DSA2102 Numerical Computation – MATLAB

- MATLAB is unable to run scripts not in the current directory (extension: *.m*)
- Change the working folder to the one you want to work in
- *Current Folder* panel shows all files in the working folder
- *Command Window* used to issue commands that create variables or call functions

Create a matrix	`A = [1, 2, 3; 4, 5, 6; 7, 8, 9]`
	`    1  2  3`
	`A = 4  5  6`
	`    7  8  9`
		MATLAB adds variable A to the workspace and displays the result in the *Command Window*

Compute eigenvalues of a matrix	`e = eig(A)`	- Using built-in math function *eig* - If no output variable (e.g. *e*) is defined, MATLAB uses the variable *ans* (short of answer) to store the results of the computation

Suppressing output	`% End statement with semicolon`
	`d = det(A);`	

Overwriting variables	`A = [1, 2, 3; 4, 5, 6; 7, 8, 9]`
	`% Transpose of A with all entries + 1`
	`A = A' + 1`
	`    2  5   8`
	`A = 3  6   9`
	`    4  7  10`
		Once a variable has been created, it can be reassigned

Entering long statements	`a = 1 -1/2 + 1/3 – 1/4 + 1/5 – 1/6 ...`
	`    + 1/7 – 1/8 + 1/9`
	`a = 0.7456`
		When a statement does not fit on one ine, use an ellipsis (*three periods* …) followed by *Return/Enter* to indicate that the statement continues on the next line

Entering multiple statements per line	`A = [1, 2, 3; 4, 5, 6; 7, 8, 9]; d = det(A), a = cos(d)`
	`d = 6.6613e-16`
	`a = 1`
		Use commas (,) or semicolons (;) to enter multiple statements at once

Clear Command Window	`clc`	Without deleting variables

Format of Display / Control format of values displayed	`x = [4/3, 1.23456e-6]`
	`% Display 4 d.p.`
	`format short`
	`x = 1.3333  0.0000`
	
	`% Display 15 d.p.`
	`format long`
	`x =`
	`1.333333333333333  0.000001234560000`
	
	`format short e`
	`x = 1.3333e+00  1.234e-06`
	
	`format long e`
	`x =`
	`1.333333333333333e+00`
	`1.234560000000000e-06`
		- By default, MATLAB only displays 4 decimals in the result of calculations
		- Note that the command affects only how numbers are displayed, not how MATLAB computes or saves them

Workspace	% View list of variables in the workspace who % With more details of variables % Size, bytes, class, attributes whos % Delete variable A only clear A % Delete all variables in the workspace Clear % Check current directory pwd % Check files ls	- *Workspace* contains variables that you created within or imported into MATLAB from data files - Variables appear on the *Workspace* panel
Save workspace variables	% Save data in current folder with file name “myfile.mat” save myfile.mat % Restore data from a MAT-file into the workspace load myfile.mat	- Workspace variables do not persist after you exit MATLAB, save data for later use with *save* - Saving preserves the workspace in current working folder in a compressed file with *.mat* extension, called a MAT-file - Run saved codes in current working folder by typing its file name
Getting help	help eig	- All MATLAB functions and commands have supporting documentation that includes examples and describes how to use them - E.g. function inputs, outputs, calling syntax - Access the complete MATLAB documentation, click the icon *?* on the desktop toolbar - Access the documentation for a specific command or function, use the *help* command in the *Command Window*
Array creation	% Create an array with 4 elements in a single row (i.e. row vector) a = [1, 2, 3, 4] a = 1 2 3 4 % Create a matrix with multiple rows, separate the rows with semicolons a = [1, 2, 3; 4, 5, 6] a = 1 2 3 4 5 6	- All MATLAB variables are multi-dimensional *arrays* - A *matrix* is a 2-dimensional array - MATLAB is designed to operate primarily on whole matrices and arrays

Concatenation of arrays	a = [1, 2, 3; 4, 5, 6]
	b = [11, 12, 13; 14, 15, 16]
	
	% Horizontal concatenation
	A = [a, b]
	A = (2 by 6 matrix)
	
	% Vertical concatenation
	A = [a; b]
	A = (4 by 3 matrix)
		- Process of joining arrays to make larger ones
		- Horizontal concatenation: arrays must have same number of rows
		- Vertical concatenation: arrays must have same number of columns

Deleting rows or columns	a = [1, 2, 3; 4, 5, 6] $a = \begin{matrix}1 & 2 & 3\\ 4 & 5 & 6\end{matrix}$ % Delete column 2 a(:, 2) = [] $a = \begin{matrix}1 & 3\\ 4 & 6\end{matrix}$	Rows or columns can be deleted from a matrix using a pair of square brackets
Creating basic matrices	% ones (all ones) % zeroes (all zeroes) % eye (identity matrix) eye(3,3) $ans = \begin{matrix}1 & 0 & 0\\ 0 & 1 & 0\\ 0 & 0 & 1\end{matrix}$ % rand (uniformly dist rand elements) randn(2, 3) $ans = \begin{matrix}-0.4336 & 3.5784 & -1.3499\\ 0.3426 & 2.7694 & 3.0349\end{matrix}$ % randn (normally dist rand elements) % Is Gaussian random % 4-by-4 magic square A = magic(4) A = (4 by 4 matrix)	
Array Indexing	% Specify row and column subscripts to refer to a particular element A(3, 2) = -7 A = (4 by 4 matrix) % List elements in the 2nd through 4th rows and the 3rd column of A A(2:4, 3) $ans = \begin{matrix}10\\ 6\\ 15\end{matrix}$ % Specify all elements in 2nd row A(2, :) ans = (1 by 4 matrix)	- Specify row and column subscripts to refer to a particular element - Refer to multiple elements of an array using the colon operator (:), which allows one to specify a range in the form of *start : end* - The colon alone (:), without start and end values, specifies all the elements in that dimension

Matrix and Array operations	A = [1, 2, 3; 4, 5, 6; 7, 8, 9]
	
	% Raise the value of each entry by 10
	A+10
	      11 12 13
	ans = 14 15 16
	      17 18 19
	
	% Compute the cosine of each entry
	cos(A)
	       0.5403 −0.4161 −0.9900
	ans = −0.6536 0.2837 0.9602
	       0.7539 −0.1455 −0.9111
	
	% Transpose a matrix (use a single quote)
	A′
	      1 4 7
	ans = 2 5 8
	      3 6 9
	
	% Standard matrix operators
	p = A*inv(A)
	    1.0000 0 −0.0000
	p = 0 1.0000 0
	       0 0 1.0000
		- MATLAB allows processing of all the entries in a matrix using a single arithmetic operator or function
		- Standard matrix operations (e.g. addition, subtraction, multiplication, power) can be performed using the +,-,* and ^ operators

	A = [1, 2, 3; 4, 5, 6; 7, 8, 10]
	
	% Element-wise multiplication
	p = A.*A
	     1 4 9
	p = 16 25 36
	    49 64 100
	
	% Raise each element of A to the third power
	p = p.^3
	      1 8 27
	p = 64 125 216
	    343 512 1000
		To perform **element-wise operations (i.e. component-wise operations)** rather than the standard matrix operations use the ".<*op*>" operator

Matrix functions	% Dimension of a matrix
	size(A)
	% Determinant
	det(A)
	% Diagonal matrices, or diagonals of a matrix
	diag(A)
	% Eigenvalues and eigenvectors
	eig(A)
	% Matrix inverse
	inv(A)
	% Matrix norms
	norm(A)
	% Matrix rank (number of linearly independent rows/columns)
	rank(A)
	% Sum of diagonal elements
	trace(A)	

Solving linear equations	A = rand(3, 3); b = ones(3, 1);
	% Solution to Ax = b
	x = A\b
	    0.3919
	x = 0.2119
	    0.7508
		- Solution to system of linear equations can be computed using the backslash (\) operator
		- The numerical algorithm behind this operator is Gaussian elimination

<table>
<tr><td rowspan="11">2d plots</td><td colspan="3">% Plot function sin(x) from 0 to 2π
% Start : increment : end
% x is a row vector of 201 elements
x = 0 : pi/100 : 2*pi
y = sin(x)
plot(x, y)

% Label axes
xlabel(‘x’)
ylabel(‘sin(x)’)
title(‘Plot of the sine function’)

% Displays plot
figure(1)</td><td colspan="3">- Given 2 vectors of the same length,
$x = (x1, x2, \ldots, xN)$
and
$y = (y1, y2, \ldots, yN)$,
the <b><i>plot</i></b> command produces a graph of y versus x
- MATLAB locates the points
(xi, yi)
with $i = 1, 2, \ldots N$,
then joins them by straight lines</td></tr>
<tr><td colspan="3">% Plot command
plot(x, y, ‘style_colour_marker’)
% where style_colour_marker is a triplet of values from the table below</td><td colspan="3">- Line specification
- By default, MATLAB uses blue colour and solid line in a single 2d plot
- Line styles, colours and markers can be specified using a third argument in the <b><i>plot</i></b> command</td></tr>
<tr><td>SYMBOL</td><td>Colour</td><td>SYMBOL</td><td>Line Style</td><td>SYMBOL</td><td>Marker</td></tr>
<tr><td>k</td><td>black</td><td>-</td><td>solid</td><td>+</td><td>plus sign</td></tr>
<tr><td>r</td><td>red</td><td>--</td><td>dashed</td><td>o</td><td>circle</td></tr>
<tr><td>b</td><td>blue</td><td>:</td><td>dotted</td><td>*</td><td>asterisk</td></tr>
<tr><td>g</td><td>green</td><td>-.</td><td>dash-dot</td><td>.</td><td>point</td></tr>
<tr><td>c</td><td>cyan</td><td>none</td><td>no line</td><td>x</td><td>cross</td></tr>
<tr><td>m</td><td>magenta</td><td></td><td></td><td>s</td><td>square</td></tr>
<tr><td>y</td><td>yellow</td><td></td><td></td><td>d</td><td>diamond</td></tr>
<tr><td colspan="3">% Plot sine function using a red dashed line
plot(x, y, ‘r--')

% Close the figure
close;

% Hold multiple plots in one figure
x = 0 : pi/100 : 2*pi
y = sin(x)
plot(x, y)
hold on;
y2 = cos(x);
% To distinguish the 2 graphs
plot(x, y, ‘r:’)
legend(‘sin’ , ‘cos’)

% Stop plotting on the same figure
hold off;</td><td colspan="3">Add plots to an existing figure using the <b><i>hold</i></b> command</td></tr>
</table>

3d plots	% Generate grid-points % meshgrid(x-dir, y-dir) % where x, y are matrices (containing the coordinates of grid-points) [X, Y] = meshgrid(-2 : .2 : 2, -2 : .1 : 2); % Transform domain specified by 2 vectors in arguments into arrays X and Y % that can be used for the evalf of functions of 2 variables Z = X.*exp(-Xˆ2 – Y.ˆ2); surf(X, Y, Z)	- 3-dimensional plots display a surface defined by the function of 2 variables, *z = f(x, y)* - Domain of Z is [-2, 2] × [-2, 2] where Z = xe^(-x^2-y^2)

Programming - Scripts	% Example of a script
	% Add comments after the percentage symbol
	n = 50;
	% Generates elements from unif(0,1)
	% Creates a column vector of length 50
	r = rand(n, 1);
	% No need to specify x-coordinates (default: natural numbers)
	plot(r)
	% Draw a horizontal line on the plot at the mean
	m = mean(r);
	hold on; plot([0,n],[m,m]);
	hold off;
	Title(’Mean of Random Uniform Data’)
		- A *script* is a file with a ***.m*** extension that contains multiple sequential lines of MATLAB commands and function call
		- To create a script, use the MATLAB editor:
		(File -> New -> Script)
		- Save the script with a name (e.g. *plotrand.m*) in the current folder
		- Run the script by typing its name at the command line
		- When trying to run a script or a user-defined function, MATLAB looks for the file in the current folder (or a folder on the search path)
		- Scripts can operate on existing variables in the workspace, or they can create new variables on which to operate
		- Variables created in a script are added to the workspace (this may have undesirable effects – for example, existing variables in the workspace may be overwritten)

Programming – Functions	% Creating your own functions
	function [f,s] – factorial2(n)
	% Returns the factorial of 2*n and the sum of integers from 1 to n
	% Compute a factorial value
	N2 = 2*n;
	f = prod(1:N2);
	s = sum(1:n);
	end
		- *Functions* are files that can accept input arguments and return output arguments
		- The first line starts with the keyword ***function***, followed by the output arguments (*f* and *s*), the function name (*factorial2*) and input arguments (*n*)
		- The next several comment lines are printed when typing *help factorial2* in the command window
		- The rest of the file is the executable MATLAB code defining the function
		- * The names of the file and of the function should be the same
		- Functions operate on variables within their own workspace, separate from the workspace that you access in the MATLAB command window
		- (Different from scripts) Variables created within the function is not created in the workspace

	% Anonymous function
	fun = @(x, y) x*sin(y);
	
	fun(2, -3)
	ans = -0.2822
		- An *anonymous function* is a simple form of the MATLAB function that is defined within a *single* statement
		- Allows the creation of simple functions without having to create a file
		- Syntax for creating an anonymous function:
		f = @(arglist) expression

Programming – Control flow	% if-else statements
	if min(e) > 0
	display(...)
	end
	
	if min(e) > 0
	display(...)
	elseif max(e) <= 0
	display(...)
	else
	display(...)
	end
		- Conditional control:
		if, else, switch
		- Conditional statements enable the selection of which block of code to execute at run time
		- Loop control:
		for, while, continue, break
		- In the "for … end" loop, a group of statements is repeated at a fixed, predetermined number of times
		- *while-loops* repeats a group of statements an indefinite number of times under the control of a logical condition
		- The *continue* statement passes control to the next iteration of the *for* loop or *while* loop in which it appears, skipping any remaining statements in the body of the loop
		- The *break* statement terminates a *for* or a *while* loop, and passes the control to the 1st statement after the corresponding *end*
		- Program termination:
		return
		- The *return* command terminates the program before it runs to completion
		- One can insert a *return* statement within the called function to force an early termination and to transfer control to the invoking function or keyboard

	% for-loop statements % n runs from 3 to 32 with increment of 1 for n = 3 : 32 % Rank elements by their value r(n) = rank(magic(n)); end % Display value of r r	

```
% Nested for-loops
m = 5; n = 8;
for i = 1 : m
   for j = 1 : n
      h(i, j) = 1/(i-j);
   end
end
```

```
% while-loop statements
% Example: Finding a zero of a
polynomial using interval bisection
a = 0; fa = -5; b = 3; fb = 16;
while b-a > eps*b
   x = (a+b)/2;
   fx = x^3 -2*x -5;
   if sign(fx) == sign(fa);
      a = x; fa = fx;
   else
      b = x; fb = fx;
   end
end
x
```

```
% Loop with continue statement
% a is a random vector of length 100
containing numbers from a std normal
distribution
a = randn(100);
count = 0;
for i = 1 : 100
   % Checks the sign
   if a(i) <= 0
      continue
   end
   count = count + 1;
   b(count) = log(a(i));
end
% b is a vector containing log of
positive random numbers
b
```

```
% Loop with break statement
% Improved program to find a zero of
a polynomial
a = 0; fa = -5; b = 3; fb = 16;
while b-a > eps*b
   x = (a+b)/2;
   fx = x^3 -2*x -5;
   if fx == 0
      break;
   elseif sign(fx) == sign(fa)
      a = x; fa = fx;
   else
      b= x; fb = fx;
   end
end
```

Relational and logical operators	>	Greater than	^=	Not equal to
	<	Less than	&	AND operator
	>=	Greater than or equal to	|	OR operator
	<=	Less than or equal to	^	NOT operator
	==	Equal to	=	Note that equals sign is reserved for assignment and is NOT a logical operator

Vectorisation	% Example of using matrices more efficiently
	% Initial, less efficient for-loop
	for k = 1 : 1000
	% x increases by 0.01 with each iteration
	x(k) = 0.01*k;
	y(k) = log10(x);
	end
	
	% A vectorised and more efficient version which achieves the same purpose
	x = 0.01 : 0.01 : 10;
	% Computes the log of each element in x
	y = log10(x);
			- One way to make MATLAB programs run faster is to *vectorise* the algorithms
			- Very often, *for* loops can be replaced by more efficient matrix operations	

Pre-allocation	% Example of making *for-loop* execute significantly faster
	
	% r is a column vector of zeroes
	r = zeroes(32, 1);
	for n = 1 : 32
	r(n) = rank(magic(n));
	end
	% Without pre-allocating r, the length of r(n) increases with each iteration which is very slow
			- *for* loops can be made to go faster by *pre-allocating* any vectors or arrays in which output results are stored
			- In the example, without pre-allocation, the MATLAB interpreter enlarges the vector *r* by one element each iteration through the loop
			- Vector pre-allocation eliminates this step and results in faster execution	