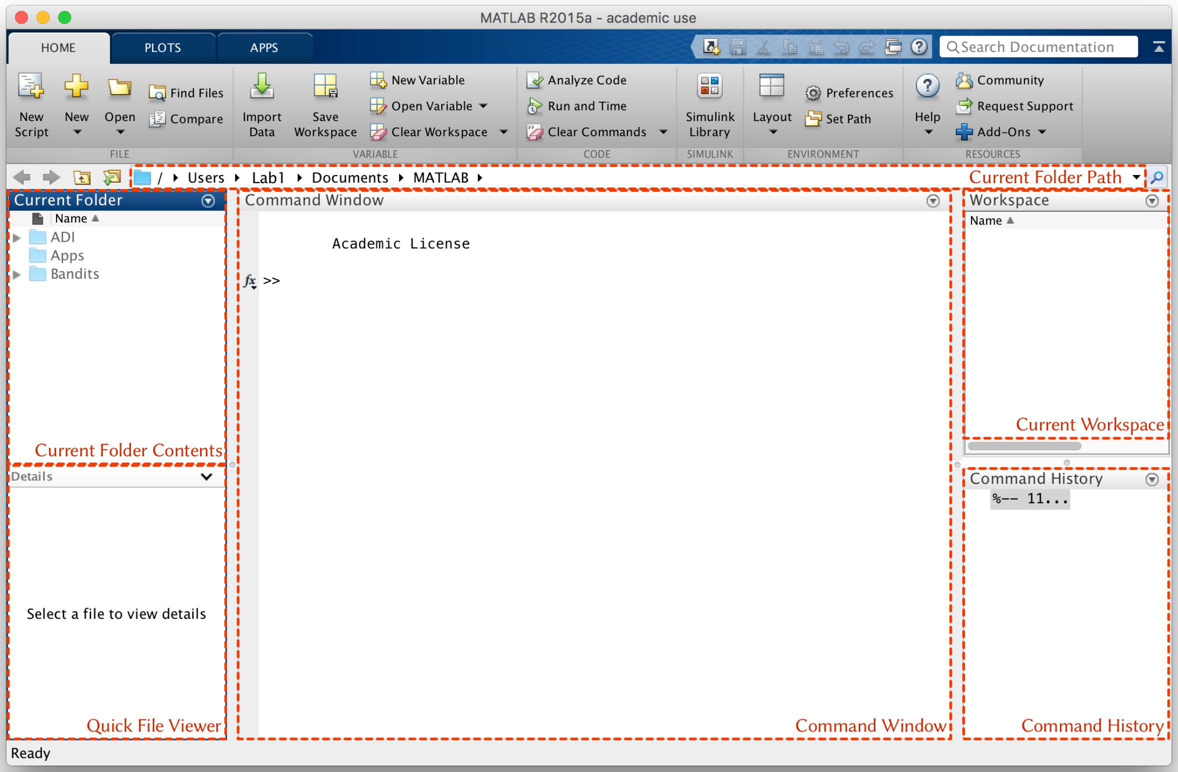
Lab 1. Introduction to Matlab

In the end of the lab, you should submit all code written in the tasks marked as **Activity n**, *together with the corresponding outputs and any replies to specific questions posed to the e-mail* **adi.tecnico@gmail.com**. *Make sure that the subject is of the form [<group n.>] LAB <lab n.>.*

1. Basic interaction

Matlab is a software package for numerical computation. It was designed originally for matrix manipulations, and hence its name: *MATrix LABoratory*. When you launch Matlab, you will see the Matlab window, reproduced in Fig. 1.



```
Academic License

>>
```

```
%-- 11...
```

Fig. 1. Matlab interface.

Besides the toolbar, there are 6 panels that typically show up, to know:

- *The Command Window panel.* It is in the command window that you can interact directly with Matlab.

- *The Current Workspace panel.* This panel, on the top right of the Command Window, depicts the variables existing in the current workspace.
- *The Command History panel.* This panel, on the bottom right of the Command Windows, lists the most recent commands, and can be used to repeat any sequence of recent commands.
- *The Current Folder panel.* This panel, on the top left of the Command Window, shows the contents of the current folder (where Matlab is currently running).
- *The Details panel.* This panel, on the bottom left of the Command Window, provides information (when available) regarding any file selected in the Current Folder panel.
- *The Current Path bar.* This bar, located above the Command Window, indicates the current folder (where Matlab is currently running) and corresponding path.

For now, let us focus on the Command Window, where you can observe the Matlab prompt: ">>". This command prompt allows you to interact directly with the Matlab engine, much like in interpreted languages like Python.

Try to reproduce the following interaction in the command window.

```
>> 2
ans =
     2
>> whos
  Name      Size            Bytes  Class     Attributes
  ans       1x1                 8  double
>> 2.0
ans =
     2
>> whos
  Name      Size            Bytes  Class     Attributes
  ans       1x1                 8  double
>>
```

Several aspects are worth noting in the interaction above:

(i) Unless if explicitly stated, expressions are evaluated to a variable `ans`.

(ii) The command `whos` shows all variables currently in the workspace. You can compare the output of this command with the contents of the Workspace panel.

(iii) Most numerical entities in Matlab are double-precision matrices. In the case of a scalar quantity (such as 2), it corresponds to a 1 x 1 matrix.

(iv) If a quantity is an integer, even if represented internally as a double, Matlab may represent it externally more compactly.

As in other languages, you may treat Matlab command window as a calculator where you can perform complex computations. For example, you can reproduce the following interaction, where integer, real and even complex number manipulation are performed seamlessly. Note, in particular, that Matlab knows the irrational number `pi` and the complex number `i`.

```
>> 5 * 2

ans =

   10

>> cos(2 * pi)

ans =

     1

>> log(1)

ans =

     0

>> exp(4)

ans =

   54.5982

>> sqrt(2)

ans =

    1.4142

>> sqrt(-1)

ans =

   0.0000 + 1.0000i

>> i * i

ans =

    -1

>>
```

In many aspects, Matlab is just like any other programming language. For example:

- You can define *variables* in Matlab. Matlab is *weakly typed*, meaning that variables do not have an associated type — instead, they are more like labels that you assign to quantities of interest.
- You can use *conditional statements* in Matlab. In particular, you have available both **`if`** and **`switch`** statements.
- You can use *cycles* in Matlab. In particular, you have available both **`for`** and **`while`** loops.
- Both conditional statements and loops must terminate with an **`end`** statement.

The following interaction illustrates the application of these different constructs.

```
>> a = 0

a =

     0

>> while a < 2 * pi
fprintf('%.3f radians correspond to %.1f degrees.\n', a, a * 90/pi);
a = a + 0.5;
```

```
end
0.000 radians correspond to 0.0 degrees.
0.500 radians correspond to 14.3 degrees.
1.000 radians correspond to 28.6 degrees.
1.500 radians correspond to 43.0 degrees.
2.000 radians correspond to 57.3 degrees.
2.500 radians correspond to 71.6 degrees.
3.000 radians correspond to 85.9 degrees.
3.500 radians correspond to 100.3 degrees.
4.000 radians correspond to 114.6 degrees.
4.500 radians correspond to 128.9 degrees.
5.000 radians correspond to 143.2 degrees.
5.500 radians correspond to 157.6 degrees.
6.000 radians correspond to 171.9 degrees.

>> for i = 1 : 5
a = sqrt(a)
end

a =

    2.5495

a =

    1.5967

a =

    1.2636

a =

    1.1241

a =

    1.0602

>> if a > 1
fprintf('%.2f is larger than 1.\n', a);
else
fprintf('%.2f is smaller than or equal to 1.\n', a);
end

1.06 is larger than 1.
```

Note, additionally, that:

- The instruction **`fprintf`** is used to write information to the screen. The syntax is essentially similar to that of **`printf`** in the C programming language. In particular, you can format the output to match your specific needs.
- When terminating an expression with a "**`;`**" the corresponding output is suppressed. Otherwise, the result is displayed on the screen.
- Later on, we will look more carefully at the expression **`i = 1 : 5`** appearing in the **`for`** loop.

2. Matrices

As stated earlier, Matlab was originally designed to manipulate matrices. For this reason, data manipulation in Matlab is done, essentially, in the form of matrix manipulation. In particular, Matlab offers a number of matrix manipulation operations that, if used proficiently, render computations significantly faster.

Matrices in Matlab are represented as sequences of numbers between square brackets. Rows of a matrix are separated by a semi-colon (“`;`”). There are also several commands to create particular matrices, such as the identity (`eye`), an all-zeros matrix (`zeros`) or an all-ones matrix (`ones`).

```
>> A = [1, 2, 3; 4, 5, 6]
A =
     1     2     3
     4     5     6
>> B = eye(3)
B =
     1     0     0
     0     1     0
     0     0     1
>> C = zeros(2, 3)
C =
     0     0     0
     0     0     0
>> D = ones(4);
D =
     1     1     1     1
     1     1     1     1
     1     1     1     1
     1     1     1     1
>>
```

You can now easily perform standard algebraic operations, such as matrix sum or products. You can also perform indexing, slicing, and other operations, as illustrated in the following program (you can copy and paste it directly to the Matlab Command Window).

```
% Matrix creation
A = [1, 2, 3; 4  5  6;7, 8  9]
B = 1:3
C = diag(B)

D = A + eye(3) % Matrix sum
A'             % Transpose
```

```
E = A * B'      % Matrix-vector product
B * A           % Vector-matrix product
inv(D)          % Inverse

% Concatenation
[D, E]
[A; D]

% Indexing
A(1)
A(5)
A(2, 3)

% Slicing
A(2:3,1:2)
A(B)
A(B')
```

The corresponding output should be:

```
A =
     1     2     3
     4     5     6
     7     8     9
B =
     1     2     3
C =
     1     0     0
     0     2     0
     0     0     3
D =


     2     2     3
     4     6     6
     7     8    10
ans =
     1     4     7
     2     5     8
     3     6     9
```

```
E =
    14
    32
    50
ans =
    30    36    42
ans =
   -6.0000   -2.0000    3.0000
   -1.0000    0.5000   -0.0000
    5.0000    1.0000   -2.0000
ans =
     2     2     3    14
     4     6     6    32
     7     8    10    50
ans =
     1     2     3
     4     5     6
     7     8     9
     2     2     3
     4     6     6
     7     8    10
ans =
     1
ans =
     5
ans =
     6
ans =
     4     5
     7     8
ans =
     1     4     7
ans =
     1
     4
     7
```

Several observations are in order:

- When specifying a matrix row, elements can be separated by commas (“`,`”) or spaces (“ ”).
- The character “%” is used for comments: everything after the “%” is treated as a comment.

- The function **diag** can be used to build a diagonal matrix from a vector or extract a diagonal from a matrix. You can know more about this (and other) functions in Matlab by typing **help <function name>**. For example, to know more about the **diag** function, you can type **help diag** in the Matlab prompt, to get something like the following interaction.

```
>> help diag

diag Diagonal matrices and diagonals of a matrix.
    diag(V,K) when V is a vector with N components is a square matrix
    of order N+ABS(K) with the elements of V on the K-th diagonal. K = 0
    is the main diagonal, K > 0 is above the main diagonal and K < 0
    is below the main diagonal.

    diag(V) is the same as diag(V,0) and puts V on the main diagonal.

    diag(X,K) when X is a matrix is a column vector formed from
    the elements of the K-th diagonal of X.

    diag(X) is the main diagonal of X. diag(diag(X)) is a diagonal matrix.

    Example

       m = 5;
       diag(-m:m) + diag(ones(2*m,1),1) + diag(ones(2*m,1),-1)
    produces a tridiagonal matrix of order 2*m+1.

    See also spdiags, triu, tril, blkdiag.

    Other functions named diag

    Reference page in Help browser
       doc diag

>>
```

  Note that, besides the help in itself, Matlab also provides links to other related functions as well as to the official online documentation.

- You can add, multiply, transpose, invert matrices much as you would in linear algebra.
- Indexing and slicing, in Matlab, are quite powerful operations. For example, you can use a matrix to index another. Note that, unlike other programming languages, *Matlab indices start at 1.*

The ability to leverage the powerful indexing and vectorization capabilities of Matlab is key to producing efficient code. It takes some time to get used to this programming philosophy, but once you do, you will notice a significant improvement in the performance of your Matlab code.

Use Matlab's help to learn more about:

- the **size** function
- the **length** function
- the **find** function
- the **repmat** function

The impact of good vectorization in the efficiency of Matlab code is illustrated in the following program (you can copy and paste it directly to the Matlab Command Window).

```
A = rand(1e3); % 1000 x 1000 random matrix with elements drawn uniformly from [0, 1]
B = rand(1e3);
C = zeros(size(A));
tic;
for i = 1:size(A, 2)
  for j = 1:size(A, 1)
    C(i, j) = A(i, j) + B(i, j);
  end
end
t1 = toc;
tic;
C = A + B;
t2 = toc;
fprintf('Standard computation: %.3f seconds.\n', t1);
fprintf('Vectorized computation: %.3f seconds.\n', t2);
```

The output will be something like (actual values may vary depending on the machine you use):

```
Standard computation: 0.118 seconds.
Vectorized computation: 0.004 seconds.
```

Besides illustrating the importance of optimizing Matlab code to take full advantage of its matrix manipulation capabilities, the previous program introduces several additional elements of the Matlab syntax, such as:

- The `rand` function.
- The `tic` and `toc` operations.

Activity 1.

Compare the time necessary to compute the cumulative sum of a 10,000 × 1 random vector using:

- a `for` loop;
- a vectorized operation.

For the latter, you may find useful the function `cumsum`.

Activity 2.

Compute in Matlab a 20 x 1 vector **y** where the *n*th entry is given by:

$$\mathbf{y}_n = \prod_{i=1}^{n} \sum_{j=1}^{i} j^3$$

You should use *no cycles* in your computation.

3. Plotting

Matlab offers a number of plotting routines that are ideal to display scientific data. Run the following program (copy it directly to the Command Window), which generates 100 perturbed samples from the function

$$f(x) = 2x$$

and uses these samples to estimate the function *f*.

```
1   % Create data
2   X = 100 * rand(100, 1);
3   Y = 2 * X + 5 * randn(100, 1);
4
5   % Estimate linear relation between X and Y
6   Xaux = [X ones(100, 1)];
7   Mest = pinv(Xaux) * Y;
8   Yest = Xaux * Mest;
9
10  % Plot commands
11  figure;            % Create new figure window
12  plot(X, Yest);     % Plot X vs Yest
13  hold on;           % Hold the axis for future plots
14  plot(X, Y, 'x');   % Plot X vs Y, each point represented as a "x"
15
16  xlabel('Input X');
17  ylabel('Output Y');
18  title('Linear regression');
```

Upon running the program above, you should observe that Matlab opens a new window similar to the one depicted on Fig. 2.

Consider more carefully the program above, where we included line numbers for easier reference.

- On lines 2 and 3, the vectors **`X`** and **`Y`** are created, using mostly functionalities that you already encountered in Sections 1 and 2. The novelty is the function **`randn`** which is similar to the function **`rand`** except on their underlying distribution: while **`rand`** generates random numbers uniformly from the interval [0, 1], **`randn`** generates normally distributed random numbers with mean 0 and a standard deviation of 1.
- Lines 6-8 estimate a linear relation between **`X`** and **`Y`** using the data already created. Do not worry about the actual computations, and simply observe the use of matrix concatenation in line 6, and the **`pinv`** function in line 7. The function **`pinv`** computes the *Moore-Penrose pseudo-inverse* of a matrix.[1]

[1] http://en.wikipedia.org/wiki/Moore—Penrose_pseudoinverse

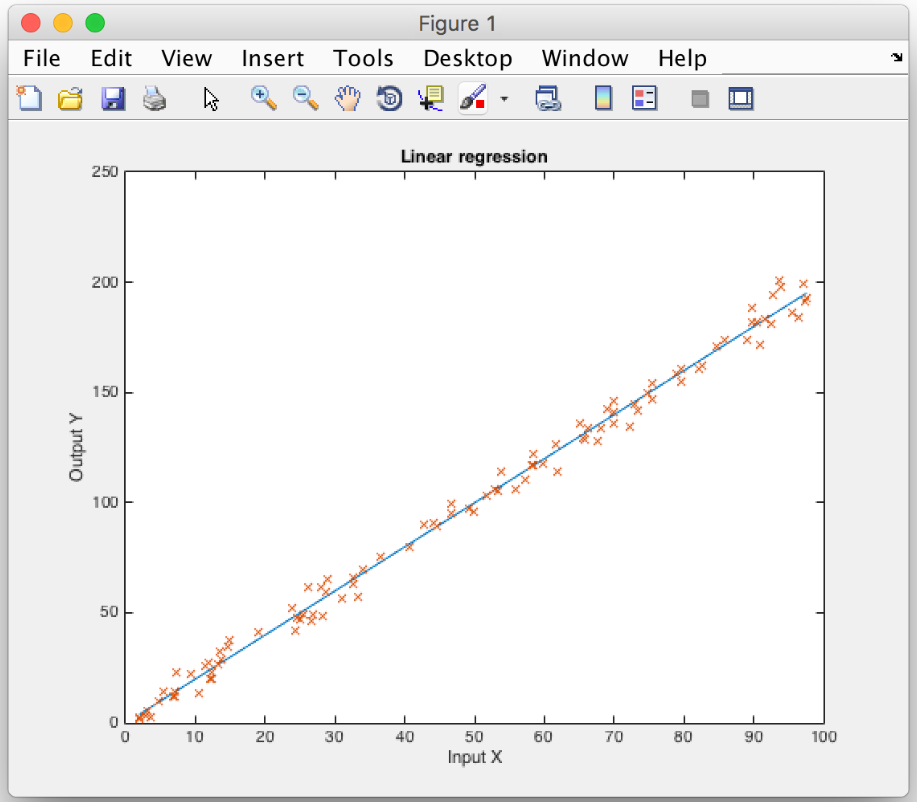


Fig. 2. Result of the program to estimate a linear function.

- Lines 11 through 14 contain the actual plotting commands. In particular:
  - The **`figure`** command in line 11 creates a new (blank) figure window. As can be seen in Fig. 2, figure windows are numbered. You can also use the **`figure`** command to bring to the foreground a figure window previously created, by providing the corresponding number as an argument.
  - The **`plot`** command in line 12 is responsible for displaying the continuous line in Fig. 2. In here, it is used with its most basic syntax. However, the **`plot`** command has a very rich syntax, and you can type **`help plot`** to know more about this useful function.
  - To avoid subsequent plot commands to erase the existing plot, we turn on the "hold" option in the current axis (the object "containing" the plot), through the **`hold`** command in line 13.
  - The plot command in line 13 plots the original data. Note how the line specification **`'x'`** indicates that, unlike the plot in line 11, this data should not be plotted continuously but instead marking each data-point with an "×".
- Finally, the commands in lines 16 to 18 are used to include additional information in the plot, such as the labels for both axis and the title for the plot.

Other useful plot commands are listed in Section 5.

Activity 3.

Recall the definition of y(n) in Activity 2:

$$y(n) = \prod_{i=1}^{n} \sum_{j=1}^{i} j^3$$

Plot the value of *y* against *n* for *n* between 1 and 20 (inclusive) using a linear plot (obtained with the command `plot`) and a logarithmic plot (using the command `semilogy`). Stack both plots together using a `subplot`, and comment on the observed differences.

What would you argue to be the usefulness of semi-logarithmic plots?

Activity 4.

A hospital is conducting a study on obesity in adult men and, as part of that study, tested the age and body fat for 18 randomly selected adults, with the following results:

Age	38	27	48	17	33	32	38	38	26
% Body fat	17.1	20.7	25.2	13.6	13.4	19.8	11.7	17.3	15.6

Age	34	33	41	45	46	26	35	22	23
% Body fat	20.5	22.3	20.2	17.5	22.7	7.5	27.2	9.2	16.1

Write down the data above as two row-vectors `age` and `fat`, and replicate the procedure used to generate Fig. 2 to estimate a linear relation between the two quantities. Plot the relation that you estimated against the real data, as in Fig. 2.

Activity 5.

(a) Generate a 2 x 2 random *stochastic matrix*. A stochastic matrix is a matrix whose elements all lie in the closed interval [0, 1] and whose rows add to 1.
Suggestion: you can do this by generating a random matrix and then dividing each row by its sum.

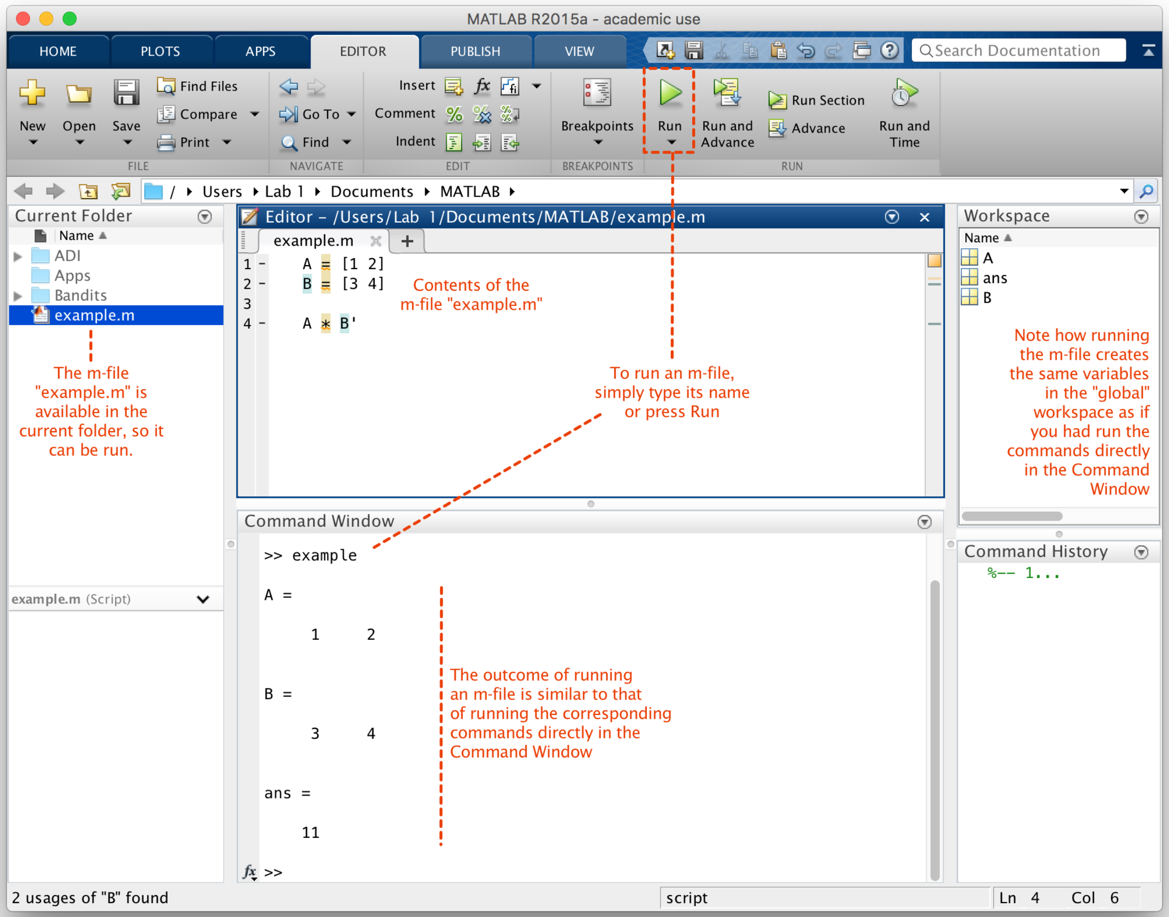
(b) Compute its eigenvalues, and show that $|\lambda_i| \leq 1$ for all eigenvalues λ_i.

(c) Repeat this process a number of times to observe that the eigenvalues of stochastic matrices always lie in the (complex) unit circle.

4. Scripts and functions

So far, your programs in Matlab were introduced directly into the Command Window. However, for larger programs this is hardly practical. Matlab offers the possibility of writing down your program in *Matlab* (script) *files* also known as *m-files*.

An *m*-file essentially contains Matlab code that can be run by either pressing the “Run” button in the “Editor” toolbar, or by typing in the command window the name of the *m*-file. Consider the screenshot in Fig. 3.



```
A = [1 2]
B = [3 4]

A * B'
```

Contents of the m-file "example.m"

The m-file "example.m" is available in the current folder, so it can be run.

To run an m-file, simply type its name or press Run

Note how running the m-file creates the same variables in the "global" workspace as if you had run the commands directly in the Command Window

```
>> example

A =

     1     2


B =

     3     4


ans =

    11

>>
```

The outcome of running an m-file is similar to that of running the corresponding commands directly in the Command Window

Fig. 3. Matlab window with an m-file “example.m” open for edition.

To reproduce the figure above, in the “Home” toolbar in the Matlab window press the button “New script”. This should split the panel where the Command Window lies in two, as in Fig. 3. You can then type into the newly opened Editor window the commands above. Save your file and then run it, to observe the interaction in Fig. 3.

Matlab files are also used to define functions. Each function in Matlab corresponds lies in a dedicated *m*-file which should contain, as its first line, the function header. The function header has the overall structure:

`function` [<*output arguments*>] = <*function name*> (<*input arguments*>)

A function to determine whether an argument is a prime number could be written as follows:

```matlab
function [out] = isprime(n)
    % ISPRIME   True for prime numbers.
    %   ISPRIME(X) returns 1 when X is a prime number
    %   and 0 when it is not.
    if n < 2
        out = false;
    else
        out = ~any(mod(n, 2:(n-1)) == 0);
    end
end
```

Several aspects are noteworthy in the piece of code above:

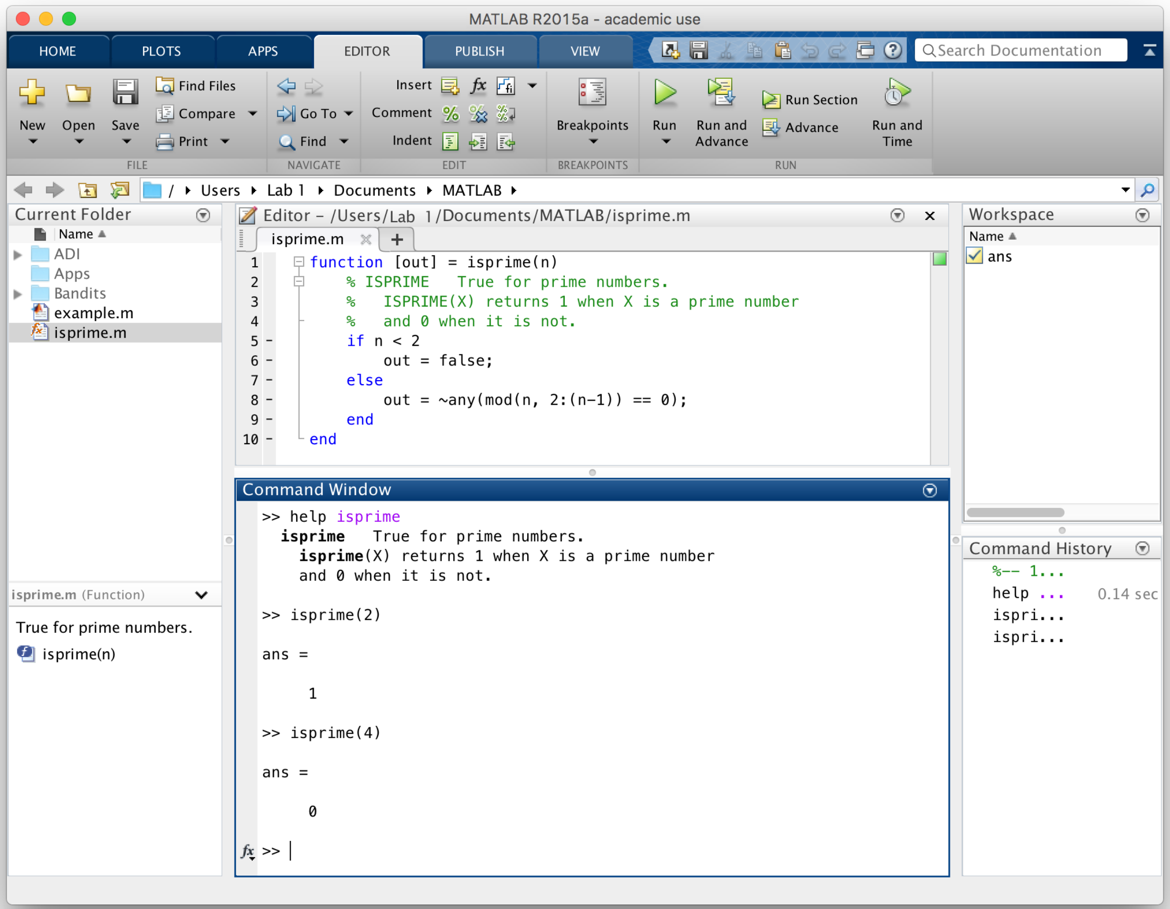
- The return value(s) of the function is specified between square brackets immediately after the `function` keyword. This implies that the corresponding variables *must be initialized/assigned somewhere in the function body*. The function will return the last value assigned to the output variables. In the example above, `out` is the output variable, which will take either the value `false` (0) or the result of the expression `~any(mod(n, 2:(n-1)) == 0)`.
- Note the use of the constant `false`. Like the constant `pi` seen before, the constants `false` and `true` correspond to the (logical) values 0 and 1, respectively. This is clear in the following interaction:

  ```
  >> true
  ans =
       1
  >> false
  ans =
       0
  >> whos
    Name      Size            Bytes  Class      Attributes
    ans       1x1                 1  logical
  >>
  ```

- Note the use of the functions `mod`, `any` and the negation operation `~`. Use Matlab's help to learn more about these operators.
- Finally, note how the code above takes advantage of vectorization to avoid cycles.

In order to use this function, it must be stored in an m-file named `isprime.m`. Matlab will be able to find it as long as it lies in Matlab's current folder or in the Matlab path.

Figure 4 illustrates the application of the function above.



```
function [out] = isprime(n)
    % ISPRIME   True for prime numbers.
    %   ISPRIME(X) returns 1 when X is a prime number
    %   and 0 when it is not.
    if n < 2
        out = false;
    else
        out = ~any(mod(n, 2:(n-1)) == 0);
    end
end
```

```
>> help isprime
  isprime   True for prime numbers.
    isprime(X) returns 1 when X is a prime number
    and 0 when it is not.

>> isprime(2)

ans =

     1

>> isprime(4)

ans =

     0

>> |
```

Fig. 4. Application of function `isprime`.

Once again, several aspects are important in the interaction above:

- Observing the Current Folder panel, notice the different icons in m-files corresponding to scripts (such as **`example.m`**) and m-files corresponding to functions (such as **`isprime.m`**).

- When you select an m-file in the Current Folder panel, note that the Quick File Viewer panel now shows the first help line for the function.

- *As soon as you save the m-file* **`isprime.m`**, *Matlab knows the function* **`isprime`**:

    - If you type **`help isprime`**, Matlab prints the corresponding help (the comment block in the beginning of the function);

    - If you make a call to the function, Matlab will compute the corresponding value.

  None of this requires you to "run" the file.

- A key difference between a script file and a function file is that *function files do not change the global workspace* (observe the Workspace panel after the execution of the function in Fig. 4). In other words, variables defined within a function file are local to that function.

Activity 6.

Write down a function **`generatetm`** that receives as an input argument a positive integer n and returns a $n \times n$ stochastic matrix (see **Activity 5** for a definition of stochastic matrix).

Activity 7.

Rewrite the function **`isprime`** so that it receives a vector/matrix of integers and outputs a vector/matrix with the same dimension and where the entry ij is **true** if the element ij in the input argument is prime, and **false** otherwise.

With your function it should be possible to obtain the following interaction:

```
>> isprime(2)

ans =

     1

>> isprime([1 2; 3 4])

ans =

     0     1

     1     0

>> isprime(1:5)

ans =

     0     1     1     0     1

>>
```

5. Reference sheet

List of common matrix operations

Operation	Meaning	Examples	Result

`+`	Matrix sum
	Sum of a constant
		`>> [1 2 3] + [4 5 6]`
		`>> [1 2 3] + 4`
			`[5 7 9]`
			`[5 6 7]`

`-`	Matrix difference
	Subtraction of a constant
		`>> [1 2; 3 4] - eye(2)`
		`>> [1 2; 3 4] - 2`
			`[0 2; 3 3]`
			`[-1 0; 1 2]`

`*`	Matrix product
	Scalar product
		`>> [1 2; 3 4] * [2; 3]`
		`>> [1 2; 3 4] * 2`
			`[8; 18]`
			`[2 4; 6 8]`

`.*`	Matrix product (elementwise)	`>> [1 2; 3 4] .* [5 6; 7 8]`	`[5 12; 21 32]`
`^`	Matrix power ($\mathbf{A}^n = \mathbf{A} \times \ldots \times \mathbf{A}$)	`>> [1 2; 3 4]^2`	`[7 10;15 22]`
`.^`	Element exponentiation	`>> [1 2; 3 4].^2`	`[1 4; 9 16]`
`\`	Left division ($\mathbf{A} \setminus \mathbf{B} = \mathbf{A}^{-1} \times \mathbf{B}$)	`>> [1 2; 3 4] \ [1; 1]`	`[-1; 1]`
`.\`	Left division (elementwise)	`>> [1 2; 3 4] .\ [5 6; 7 8]`	`[5 3; 2.333 2]`
`/`	Right division ($\mathbf{A} / \mathbf{B} = \mathbf{A} \times \mathbf{B}^{-1}$)	`>> [1 2; 3 4] / [5 6; 7 8]`	`[3 -2; 2 -1]`
`./`	Right division (elementwise)	`>> [5 6; 7 8] ./ [1 2; 3 4]`	`[5 3; 2.333 2]`

`:`	Range	`>> 1:0.1:2`
		`>> 1:5`
			`[1.0 1.5 2.0]`
			`[1 2 3 4 5]`

`'`	Transpose	`>> [1 2; 3 4]'`
		`>> [1 2 3]'`
			`[1 3; 2 4]`
			`[1; 2; 3]`

`inv`	Inverse	`>> [1 2; 3 4]`	`[-2 1; 1.5 -0.5]`

`pinv`	Moore-Penrose pseudo-inverse	`>> pinv([1 2; 3 6])`
		`>> pinv([1 2])`
			`[.02 .06; .04 .12]`
			`[.2; .4]`

`rank`	Rank of a matrix	`>> rank([1 2; 3 4])`
		`>> rank([1 2; 3 6])`
			`2`
			`1`

`sum`	Sum of elements	`>> sum([1 2; 3 4])`
		`>> sum([1 2; 3 4], 2)`
			`[4 6]`
			`[3; 7]`

`cumsum`	Cumulative sum of elements	`>> cumsum([1 2; 3 4])`
		`>> cumsum([1 2; 3 4], 2)`
			`[1 2; 4 6]`
			`[1 3; 3 7]`

`find`	Find indices of non-zero elements	`>> find(eye(2))`
		`>> find([1 2; 3 4] < 3)`
			`[1; 4]`
			`[1; 3]`

`min`	Find smallest element	`>> min([1 2; 3 4])`
		`>> min([1 2; 3 4], [], 2)`
			`[1 2]`
			`[1; 3]`

`max`	Find largest element	`>> max([1 2; 3 4])`
		`>> max([1 2; 3 4], [], 2)`
			`[3 4]`
			`[2; 4]`

`abs`	Absolute value	`>> abs([-1 2; -3 4])`	`[1 2; 3 4]`

`norm`	Matrix norm	`>> norm(ones(1, 2))`
		`>> norm([1 2; 3 4])`
			`1.4142`
			`5.465`

`size`	Matrix dimensions	`>> size([1 2; 3 4; 5 6])`
		`>> size([1 2; 3 4; 5 6], 1)`
			`[3 2]`
			`3`

`length`	Length of a matrix: length(A) = max(size(A))	`>> length([1 2;3 4;5 6])`	`3`

Common workspace and session control commands

Operation	Meaning	Usage
`clc`	Clear command window	`>> clc`

`clear`	Clear variables from memory	`>> clear x   % Clears variable x`
		`>> clear all % Clears all variables`

`exist`	Checks the existence of an entity	`>> exist('A','var')  % Is there a var. A?`
		`>> exist('A','file') % Is there a file A?`

`who`	Lists current variables	`>> who`
`whos`	Lists current variables (detailed)	`>> whos`

Useful plot commands

Operation	Meaning	Usage

`figure`	Open new figure window	`>> figure          % Open new figure`
		`>> figure(1)       % Open figure 1`

`close`	Close figure window	`>> close(1)        % Close figure 1`
		`>> close all       % Close all figures`

`clf`	Clear figure window	`>> clf             % Clear current figure`
		`>> clf(1)          % Clear figure 1`

`plot`	Generate *x*-*y* plot	`>> plot(X, Y)      % Plot X x Y`
`semilogx`	Generate *x*-*y* plot with logarithmic *x*-axis	`>> semilogx(X, y)  % Plot log(X) x Y`
`semilogy`	Generate *x*-*y* plot with logarithmic *y*-axis	`>> semilogy(X, Y)  % Plot X x log(Y)`
`loglog`	Generate logarithmic *x*-*y* plot	`>> loglog(X, Y)    % Plot log(X) x log(Y)`
`bar`	Generate bar plot	`>> bar(X, Y)       % Bar-plot X x Y`
`step`	Generate stem plot	`>> stem(X, Y)      % Stem-plot X x Y`
`stairs`	Generate stair plot	`>> stairs(X, Y)    % Stair-plot X x Y`
`subplot`	Generate/activate subplot window	`>> subplot(2, 1, 1) % 2 x 1 plot window`

`hold`	Turn plot holding on/off	`>> hold on`
		`>> hold off`

`grid`	Turn grid on/off	`>> grid on`
		`>> grid off`

`title`	Set axis title	`>> title('This is a title')`
`xlabel`	Set *x*-axis label	`>> xlabel('X')`
`ylabel`	Set *y*-axis label	`>> ylabel('Y')`
`legend`	Set plot legend	`>> legend('This is a line')`