ELEN E3084: Signals and Systems Lab
Lab I: Introduction to MATLAB

1 Introduction

The purpose of this lab is to familiarize us with basic MATLAB functionality. We will do that by focusing on some easy to follow tasks that will help us feel comfortable with MATLAB both as a programming language as well as a development environment. For a more comprehensive introduction we recommend reading the free on line tutorial *Getting Started* from Mathworks, the makers of MATLAB.

```
http://www.mathworks.com/access/helpdesk/help/techdoc/matlab.shtml
```

However, after we gain some experience we will rarely rely on particular tutorials or books, since the available online help and the ability of searching for new options is satisfactory for most problems. In this as well as the subsequent labs we will use MATLAB version 6.5 also known as R13.

2 Running and configuring MATLAB

2.1 Configuring the integrated development environment

MATLAB in its early versions started as a simple interactive UNIX shell in which users could execute mathematical operations. In its most current version, MATLAB includes a highly configurable Integrated Development Environment (IDE) which facilitates writing, testing, debugging and even optimizing the calculations we need to perform. We can run MATLAB in UNIX by typing:

```
matlab
```

at the shell's command prompt. A graphical interface with 3 main windows appears. The *Command Window* with the command prompt `>>` is where we execute our code. The *Launch Pad* window provides easy access to tools, demos, and documentation while the *Command History* window displays the commands we have recently typed. History is very useful since by simple scrolling we can look at our work several days back, an electronic form of lab notebook. Under the *View* menu we can find all the windows that can be displayed. We can open any those windows and selectively float, dock or even tab list them inside the IDE. Any changes we make to the IDE will be saved when we quit MATLAB by typing:

```
>> quit
```

at MATLAB's prompt or by selecting *Exit* under the *File* menu. One last thing to note is that if we don't need the IDE we can run MATLAB using the command:

```
matlab -nosplash -nodesktop
```

at the UNIX shell.

To Do 1

Customize the IDE.
We will now make our MATLAB IDE look like the one in the figure 1. We can do that in a few easy steps. First close the *Launch Pad* window, as it is not very useful, by clicking at the x box. Then open the *Workspace* window which you can find under the *View* menu. Then open an m-file, for example `lpc.m`, by typing in the command prompt:

```
>> open lpc;
```

This will launch the MATLAB file editor which we should dock in the main IDE by selecting *Dock* under the *View* menu. Notice that when we open a second file, for example `prony.m`, by typing:

```
>> open prony;
```

it automatically opens as a tab list. In order to move all the docked windows in the IDE we drag and drop them to the desired position. The placeholder changes automatically as we move the windows to denote the docking position.

The benefits of having our IDE setup this way will be apparent as we start writing, testing and debugging our code. Make sure that after moving around the windows, your IDE looks like the one in figure 1 and then type `quit` to close MATLAB and save your IDE configuration.

2.2 Setting the path

MATLAB encapsulates most of its functionality in system- or user-defined text files that contain commands. Those files (like `lpc.m` in the previous section) are called m-files. In order for MATLAB to be able to find those files they need be in the so-called path. Since we will be writing some basic functions for this lab series it is wise to create a directory structure and organize our code for future reference. This is a good practice for all the courses that require MATLAB programming.

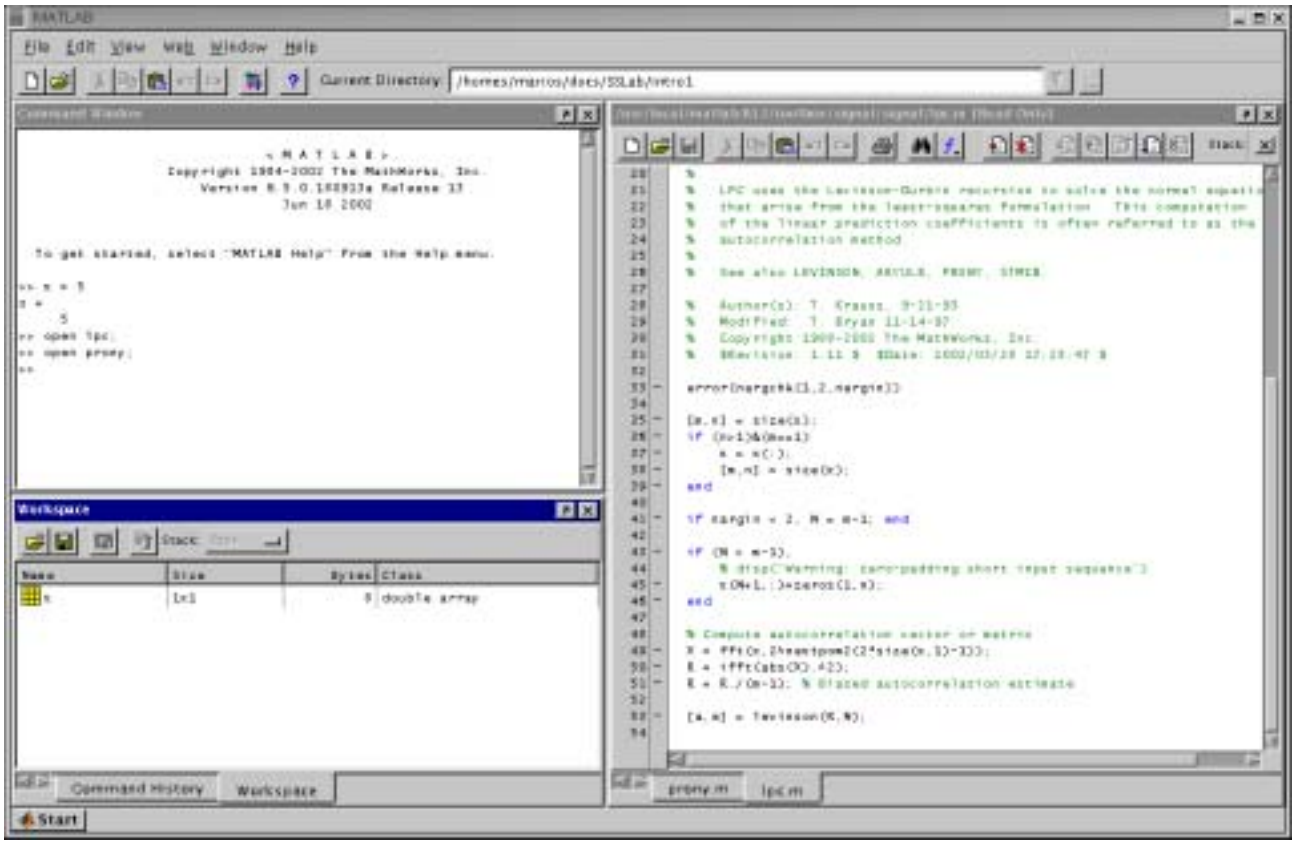

Figure 1: A customized development environment

To Do 2

Create a directory structure and set the path.
First we make sure we are in our home directory by typing:

```
>> cd ~;
```

Then we create the following directories and files.

```
>> mkdir matlab;
>> cd matlab;
>> !touch startup.m;
>> mkdir elen_e3084;
>> cd elen_e3084;
>> mkdir lab1;
>> mkdir lab2;
>> mkdir lab3;
>> mkdir lab4;
```

```
>> mkdir lab5;
>> mkdir lab6;
```

Now we have a complete directory structure were we can store our work for all the subsequent lab exercises. Moreover we created a file called `startup.m` in the `matlab` directory which has a special functionality. This file is automatically called by MATLAB at launch. This is the best place were we can set our path so that MATLAB can find the m-files we will write.

Edit `startup.m` by typing:

```
>> cd ~/matlab;
>> edit startup;
```

We can see how our IDE configuration helps us. `startup.m` is opened by the editor in a tab right next to `lpc.m` and `prony.m` m-files something that facilitates editing since we can be testing code in the *Command Window* and copy-paste it in the editor. Next in `startup.m` we type the following commands:

```
disp('This startup.m file has been modified for the E3084 lab');
addpath('~/matlab/elen_e3084/');
addpath('~/matlab/elen_e3084/lab1');
addpath('~/matlab/elen_e3084/lab2');
addpath('~/matlab/elen_e3084/lab3');
addpath('~/matlab/elen_e3084/lab4');
addpath('~/matlab/elen_e3084/lab5');
addpath('~/matlab/elen_e3084/lab6');
```

Save the `startup.m` file. Now typing:

```
>> startup;
```

at the command prompt executes the file we just saved and the directories we created are now in the path and thus searchable by MATLAB. We essentially created our first m-file. Show the TA the `startup.m` file.

Finally we will now start a diary function that will capture all our commands from now on. Go to the `lab1` directory using the `cd` command the same way as in UNIX. Then type:

```
>> diary on;
```

Please note that whenever you see examples of code in the sections that follow you should experiment with them and try to reproduce some of them. All the commands you type will be collected by the diary function and will have to be submitted at the end of the lab session. This is just a simple way for us to see that you spent some time experimenting and it's not supposed to be a hard grading policy.

3 Elements of programming

3.1 Getting help

Before we start presenting the basic programming constructs we should feel comfortable with the help system as it will prove to be our best friend. The most basic command we will use is `help`. Typing:

```
>> help help;
```

at the MATLAB prompt we get:

```
 HELP On-line help, display text at command line.
    HELP, by itself, lists all primary help topics.  Each primary topic
    corresponds to a directory name on the MATLABPATH.

    "HELP TOPIC" gives help on the specified topic.  The topic can be a
    command name, a directory name, or a MATLABPATH relative partial
    pathname (see HELP PARTIALPATH).  If it is a command name, HELP
    displays information on that command.  If it is a directory name,
    HELP displays the Table-Of-Contents for the specified directory.
    For example, "help general" and "help matlab/general" both list
    the Table-Of-Contents for the directory toolbox/matlab/general.

    HELP FUN displays the help for the function FUN.

    T = HELP('topic') returns the help text in a '\n' separated string.

    LOOKFOR XYZ looks for the string XYZ in the first comment line
    of the HELP text in all M-files found on the MATLABPATH.  For all
    files in which a match occurs, LOOKFOR displays the matching lines.

    MORE ON causes HELP to pause between screenfuls if the help text
    runs to several screens.

    In the online help, keywords are capitalized to make them stand
    out.  Always type commands in lowercase since all command and
    function names are actually in lowercase.

    For tips on creating help for your m-files 'type help.m'.

    See also LOOKFOR, WHAT, WHICH, DIR, MORE.
```

This is the general style of help. It not only gives a detailed description of the topic often including examples, but refers to related topics as well. For example, if you want further information about the `lookfor` command, entering:

```
>> help lookfor;
```

gives:

```
 LOOKFOR Search all M-files for keyword.
    LOOKFOR XYZ looks for the string XYZ in the first comment line
    (the H1 line) of the HELP text in all M-files found on MATLABPATH.
    For all files in which a match occurs, LOOKFOR displays the H1 line.

    For example, "lookfor inverse" finds at least a dozen matches,
    including the H1 lines containing "inverse hyperbolic cosine"
    "two-dimensional inverse FFT", and "pseudoinverse".
    Contrast this with "which inverse" or "what inverse", which run
    more quickly, but which probably fail to find anything because
    MATLAB does not ordinarily have a function "inverse".

    LOOKFOR XYZ -all  searches the entire first comment block of
    each M-file.

    In summary, WHAT lists the functions in a given directory,
    WHICH finds the directory containing a given function or file, and
    LOOKFOR finds all functions in all directories that might have
    something to do with a given key word.

    See also DIR, HELP, WHO, WHAT, WHICH.
```

This is the style of help that we can get displayed on the prompt. As you probably noticed the screen is cluttered with text and we need to scroll up in order to read the full article. In order to present the text a page at a time we can type:

```
>> more on;
```

in which case the text is presented in pages so its easier to read. We can disable this feature by typing:

```
>> more off;
```

An arguably better way to get help is to use the HTML version. The command `doc` works the same way as `help` with the difference that it launches a window displaying the HTML version of the help file. This way is preferable as links to related commands as well as graphics are included. Typing:

```
>> doc help;
```

gives us the HTML version. Typing `help` without arguments gives us a full list of topics and toolboxes with short descriptions.

3.2 Variables

3.2.1 Defining variables

As we have seen in previous sections we can type commands and execute them in the command prompt `>>`. Now if we type:

```
>> x = 5
x =
     5
```

we define a variable `x` and assign the value 5. Notice that we don't need to define the type (such as string or number) of the variable. MATLAB is able to infer the type. So now when we type:

```
>> x
x =
     5
```

we get the value that we previously assigned. Notice that the variable along with its type and size appear on the *Workspace* window that we previously docked on the main MATLAB IDE.

Any string starting with a letter can be a variable such as `m, a, test, TEST` etc. Strings starting with other characters such as `1alfa, -a or a+` cannot be variables. For example, when we type `1alfa` the error message:

```
>> 1alpha = 5
??? 1alpha = 5
     |
Error: Missing operator, comma, or semicolon.
```

appears. It is important to note that variable names are case sensitive which means that `a` and `A` are two different variables.

3.2.2 Suppressing variables

When we type a command or a variable name, its value appears on the screen. Sometimes this is inconvenient. The display of the value can be suppressed by putting a semicolon after the command or a variable name. In general it is preferable to add the semicolon especially when we define variables so that we don't clutter the command prompt. For example we could have defined the previous variable `x` by typing:

```
>> x = 5;
```

thus suppressing its output.

3.2.3 Clearing variables

The command:

```
>> clear x;
```

clears the variable from the workspace. This is useful for example when we want to reclaim system memory. We can also clear variables when we want to make sure that any old values are not going to be used by accident.

3.2.4 Built-in variables

MATLAB includes a number of special variables that are built in the system. For example, entering `pi` gives us the value of π.

```
>> pi
ans =
    3.1416
```

It is possible to overwrite those special variables. For example, typing `pi = 5` assigns a new value, thus loosing its previous, built-in value. The original values of the built-in variables can be returned by using the `clear` statement as illustrated below:

```
>> pi = 1
pi =
     1
>> i = 2
i =
```

```
     2
>> j = 3
j =
     3
>> ans = 4
ans =
     4
>> clear pi i j ans
>> pi
ans =
    3.1416
>> i
ans =
        0 + 1.0000i
>> j
ans =
        0 + 1.0000i
>> ans
ans =
        0 + 1.0000i
```

3.2.5 Numerical values

MATLAB internally uses 8-byte floating point numbers to store arithmetic variables. This can be verified by looking at the `Workspace` window under the column `Bytes`. However, entering or displaying numbers is subject to formatting for presentation purposes. For example:

```
>> 0.000000000000001
ans =
  1.0000e-015
```

In order for this long decimal number to be displayed better, MATLAB uses mantissa-exponent notation. In fact the exponent can be separated with either `e` or `E`. We can also use this formating as an input method:

```
>> 1.000000e-14
ans =
  1.0000e-014
>> 1.00E-14
ans =
  1.0000e-014
```

```
>> 1E-14
ans =
  1.0000e-014
```

It should be emphasized at this point that the way numbers are displayed does not affect the internal accuracy with which they are stored or manipulated internally by MATLAB. In fact we can increase the accuracy by which variables are displayed by typing:

```
>> format long;
```

in which case the previous examples become:

```
>> 1.000000e-14
ans =
    1.000000000000000e-014
>> 1.00E-14
ans =
    1.000000000000000e-014
>> 1E-14
ans =
    1.000000000000000e-014
```

For a complete description you can get help by typing: `help format`. We can return to the default formating by typing:

```
>> format short;
```

Finally, it is worth noting that MATLAB displays integers whenever possible. For example:

```
>> 3.0 * 2
ans =
     6
>> 6 / 3.0
ans =
     2
```

This is a nice example of how MATLAB is able to infer the type of the variable.

3.2.6 Strings

In addition to numerical variables, a sequence of characters referred to as *strings* or *text* can also be used in MATLAB. Characters enclosed in single quotes (apostrophes) are defined as strings. In the simplest case strings can be used for displaying messages:

```
>> x = 'This is a message'
x =
This is a message
```

Several strings can be concatenated by using square brackets as illustrated below:

```
>> st1 = 'aaa';
>> st2 = 'bb';
>> st3 = 'ccccc';
>> stf = [st1 st2 st3 st2]
stf =
aaabbcccccbb
```

This use of square brackets is essentially a way to create vectors discussed later. The important thing here is that strings inside the square bracket have to e separated by spaces or commas.

Numbers and numerical variables can be converted into strings by using the `num2str` function. The general form is `num2str(x,n)` where `x` is the numerical variable and the integer `n` denotes the maximum digits of precision. For example:

```
>> x = sqrt(2);
>> num2str(x,3)
ans =
1.41
>> num2str(x,12)
ans =
1.41421356237
```

3.2.7 The `disp` command

As discussed variables can be displayed by entering the variable without semicolon. A more flexible, and generally more favored way of displaying variables is to use the `disp(var)` command where `var` stands for an arbitrary variable. For example:

```
>> x = 3;
>> disp(x)
     3
```

Finally, combined messages can be created by converting numerical variables into strings and concatenate them with the string variables containing the message. For example:

```
>> mes = 'The value of x is: ';
>> x = sqrt(2);
>> d = [mes num2str(x,10)];
>> disp(d)
The value of x is: 1.414213562
```

3.3 Operators

One can see MATLAB as a glorified calculator. In this sense operators are the special characters that provide MATLAB the calculator functionality. The most basic operators are:

```
  Arithmetic operators.
    plus       - Plus                              +
    uplus      - Unary plus                        +
    minus      - Minus                             -
    uminus     - Unary minus                       -
    mtimes     - Matrix multiply                   *
    times      - Array multiply                   .*
    mpower     - Matrix power                      ^
    power      - Array power                      .^
    mldivide   - Backslash or left matrix divide   \
    mrdivide   - Slash or right matrix divide      /
    ldivide    - Left array divide                .\
    rdivide    - Right array divide               ./
    kron       - Kronecker tensor product        kron

  Relational operators.
    eq         - Equal                            ==
    ne         - Not equal                        ~=
    lt         - Less than                         <
    gt         - Greater than                      >
    le         - Less than or equal               <=
    ge         - Greater than or equal            >=

  Logical operators.
                 Short-circuit logical AND        &&
                 Short-circuit logical OR         ||
    and        - Element-wise logical AND         &
    or         - Element-wise logical OR          |
    not        - Logical NOT                      ~
    xor        - Logical EXCLUSIVE OR
```

```
    any        - True if any element of vector is nonzero
    all        - True if all elements of vector are nonzero

  Bitwise operators.
    bitand     - Bit-wise AND.
    bitcmp     - Complement bits.
    bitor      - Bit-wise OR.
    bitmax     - Maximum floating point integer.
    bitxor     - Bit-wise XOR.
    bitset     - Set bit.
    bitget     - Get bit.
    bitshift   - Bit-wise shift.

  Set operators.
    union      - Set union.
    unique     - Set unique.
    intersect  - Set intersection.
    setdiff    - Set difference.
    setxor     - Set exclusive-or.
    ismember   - True for set member.
```

The use of most operators is self explanatory but if you need more help in any of them you can use the help system, e.g. `help mtimes` gives:

```
 *   Matrix multiply.
    X*Y is the matrix product of X and Y.  Any scalar (a 1-by-1 matrix)
    may multiply anything.  Otherwise, the number of columns of X must
    equal the number of rows of Y.

    C = MTIMES(A,B) is called for the syntax 'A * B' when A or B is an
    object.

    See also TIMES.
```

You can try to perform some simple operations like:

```
>> a = 5;
>> b = 3;
>> a + b
ans =
     8
>> 2 ^ 4
```

```
ans =
    16
>> 1 | 0
ans =
     1
```

3.4 Elementary functions

The elementary functions can be found by typing `help elfun`. The most basic functions follow.

```
  Elementary math functions.

  Trigonometric.
    sin         - Sine.
    sinh        - Hyperbolic sine.
    asin        - Inverse sine.
    asinh       - Inverse hyperbolic sine.
    cos         - Cosine.
    cosh        - Hyperbolic cosine.
    acos        - Inverse cosine.
    acosh       - Inverse hyperbolic cosine.
    tan         - Tangent.
    tanh        - Hyperbolic tangent.
    atan        - Inverse tangent.
    atan2       - Four quadrant inverse tangent.
    atanh       - Inverse hyperbolic tangent.
    sec         - Secant.
    sech        - Hyperbolic secant.
    asec        - Inverse secant.
    asech       - Inverse hyperbolic secant.
    csc         - Cosecant.
    csch        - Hyperbolic cosecant.
    acsc        - Inverse cosecant.
    acsch       - Inverse hyperbolic cosecant.
    cot         - Cotangent.
    coth        - Hyperbolic cotangent.
    acot        - Inverse cotangent.
    acoth       - Inverse hyperbolic cotangent.

  Exponential.
    exp         - Exponential.
```

```
    log         - Natural logarithm.
    log10       - Common (base 10) logarithm.
    log2        - Base 2 logarithm and dissect floating point number.
    pow2        - Base 2 power and scale floating point number.
    realpow     - Power that will error out on complex result.
    reallog     - Natural logarithm of real number.
    realsqrt    - Square root of number greater than or equal to zero.
    sqrt        - Square root.
    nextpow2    - Next higher power of 2.

  Complex.
    abs         - Absolute value.
    angle       - Phase angle.
    complex     - Construct complex data from real and imaginary parts.
    conj        - Complex conjugate.
    imag        - Complex imaginary part.
    real        - Complex real part.
    unwrap      - Unwrap phase angle.
    isreal      - True for real array.
    cplxpair    - Sort numbers into complex conjugate pairs.

  Rounding and remainder.
    fix         - Round towards zero.
    floor       - Round towards minus infinity.
    ceil        - Round towards plus infinity.
    round       - Round towards nearest integer.
    mod         - Modulus (signed remainder after division).
    rem         - Remainder after division.
    sign        - Signum.
```

Remember to use the `help` command to get more information for any of those functions.

```
>> help exp

 EXP    Exponential.
    EXP(X) is the exponential of the elements of X, e to the X.
    For complex Z=X+i*Y, EXP(Z) = EXP(X)*(COS(Y)+i*SIN(Y)).

    See also LOG, LOG10, EXPM, EXPINT.
```

Here are some examples:

```
>> exp(1)
```

```
ans =
    2.7183
>> sin(pi/2)
ans =
     1
>> asin(1)
ans =
    1.5708
```

3.5 Flow control

3.5.1 The `for` statement

Repeating statements for an arbitrary number of times is one of the most important features of most programming languages. It used to be the case that MATLAB was not able to repeat statements efficiently but this is not true anymore. Starting with version 6.5 MATLAB includes a JIT-compiler which is responsible for optimizing loops.

The `for` loop repeats a group of statements for a fixed, predetermined number of times. A matching `end` denotes the end of the loop. For example:

```
>> for I = 1:5
s(I) = sqrt(I);
end
>> s
s =
    1.0000    1.4142    1.7321    2.0000    2.2361
```

The semicolon terminating the inner statement suppresses repeated printing, and the `s` after the loop displays the final result. Nested loops are good to be indented for readability. The MATLAB editor smart-indents nested loops and other constructs as we type. For example indentation makes the following loop more readable.

```
for I = 1:M
   for J = 1:N
      Q(I,J) = (I+J)^2;
   end
end
```

3.5.2 The `if` statement

The `if` statement executes a group of commands depending on the value of a logical expression. The `else` and `elseif` constructs are optional and provide additional branching. For example:

```
>> x = 3;
>> if x < 0
disp('negative');
else
disp('non-negative');
end
non-negative
>>
```

Indentation again can make the `if` statement more readable.

3.6 Scripts and functions

Using the command prompt is definitely recommended for temporary calculations however it is fundamental to be able to save code so that we can run it again in future MATLAB sessions. The saved MATLAB files are called m-files and must have an extension `.m`. We have already created our first m-file, `startup.m` in a previous section.

One can edit m-files using any text editor, such as EMACS but we recommend using the editor provided with the IDE. As we start working with m-files we will appreciate the customized IDE we have already created. You can imagine that testing code interactively on the command prompt and then copy-paste it to the editor right next to the command prompt is a very efficient way to work.

All the constructs we have learned so far are valid in an m-file. In addition we can now have comments in our code which definitely help for future reference. There is nothing worse than reading our own code 2 months after we wrote it only to realize that we don't remember what we wanted to calculate. Lines that start with `%` are treated as comments.

It is important to state that there are two types of m-files. Scripts and functions. Quoting from MATLAB help.

- Scripts do not accept input arguments or return output arguments. They operate on data in the workspace.
- Functions can accept input arguments and return output arguments. Internal variables are local to the function.

It is preferable to think and organize our code in terms of functions. Scripts might seem easier to write but they are very difficult to debug as every variable they use is stored in the workspace. Writing modular code using functions is a skill that we definitely need in our programming.

As an example assume that we need to write a function with name `myfun` that takes three input arguments and returns two output arguments. Then we should have as first line the function definition `[out,N] = myfun(x,time,freq)` and we should save the function in a file with filename `myfun.m`. Notice that the function name is the same as the filename as this is the only way for MATLAB to find and execute the function.

A good practice is to include comments right on the second line of the m-file. Those comments are automatically displayed by MATLAB if we were to ask for help by typing `help myfun`. We will demonstrate some meaningful help comments below.

To Do 3

Write 3 functions that calculate the factorial.
In the first function we are going to use the `for` loop. First check which directory you are in by typing `pwd`. If you are not in the `lab1` directory go there using the command `cd`. Create an m-file by typing:

```
>> !touch fact_for.m
```

Now open the file by typing `open fact_for.m`. Type the following code in the m-file:

```
function y = fact_for(x)
% FACT_FOR Calculates the factorial using a for loop
%   y = fact_for(x)
%
%   x: a positive integer
%   y: x's factorial

% Initialize y
y = 1;

% Do the loop
for I = 1:x
    y = y*I;
end
```

Save and verify that the function gives the right result.

If we get help for this function by typing `help fact_for` what is displayed on the command prompt?

What do you get if you type `fact_for(10)`?

How about `fact_for(1000)`? In this case the result is larger than the largest number that MATLAB can represent so it substitutes the result with infinity.

How about `fact_for(-10)`? Is this result correct? If not can you suggest code that would remedy this?

Now for the second function we will use the programming concept of recursion. Create a second file by typing `!touch fact_rec.m` Now open the file by typing `open fact_rec.m`. Type the following code:

```
function y = fact_rec(x)
% FACT_REC Calculates the factorial using a recursive call
%   y = fact_rec(x)
%
%   x: a possitive integer
%   y: x's factorial

if x == 0 | x == 1
    y = 1;
else
    y = x * fact_rec(x-1);
end
```

Try to understand what this code does. Notice that the function calls itself.

What do you get if you type `fact_rec(10)`?

How about `fact_rec(1000)`? In this case we get an error which means that we cannot have arbitrarily deep recursions.

Finally for the third function we are not going to provide the code. You should create a function called `fact_whi` from scratch that implements the factorial using the `while` construct instead of `for` or the recursion. Since we haven't introduced `while` you should get help and see how it is used. Don't forget to add meaningful comments similar to the ones in the previous two functions. You should use different input/output variable names.

Show the TA all three functions.

At this point you should turn off the diary function we started in the beginning of this lab session. You can do that by typing:

```
>> diary off;
```

This will create the diary file that contains all the commands you typed in the command prompt as well as all the output they generated. You should show the TA the diary file.