|  |
| --- |
| School of Electronic & Electrical Engineering  FACULTY OF ENGINEERING |

ELEC5681 – Programming

(MATLAB Lab 4 notes)

Function program; work with Matrices in MATLAB, Finding roots, (and optional: Differentiation)

**Contents**

[1. Functions 3](#_Toc20749437)

[2. Matrices in MATLAB 5](#_Toc20749438)

[Matrix operations 5](#_Toc20749439)

[Matrix types 6](#_Toc20749440)

[Matrix transpose 7](#_Toc20749441)

[Matrix determinant 7](#_Toc20749442)

[Matrix inversion 7](#_Toc20749443)

[Solving systems of linear equations 8](#_Toc20749444)

[3. Finding roots (Bisection, MATLAB fzero and Newton’s method) 10](#_Toc20749446)

4. Differentiation ………………………………………………………………………………………………. 12

# Functions

**Function Files**

# Function Files

Functions are useful for complicated program to be broken down into smaller parts. Also If series of statements is to be used many times

Syntax:

function [output variables]=function\_name(input variables)

Statements describing function

…

end

Function file name is to be saved as function\_name.m and the first executable statement must be “function”.

If there is more than one output value, one needs to put output variables in [] bracket. If there is only one output variable the brackets are not necessary. Do not use input command within a function, since function arguments are its inputs. Matlab does not have specifiers of data types for its variables, therefor if your function operates with whole numbers (integers) you need to provide protection of the function in it.

Variables defined and manipulated inside the function subroutine are local to the function.

Please use Matlab help function in the command window to read more about syntaxes. Please read also ELEC5681 MATLAB Lab3 notes.

Example: Function for sum of n integer positive numbers

function mysum=my\_sum\_fun(n)

if (n<0)

error('Only positive input is accepted!')

end

if floor(n)~=n

error('Only integer values are accepted')

end

if numel(n)>1

error('Only single integer values are accepted, array/matrix input is prohibited')

end

mysum=0;

for i=1:n

mysum=mysum+i;

end

end

The first two if statements protect the function from negative and decimal input, and the third if statement protects the function from an array/matrix input. Since Matlab does not have data type specifiers, unskilled user may attempt sending illogical input to the function. Note that this can create a lot of execution problems, for instance, **for** (or **while**) loop would not run correctly if n was an array or a matrix, however you will not get an error message! The first iteration when i=1 will execute as mysum=mysum+1; and output of this function will be mysum=1 with no error messages from Matlab! As a creator of the function you need to be very careful and predict when you actually want your input to be an array and when not. Matlab has “dot” operators as ./ and .\* to deal with the arrays which is useful when dealing with mathematical expressions, however if you forget the dot, illogical output may occur with no errors on the screen! Generally “dot” operations simply save you from writing a **for** loop that iterates through the array and performs \* or / operation on each element.

Make sure you fully understand the usage of these operators.

* Write down MATLAB function subroutine which defines the following functions (either in one .m file or separately):

Write then the main MATLAB program, which evaluates functions for x between -4π and 4π, and plot the functions.

* Construct a simple MATLAB function program ***my\_factorial*** which calculates product of the first *n* positive integer numbers (i.e. find factorial *n!*).

*- Make sure you protect the code from illogical use (n must be positive integer)*

*- Test the function by comparing with built in MATLAB function factorial(n)*

*- Make sure you test your function for any limitations.*

* If you check the MATLAB help page on **plot** function, you’ll discover numerous ways how to set up your graphs by typing code. If you learn how to do this once, you can then copy paste the set-up code for every future use when you need nicer graphs for your report (the default behaviour of MATLAB’s **plot** function has very thin lines and very small font for example). A better approach is to write a custom function that plots your data and contains such copy paste code.

Construct a function that extends MATLAB’s plot function. The function needs to have at   
 least following arguments:

function z=better\_plot(x,y,LineSpec,LineThickness,xlab,ylab,FontSize,enablegrid)

where x and y are data arrays that you wish to plot.

LineSpec is a string defining colour, marker and line type (same as in ordinary **plot** function)

LineThickness is a positive integer number - thickness of the data line you wish

xlab is a string describing the label of your x axis

ylab is a string describing the label of your y axis

FontSize is the size of font you wish to use for your axes

Enablegrid is a logical value that disables grid (0) or enables it (any numeric value that is not 0)

Test this function on data of your choice.

*Note: Since MATLAB is defaulting data types, it is relatively hard to validate if user is really sending input as string (word in single quotes as ‘word’). For simplicity, assume all input is valid, so you do not need to worry about this, however do pay attention when testing the function, all string arguments in this function need to be provided in single quotes and LineSpec variable needs to follow syntax that MATLAB’s own LineSpec properties follow (check help page on* **plot** *function).*

*MATLAB is also not allowing to leave out a function argument (setting a default value behaviour is complicated in MATLAB in comparison to C/C++ programming language), so even when you do not wish to set some variable when using the function above, you still need to send empty string as the argument (i.e empty quotes ‘* **’** *)*

Example of how to use the function better\_plot (that you need to create):

x=linspace(0,2\*pi);

y=sin(x);

figure (1)

better\_plot(x,y,'',5,'x','y',24,1)

figure(2)

better\_plot(x,y,'-\*r',7,'x','sin(x)',18,0)

Code output:

Figure 1: Figure 2:



Notice how all settings got applied, if you do not want some setting to apply, just pass empty string to the function (as is was done for LineSpec for figure 1.

All settings for figure 2 got applied as instructed – Line spec is saying graph line should be red full line with \* as data markers, ylabel is saying sin(x), there is no grid, and font is smaller.

In both examples, this is still low plot quality as text on axes is hardly readable and data lines are too thin, however depending on the size of figures you wish to place in the report, you can easily tweak this function so it generates great figures. You can also add more plot settings, as setting text on axes to be bold etc.

Note that if you wish some setting to always be applied and not controlled as function argument, you can do that. Just do not pass as an argument, and define it in the function itself as all variables you create in the function files are local (visible only in that function file). Also never name a variable in your function the same as some variable that is passed as the function argument, because then you would be overwriting the function argument.

In this way you can customize plot function settings however you like or need and this is very useful in future whenever you need to create graphs for your reports!

* Using ***my\_factorial*** function from the previous subtask, construct new MATLAB function program ***my\_sin*** which evaluates using Maclaurin expansion:

The number of terms *n* in this expansion needs to be provided by the user of the function as an positive integer value and value of *x* must be a single numeric value and not an array or   
matrix. Examine numerical limitations of this function and comment on their origin.

* Write a main MATLAB program in which you test the ***my\_sin***function from the previous subtask by comparing with MATLAB built in function *sin(x).* Create a variable that represents the numerical error () and then examine the effect of number of terms in your *my\_sin* function and the effect of *x* –axis you are using. You may find the ***semilogy()*** command more useful than ***plot()***. This displays the y-axis on a logarithmic scale.

*- Test your code for any limitations and comment on their origin*

*- If you discovered some limitations, make sure you protect the code from them.*

*Note: Read more about numerical overflow. In MATLAB variables have double data type by default and every time a function returns a value above or below the double data type range the variable becomes Inf, for example exp(709)=* *8.2184e+307 and exp(710)=Inf. The issue with MATLAB is that it will not warn you about Infs and NaNs (not a number variable) and it is up to the programmer to search for potential overflow and protect the code.*

*Hint: Function you need to create in has two inputs: number\_of\_terms and x. Both of these should be single numeric values (positive integer and double data type). In order to examine their effect in this example you need to run your code for multiple values of x and number\_of\_terms. Expectation is to do this automatically. The analytic approach is:*

*a) Fix x to some value, create array consisting of various number\_of\_terms and run a loop   
 calling your function for each value, calculate the numerical error, plot the numerical   
 error versus array of various numbers of terms and comment on your observations.*

*b) For examining effect of x, you should fix number\_of\_terms to some value, create array of   
 x values, run a loop for each value, calculate the numerical error, plot numerical error   
 versus array of x values and comment on your observations.*

*c) Finally, if you noticed some interlay of the effects you should repeat steps a) and b) by   
 changing the fixed terms or array range and then explaining how both parameters affect   
 your function at the same time, and what are the limitations.*

*.*

# Matrices in MATLAB

*(each blue shaded task is worth 1 mark)*

An array is effectively a **row vector** or a **1 x *n*** matrix i.e.

>> rowvector = [3 5 7]

rowvector =

3 5 7

In MATLAB you can enter a ***n* x 1** matrix (or column vector) using the following syntax

>> columnvector = [2;4;6]

columnvector =

2

4

6

Using the above information on inputting matrices as a starting point, enter the following matrices in MATLAB.

## Matrix operations

You can carry out basic operations on matrices such as addition/subtraction and multiplication using MATLAB but you need to be careful regarding the matrix dimensions. You can find out the matrix dimensions using the size() command i.e.

>> size(A)

ans =

3 3

>> size(B)

ans =

2 3

Try and add *A* and *B* together. What happens? You should get an error message like the following:

>> A+B

??? Error using ==> plus

Matrix dimensions must agree.

You can only add matrices of the same size. Similarly if you try and multiply *A* and *B*, you will also get an error:

>> A\*B

??? Error using ==> mtimes

Inner matrix dimensions must agree.

You have probably seen this error before when you accidently used the \* (or /) operation instead of .\* (or ./) when using arrays. Remember, the dot . tells MATLAB to operate on the individual array elements rather than the arrays as a whole. In this case, we are trying to multiply (3 x 3) x (2 x 3) matrices together. In this case, our inner dimensions (underlined) are 3 and 2 and so they don’t agree and multiplication is not possible. What happens if we multiply them in the opposite order?

>> B\*A

ans =

50 44 14

188 170 68

This time we are doing (2 x 3) x (3 x 3) and so our inner dimensions agree. Remember that matrices in general **do not commute!**

Multiplying by a scalar is much simpler i.e. we don’t have to worry about pre- or post-multiplication

>> 2\*A

ans =

8 10 12

22 18 2

16 14 4

## Matrix types

A diagonal matrix is a square matrix in which all elements are equal to zero apart from the main diagonal. Suppose you want to enter a diagonal matrix you could do the following:

>> C = [6 0 0;0 3 0;0 0 9]

C =

6 0 0

0 3 0

0 0 9

What happens if you wanted to enter a 4 x 4, or a 100 x 100, or even a 1000 x 1000 diagonal matrix (which can easily occur in some complex Engineering problems)? This would very quickly get very messy.

* Use the help files to find the command ***diag*** which makes it easier to enter diagonal matrices and use it to produce a 5 x 5 diagonal matrix. (A *n* x *n* squarematrix is also known as a square matrix of order *n*). Don’t forget to copy-paste MATLAB input/output commands in your reports (e.g use snipping tool)

The unit (or identity) matrix is a special case of diagonal matrix in which all the elements on the main diagonal are equal to one.

* Use the MATLAB help to find command ***eye*** do this and use it to produce a unit matrix of order 3. (If you use the same command as you did to make the diagonal matrix you won’t get any marks!)

A zero (or null) matrix is a matrix in which all elements are equal to zero.

* Use the MATLAB help to find command ***zeros*** to do this and use it to produce a 3 x 2 zero matrix. (If you use the same command as you did to make the diagonal matrix and set the diagonal to be equal to zero you won’t get any marks! )

As the name suggests, a matrix of one’s is a matrix in which all the elements are equal to one. This is sometimes called a unit matrix so make sure you don’t get it mixed up with an identity matrix!

* Use the MATLAB help to find command ***ones*** to produce at matrix of ones and use it to produce a   
   2 x 4 unit matrix.

Create following matrices in MATLAB command line;

,

Perform the following operations in MATLAB (include both your input/output commands in the report):

* , What’s the difference between and , what ^ and .^ do?
* , What’s the difference between and ?

## Matrix transpose

By interchanging the rows and columns in a matrix we obtain the matrix transpose.

Create the following matrix:

* Use the MATLAB help to find out how to calculate the transpose of a matrix ***A, (command A’*** ) and transpose the matrix A. Calculate **A**(**AT**). What do you notice about the resulting matrix?

## Matrix determinant

The determinant of a matrix ***A***is a number and often denoted by |***A***| or det ***A****.*

* Find the determinant of ***A*** using the MATLAB help documents. Is ***A*** singular?

## Matrix inversion

Being able to invert a matrix allows us to do matrix division and allows us to solve systems of linear equations.

* Calculate the inverse of a matrix ***A***using the MATLAB help documents to find about command   
   ***inv(A)***

## Solving systems of linear equations

We can write the following system of linear equations in matrix form

**A x = b**

If we know ***A*-1** exists and the determinant of ***A*** is non-zero (i.e ***A*** is non-singular) and so we know a unique solution exists which can be found using ***x = A-1 b***

* Use MATLAB to calculate the values of *x1, x2* and *x3*.

# Finding roots (Bisection, MATLAB fzero and Newton’s method)

Please see first ‘ELEC5681M Lab 3 notes.

* Using the code for the *bisection method* provided in Matlab Lab3 notes calculate the real roots of the function using 5 iterations. To determine the intervals where the function changes sign use the graphical method.

*- Discuss interval width and number of iterations.*

* Read about MATLAB ***fzero*** program. Write a program that performs incremental search (finds all zeros of a function automatically) by using ***fzero.*** Incremental search checks the root of a function only on intervals where function changes sign. Test your program by finding all zeros of function given above.

Note that input of this program should be general function *f.* You may use ***input()*** command and instruct your user to provide you a function of x in format ***@(x) f(x)****.* Thus inputting the function would be possible if user types, for example, ***@(x) x^4-2\*x^2+x*** after being prompted by input command

* Review *Newton’s* method function code from the lab notes. Focus on ***mynewtontol***function specifically and write a MATLAB code that finds all roots of the given function automatically. In order to do this you need to make array for the x-axis and send each point of *x(i)* as initial guess to *mynewtontol*  function, your output (the roots) will also be an array. Use tolerance of 1×10-6.  
     
   - Use your code to find the roots of the function ***f(x) = x3 +4x2 -7x – 20*** in the interval between -6 and 6.Plot the function for visual check of your code. *What do you notice about the output of your code? Why do you have repetitive roots?* Check MATLAB’s *round* and *unique* function and combine them in order to filter repetitive values. *What are the issues with this filtration?*- In order to avoid repetition of the roots, modify your code so that you send initial guess only when your function changes sign doing, therefore, the incremental search algorithm and rerun your code. *What do you notice now about the output of your code?*  
    
  - Furthermore test your code for the function ***f(x) = x2 - 4x +4***. Plot the function to check where the root is.  
    
  *-* Discuss incremental search and bracketing procedure and what are potential problems. Illustrate potential incremental search hazards plotting in MATLAB different functions of your choice.
* Write a MATLAB program that determines how many iterations *Newton’s method* takes to have the tolerance 1×10-6 with various initial values for the root ***x0***. You will need to make x-axis array, send it to ***mynewtontol***function and output number of iterations for every guess *x(i)*.   
    
  - Test your code for the function  ***f(x) = x3 +4x2 -7x – 20*** and plot number of iterations needed for different guesses. *What do you notice?*   
    
  *-* Review the theory of Newton’s method and plot **f(x) / f’(x)**.Compare this figure with your iterations figure, what do you notice? *Can you explain why this happened?*- Test your code for the function ***f(x) = x2 - 4x +4*** and repeat the procedure you did with the previous function. *What do you notice now?*
* Write a MATLAB program that determines how many iterations *Newton’s method* takes to satisfy various tolerances ***tol*.**In this task tolerance needs to be an array (for example from 1×10-14 to 0.1 using command: tol=10.^[-14:-1] ), while initial guess is fixed.

- Run your code for the function ***f(x) = x3 +4x2 -7x – 20,*** use *x*0 = 2 as the initial guess in the *Newton’s method.* Plot (use *semilogx* function) number of iterations needed for different tolerances. *What do you notice?*

# Optional: Differentiation (not be assessed in ELEC5681M MATLAB test questions)

To differentiate to give you is easy. In MATLAB, we will first cover how to do this numerically, i.e. to get the numerical value of the derivative at a given point.

For example, say we want to calculate the derivative for at . Of course, we can do this in our head and get . So how would we do this in MATLAB?

>> x=5;

>> dx=0.1;

>> fx=x^2;

>> fxx=(x+dx)^2;

>> dfdx=(fxx-fx)/dx

dfdx =

10.099999999999980

First we have defined our variable *x*. We have then defined the small step *δx*. Then we calculated the function at *x* and at *x+ δx*. Finally we calculate the derivative to give us an answer of 10.0999999999. (*NOTE: In order to display so many decimal places, use MATLAB command* Format long*)*. This is relatively close to the exact answer of 10. So how do we make it more accurate? - By making the step size smaller.

>> x=5;

>> dx=0.01;

>> fx=x^2;

>> fxx=(x+dx)^2;

>> dfdx=(fxx-fx)/dx

dfdx =

10.009999999999764

Note how this is even closer to the exact answer.

* Write a MATLAB function program to numerically differentiate (do not use the analytical result within this function) . The inputs of the function should be value ***x*** *and step* ***dx.***

Write a main code in which you analyse the effect of the step ***dx*** while ***x*** is being fixed at ***x=****,*by plotting (use *loglog())* the numerical error (ther error is absolute value of difference between exact analytical result and numerical result: ) for step sizes 1, 0.1, 10-2, 10-3...10-6.

*- Create an array x=[-2pi, 2pi] and use your function to find the derivative of*

*for every x for step sizes 2, 1, 0.5, 1, 1e-3. Plot on the same graph the numerical error versus x for different step sizes. What do you notice? Where is error the highest?*

*Hint: To plot a graph with multiple traces you need to use hold command. For the second graph   
 you therefore need to calculate the numerical error versus x while dx is being fixed, plot it,   
 hold the plot, and repeat the previous code for different dx. The neat way to do this is to use a loop that iterates through dx values, or you can do it manually, but keep in mind that such   
 solutions neither tidy nor optimal.*

Now comes the really useful part. You are now going to learn how to *symbolically* differentiate a function. Whereas numerical differentiation only gives you the value of the derivative at a particular point, differentiating a function symbolically gives you the actual function itself! i.e. considering if we want to differentiate .

>> syms x

>> y=sin(x);

>> diff(y)

ans =

cos(x)

First we define a *symbolic object* called *x*. We then write the function we want to differentiate, then use the diff() function to give us the derivative. This doesn’t just work with simple functions either. Consider. You have learned how to do this using the product rule; to do this in MATLAB you would simply do the following

>> syms x

>> y=sin(x)\*exp(5\*x)

>> diff(y)

ans =

exp(5\*x)\*cos(x) + 5\*exp(5\*x)\*sin(x)

* Symbolically differentiate using MATLAB and check with the solution derived by hand.

**-** Store the derivative in a variable and use the pretty() function. What does it do?

Remember though, this is useful for checking your answers etc.