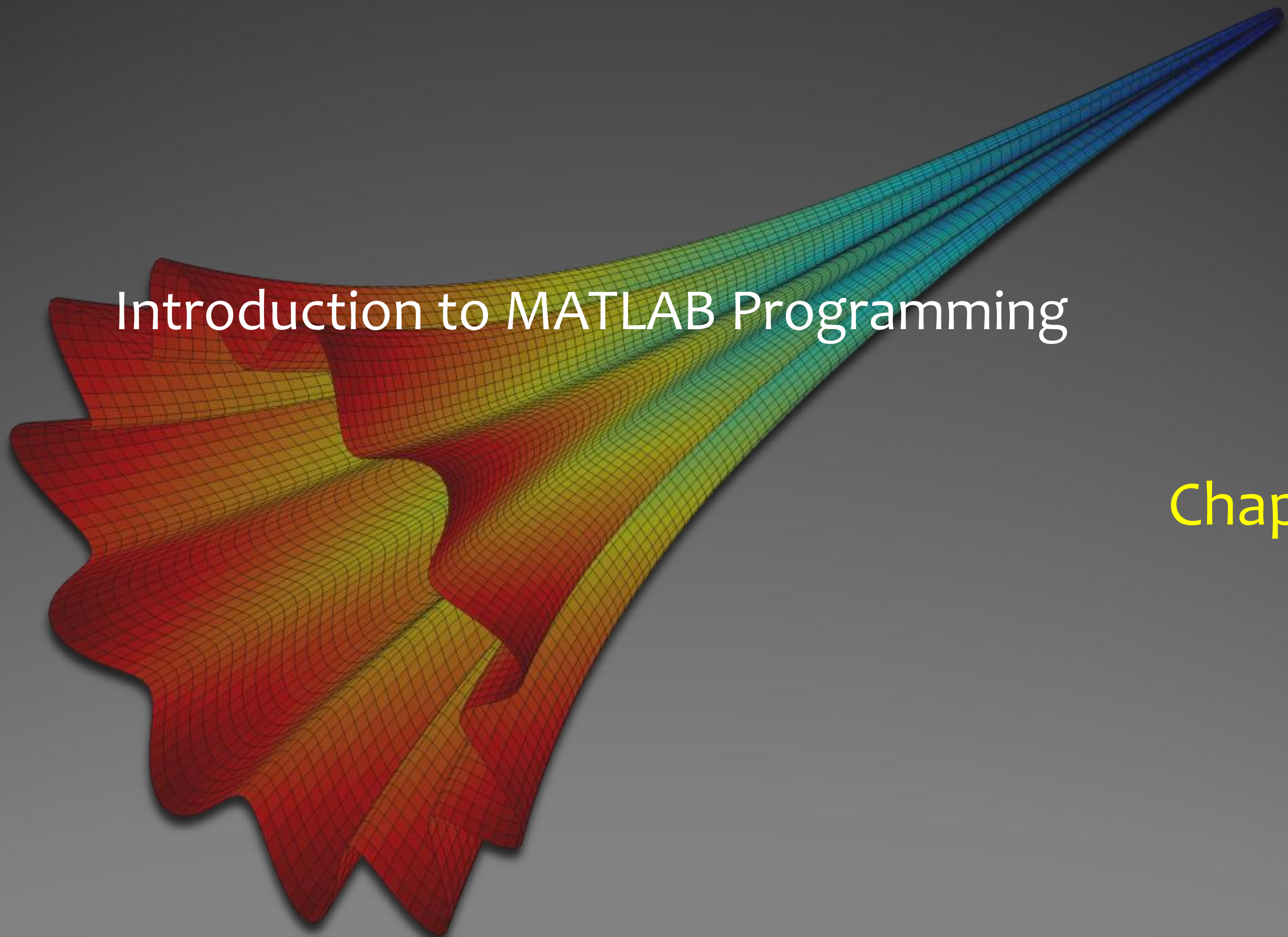
Introduction to MATLAB Programming



Chapter 3

Algorithms

- An ***algorithm*** is the sequence of steps needed to solve a problem
- ***Top-down design*** approach to programming: break a solution into steps, then further refine each one
- Generic algorithm for many programs:
  1. Get the input
  2. Calculate result(s)
  3. Display the result(s)
- A ***modular*** program would consist of functions that implement each step

Algorithms and Control Structures

There are three categories of algorithmic operations:

Sequential operations: Instructions executed in order.

Conditional operations: Control structures that first ask a question to be answered with a true/false answer and then select the next instruction based on the answer.

Iterative operations (loops): Control structures that repeat the execution of a block of instructions.

Advantages of structured programming

1. Structured programs are easier to write because the programmer can study the overall problem first and then deal with the details later.

2. Modules (functions) written for one application can be used for other applications (this is called *reusable code*).

3. Structured programs are easier to debug because each module is designed to perform just one task and thus it can be tested separately from the other modules.

Advantages of structured programming (continued)

4. Structured programming is effective in a teamwork environment because several people can work on a common program, each person developing one or more modules.

5. Structured programs are easier to understand and modify, especially if meaningful names are chosen for the modules and if the documentation clearly identifies the module's task.

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; longer comment blocks are contained in between %{ and %}
- In particular, the first comment line in a script is called the "H1 line"; it is what is displayed with **help**
- Proper selection of variable names to reflect the quantities they represent.

Steps for developing a computer solution:

1. State the problem concisely.
2. Specify the data to be used by the program. This is the “input.”
3. Specify the information to be generated by the program. This is the “output.”
4. Work through the solution steps by hand or with a calculator; use a simpler set of data if necessary.

Steps for developing a computer solution (continued)

5. Write and run the program.

6. Check the output of the program with your hand solution.

7. Run the program with your input data and perform a reality check on the output.

8. If you will use the program as a general tool in the future, test it by running it for a range of reasonable data values; perform a reality check on the results.

Finding Bugs

Debugging a program is the process of finding and removing the “bugs,” or errors, in a program. Such errors usually fall into one of the following categories.

1. Syntax errors such as omitting a parenthesis or comma, or spelling a command name incorrectly. MATLAB usually detects the more obvious errors and displays a message describing the error and its location.

2. Errors due to an incorrect mathematical procedure. These are called *runtime errors*. They do not necessarily occur every time the program is executed; their occurrence often depends on the particular input data. A common example is division by zero.

To locate a runtime error, try the following:

1. Always test your program with a simple version of the problem, whose answers can be checked by hand calculations.

2. Display any intermediate calculations by removing semicolons at the end of statements.

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
- Use %#x where # is an integer and x is the conversion character to specify the ***field width*** of #
- %#.#x specifies a field width and the number of decimal places
- %.#x specifies just the number of decimal places (or characters in a string); the field width will be expanded as necessary

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**

Examples of **fprintf**

- Expressions after the format string fill in for the place holders, in sequence

```
>> fprintf('The numbers are %4d and %.1f\n', 3, 24.59)
The numbers are    3 and 24.6
```

- It is not the case that every **fprintf** statement prints a separate line; lines are controlled by printing \n; e.g. from a script:

```
fprintf('Hello and')
fprintf(' how \n\n are you?\n')
```

- would print:

```
Hello and how

 are you?
>>
```

Scripts with I/O

- Although input and output functions are valid in the Command Window, they make most sense in scripts (and/or functions)
- General outline of a script with I/O:
  1. Prompt the user for the input (suppress the output with ;)
  2. Calculate values based on the input (suppress the output)
  3. Print everything in a formatted way using **fprintf** (Normally, print both the input and the calculated values)
- Use semicolons throughout so that you control exactly what the execution of the script looks like

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); vectors named x and y would be common – or, if x is to be 1,2,3,etc. it can be omitted

  plot(x,y) or just plot(y)
- 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 greeen, '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
- **bar** bar chart
- Note: make sure to use enough points to get a “smooth” graph

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 (covered in Chapter 9)

load and **save**

- To read from a file into a matrix variable:

    load filename.ext

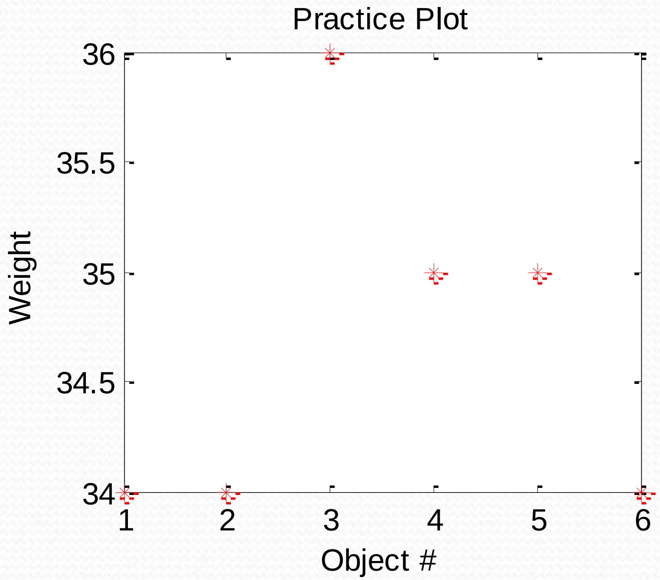
    - Note: this will create a matrix variable named “filename” (same as the name of the file but not including the extension on the file name)
    - This can only be used if the file has the same number of values on every line in the file; every line is read into a row in the matrix variable
- To write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

    save filename matrixvariablename –ascii
- To append the contents of a matrix variable to an existing file:

    save filename matrixvariablename –ascii -append

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

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 (Note: suppress the output)
- The names of the input and output arguments follow the same rules as variables, and should be mnemonic

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*
  - A variable with the same name as the output argument could also be used
- >> 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*

Notes

- You can pass multiple input arguments to a function
- Variables that are used within a function (for example, for intermediate calculations) are called *local variables*

MATLAB Programs

- Note: a function that returns a value does NOT normally also print the value

- A function can be called from a script

- This combination of a script (stored in an M-file) and the function(s) (also stored in M-files) that it calls is a *program*

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.
- Also, we will write the function!

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 (this is a bad idea, however)
- Functions have their own workspaces – so local variables in functions, input arguments, and output arguments only exist while the function is executing

Commands and Functions

- Commands (such as format, type, load, save) are shortcut versions of function calls
- The command form can be used if all of the arguments that are passed to the function are strings, and the function is not returning any values.
- So,

    fnname   string

- and

    fnname('string')

- are equivalent

Common Pitfalls

- Spelling a variable name different ways in different places in a script or function.
- Forgetting to add the second ‘s’ argument to the **input** function when character input is desired.
- Not using the correct conversion character when printing.
- Confusing **fprintf** and **disp.** Remember that only **fprintf** can format.
- Not realizing that **load** will create a variable with the same name as the file.

Programming Style Guidelines

- Use comments to document scripts and functions
- Use mnemonic identifier names (names that make sense, e.g. *radius* instead of *xyz*) for variable names and for file names
- Put a newline character at the end of every string printed by **fprintf** so that the next output or the prompt appears on the line below.
- Put informative labels on the x and y axes and a title on all plots.
- Keep functions short – typically no longer than one page in length.
- Suppress the output from all assignment statements in functions and scripts.
- Functions that return a value do not normally print the value; it should simply be returned by the function.
- Use the array operators .*, ./, .\, and .^ in functions so that the input arguments can be arrays and not just scalars.