Systems Theory Laboratory Assignment 1: Introduction to Matlab

1 Introduction

MatLab, developed by Mathworks Inc. (www.mathworks.com) is a commercial software environment for scientific and engineering calculations. The name stands for "**Mat**rix **Lab**oratory" because the language is aimed at the solution of problems involving vectors and matrices. This chapter is a very short introduction to the fundamentals of Matlab that may be useful for solving the exercises proposed in this handbook.

2 Using Matlab

When Matlab is launched, the main interface looks as the one shown in Figure 1.

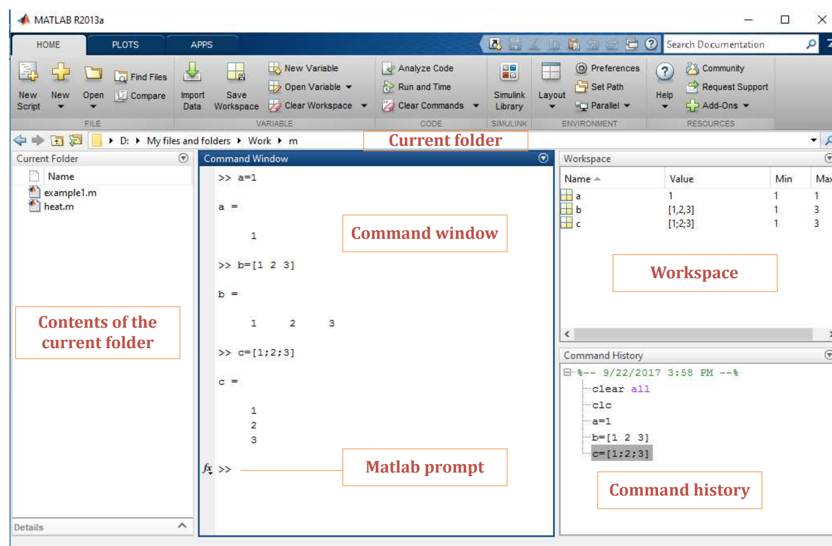

Figure 1: The Matlab desktop

The panels from the Matlab desktop are used as follows:

- Command window - to enter commands at the prompt indicated by two right arrows (>>)
- Current folder - to access and manipulate the files
- Workspace - to explore the data created in the current work session
- Command history - to explore the history of commands

To quit Matlab, type *quit* or *exit* at the command prompt.

To break the execution of current operation: press and hold Ctrl+C.

3 Statements and variables

Statements have the form:

```
>> variable = expression
```

The sign equals "=" implies the assignment of the expression to the variable.
In the following example, the value 1 is assigned to the variable a after **Enter** is pressed.

```
>> a = 1
a =
     1
```

The value of the variable is displayed after the statement is executed. If the statement is followed by a semicolon (;), the value is assigned to the variable but the echo is suppressed.

```
>> b = 2;
```

Both variables a and b have values at the current moment, as can be seen either in the *Workspace* window or by entering the name of the variable at the command prompt without a semicolon.

Matlab is case sensitive for variable names.
A variable name starts with a letter, followed by letters, digits or underscores.

Matlab can be used in *calculator mode*. If the variable name and the equal sign are omitted, the result is assigned to a generic variable called *ans* (from *answer*). This variable will be overwritten for every new calculation.

```
>> 2.5*3
ans =
   7.5000
>> 10-2
ans =
    8
```

4 Arrays and matrices

Matlab variables are multidimensional *arrays*, no matter what type of data, [2].
A *row vector*, or an array of elements on a single row, can be created by entering each element, separated by space or comma (,), between square brackets.

```
>> a = [1 2 3 4 5 6]

   a =
       1   2   3   4   5   6
```

A *column vector*, or an array with multiple rows, can be created by entering the elements separated by a semicolon (;), also between square brackets.

```
>> b = [1;2;3;4]

   b =
       1
       2
       3
       4
```

To create a matrix, separate the elements on a row with space or comma and the rows with semicolons.

```
>> A = [1 2 3 4;5 6 7 8;9 10 11 12]

   A =
          1     2     3      4
          5     6     7      8
          9    10    11     12
```

To create a regularly-spaced row vector with values between an *initial_value* and a *final_value* using an increment *inc*, state:

```
>> vector = initial_value:inc:final_value
```

If the increment is missing, the statement will create a unit-spaced vector. For example:

```
>> x=1:5
x =
     1     2     3     4     5

>> y=0:0.2:1
y =
         0    0.2000    0.4000    0.6000    0.8000    1.0000

>> z=10:-1:5
z =
    10     9     8     7     6     5
```

The elements in a matrix can be referenced by specifying the row and column number between parentheses (round brackets):

```
>> A(row,column)
```

The element on the second row and third column of the matrix A created before, is displayed below:

```
>> A(2,3)

ans =
     7
```

A colon used instead of the row or column number, refers to *all* the elements in a row or column of a matrix. All elements on the third column of the matrix A are obtained from:

```
>> A(:,3)

ans =
     3
     7
    11
```

Two or more arrays can be concatenated between square brackets. The row vectors x and y created before can be concatenated as follows:

```
>> [x y]
ans =
  Columns 1 through 8
    1.0000   2.0000   3.0000   4.0000   5.0000   0   0.2000   0.4000

  Columns 9 through 11
    0.6000   0.8000   1.0000
```

A new value is appended at the end of the third column of matrix A and the result is assigned to a new variable Am:

```
>> Am = [A(:,3); 22]

Am =
     3
     7
    11
    22
```

5 Operators

The usual mathematical operators can be used in expressions and the order of arithmetic operations can be altered by using parentheses. The common operators are given in Table 1.

Symbol	Role	Symbol	Role
+	Addition	.*	Element-wise multiplication
−	Subtraction	./	Element-wise division
*	Multiplication	.^	Element-wise power
/	Division	'	Transpose
^	Power	\	Left-division

Table 1: Mathematical operators

Suppose you would like to add 2 to each of the elements in vector a:

```
>> a=[1 2 3 4 5 6];
>> b = a + 2
     b =
          3   4   5   6   7   8
```

Two vectors can be added together if they are the same length. The vectors a and b created above can be added into vector c as:

```
>> c = a + b
     c =
          4   6   8   10   12   14
```

Addition and subtraction of matrices of the same size work the same way.

```
>> A=[1 0;0 1], B=[0 1; 1 0], C=[1 2; 3 4], D=A+B-C
A =
     1     0
     0     1
B =
     0     1
     1     0
C =
     1     2
     3     4
D =
     0    -1
    -2    -3
```

The matrices B and C can be multiplied together. Remember that order matters when multiplying matrices. Also, remember that matrix multiplication can only be applied when the number of columns in the first matrix equals the number of rows in the second.

```
>> B*C
ans =
     3     4
     1     2
>> C*B
ans =
     2     1
     4     3
```

A square matrix, like C, can be multiplied by itself several times by raising it to a given power.

```
>> C^3
     ans =
         1    8
         27   64
```

Two matrices can be multiplied element-by-element (element-wise) using the .* operator, if they have the same size. They can also be divided element-by-element using the ./ operator

```
>> B.*C
ans =
     0     2
     3     0
>> B./C
ans =
         0    0.5000
    0.3333         0
```

Each element in a matrix can be raised to a given power, using the element-by-element (element-wise) power.

```
>> C.^3
ans =
     1     8
    27    64
```

The transpose of a matrix is computed using the apostrophe key:

```
>> E = C'
C =
     1     2
     3     4
```

The matrix left division operator (\) solves the system of linear equations $A * x = b$, if the solution exists. For example, consider the system of linear equations:

$$\begin{cases} 4x_1 + 3x_2 &= 5 \\ 2x_2 + x_2 &= 6 \end{cases}$$

that can be written in the matrix form:

$$\begin{bmatrix} 4 & 3 \\ 2 & 1 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} 5 \\ 6 \end{bmatrix}, \text{ where } A = \begin{bmatrix} 4 & 3 \\ 2 & 1 \end{bmatrix}, \; x = \begin{bmatrix} x_1 \\ x_2 \end{bmatrix}, \; b = \begin{bmatrix} 5 \\ 6 \end{bmatrix}$$

The solution of the linear system $A * x = b$, in Matlab, can be obtained as follows:

```
>> A=[4 3; 2 1]; b=[5;6]; x=A\b
x =
    6.5000
   -7.0000
```

6 Complex numbers

Matlab can manipulate complex numbers consisting of a real part and an imaginary part. The basic imaginary unit, equal to square root of -1 is stored as any of the variables i or j. Notice that these two variables are not protected and can be overwritten. A complex number $x = 2+3i$ and its conjugate $y = 2-3i$ can be introduced using either i or j as the imaginary unit:

```
>> x=2+3i, y=2-3j
x =
   2.0000 + 3.0000i
y =
   2.0000 - 3.0000i
```

7 Built-in functions

Matlab includes many standard built-in functions for math operations. The source code for built-in functions is typically not available for the user. To determine the usage of any function, or the input and output arguments, use *help function_name* or *doc function_name*. *Help* will display the information in the command window, while *doc* opens a new window containing the function description, possibly some examples of usage and other information.

```
>> help det
>> doc inv
```

Some common trigonometric functions are shown in Table 2. Notice that the function input arguments as well as the results are expressed in radians.

Table 2: Trigonometric functions

sin(x)	Sine of the elements of x (in radians)
cos(x)	Cosine of the elements of x (in radians)
asin(x)	Arcsine of the elements of x (result in radians)
acos(x)	Arccosine of the elements of x (result in radians)
tan(x)	Tangent of the elements of x (in radians)
atan(x)	Arctangent of the elements of x (result in radians)

An example for computing the sine of the angles 90°, 60° and 30° is given below, where *pi* is a constant incorporated in Matlab.

```
>> angles_degrees=[90 60 30]
angles_degrees =
    90    60    30
>> sin(angles_degrees*pi/180)
ans =
    1.0000    0.8660    0.5000
```

A list of some commonly used mathematical functions is given in Table 3.

The real and imaginary parts and the absolute value of a complex number are computed as shown below:

Table 3: Elementary math functions

abs(x)	Absolute value of the elements of x
sqrt(x)	Square root of x
imag(x)	Imaginary part of x
real(x)	Real part of x
conj(x)	Complex conjugate of x
log(x)	Natural logarithm of the elements of x
log10(x)	Logarithm base 10 of the elements of x
exp(x)	Exponential of the elements of x

```
>> x=4+5j;
>> re=real(x), im=imag(x), ab=abs(x)
re =
     4
im =
     5
ab =
    6.4031
```

Some functions for matrix properties and manipulation are given in Table 4.

Table 4: Matrix properties and matrix manipulation

inv(A)	Inverse of a matrix A
eig(A)	Eigenvalues of the matrix A
det(A)	Determinant of matrix A
rank(A)	Rank of matrix A
eye	Builds an identity matrix
ones, zeros	Builds a ones or zeros array
diag	Builds a diagonal matrix

For a square matrix A compute, for example, the inverse and the determinant:

```
>> A=[1 2;3 4];
>> inv(A)
ans =
   -2.0000    1.0000
    1.5000   -0.5000
>> det(A)
ans =
    -2
```

A 3-by-3 identity matrix and a 1-by-3 array of ones are built as follows:

```
>> B=eye(3)
B =
     1     0     0
     0     1     0
     0     0     1
>> C=ones(1,3)
C =
     1     1     1
```

8 Polynomials

A polynomial is represented in Matlab by a row vector containing the coefficients ordered by descending power. Matlab can interpret a vector of length $n+1$ as an n-th order polynomial. If a polynomial is missing some coefficients, zeros must be entered in the appropriate place in the vector. The polynomials:

$$p(s) = s^4 + 2s^3 - 3s^2 + 4s - 5 \text{ and } q(s) = s^3 + 6$$

are entered as vectors in the following manner:

```
>> p = [1 2 -3 4 -5];
>> q = [1 0 0 6];
```

Some functions to be used for polynomials are given in Table 5.

Table 5: Functions for polynomials

roots(p)	Roots of polynomial p
polyval(p,v)	Value of polynomial p at value v
conv(p,q)	Polynomial multiplication
deconv(p,q)	Polynomial division

For example, the roots of the polynomial $p(s)$ are obtained as shown below:

```
>> roots(p)
ans =
  -3.3719 + 0.0000i
   1.1103 + 0.0000i
   0.1308 + 1.1482i
   0.1308 - 1.1482i
```

To multiply the polynomials $p(s)$ and $q(s)$, use convolution or polynomial multiplication. To divide the polynomial $p(s)$ by $q(s)$ use the function *deconv* that will return the quotient and the remainder. The output of this function is actually a set of two polynomials. The variable names of these polynomials should be enclosed between brackets when calling the function. To determine the usage of this function, type *help deconv.*

```
>> pq=conv(p,q)
pq =
     1     2    -3    13    13   -27    36   -45
>> [qu,re]=deconv(p,q)
qu =
     1     2
re =
     0     0    -3    -5   -23
```

9 Plotting

Matlab has very strong visualization capabilities, from simple two-dimensional to complex three-dimensional plots.

Table 6 shows several functions available for plotting, formatting and annotation.

The *plot* function has different forms, depending on the input arguments.

To plot the vector y versus vector x, use *plot(x,y)*. For example, to plot a sine wave as a function of time, make first a time vector and then compute the *sin* value for each element. The time vector will be a regularly-spaced row vector with values between 0 and 4π, spaced by an increment of 0.1.

Table 6: Plotting

plot(x,y)	Plots vector y versus vector x
mesh, surf	Create a 3-D mesh surface
figure	Creates a new figure window
grid, grid on/off	Toggles a grid on and off in the current figure.
title('text')	Adds 'text' at the top of the current axis
xlabel('text')	Labels the x-axis with 'text'
ylabel('text')	Labels the y-axis with 'text'
legend(string)	Displays the legend
hold, hold on/off	Toggles the hold state on and off in current figure/
subplot	Creates multiple axes in tiled positions

```
>> x=0:0.1:4*pi;
>> y=sin(x);
>> plot(x,y)
```

To plot multiple sets of data, use the general form *plot(x1,y1, x2,y2, ...)*. A sine and a cosine wave are created on the same plot in a new figure window and grid is added:

```
>> x=0:0.1:4*pi;
>> figure, plot(x,sin(x),x,cos(x))
>> grid on
```

If a figure window is open, a new plot command will clear the figure and display the new one into the current window. The function *figure* opens a new window and displays the new graph.

The line colors are automatically chosen by the function in the order specified by a Matlab variable. However, the user can change the line style and color if the *plot* function is called in the general form *plot(x1,y1,'options1', x2,y2, 'options2', ...)*. All options for various line types, markers and colors are displayed by *help plot*.

For example, the sine and cosine functions from the previous example, are plotted now using a red marker $*$ for the first set of data and a black dotted line for the second one. A legend is also displayed on the figure.

```
>> x=0:0.1:4*pi;
>> figure, plot(x,sin(x),'r*',x,cos(x),'k:')
>> grid on
>> legend('sine', 'cosine')
```

A title and axis labels can be added to a plot. In the following example the functions x^2 and x^3 are plotted against $x \in [-1, 1]$. The axes are labeled and a title is also added.

```
>> x=-1:0.1:1;
>> figure, plot(x, x.^2, x, x.^3)
>> grid on
>> xlabel('x'), ylabel('x^2, x^3'), title('x squared and x cubed')
```

Let us point out that the element-wise power operator (`.^`)was used for computing x^2 and x^3. If x was created as a row vector, a simple power operator will try to multiply the row vector by itself in the linear algebra sense resulting in an error. In this case, however, the vector to be plotted against the row vector x is one containing the squared (or cubed) values of every element in x, computed using the dot power operator.

If wanted to add a new line plot to an existing graph use the *hold on* command. Then reset the hold state to off with *hold off*.

```
>> x=-5:0.2:5;
>> figure, plot(x,sin(x), 'r'), grid on
>> hold on
>> plot(x,cos(x), 'm')
>> hold off
```

More than one sub-figures can be created in one figure window using the *subplot* command. *Subplot(m,n,p)* or *subplot(mnp)* will divide the current figure into a matrix of m-by-n rectangular sub-figures and sets the current axes in the p-th position, numbered row-wise. In the following example, a figure window is divided into a matrix with three rows and one column and sine functions of various frequencies are displayed in each one.

```
>> x=-2:0.1:2;
>> subplot(311), plot(x,sin(2*x)), xlabel('x'), ylabel('sin(2x)')
>> subplot(312), plot(x,sin(3*x)), xlabel('x'), ylabel('sin(3x)')
>> subplot(313), plot(x,sin(4*x)), xlabel('x'), ylabel('sin(4x)')
```

10 M-files

The m-files are plain text files containing Matlab commands. They can be created using the Matlab editor that will save them with the default extension *.m*. Two types of m-files can be written in Matlab:

- Scripts - store statements as those introduced in the command window and execute them sequentially. Scripts operate on data in the workspace, i.e. all variables used in a script are visible in the workspace
- User-defined functions - are program files that can accept inputs and return outputs. The variables used inside a function are local to the function.

Matlab will execute program files searching first the current directory and then the search path. All the directories in the search path are displayed when typing *path* in the command prompt.

Before editing and running new scripts or calling functions, set the current directory to your own working directory in the main Matlab interface.

It is important that the filenames do not match other Matlab functions that may be used. For example, it is strongly **not advisable** to call a script *plot.m*, *sin.m* or *roots.m*.

10.1 Scripts

To create and run a new script:

- Open the Matlab editor or select *New script* in the main Matlab interface
- Enter some commands as they would be introduced in the command window.
- Save the script in the current working directory
- Run the script in one of the following ways:
  - In the Matlab editor press *F5*
  - In the Matlab editor press the green button *Save and run*
  - Enter the filename, without the extension, to the Matlab prompt in the Command window.

It is strongly recommended to start a script with three commands: *close all*, *clear all* and *clc*. The command *close all* closes all open figure windows, the command *clear all* clears all data stored in the variables from the workspace and *clc* clears the command window.

Comments can be inserted into the program files using the percent (%) symbol. They can appear anywhere in a script (or user-defined function), and can be placed also to the end of a line of code.

A script example is presented below.

Listing 1: script_example.m

```
close all
clear all
clc
%--------------------------------------------------------------
% plot three sine functions with phase shift between them
% over a time interval between 1 and 15
%--------------------------------------------------------------
t=0:0.01:15;     % input the time vector
s1=sin(t);       % create the first sine
s2=sin(t-1);     % create the second sine shifted
s3=sin(t-2);     % create the third sine shifted some more

% place all three line plots in the same figure window
% add grid, labels on the axes and title
plot(t,s1,t,s2,t,s3, 'LineWidth',2), grid on
xlabel('time, t')
ylabel('sin(x), sin(x-1), sin(x-2)')
title('three sine functions')
```

10.2 User-defined functions

Functions are also m-files consisting of commands, but they can accept one or more input arguments and may return one or more output arguments.

Functions operate on variables stored in a local workspace, separate from the workspace that can be accessed in the main interface. The syntax of a function statement (that is also the first line in the m-file) is:

function [output arguments] = *function_name*(input arguments)

Save each function in a separate m-file. The name of the m-file has to be the same as the *function_name*.

A function that computes the sum and the product of two numbers can be written as follows:

Listing 2: sum_product.m

```
function [sum, product] = sum_product(x,y)
%sum_product - returns the sum and the product of two numbers
% input arguments:  x,y - scalar values
% output aguments:  sum  - the sum of x and y
%                   product: the product of x and y

sum = x+y;
product = x*y;

end
```

The function can be called from another function, a script file or in the command window:

```
>> [s,p]=sum_product(2,3)
or
>> [x,y]=sum_product(-4, 5)
```

11 Loops and conditional statements

Matlab language provides several loops and conditional statements for programming. The commonly used ones have the syntax:

- *for*:

for index = initial_value:increment:final_value,

statements

end

- *while*:

  while condition,
      statements
  end

- *if*:

  if condition,
      statements
  elseif condition,
      statements
  else
      statements
  end

Conditions can be expressed using relational operators. For a complete list type *help relop*. A list of commonly used operators is given in Table 7.

Symbol	Description	Symbol	Description
<	Less than	&	Element-wise logical AND
>	Greater than	|	Element-wise logical OR
==	Equal to	~	logical NOT
~=	Not equal to		
<=	Less than or equal to		
>=	Greater than or equal to		

Table 7: Relational operators

Consider for example a polynomial:

$$p(x) = x^3 + 2x^2 + 3x + k$$

where k is a variable coefficient that may take the values $k \in \{-2,\ -1,\ 0,\ 1,\ 2\}$. The following script computes the roots of the polynomial for all values of k.

Listing 3: for_example.m

```
close all
clear all
clc
% roots of a polynomial with a variable coefficient
for k = -2:2,                % k takes sequentially the values: -2, -1, 0, 1, 2
    polynomial = [1 2 3 k]; % input the polynomial coefficients
    roots(polynomial)        % compute and display the roots in the command window
end;
```

As an example for using an *if* statement, the following script will generate a random number $r \in (0,\ 1)$, checks whether the number is less than 0.5, between 0.5 and 0.8, or greater than 0.8 and displays a message for each case.

Listing 4: if_example.m

```
close all
clear all
clc
r = rand(1);    % generate a random number in the interval (0,1)
disp('----------------------------------------')
disp(r)         % displays the number
if r < 0.5,
    disp('The number is less than 0.5');
elseif r <= 0.8
    disp('The number is in the interval [0.5, 0.8]')
```

```
else
    disp('The number is greater than 0.8')
end
disp('--------------------------------------------')
```

12 Practice exercises

Exercise 1: Plot the roots of the polynomial $p(x) = x^3 + 2x^2 + 3x + k$, for the values of k: $0, 0.5, 1, 1.5, ..., 10$ in the same graphic window, using the marker "*". Use the Matlab functions: *for, roots, real, imag, plot, hold on, pause.*

Exercise 2: Write a function *plotfreq(n)* that will display the following functions on the same plot, over a time interval t = [0, 10].

$$f_1(t) = e^{-t}, \quad f_2(t) = sin(2\pi nt), \quad f_3(t) = f_1(t) \cdot f_2(t)$$

The function input argument n is the frequency of the sine function (f_2). Call the function for various values of $n \in [0.5,\ 5]$.

Exercise 3: Write a function that generates the first n Fibonacci numbers. The function will be called: *fibonacci(n)* and will return the vector: 1, 1, 2, 3, 5, ..., N (n numbers).

$$F_1 = 1, \ \ F_2 = 1, \ \ ..., \ \ F_k = F_{k-1} + F_{k-2}$$

Exercise 4: Draw a *fern* fractal in Matlab, [1]. Use the following algorithm:

References

[1] Michael Barnsley. *Fractals Everywhere.* Academic Press, 1993.

[2] MathWorks. MATLAB, The Language of Technical Computing. `https://www.mathworks.com/help/matlab`, 2017.

Algorithm 1 Fractal fern

Define the *number_of_iterations* = 10000

Define the matrices:

$$A_1 = \begin{bmatrix} 0 & 0 \\ 0 & 0.16 \end{bmatrix},\ A_2 = \begin{bmatrix} 0.85 & 0.04 \\ -0.04 & 0.85 \end{bmatrix},\ A_3 = \begin{bmatrix} 0.2 & -0.26 \\ 0.23 & 0.22 \end{bmatrix},\ A_4 = \begin{bmatrix} -0.15 & 0.28 \\ 0.26 & 0.24 \end{bmatrix},$$

$$t_1 = \begin{bmatrix} 0 \\ 0 \end{bmatrix},\ t_2 = \begin{bmatrix} 0 \\ 1.6 \end{bmatrix},\ t_3 = \begin{bmatrix} 0 \\ 1.6 \end{bmatrix},\ t_4 = \begin{bmatrix} 0 \\ 0.44 \end{bmatrix}.$$

Set the initial point $(0, 0)$ as a column vector with 2 coordinates: $X = [0; 0]$. This will be the first column in the coordinate matrix X, where each column will contain the coordinates of the points to be plotted.

Set the current point equal to the initial point: $v = X$

for $i = 1$ to $number_of_iterations - 1$ **do**

  Generate a random number n between 0 and 1 (see the Matlab *rand* function)

  Compute a new point v:

  if $n < 0.01$ **then**

    $v = A_1 \cdot v + t_1$

  else

    if $n < 0.8$ **then**

      $v = A_2 \cdot v + t_2$

    else

      if $n < 0.9$ **then**

        $v = A_3 \cdot v + t_3$

      else

        $v = A_4 \cdot v + t_4$

      end if

    end if

  end if

  Append the current point v to the coordinate matrix X (in Matlab write: $X = [X\ \ v]$)

end for

Plot the elements in the second row of the coordinate matrix X versus the elements in the first row, using a green marker ”.”