MM409/MM509 – Lab. 1
January 24, 2019 [FA]

By the end of this Lab, you will

- be familiar with basic Matlab operations;
- have used the `help` command a few times;
- know (roughly) what the following built-in functions do: `ones`, `zeros`, `isequal`, `eye`, `diag`, `find`, `sparse`, and `full`;
- have built a few functions on your own – some of which you will need to use in your assignment.

Some conventions and useful tips

Programming is very personal and everyone does it in their own way, but there are some basic conventions that you need to follow in order to become a good Matlab programmer.

- Use lower case letters for vectors and scalars;
- use upper case letters for matrices;
- blank spaces are not allowed when naming variables/scripts/functions. Use '_' instead;
- when writing a script/function, **always** comment your code and use insightful names for your variables;
- to clear the command window (without affecting the workspace) use `clc`;
- to clear the workspace (remove ALL your variables) use `clear`;
- always end a line with a '`;`' ;

Get started

Create a folder somewhere in your laptop where you will store all your Matlab files for this course. Once this is done, go back to Matlab and make sure that the path displayed leads you to the folder that you just created. If it doesn't, "move there" by using the drop-down list.

We will start by seeing how to define scalars, vectors, and matrices and we will be doing this by working directly in the command window. Everything that you define in the command window will appear as a variable in the workspace.

1. Define a scalar (also, show the effect of the semicolon):

```
>> c = 5
c =
     5
>> c = 5;
>>
```

if you now type `c` Matlab should display its value because `c` is saved in your workspace

```
>> c
c =
     5
```

to clear `c` from the workspace:

```
>> clear c
```

2. Define a row vector: `v = [1 3 4 5]` or `v = [1, 3, 4, 5]`.
3. Define a column vector: `w = [1; 2; 5; 6; 7]`.
4. Access the third entry of a vector: `v(3)`.
5. Modify the last entry of a vector: `w(end) = 1000`.
6. Define a matrix: `A = [ 1 2 3; 9 8 7; 2 5 1]`
7. Access/modify a certain element of a matrix : `A(2,2)` and `A(1,3) = 1000`
8. Access/modify a row of a matrix: `A(2,:)` and `A(1,:) = [ 5 5 5]`
9. Access/modify a column of a matrix: `A(:,1)` and `A(:,end) = [0; 1; 0]`
10. Transpose a vector/matrix: `vt = v'` and `At = A'`
11. Operations between scalars, vectors and matrices are defined in the standard way and the symbols used are: `+`, `-`, `*`, `/` and `^`. It is very important that the dimensions of the objects involved in the computations are consistent.

    As an example:

```
>> A = [1 0; 1 2];
>> B = [1 3; 1 1];
>> A*B
ans =
      1 3
      3 5
```

12. Multiplications, exponentiation and divisions can also be performed entry-wise, using `.*`, `.^`, and `./`

```
  >> A.*B
ans =
      1 0
      1 2
```

13. To check if two numbers are equal, use `==`. To compare other objects use `isequal` instead. (Why?! Try for yourself using `==`...)

```
>> a = 0; b = 1;
>> a == b
ans =
      0
>> isequal(A,At) % check if A is symmetric
ans =
      0
```

Functions

A function is a sequence of tasks that you expect to use often, for different input data, and that you do not want to rewrite from scratch every time. Functions are not application specific; e.g., you may want to write a function that computes the degree of nodes in the network you pass as input regardless of the particular structure of the network.

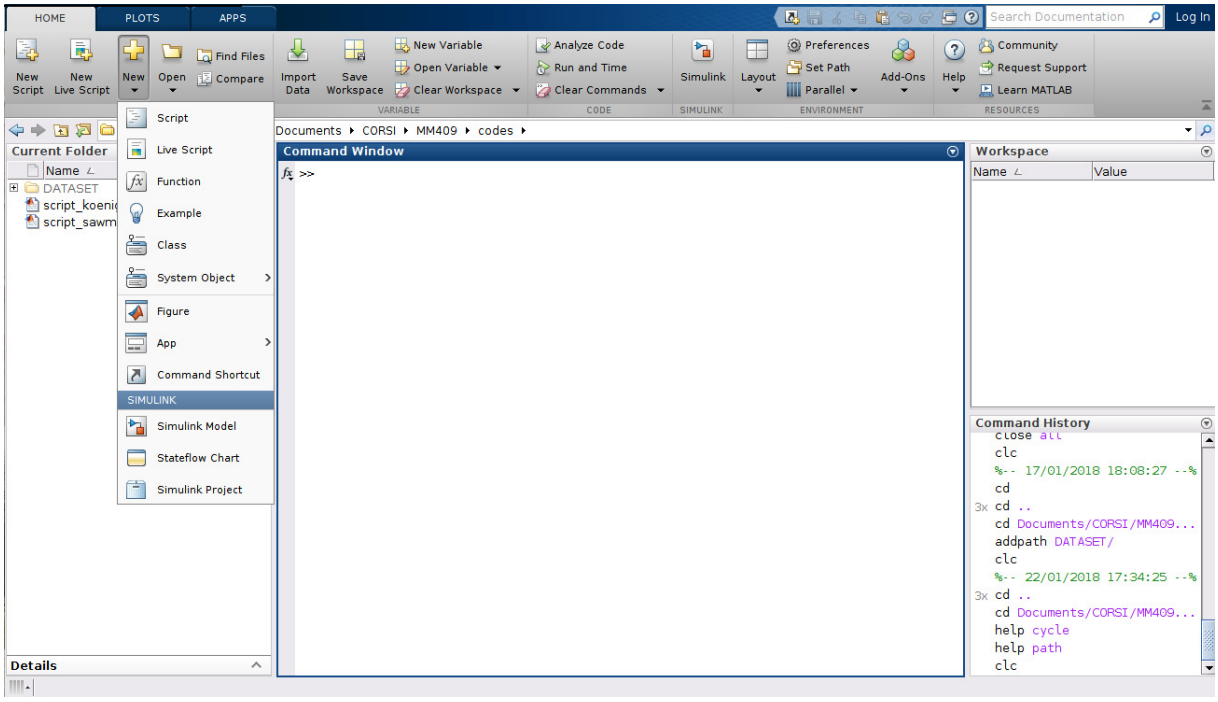
Matlab has countless built-in functions, but sometimes you will need to write your own. Today you will be writing a few to practice and to learn how to use some of Matlab's built-in functions.

It is important that you comment your code, so that it will be more easily understandable when you go back to it. To type comments in Matlab, use the symbol `%`.
Remember: when you need help to understand how a certain function works, say `eye`, you can type in the command window:

```
help eye
```

and Matlab will tell you the rest. This will work with all built-in functions and also your own functions – if you wrote them properly.

To write a new function, click on the yellow plus sign in the Home bar. A drop-down list will appear – click on "function".



In Matlab functions are structured as follows:

```
function [outputArg1,outputArg2,...]  = name_of_function(inputArg1,inputArg2,...)
   %UNTITLED Summary of this function goes here
   % Detailed explanation goes here
   List of Instructions;
end
```

They thus contain a `function` definition line and can accept input arguments and return output arguments. Their internal variables are **local** to the function: what happens inside a function does not

affect your workspace! So you do not need to be that careful when you pick the names of your variables inside a function.

For example, a function that computes the degree of the nodes in a network given the adjacency matrix looks like this:*

```
function d = degree_adj(A)
   % DEGREE_ADJ computes degree of nodes in a graph
   % D = DEGREE_ADJ(A) first checks whether the (n x n) matrix A is symmetric or not.
   % If yes, D is a (n x 1) vector containing the degree of nodes;
   % otherwise D is a (n x 2) matrix that contains the out-degree in the first column
   % and the in-degree in the second.
      if isequal(A,A') % check if A is symmetric
          d = sum(A,2); % the degrees are the row sums of A
      else
          n = size(A,1);
          d = zeros(n,2); % preallocate space for your output
          d(:,1) = sum(A,2);
          d(:,2) = sum(A,1)';
      end
end
```

Now, suppose that your workspace contains a variable named `n`. This variable will not change value if you use the function `degree_adj.m`. *What happens in a* `function` *stays in the* `function`.

The description given in lines 2 to 6 is what will appear when calling `help degree_adj` (try!).

You have now written a reusable function that allows you to compute the (in/out)degree of the nodes in any (di)graph.

YOUR TURN

Complete the following tasks. In case you have no time to finish this session, it is strongly recommended that you finish it at another time. To check that your functions work properly, use small inputs and make sure that Matlab returns the result you are expecting. For example, this is what should happen when you call the first of the functions you need to build:

```
>> complete_graph(4)
ans =
   0 1 1 1
   1 0 1 1
   1 1 0 1
   1 1 1 0
```

Tasks

- Write a function `complete_graph.m` that takes n as input and returns the adjacency matrix of K_n. [Hint: use `ones` and either `eye` or `diag`.]
- Write a function `path_graph.m` that takes n as input and returns the adjacency matrix of P_{n-1}. [Hint: check `diag` more carefully!]
- Write a function `cycle_graph.m` that takes n as input and returns the adjacency matrix of C_n.

*This function MUST be saved with the name `degree_adj.m` otherwise it won't work.

- Write a function `adj2list.m` that takes in input an adjacency matrix A and returns 1) a $m \times 3$ array L that stores in its rows the source of every edge, its target, and its weight, and 2) the number of nodes in the graph.

  Example:

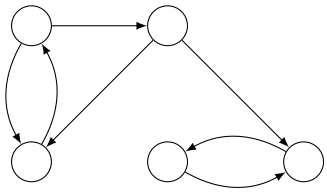
$$A = \begin{bmatrix} 0 & 1 & 2 \\ 3 & 1 & 0 \\ 0.5 & 1 & 1 \end{bmatrix}, \qquad L = \begin{bmatrix} 1 & 2 & 1 \\ 1 & 3 & 2 \\ 2 & 1 & 3 \\ 2 & 2 & 1 \\ 3 & 1 & 0.5 \\ 3 & 2 & 1 \\ 3 & 3 & 1 \end{bmatrix}$$

  [Hint: `find` – the order of the rows in L is unimportant!]

- Write a function `list2adj.m` that does the opposite of `adj2list.m`, by taking in input a $m \times 3$ list L and an integer n and returning the associated adjacency matrix. [Hint: check `sparse`. Can you make your output `full`?]

In the workspace...

- ...build the adjacency matrix of K_{10} and check that it is 9-regular;
- ...do the same for C_{10};
- now clear the workspace;
- write the adjacency matrix of the graph below and use your functions to build its adjacency list and vectors of out/in degrees.



Keep practicing

- Write a function `star_graph.m` that builds the adjacency matrix of $S_{1,n-1}$ given n as input. Label node 1 as the centre of the star.

  [Hint: start by building an empty matrix (`zeros`) and then use points 8/9 in the **Get started** section (`ones`).]

- Write a function `wheel_graph.m` that takes n and builds the adjacency matrix of a wheel graph with n nodes.

  [Hint: `cycle_graph.m`, `path_graph`, `star_graph.m`]

Later today I will upload a script on Myplace called `check_lab1.m`. Download it and save it in your folder. Then type in the command space the name of the file: `check_lab1`. It will tell you if everything has been done properly or if you need to make some adjustments to your functions.

Challenge

- Write a function `adj2inc.m` that takes in input the adjacency matrix of a graph (assume it does not contain multiple edges) and returns its incidence matrix. This is the most challenging problem. You will need to use `find` and an `if` statement. Remember that, for undirected networks, we only use one out of the two directions of the edges – to do this you may want to check `triu`. Start your function with the following instructions:

```
if ~isempty(find(diag(A))) % check if the diagonal of A is empty; if not, then ...
      A = A - diag(diag(A)); % remove the diagonal elements
      disp('Loops were removed from the network') % display this message
end
```

which are used to remove self loops from the graph associated with the adjacency matrix (i.e., makes sure that the diagonal of A is empty).

You should be able to write this function by only using the built-in functions that you have learned from completing the previous tasks. Alternatively, you can try and use a `for` loop (if you know what that is). If you need help, let me know!

If you complete this challenge, send me a functioning code and I will check it for you.