Matlab for Finance Course: Session 2

Dr. Peter A. Bebbington

Brainpool AI

peter@brainpool.ai

bebbington @brainpoolai @brainpoolai

November 16, 2024

REVIEW OF SESSION 1

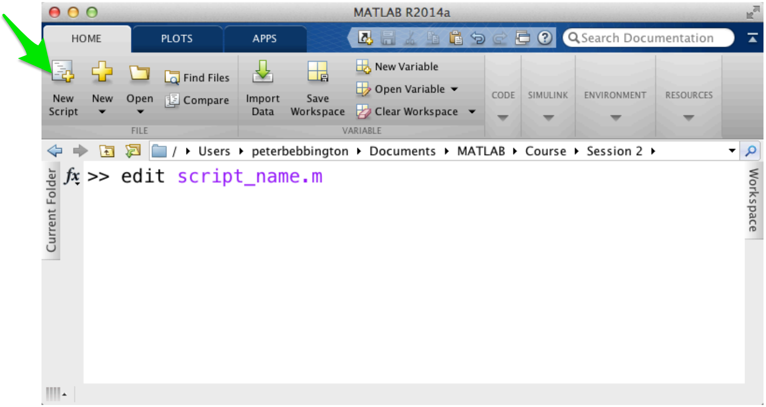
- MATLAB Desktop; Command Window, Command History, Current Directory, Launch Pad, Workspace
- Executing Basics Commands, Initialising and Defining
- Utility Commands
- Arrays, Matrices and Vectors
- Set Functions
- Addressing Array/Matrix Elements
- Solving Linear Equations
- Basic Plots

OBJECTIVE

- MATLAB Basics
  - Scripts and Editor
  - Executing Scripts
  - Debugging Tools
- Data Manipulation
  - Matrix Operations
  - Array Indexing
  - Matrix Inversion
- Programming Concepts
  - Boolean Logic
  - Control Flow
  - Error Handling
- Best Practices
  - Common Mistakes
  - Performance Tips
  - Code Organization

CREATING SCRIPTS

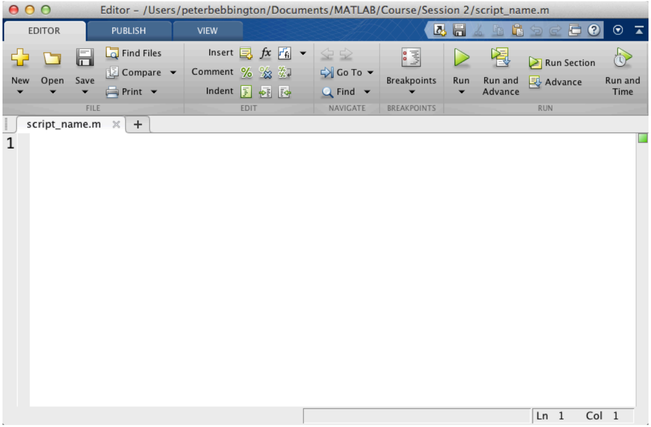
- Whilst the MATLAB console is useful for testing short code segments, you are going to want to use scripts for bigger projects.



```
>> edit script_name.m
```

EDITING SCRIPTS

- You can write your code in this window, then copy and paste into the console to make it run.
- If you make a mistake, you can fix it in the script, instead of re-typing it in the console window.



ARRAYS REVIEW

- An array is a collection of numbers in a sequence.
- Arrays can be multidimensional, but must always be rectangular.

```
X = [10 25 32 47 50 68]
```

- Think of them as vectors or matrices if it helps.
- We can operate on the whole array at once, or on individual elements by indexing.

```
X(1) = 10
X(6) = 68
```

- Think of them as vectors or matrices if it helps.
- We can operate on whole array at once, or on individual elements by indexing.

ARRAYS REVIEW CONTD.

- With 2 dimensions, it takes 2 arguments `X(r,c)`

```
X = [10 25 32 47 50 68; 45 3 98 56 11 22]
```

- Hence one can access an element by

```
X(1,2) = 25
X(2,6) = 22
```

ARRAY OPERATORS

- Create a new script by returning the following command

```
edit array_operators.m
```

- The following script performs some array operations and prints the values to the Command Window:

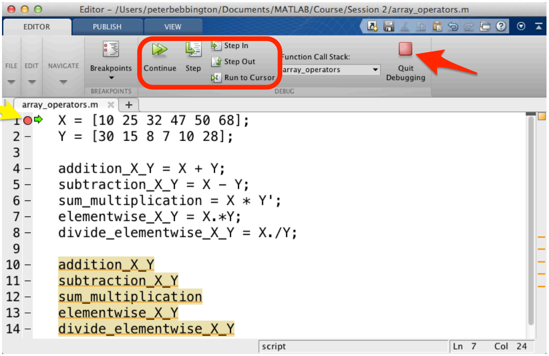
```
X = [10 25 32 47 50 68];
Y = [30 15 8 7 10 28];

addition_X_Y = X + Y;
subtraction_X_Y = X - Y;
sum_multiplication = X * Y';
elementwise_X_Y = X.*Y;
divide_elementwise_X_Y = X./Y;

addition_X_Y
subtraction_X_Y
sum_multiplication
elementwise_X_Y
divide_elementwise_X_Y
```

- Run the script by returning the following Command in Command Window:

```
>> array_operators
```

DEBUGGING

- Debugging is useful to step through your code to see how it works and possible find issues.
- Set break points (red dot) by clicking line number (yellow arrow), run the code to the break point with the `F5` or run button, step through the code with buttons in red rectangle and click quit debugging button to stop (red arrow).



```
X = [10 25 32 47 50 68];
Y = [30 15 8 7 10 28];

addition_X_Y = X + Y;
subtraction_X_Y = X - Y;
sum_multiplication = X * Y';
elementwise_X_Y = X.*Y;
divide_elementwise_X_Y = X./Y;

addition_X_Y
subtraction_X_Y
sum_multiplication
elementwise_X_Y
divide_elementwise_X_Y
```

ARRAY OPERATORS INVERSION

- Solve (for x) system of $\bar{\bar{A}}\bar{x} = \bar{\bar{B}}$ using \

$$\texttt{x=A\textbackslash B} \quad \Rightarrow \bar{x} = \bar{\bar{A}}^{-1}\bar{\bar{B}}$$

- Solve (for x) system of $\bar{x}\bar{\bar{A}} = \bar{\bar{B}}$ using /

$$\texttt{x=A/B} \quad \Rightarrow \bar{x} = \bar{\bar{B}}\bar{\bar{A}}^{-1}$$

BOOLEAN LOGIC

- At some point you will need your software to make decisions.
- These are made using simple Boolean logic tests. A Boolean test assumes that only two answers are possible; true (1) or false (0).

```
< less than
> greater than
<= less than or equal to
>= greater than or equal to
== equal to
~= not equal to
&  AND
&& AND short-
circuit
|  OR
|| OR short-
circuit
```

TRUTH TABLE

A	B	A&B	A|B	xor(A,B)	~A
0	0	0	0	0	1
1	0	0	1	1	0
0	1	0	1	1	1
1	1	1	1	0	0

SHORT-CIRCUIT

The statement shown here performs an AND of two logical terms, A and B:

```
A && B
```

- If `A` equals zero, then the entire expression will evaluate to logical `0` (`false`), regardless of the value of `B`. Under these circumstances, there is no need to evaluate B because the result is already known. In this case, MATLAB `short-circuits` the statement by evaluating only the first term.
- A similar case is when you OR two terms and the first term is `true`. Again, regardless of the value of `B`, the statement will evaluate to `true`. There is no need to evaluate the second term, and MATLAB does not do so.

BOOLEAN LOGIC AND

- The & and && operators compare two Booleans, and returns a true value only if both Booleans are true.

Example:

```
(x ~= 0) && (x < y)

 x = 5, y=6  : 1
 x = 5, y=2  : 0
```

BOOLEAN LOGIC OR

- The | and || operators compare two Booleans, and return true if either or both Booleans are true.

Example:

```
(x ~= 0) || (x < y)

 x = 5, y=6  : 1
 x = 5, y=2  : 1
 x = 0, y=0  : 0
```

BOOLEAN LOGIC EXAMPLE

```
a = [3 5 9 0]; b = [2 0 10 1];
(a > 5) & (b <= 10), (a > 5) & (b < 4)
(a > 5) | (b < 4)
xor(a >= 5, b >= 10)
~(a == 3)
```

CONDITIONAL INDEXING

- MATLAB allows you to index arrays using logical conditions
- This is a powerful way to filter data without loops

```
% Create sample data
data = [10 25 32 47 50 68];

% Find values greater than 30
big_values = data(data > 30)

% Replace values less than 30 with zero
data(data < 30) = 0

% Find indices of even numbers
even_indices = find(mod(data, 2) == 0)

% Count number of elements meeting condition
num_big = sum(data > 30)
```

FLOW CONTROL IF-ELSE

- Not only do you want your software to make decisions, you want it to act on them
- Flow control (or Branching) allows you to tell the program how to react to given situations
- The **if** statement checks to see if an argument is true, and only runs its block of code if it is

```
x = 5;
if x > 0
    disp("x is positive")
elseif x < 0
    disp("x is negative")
else
    disp("x is zero")
end
```

IF-ELSE EXAMPLE

- The if statement checks a condition and executes code based on whether it's true or false
- Multiple conditions can be checked using elseif
- The else block catches any cases not covered by if/elseif

```
a = 57;
if mod(a,7) == 0
    disp("Divisible by 7")
elseif mod(a,7) == 1 || mod(a,7)==6
    disp("Almost divisible by 7")
else
    disp("Definitely not divisible by 7")
end
```

FLOW CONTROL WHILE

- The **while** statement repeats (loops) its code block until the Boolean test condition stops being true

```
% Initialize counter
count = 1;
while count <= 5
   disp(["Iteration " num2str(count)])
   count = count + 1;
end
```

WHILE EXAMPLE

- The while loop continues executing until its condition becomes false
- The condition is checked before each iteration

```
A = zeros(10,1);
A(7) = 1;
m = 1;
while (A(m) ~= 1)
    m = m + 1;
end
```

- What is the value of `m` after while statement has finished?

FLOW CONTROL FOR

- The **for** statement takes an array as input, and loops its code block once for each element
- A loop variable takes the value of the array element in each iteration

```
% Simple for loop example
numbers = [1 2 3 4 5];
sum = 0;
for i = 1:length(numbers)
    sum = sum + numbers(i);
end
disp(["Sum is " num2str(sum)])
```

FLOW CONTROL SWITCH

- The **switch** statement takes a ”switch” variable as input, and activates the code block whose ”case” variable matches the switch
- This could be achieved with an if loop with lots of **elseif** conditions. However, when there are a lot of code blocks the switch is much faster

```
% Example of switch statement
day = "Wednesday";
switch day
    case "Monday"
        disp("Start of work week")
    case {"Saturday", "Sunday"}
        disp("Weekend!")
    otherwise
        disp("Midweek")
end
```

DATA SERIES EXAMPLE

- Find the sum of squares between two data series
- The two series are stored in row-vector arrays
- Each has the same number of data points

```
n = 10000; % size of series
series1 = sqrt(2).*randn(n,1);
series2 = sqrt(4).*randn(n,1);

% intialise the variable sum_of_sqr
sum_of_sqr = 0;

% loop through each element
for index = 1:length(series1)
    difference = series1(index) - series2(index);
    sum_of_sqr = sum_of_sqr + difference^2;
end
```

VECTORISE

- Not only is it less code, but this kind of ”vectorised” computing runs much faster than using loops

```
% Using a loop (slower approach)
for i = 1:length(series1)
   difference(i) = series1(i) - series2(i);
end
sum_of_sqr = sum(difference.^2);

% Using vectorization (faster approach)
difference = series1 - series2;
sum_of_sqr = sum(difference.^2);
```

FIBONACCI EXAMPLE

- Generate Fibonacci sequence up to 4 million
- Sum only the even values in the sequence

```matlab
%% Fibonacci
fib = [1 2];  % assign initial value to variables
fib_sum = 0;

while (fib(2) < 4e6)
    % check if largest value is even
    if (mod(fib(2),2) == 0)
        fib_sum = fib_sum + fib(2);
    end
    % generate next number in sequence
    fib = [fib(2) sum(fib)];
end
```

EXERCISERS

- Write scripts to perform the following tasks:
  1. Figure out how many terms in the sum `1+2+3+...` it requires for the sum to exceed one million.
  2. Write a program to calculate the sum `1+2+3+...+300`. Display the total after every 20 terms by using an if statement to check if the current number of terms is a multiple of 20.
  3. Write a simulator to determine the result of flipping of a coin `1000` times using a for loop, and the `rand()` command to generate a `uniform[0,1]` random number. (type `help rand' in console). How would you change the probability of getting heads/tails in your model? Can this have a vectorised solution?
  4. By modifying your simulation from (3), generate a random walk (starting from a value of 1) based on the result of each coin flip. If the coin lands on heads multiply the previous step in the walk by `1.1` to get the new step. If it lands on tails then multiply by `0.9` instead. Keep track of the result from each step. Can this have a vectorised solution?

MATLAB OPERATORS SUMMARY

- Arithmetic Operators

```
+ Addition          - Subtraction
* Matrix mult.     .* Element-wise mult.
/ Matrix div.      ./ Element-wise div.
^ Matrix power     .^ Element-wise power
```

- Relational Operators

```
== Equal   ~= Not equal
<  Less    <= Less/equal
>  Greater >= Great/equal
```

- Logical Operators

```
&& AND              || OR
~  NOT              xor() Exclusive OR
```

COMMON MISTAKES TO AVOID (1/2)

- Using wrong array dimensions

```
% This will error:
[1 2 3] + [1; 2; 3] % Different dimensions

% Fix: Make dimensions match
[1 2 3] + [1 2 3] % Both row vectors
```

- Confusing matrix and element-wise operations

```
% Matrix multiplication (inner dimensions must match)
A * B

% Element-wise multiplication (dimensions must match)
A .* B
```

COMMON MISTAKES TO AVOID (2/3)

- Forgetting to initialize variables

```
% Better practice:
sum = 0;
for i = 1:n
    sum = sum + i;
end
```

- Using wrong comparison operators

```
% Wrong: single = is assignment
if x = 5 % This is an error

% Correct: double == is comparison
if x == 5 % This checks equality
```

COMMON MISTAKES TO AVOID (3/3)

- Mixing up parentheses () and brackets []

```
% Wrong: brackets for indexing
array[1,2] = [1 2] % Error

% Correct: parentheses for indexing
array(1,2) = [1 2] % OK
```

- Note: In MATLAB:
  - () for function calls and indexing
    for array creation

ERROR HANDLING

- Use try-catch blocks to handle potential errors gracefully

```
try
    result = someFunction();
catch ME
    fprintf('Error: %s\n', ME.message);
end
```

- Common error handling patterns:
    - Input validation
    - File operations
    - Network operations
    - Matrix operations with incompatible dimensions

CELL ARRAYS

- Cell arrays can store different types of data
- Created using curly braces {}
- Useful for mixed data types

```matlab
% Create cell array
data = {1, 'text', [1 2 3], @sin};

% Access elements
number = data{1};  % Returns 1
array = data{3};   % Returns [1 2 3]

% Cell array of strings
names = {'Alice', 'Bob', 'Charlie'};
```

PERFORMANCE TIPS

- Pre-allocate arrays
- Use vectorization instead of loops
- Avoid growing arrays in loops
- Profile code using `profile on/off`

```matlab
% Bad practice (slow)
for i = 1:1000
    arr(i) = i; % Growing array
end

% Good practice (fast)
arr = zeros(1, 1000); % Pre-allocate
for i = 1:1000
    arr(i) = i;
end

% Even better (vectorized)
arr = 1:1000;
```