Technological University Dublin
ARROW@TU Dublin

Instructional Guides

School of Multidisciplinary Technologies

2013

A MATLAB Primer in Four Hours with Practical Examples

Jerome Casey
jerome.casey@tudubln.ie

Follow this and additional works at: https://arrow.tudublin.ie/schmuldissoft

 Part of the Computer Sciences Commons, and the Engineering Commons

Recommended Citation

Casey, J. (2013). A MATLAB primer in four hours with practical examples. Software guide for undergraduate students. Technological University Dublin

This Other is brought to you for free and open access by the School of Multidisciplinary Technologies at ARROW@TU Dublin. It has been accepted for inclusion in Instructional Guides by an authorized administrator of ARROW@TU Dublin. For more information, please contact arrow.admin@tudublin.ie, aisling.coyne@tudublin.ie, gerard.connolly@tudublin.ie, vera.kilshaw@tudublin.ie.

A MATLAB Primer in 4 Hours with Practical Examples

Engineering Computing

DT023-1, DT025-1

Jerome Casey

Table Of Contents

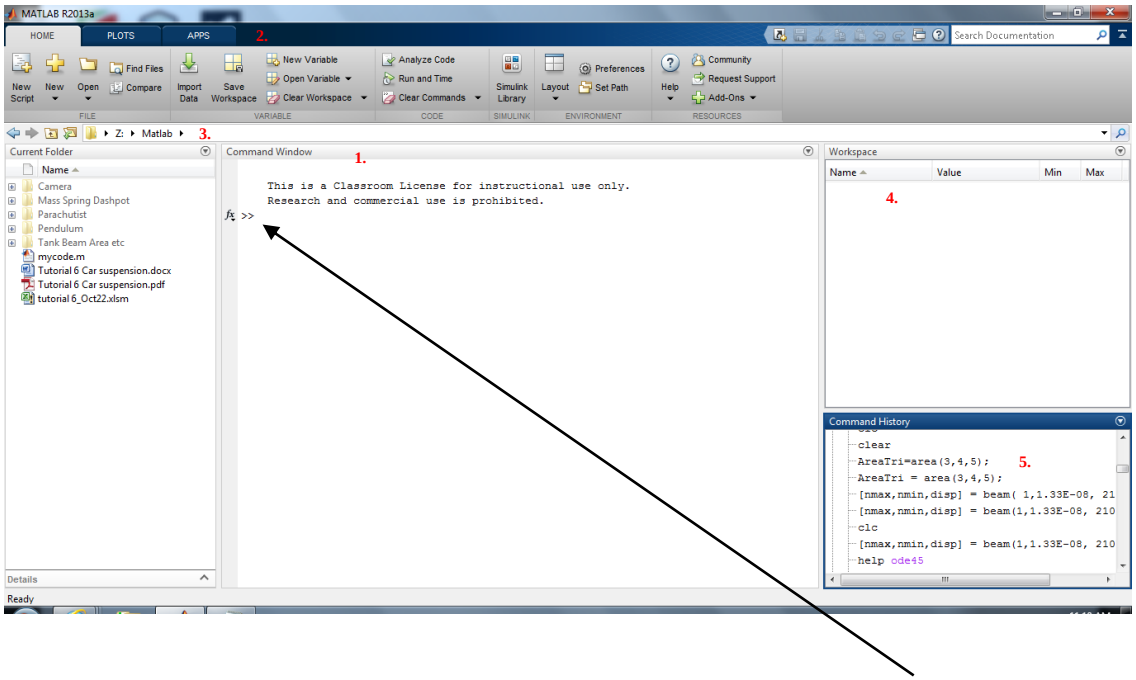
MATLAB EXERCISE 1 1
INTRODUCTION TO THE MATLAB INTERFACE 1
MATLAB EXERCISE 2 2
NUMERICAL METHODS: WORKING WITH MATRICES 2
MATLAB EXERCISE 3 7
ENTERING & PROCESSING DATA IN MATLAB, SAVING DATA TO EXTERNAL FILES 7
MATLAB EXERCISE 4 10
READING FROM EXTERNAL FILES / WORKING WITH PLOTS: LINE COLORS & STYLES, TITLES 10
MATLAB EXERCISE 5 13
WORKING WITH SUBPLOTS 13
MATLAB EXERCISE 6 14
LOADING AND WRITING RESULTS TO EXTERNAL FILES: LOAD & SAVE COMMANDS, VARIABLE EDITOR ... 14
MATLAB EXERCISE 7 16
WRITING SCRIPTS & FUNCTIONS 16

Matlab Exercise 1

Introduction to the Matlab Interface – Setting the Current Folder

"MATLAB" stands for "Matrix Laboratory." It is an interactive software program for performing numerical computations. It was initially designed by Cleve Moler in the 1970s for use as a teaching tool, but it has since become a very successful commercial package produced by The MathWorks Ltd. [1] One of MATLAB's best features, from the point of view of the computational scientist, is its large built-in library of numerical routines and graphical visualisation tools.

Click on the desktop shortcut to open MatlabR2013a. The main window is the MATLAB **Command Window**, where you write your instructions.



In the MATLAB **Command Window**, Matlab will execute the instructions after the **Command prompt** » when you press 'Enter' on your keyboard. You can also use the arrow key ↑ to cycle back through previously entered commands.

Set the desktop layout to the default setting by selecting **Desktop > Desktop Layout > Default**.

Note: Create a Matlab folder in your Computing folder of your Home folder and set the **Current Folder** (in 3. above) to where you will save your files such as: z:\Computing\Matlab

In these exercises you will cover:

- The MATLAB Desktop and Desktop Tools including the *Command Window*, the *Launch Pad*, the *Help Browser*, the *Current Folder Browser* and the *Editor/Debugger*.
- Matrices in MATLAB: Generating Matrices, Sum, Transpose, Subscripts, concatenation, deleting rows and columns and the Colon Operator. Expressions: Variables, Numbers, Operators.
- Scripts & Functions, the Load command, M-Files.
- Demonstration of Numerical methods.

[1] see www.mathworks.com for more examples and help on code.

Matlab Exercise 2

Numerical Methods: Working with Matrices

Creating Matrices:

Informally, the terms matrix and array are often used interchangeably. More precisely, a matrix is a two-dimensional rectangular array of real or complex numbers that represents a linear transformation. The linear algebraic operations defined on matrices have found applications in a wide variety of technical fields.

MATLAB has dozens of functions that create different kinds of matrices. Two of them can be used to create a pair of 3-by-3 example matrices. The first example matrix is **symmetric**:

```
>> A = pascal(3)

A =
     1     1     1
     1     2     3
     1     3     6
```

Info:
An nxn matrix that has the same number of rows and columns is called a **square** matrix.

A matrix is said to be **symmetric** if it is equal to its transpose (i.e. it is unchanged by transposition).

The second example is not symmetric:

```
>> B = magic(3)

B =
     8     1     6
     3     5     7
     4     9     2
```

The command >> **sum**(B) sums the elements in each column to produce:

```
    15    15    15
```

The command >> **sum**(B')' transposes the matrix, sums the columns of the transpose, and then transposes the results to produce the row sums:

```
    15
    15
    15
```

The command >> **sum**(**diag**(B)) sums the **main diagonal** of B, which runs from the upper left element to the lower right element, to produce 15

An m x n matrix is a rectangular array of numbers having m rows and n columns. A **column vector** is an m-by-1 matrix, a **row vector** is a 1-by-n matrix and a **scalar** is a 1-by-1 matrix. Enter the following statements:

>> col = [3; 1; 4]	produces	a column vector, >> col = 3 1 4
>> row = [2 0 -1]	produces	a row vector, >> row = 2 0 -1
>> s = 7	produces	a scalar: >> s = 7

Note the square brackets create a matrix; rows are delineated by a semi-colon and columns by a space.

Accessing Matrix Elements:

A 2D matrix is specified first by rows and then by columns.

```
>> col1 = A(: , 1)   % select all row entries for column 1
col1 =
   1
   1
   1
>> elem1 = A(2 , 2)   % select element at row 2 and column 2
elem1 =
   2
```

Note the **:** operator denotes 'all'.

Adding and Subtracting Matrices:
Addition and subtraction of matrices is defined just as it is for arrays, element-by-element. To add 2 matrices together they must have the same dimensions:

```
>> C = A + B
C =
     9     2     7
     4     7    10
     5    12     8
```

Addition and subtraction require both matrices to have **the same dimension**, or one of them to be a **scalar**. If the dimensions are incompatible you will get an error. Add a scalar to each element of a matrix:

```
>> D = C + 1
D =
    10     3     8
     5     8    11
     6    13     9
```

Vector Products and Transpose:
For real matrices, the transpose operation interchanges $a_{i\,j}$ and $a_{j\,i}$. In other words, transposing a vector changes it from a row to a column vector and vice versa. The extension of this idea to matrices is that transposing interchanges rows with the corresponding columns: the 1st row becomes the 1st column, and so on. MATLAB uses the apostrophe operator (') to perform a complex conjugate transpose, and the dot-apostrophe operator (.') to transpose without conjugation. For matrices containing all real elements, the two operators return the same result.

```
>> Z = B'
Z =
     8     3     4
     1     5     9
     6     7     2
```

Deleting Matrix Rows and Columns:

```
>> F = magic(5) % create this 5 x 5 matrix
F =
    17    24     1     8    15
    23     5     7    14    16
     4     6    13    20    22
    10    12    19    21     3
    11    18    25     2     9

>> F(3,:)=[]; % delete row 3
>> F(:, 5)=[]; % delete column 5
% F is now a 4x4 matrix
```

Tip:
The **semi-colon suppresses** the output to the Command Window.
Double-click on the variable F when it appears in the **Workspace** to open it in the **Variable Editor** and watch the values update as you enter the commands.

Matrix Concatenation:

```
>> Z = ones(4,1) % create this 4x1 matrix of ones
>> F= [F  Z] % concatenate the 4x1 matrix to the end of the 4x4 matrix
```

Matrix Multiplication:
MATLAB uses a single asterisk to denote matrix multiplication. The next two examples illustrate the fact that matrix multiplication is **not commutative**; AB is usually not equal to BA:

```
>> X = A*B, Y = B*A

X =
    15    15    15
    26    38    26
    41    70     3

Y =
    15    28    47
    15    34    60
    15    28    43
```

Dot-Product of Matrices:
Instead of doing a matrix multiply, we can multiply the corresponding elements of two matrices or vectors using the .* operator. The **dot-product** is also known as **element-wise multiplication**. The dot product works as for vectors: corresponding elements are multiplied together, thus the matrices involved must have the same size:

```
>> Dot = A .* B
Dot =
    8    1    6
    3   10   21
    4   27   12
```

Sparse Matrices:
Sparse matrices are usually large matrices that have only a very small proportion of non-zero entries.

Create a sparse 5x4 matrix *S* having only 3 non-zero values: $S_{1,2} = 10$, $S_{3,3} = 11$ and $S_{5,4} = 12$.
First create 3 vectors containing the ith index, the jth index and the corresponding values of each term and then use the **sparse** command.

```
>>i = [1, 3, 5]; j = [2, 3, 4];
>>v = [10 11 12];
>>S = sparse(i,j,v)

S =
  (1,2)       10
  (3,3)       11
  (5,4)       12

>> Sfull = full(S)

Sfull =
    0   10    0    0
    0    0    0    0
    0    0   11    0
    0    0    0    0
    0    0    0   12
```

The matrix *Sfull* is a "full" version of the sparse matrix S.

Solving Simultaneous Equations:
A general system of linear equations can be expressed in terms of a co-efficient matrix A, a right-handside (column) vector b and an unknown (column) vector x as follows:

A * x = b

or, component wise, as:

$a_{1,1}x_1 + a_{1,2}x_2 + ... a_{1,n}x_n = b_1$
$a_{2,1}x_1 + a_{2,2}x_2 + ... a_{2,n}x_n = b_2$
...
$a_{n,1}x_1 + a_{n,2}x_2 + ... a_{n,n}x_n = b_n$

The solution is obtained by $x = A^{-1} * b$

Given

3v -3w +6x -2y+z = 14
3v -6w +x -y +z = 25
2v -4w +4x -4y+3z = 5
3v -6w +5x -y +2z = 30
2v -4w +9x +y +z = 30

Solve for v, w, x, y and z using the matrix functions

Enter the following in the **Command Editor**:

```
>> A = [3 -3 6 -2 1; 3 -6 1 -1 1; 2 -4 4 -4 3; 3 -6 5 -1 2; 2 -4 9 1 1];
>> b = [14; 25; 5; 30; 30];
>> x = inv(A)*b
```

(Though formally correct >> x = A \ b is generally used in Matlab)

```
x =
    6.3333
   -1.3333
         0
    7.0000
    5.0000
```

The inverse looks better if it is displayed with a rational format.

```
>> format rat
>> x

x =
     19/3
     -4/3
      0
      7
      5
```

The following statement restores the output format to its default.

```
>> format short
```

Diagonal Matrices:
A diagonal matrix contains non zero diagonal entries with zeros everywhere else. D is a 3x3 diagonal matrix. To construct this in Matlab, we could type it in directly as follows:

```
>> D = [-3 0 0; 0 4 0; 0 0 2]
D =
    -3     0     0
     0     4     0
     0     0     2
```

However this becomes impractical when the dimension is large (e.g. a 100 x100 diagonal matrix). We then use the **diag** function. Create a row vector d, say, containing the values of the diagonal entries (in order) then diag(d) gives the required matrix:

```
>> d = [-3 4 2], D = diag(d)
```

On the other hand, if A is any matrix, the command diag(A) extracts its diagonal entries:

>> H= **diag**(A)

H =

3
-6
4
-1
1

	1	2	3	4	5
1	3	-3	6	-2	1
2	3	-6	1	-1	1
3	2	-4	4	-4	3
4	3	-6	5	-1	2
5	2	-4	9	1	1

Initialising Matrices:

The functions **ones** and **zeros** are typically used to initialise a matrix - the size of which you may already know, but the values which may change during processing. Pre-allocation of memory by initialising a matrix allows for more efficient programs.

>> P= **ones**(2,3)

P=

1 1 1
1 1 1

>> Q= **zeros**(3,2)

Q =

0 0
0 0
0 0

>> R= **ones**(size(C))

R =

1
1
1
1
1

This example shows how we can construct a matrix based on the size of an existing one.

The Identity Matrix:

The n x n identity matrix is a matrix of zeros except for having ones along its leading diagonal (top left to bottom right).

>> I = **eye**(3)

I =

1 0 0
0 1 0
0 0 1

This is called **eye**(n) in Matlab since mathematically it is usually denoted by I.

Using Nested For Loops in 2 Dimensional Matrices:

A nested for loop is typically used to cycle through the entries of a 2 dimensional (m x n) matrix in so called raster order. The outer for loop controls the movement along the rows, beginning at the first row, whilst the inner loop controls the movement along the columns. The following code uses the matrix A created previously:

```
% update the diagonal elements to 10
[numRows,numCols] = size(A); % A was created above
for i = 1: numRows
    for j = 1: numCols
        if i==j
          A(i , j) =10;
        end
    end
end
```

Other important quantities in computational linear algebra that have functions in Matlab are matrix norms, trace, rank, condition number, eigenvalues and singular values.

Matlab Exercise 3

Entering & Processing Data in Matlab, Saving Data to External Files

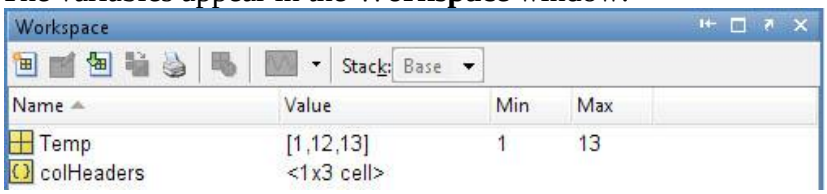
The basic arithmetic operators are + - * / ^ and these are used in conjunction with brackets ( ).

Entering Data:

This task is to enter Temperature data [2] for the first 10 days of the Month of April for various locations in the country and carry out some processing of the data. First create the variables:

```
>>colHeaders={'April' 'Dublin' 'Kilkenny'};
>> Temp =[1,12,13];
```

The variables appear in the **Workspace** window:



Method 1: - Entering Data via the Variable Editor

The **Variable Editor** allows you to view array contents in a table format and edit the values. Open the `Temp` variable that you just created in the **Variable Editor** by double clicking on it in the **Workspace** pane and continue entering the following data:

1	**12**	**13**
2	15	14
3	22	20
4	22	20
5	23	21
6	22	22
7	21	21
8	20	19
9	15	16
10	12	14

Method 2 - Entering Data by importing from an External file:

Delete the `Temp` variable from the workspace.

Create an Excel file with the same data above laid out in 3 columns. Search the help to see how to import the numerical data into MATLAB from Excel and assign it to the `Temp` variable.

Processing Data in Matlab:

Calculate the following:

(i) the average temperature over the 10 days for a location

(ii) the average temperature on each day for both locations

You can process data in Matlab by calling on a myriad of built-in functions. Later you will also create your own functions.

[2] Met Éireann use Matlab to process data, see http://www.met.ie/climate-ireland/rainfall.asp and the R package http://www.met.ie/climate/dataproducts/Estimation-of-Point-Rainfall-Frequencies_TN61.pdf

Enter the following at the **Command** prompt:

```
>> avgTempLocation1 = mean(Temp(:,2))
avgTempLocation1 =
   18.4000
>> avgTempLocation2 = mean(Temp(:,3))
avgTempLocation2 =
    18

>> avgTemp = mean(Temp(:,2:3),2)
avgTemp =
   12.5000
   14.5000
   21.0000
   21.0000
   22.0000
   22.0000
   21.0000
   19.5000
   15.5000
   13.0000
```

Notes:

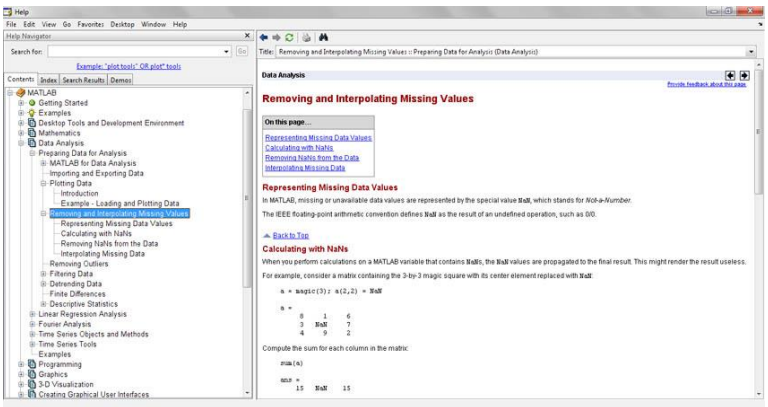
- The built in mean function was called to calculate the average (mean).
- `Temp` is actually an array variable, similar to a matrix with 10 rows and 3 columns.
- A matrix is referenced by specifying the row reference followed by the column reference, separated by a comma. e.g. `Temp(:,2)` accesses all the rows of column 2, i.e. column 2. The colon operator **:** is shorthand for all.
- The various calls to mean here calculate the average of a column or the average across a row. Can you see which is which? Check the help to see the various forms of the mean function.

Help Browser:
For Help on Matlab functions enter the following after the command prompt and it will print within the pane:

```
>>help mean
```

Alternatively select **Help/Product Help** and type in the query in the *Search for* box. e.g. type in *mean* and hit return. The help returns info on the *mean* function.

Take time to explore the Help. You can teach yourself Matlab from all the examples shown in the various sections. In particular have a look at the sections: Getting Started, Examples, Mathematics, Data Analysis, Programming, Graphics, Functions.



以上内容仅为本文档的试下载部分，为可阅读页数的一半内容。如要下载或阅读全文，请访问：https://d.book118.com/087144106013006062