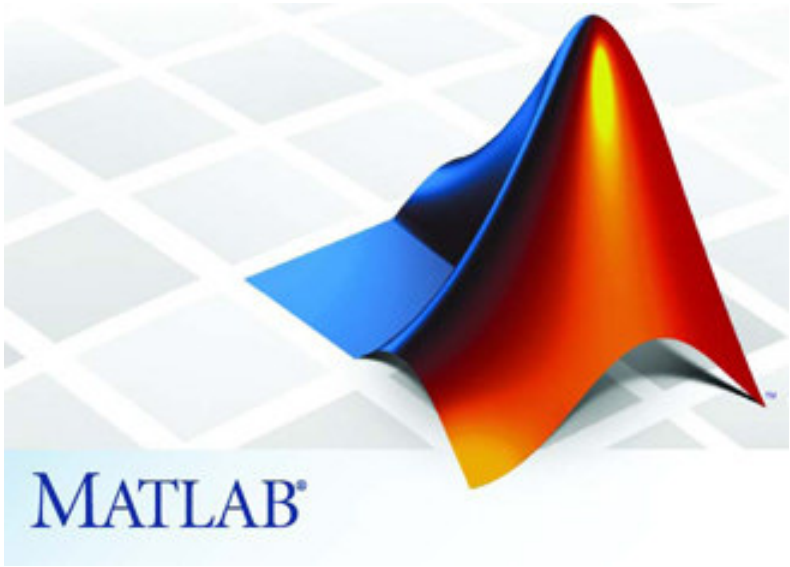
INTRODUCTION TO MATLAB



Course of
Intelligent Systems
a.y. 2015/2016

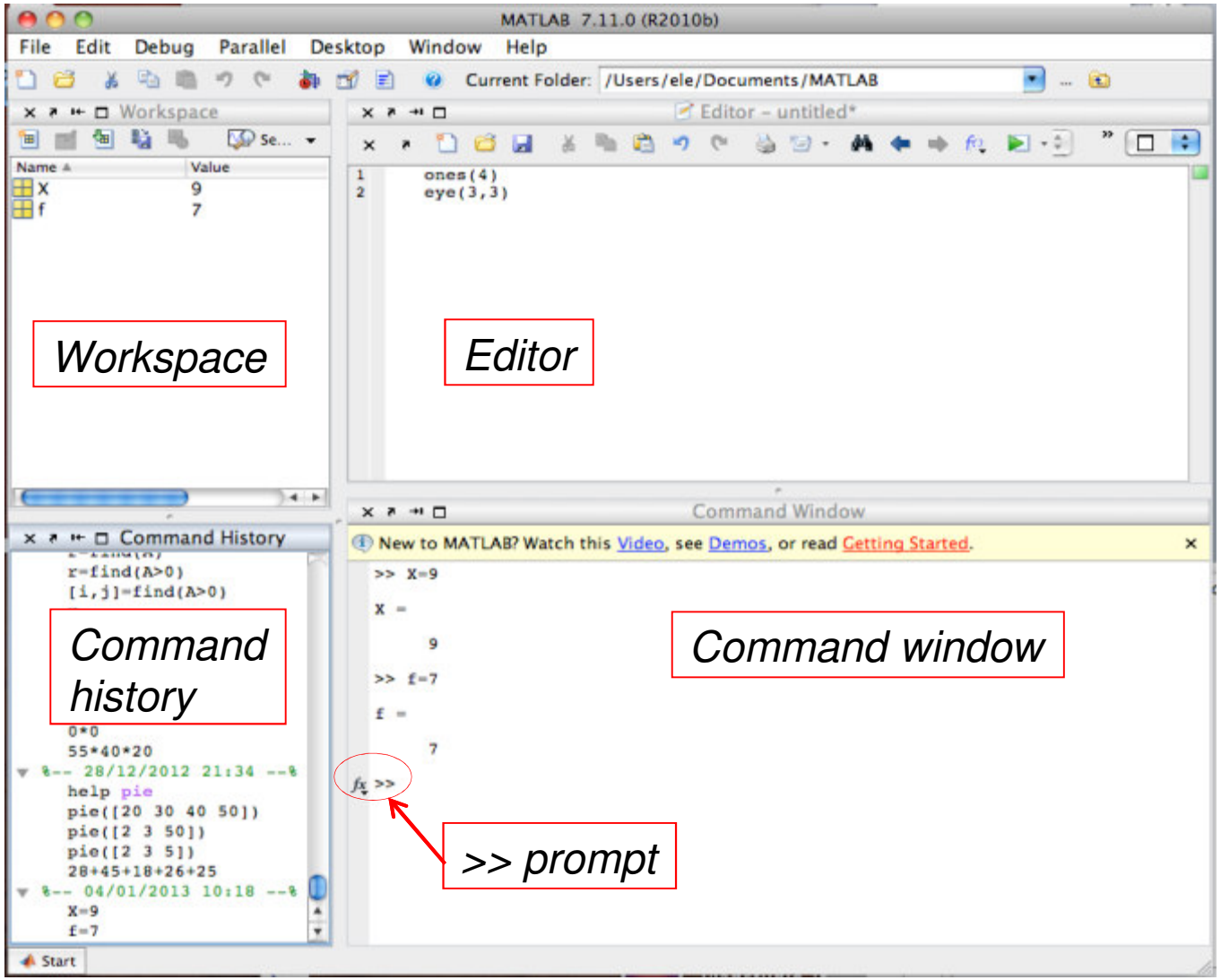
Ing. Eleonora D'Andrea

eleonora.dandrea@for.unipi.it

What is Matlab?

- *Mat*rix *Lab*oratory, interactive, matrix-based environment for scientific and engineering calculations
- Matrix: fundamental data structure
- Desktop tools in Matlab
  - *Command window*: to insert commands at the prompt **>>**, run functions or display results and error messages
  - *Editor:* to write or edit source code in a friendly framework (colored keywords, highlighted errors, tools for debugging)
  - *Workspace:* contains the set of variables *currently* defined (*in the current working session* : time interval from startup); a variable value can be viewed (in the *Variable Editor*) by double-clicking its name
  - *Command history:* lists all previously run commands in current and previous working sessions

Matlab's desktop tools



Main elements of language

1. **Variables**: Matlab works with one kind of object: the matrix (**each variable is treated as a matrix**)
   - **Matrix** is indicated with `A, B`,...
   - **Array** (or vector) is treated as matrix with one only row (or column) `a, b, ...`
     - the terms *array* and *matrix* are used interchangeably
   - **Scalar** is a number, treated as a matrix 1x1
2. **Functions**
   - built-in: already defined and ready to use , *mathematical*, *statistical*, etc. (`sum, max, min`...)
   - user-defined: can be written according to rules
3. **Operators**
   - **Arithmetic**: + addition, - subtraction, * multiplication, / division, ^ power, ' transpose
   - **Special**: : colon, . dot, [ ] square brackets
4. **Statements** are made of functions + operators + variables

Some important notes

- Matlab is *case sensitive,* `a` (lowercase) is not tha same as `A` (uppercase)
- Matlab is an interpreted language
  - each instruction is translated in machine code and executed
  - there is no compiler
- The allocation of matrices and arrays is dynamic
  - no preallocation is required for variables
  - no declaration is required about dimensions and type of variables
    - it can be *useful* to preallocate very large matrices to speed up the code (when using program loops)

Statements

- Matlab statements are of the form:
  - `variable_name = expression;`
  - `expression;`
  - `x=log(y)+2;` complete statement
- Expressions are composed from operators, functions, variable names, and scalars
  - `c=3+2;` the evaluation of the expression (`3+2`) produces a result which is assigned to variable `c`
  - `3+2;` If `c` and the '=' are omitted, the result is assigned to the predefined variable *ans*
  - Suppressing the output
    - If the semicolon '**;**' at the end of the statement is omitted, the result is always displayed on the screen
    - to suppress the output, use the semicolon '**;**' at the end of the statement
    - It is useful to suppress the output when managing large matrices or performing multiple operations

Entering matrices

- Matrices can be entered in different ways:
- Entered as list of elements
  - separated by space or comma between square brackets
  - each row ends with a semicolon '**;**'
  - Matrix `A=[1 2 3; 4 5 6; 7 8 9]`
  - Array `b=[10 20 30]`

$$A = \begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{pmatrix}$$

- Loaded from numeric data files
  - `B=load data.dat, B=load data.txt`
- Generated with built-in or user-defined functions
  - `C=zeros(3,5)` 3x5 matrix of 0's elements
  - `O=ones(1,5)` 1x5 matrix of 1's elements
  - `M=magic(3)` magic square, sum of the elements on rows, columns, diagonals is equal
  - `R=rand(4,5)` 4x5 matrix of randomly generated values in [0 ,1]

Entering matrices: examples

```
Z = zeros(2,4)
Z =
     0     0     0     0
     0     0     0     0

F = 5*ones(3,3)
F =
     5     5     5
     5     5     5
     5     5     5
```

```
>> M=magic(3)

M =

     8     1     6
     3     5     7
     4     9     2
```

```
>> rand(3,1)

ans =

    0.3404
    0.5853
    0.2238
```

Subscripts (indexes)

the index starts from 1 not 0!

- A subscripts notation between round brackets is used to access elements or submatrices in a given matrix:

- row-column notation (two indexes, most used)
  - we specify the name of the matrix and then the number of row and column of the element
  - `A(i,j)` extracts from *A* the element in position (`i`,`j`) where `i` is the row index and `j` is the column index
  - `A(2,3)` is '6', the element in row 2, column 3

$$A = \begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{pmatrix}$$

- linear notation (a single index)
  - we specify the name of the matrix and then the index of the element, as if the elements were arranged in a long column vector, composed by the columns of the matrix, arranged one above the other
  - `A(k)` extracts the *k*-th element
  - `A(8)` is '6', the 8-th element

$$A = \begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{pmatrix}$$

Colon ':' operator

- The colon operator ':' is used mainly in two ways:
- For generating a equally spaced vector
  - `t =(start_val:end_val)` with default increment 1
  - `t =(4:10)` gives t = 4 5 6 7 8 9 10
  - `t2 =(start_val:increment:end_val)` with increment 2
  - `t2 =(4:2:10)` gives t2 = 4 6 8 10
- For selecting a submatrix (a portion of a matrix)
  - `a = t(1:3)` selects the elements in *t* from 1 to 3 and produces a = 4 5 6
  - `a = t(:)` selects all elements in *t* (":" used alone means "all", no selection)

Access to elements with subscripts and colon operator

- A subscripts notation is used to access elements in a matrix
- Access a single element
  - `A(2,3)` is '6', the element in row 2, column 3
- Access a submatrix
  - `B=A(1:2,2:3)` is the submatrix in *A* consisting of rows 1 and 2, and columns 2 and 3
  - `C=A(:,2)` is the submatrix (subarray) in *A* consisting of *all* elements of column 2
- the colon operator '**:**' used alone as index of a row (column) denotes all the elements of the row (column)

$$A = \begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{pmatrix}$$

$$B = \begin{pmatrix} 2 & 3 \\ 5 & 6 \end{pmatrix}$$

$$C = \begin{pmatrix} 2 \\ 5 \\ 8 \end{pmatrix}$$

Square brackets '[ ]' operator

- The square brackets [ ] operator is used mainly in two ways:
- Concatenation of matrices: two matrices are joined to make a bigger matrix, by placing the one side by side to the other (A and B have same number of rows)
  - `AB=[A B]`

$$A=\begin{pmatrix}1 & 2\\ 5 & 6\end{pmatrix}, B=\begin{pmatrix}3 & 4\\ 7 & 8\end{pmatrix}, \quad AB=\begin{pmatrix}1 & 2 & 3 & 4\\ 5 & 6 & 7 & 8\end{pmatrix}$$

- Deleting rows or columns from a matrix
  - `AB(:,2)= []` deletes the second column (by assigning the empty element [ ]) and directly modifies `AB`
    - the colon operator as row subscript means that all rows are considered.

$$AB=\begin{pmatrix}1 & 3 & 4\\ 5 & 7 & 8\end{pmatrix}$$

Matrix linear algebra operations

- The mathematical operations defined on matrices are the same of linear algebra
- Addition `A+B` and subtraction `A-B` act on involved matrices element by element
  - (A and B must have same size)
- Multiplication `A*B` and division `A/B` act between entire matrices following the rules of linear algebra
  - (A and B must have compatible sizes)
  - If the dimensions of the involved matrices in the operation are incompatible, an error message is showed
- Power `A^n`
  - corresponds to repeat *n* times the multiplication of *A* by itself
- Transpose `A'`
  - the transpose matrix is the matrix with columns and rows reversed

Element by element operations

- Operations on matrices can also be performed between the corresponding elements of the matrices involved, by using the dot '.' operator before the arithmetic operator
  - Addition and subtraction are already element-wise operations!
  - Multiplication element by element `A.*B`
  - Division element by element `A./B`
  - Power element by element `A.^`
- ***A.*B*** is different from ***A*B*** because the former is a *element-wise* multiplication , the latter is a row-column multiplication

$$A * B = \begin{pmatrix} 1 & 2 \\ 2 & 1 \end{pmatrix} * \begin{pmatrix} 1 & 1 \\ 2 & 2 \end{pmatrix} = \begin{pmatrix} 5 & 5 \\ 4 & 4 \end{pmatrix}$$

$$A .* B = \begin{pmatrix} 1 & 2 \\ 4 & 2 \end{pmatrix}$$

Scalar operations

- A scalar in Matlab is a 1x1 matrix
- The operation which involves a matrix and a scalar is applied between the scalar and each element of the matrix
- Scalar operations (involving a scalar *s*) are
  - Addition
  - Subtraction
  - Division
  - Multiplication

$$A = s + \begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix} = \begin{pmatrix} s+1 & s+2 \\ s+3 & s+4 \end{pmatrix}$$

$$A = s \cdot \begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{pmatrix} = \begin{pmatrix} s & 2s & 3s \\ 4s & 5s & 6s \\ 7s & 8s & 9s \end{pmatrix}$$

Strings and string functions

- A string is a matrix whose elements are charcaters surrounded by single quotes ' '
  - `name = 'Mario'`
- `disp(s)` is the standard function used to display a string *s*
- `s=sprintf('the result is: %d',r)` writes formatted data (a string and a double value `r`) to a string `s`
  - useful when we need to insert carriage returns or values of variables in a string
  - sprintf('first line \n second line') produces a string on two lines
- `num2str(x)` converts the number `x` to a string
  - useful when dispaying numerical results within a string with `sprintf`
- `strcat(s1,s2)` concatenates two strings `s1,s2`
  - strcat('one', ' two') produces the string 'one two'
- `strcmp(s1,s2)` compares two strings and return a true (1) or false (0) value

Structures and cell arrays /1

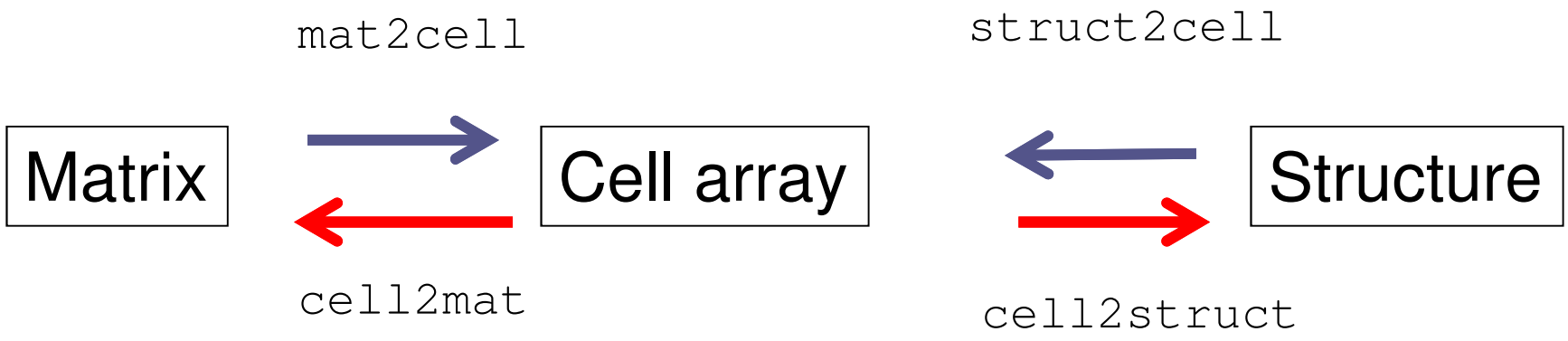
- Cell arrays and structures are two useful Matlab objects
- they are multidimensional arrays used to store data related to each other of different size and type
  - for example, storing the linguistic and numeric information about a student (name, surname, age, marks) is not allowed in a matrix (elements of the same kind)
- Structure
  - organized by **fields**
  - creation or access to the fields is made by using the **name** of the field and the . dot operator
- Cell array
  - organized by **cells** (regular arrangement)
  - creation or access to the cells is made by using **subscripts** between curly brackets '{ }'

Structures and cell arrays /2

- Structure: an example
  - `mystruct=struct` builds an empty structure with name '*mystruct*'
  - `mystruct.marks=[1 2 3; 4 5 6]` adds to the structure a field named `marks` and assigns a value to `marks` (2x3 matrix)
  - `mystruct.name='text'` adds to the structure a field named `text` and assigns a string value to `text`
  - `mystruct.age=30` adds to the structure a field named `age` and assigns a scalar value to `age`
- Cell array : an example
  - `mycellarray=cell(2,1)` builds an empty cell array of size 2x1 with name '*mycellarray*'
  - `mycellarray{1,1}=[1 2 3]` assigns a value of type vector to the cell in position {1,1}
  - `mycellarray{2,1}=['text']` assigns a value of type string to the cell in position {2,1}

Structures and cell arrays /3

- Conversion between matrices, cell arrays, and structures
  - `C=mat2cell(M), M=cell2mat(C)`
  - `C=struct2cell(S), S=cell2struct(C)`



Dot '.' operator

- The dot operator is used mainly in two ways:
- to perform element by element (element-wise) operations by alterating the behaviour of some mathematical operations
  - `A.*B`
- to access the fields in structures (Matlab objects)
  - `mystruct.A`

Flow control statements

- In Matlab we can find the flow control statements
- They operate like those in many computer languages, by altering the default sequencial flow control
- Conditional statements
  - `if...else` statement
  - `switch` statement
- Iterative statements (program loops)
  - `for` loop
  - `while` loop
  - `break` statement, is used to force the exit from a loop
  - `continue` statement, is used to go to the next loop iteration by skipping the remaining statements of the current iteration

`if…else` statement /1

- The `if` statement evaluates a condition (logical expression) and executes a statement (or group of statements) only if the specified condition is *true* (otherwise the statement is skipped)
- The `end` keyword terminates the `if` statement
- The keywords `elseif` and `else` are optional and provide alternative branches

(standard version)

```
if condition
    statement
end
```

```
if (x>0)
 y=sqrt(x);
end
```

`if…else` statement /2

(complete version)

```
if condition1
     statementA
elseif condition2
     statementB
else
     statementC
end
```

(mutually exclusive conditions)

The *statementA* is executed only if *condition1* is true, otherwise *condition2* is checked and *statementB* is executed

If no condition is true, *statementC* is executed

`switch` statement /1

- The *`switch`* statement evaluates a variable or an expression and executes a statement (or group of statements) based on the value of the expression
- It is used when several conditions must be handled
- The keywoard `case` and `otherwise` identify groups of statements
- The `end` keyword terminates the `switch` statement

`switch` statement /2

```
switch expression
    case value0
         statementA
    case value1
         statementB
  otherwise
          statementC
end

switch x
    case 0
      disp('x is 0');
    case {1,2}
      disp('x is 1 or 2');
    otherwise
      disp('error');
end
```

The *expression* is evaluated and a value is obtained. The program flow enters the corresponding case, by executing the specified *statement*

If no **case** is verified, *statementC* is executed

Only the first matching case is executed!

`for` loop

- The *`for`* loop repeates the execution of a statement (or group of statements) a fixed number of times, as specified by the *`condition`*
- The `end` keyword terminates the `for` statement

```
for condition
   statement
end
```

```
x=0;
for i=1:10,
    x=x+1;
end
```

`while` loop

- The *`while`* loop repeates the execution of a statement (or group of statements) an indefinite number of times, based on the control of a *`relation`* (logical condition)
- The *`statement`* is executed as long as the *`relation`* is true
- The `end` keyword terminates the `while` statement

```
while relation
    statement
end
```

```
x=0;
while (x<10)
      x=x+1;
end
```

Relational and logical operators /1

- The conditions in flow control statements can be expressed through relational and logical operators
- **Relational operators**
  - < less than, <= less than or equal
  - > greater than, >= greater than or equal
  - == equal, ~= not equal
  - Note that, a **single equal '='** is used in **assignments** (a=3), while a **double equal '=='** is used in **comparisons** (a==3) ?
- **Logical operators** (are used to combine two or more relations)
  - & and
  - | or
  - ~ not
- If the comparison involves matrices or arrays, the comparison is done element by element

Relational and logic operators /2
An example

- Multiple conditions are combined with a logical operator
- the statement in the body of the `if` is executed only if the two conditions are simultaneously met
  - the logical operator is the logic AND (&)

```
if ((x>0) & (y~=10))
    z=y+sqrt(x);
end
```

  - Always use round parentheses to set priority!
  - Result after evaluating a condition:
  - 1 (or any value different from 0) ⟹ TRUE
  - 0 ⟹ FALSE

Scalar functions

- built-in Matlab functions operating on **scalars**
- If applied to a **matrix** (or **vector**), they operate ***element-wise*** and produce a **matrix** (or **vector**) containing the result of the application of the function to the elements of the matrix
  - sine `sin(x)`, cosine `cos(x)`, …
  - exponential base e `exp(x)`
  - natural logarithm `log(x)`
  - absolute value `abs(x)`
  - Example: square root `sqrt(x)`
    - `sqrt(4)` returns `2`
    - `sqrt(A)`, with `A=[4 16 9 4]`, returns `2 4 3 2`

Array functions /1

- built-in Matlab functions operating on **arrays** (row array or column array)
- If applied to a **matrix**, they operate column by column and produce a row array containing the result of the application of the function to the columns of the matrix
  - Matlab operates by working on columns
- `max(x), min(x), sum(x), prod(x), mean(x),std(x)...`
  - the behavior changes when applied to matrices or vectors

Array functions /2

- `sum(A)` calculates the sum of elements
  - If *A* is a **vector**, *sum*(*A*) gives a **scalar** of value equal to the sum of the elements of the vector, if *A*=[1 2 3], sum(*A*)=6
  - If *A* is a **matrix**, *sum*(*A*) gives a **vector** containing the sum by columns of the elements of the matrix,

    if *A*=[1 2 3; 4 5 6] , sum(*A*)=[5 7 9]
- `mean(A), prod(A)`
  - they compute arithmetic mean and product of a vector or a matrix in the same manner
- `max(A), min(A)` extract the maximum and minimum value among elements
  - If *A* is a **vector**, *max*(*A*) extracts the maximum element in *A* (a **scalar**)
  - If *A* is a **matrix**, *max*(*A*) gives a **vector** containing the maximum values extracted by columns of matrix *A*

Other useful functions

- Rounding functions
  - `round(x)` rounds the value *x* to the nearest integer
  - `floor(x)` rounds the value *x* to the nearest *lower* integer
  - `ceil(x)` rounds the value *x* to the nearest *higher* integer
- `size(A)` returns the size of *A* as a 1x2 row vector containing the number of rows and columns of *A*
- `roots(c)` computes the roots of the n-degree polynomial whose coefficients are given in *c*

`find` function

- The `find` function returns the position of the elements in a matrix that meet a given condition
- `find(cond)` returns the **indices** of the elements which satisfy the condition `cond` on the specified array
- Behaviours of the `find` function:
  - `r=find(A>0)` finds the **linear indices** of positive elements in *A* (we assign the result to a single variable `r`)
  - `[i,j]=find(A>0)` finds the **row and column indices** of positive elements in *A (* we assign the result to a couple of variables `i` and `j`)

$$A = \begin{pmatrix} -1 & -2 & -3 \\ -4 & -5 & -6 \\ -7 & 8 & 9 \end{pmatrix}$$

r = 6 9 (linear indexes)

i = 3 3 (row indexes)
j = 2 3 (column indexes)

M-files - file.m

- M-files are files containing source code written in the Matlab language
- They have extension .m
- Two kinds of M-files exist:
  - Script file
  - Function file

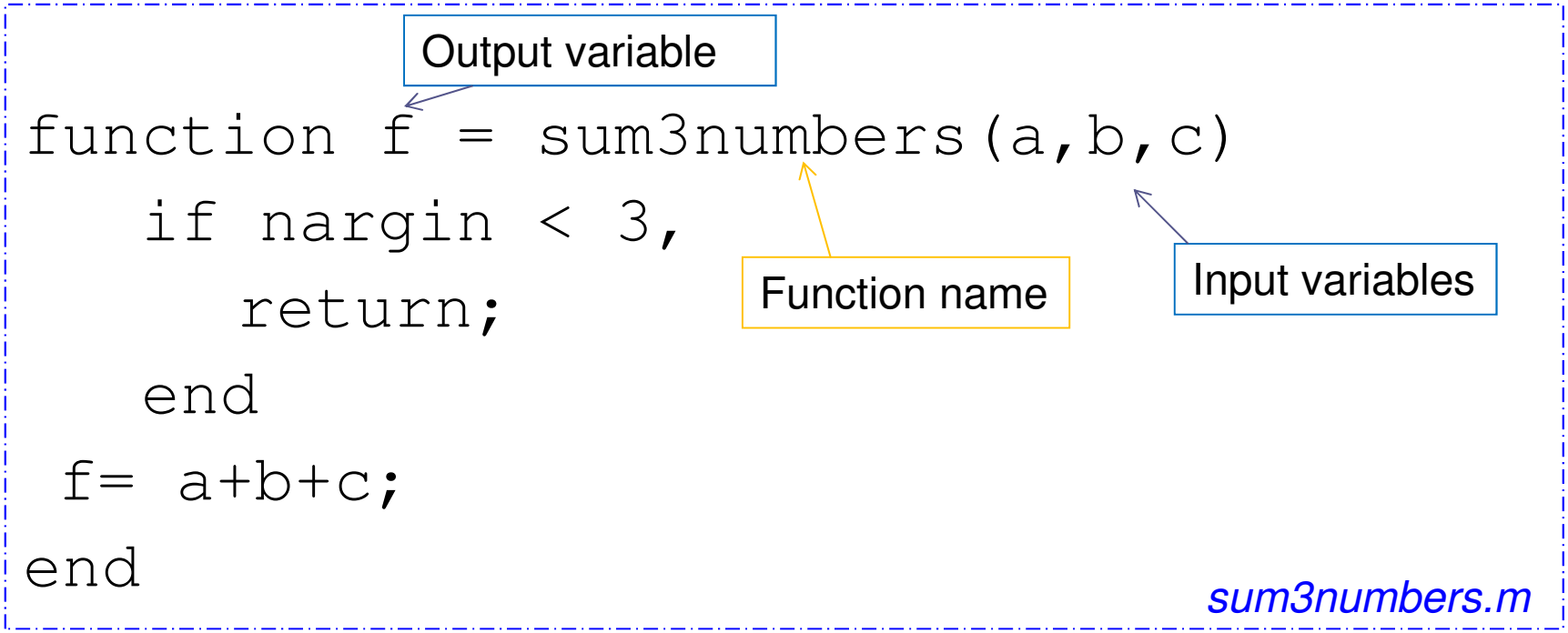
Script file

- Script file
  - consists of a sequence of statements without any initial declaration
  - can be executed by entering its name (the name of the script file) in the *Command window*
  - its invocation produces the execution of the statements
    - Invoking a script is the same as writing directly the statements of the script into the *Command Window!*
  - Variables in the script are **global** (visible outside the script)
  - does not return output arguments and does not accept input arguments

Function file

- Function file
  - An initial declaration is required
    - the declaration starts with the keyword `function`
    - the declaration contains the function name, the input arguments and the output arguments
    - `function f= function_name(var1, var2)`
    - `function [a,b,c]= function_name(var1, var2)`
  - can be called from other functions or scripts
  - Variables in the function are **local** (except if the global scope is defined)
  - The name given to the function must match the function file name !

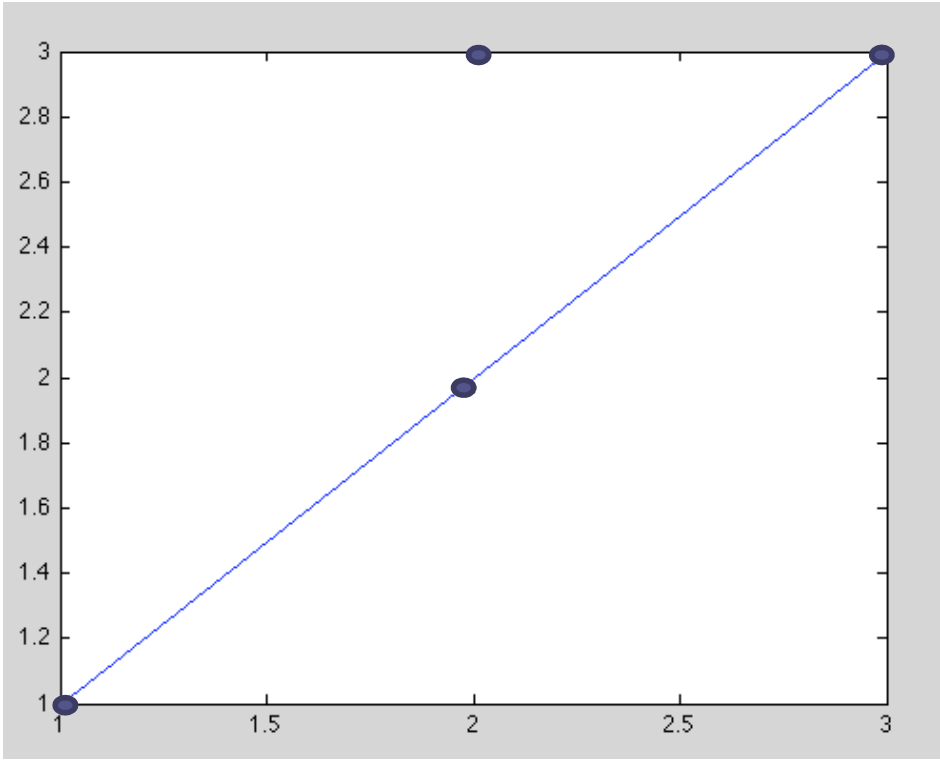
User-defined function 'sum3numbers'



- nargin = "**n**umber of **in**put **arg**uments"
- `return` causes a forced exit from the function if less than three input arguments are specified
- The name of the file ***must be*** *sum3numbers.m*!!!

Graphics in Matlab / 1

- The main function is `plot`
- It creates linear two-dimensional graphics given data points with coordinates (x,y)
  - `plot(2,3)` opens a figure window and draws the **point** (2,3)
    - with possible customization
  - `plot(x,y)` opens a figure window (if no figure windows are already opened) and draws the **points connected with a line** identified by the vectors `x=[1 2 3]` and `y=[1 2 3]`



Graphics in Matlab / 2

- Change the appearance of the graph components
  - both from the command line and from the GUI (command `plottools`)
  - `plot(x,y,'color_line_marker')` produces a line graph of *y* versus *x* and at the same time changes the appearance of the line, by setting a color, a line style and a marker symbol for data

Specifier	Color
r	Red
g	Green
b	Blue
c	Cyan
m	Magenta
y	Yellow
k	Black
w	White

Specifier	Line Style
-	Solid line (default)
--	Dashed line
:	Dotted line
-.	Dash-dot line

Specifier	Marker Type
+	Plus sign
o	Circle
*	Asterisk
.	Point
x	Cross
'square' or s	Square
'diamond' or d	Diamond
^	Upward-pointing triangle
v	Downward-pointing triangle

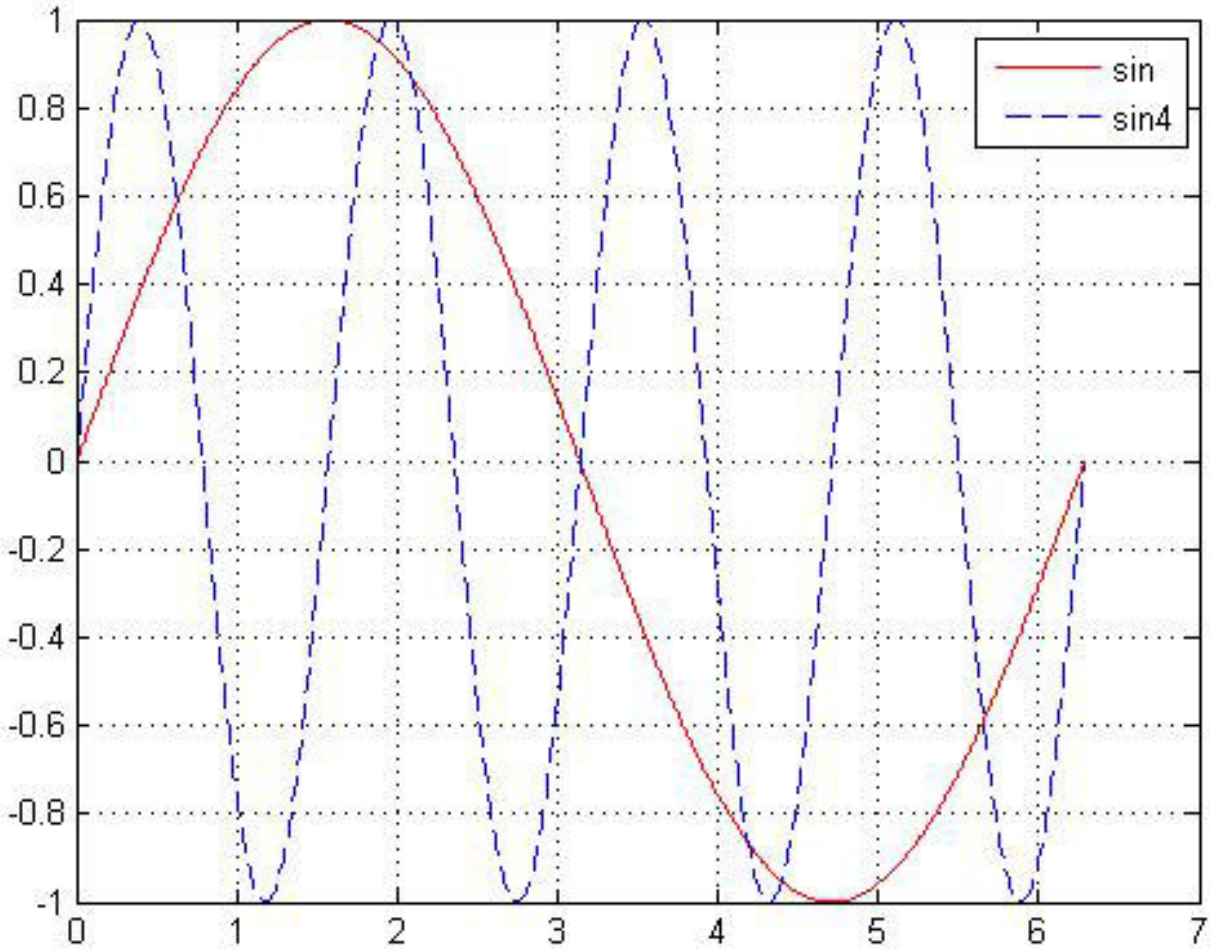
Graphics in Matlab / 3

- Add information to the graph
- Labels
  - `xlabel('text on x-axis')` adds a text label on x-axis or y-axis
  - `ylabel('text on y-axis')`
- Title
  - `title('title of the figure')` adds a title to the figure
- Legend
  - `legend('y versus x')` inserts a legend on the figure

Plot of the sine function /1

- `x=0:.01:2*pi` ; creates a vector of abscissa values ranging from 0 to 2pi
- `y1=sin(x);` computes the sine values of *x*
- `plot(x,y1,'r-');` plots the function with a red (r) continuous (-) line and no marker specified
- `hold on` allows to add plots on the same figure, by holding the current figure, without replacing the previous plots
- `y2=sin(4*x);` computes the sine values of 4**x*
- `plot(x,y2,'b--');` plots the function with a blue (b) dashed (--) line and no marker specified
- `grid on` shows a grid on the figure
- `legend('sin', 'sin4');` sets the legend

Plot of the sine function /2



Bar diagrams

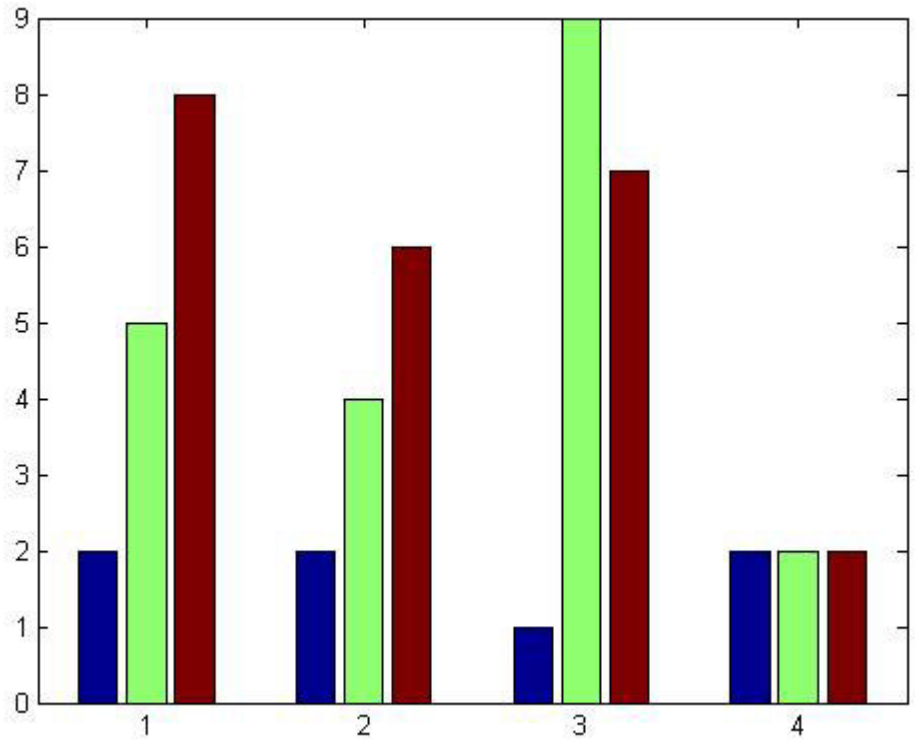
- Bar diagrams are used when we have groups of values to compare
- A bar graph displays the values in a matrix as vertical bars
- `bar(y)` builds the bar diagram of the values in `y`
- if `y` is a matrix 4x3, `bar` draws 4 groups of 3 bars each
  - on the abscissa is the index of the group

exams

blue *green* *red*

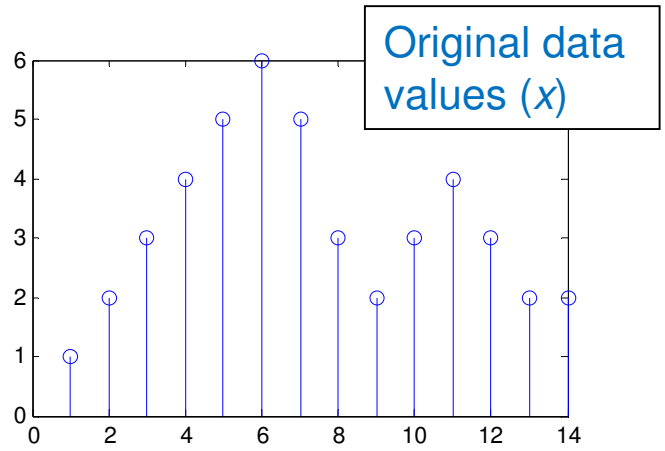
```
y = [2 5 8;    student 1
     2 4 6;    student 2
     1 9 7;    student 3
     2 2 2]    student 4
```

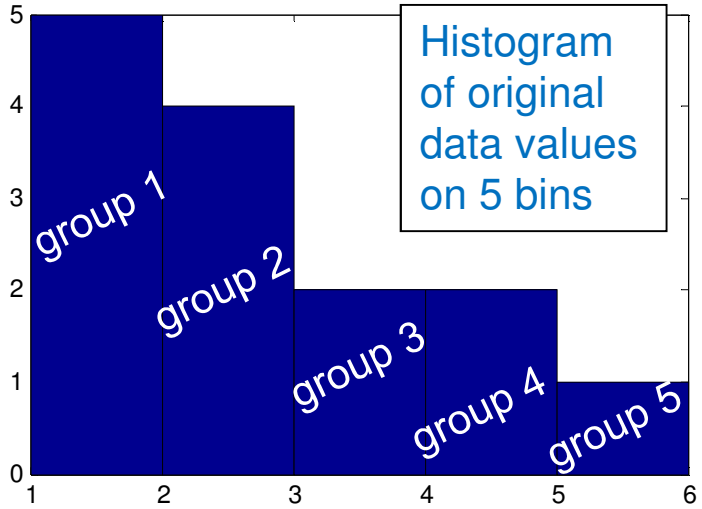
compare the marks of 4 students in 3 exams



Histograms /1

- Often is useful to consider groups (classes) on data
- A histogram shows the distribution in classes of the values of a variable
- `hist(x,c)` without output arguments
  - produces the histogram plot of the grouped data
  - groups the data `x` into:
    - `c` equally spaced bins, if `c` is a scalar
    - *n* bins having central values given in `c` , if `c` is a vector of length *n*
    - if `c` is omitted the default value is 10 bins





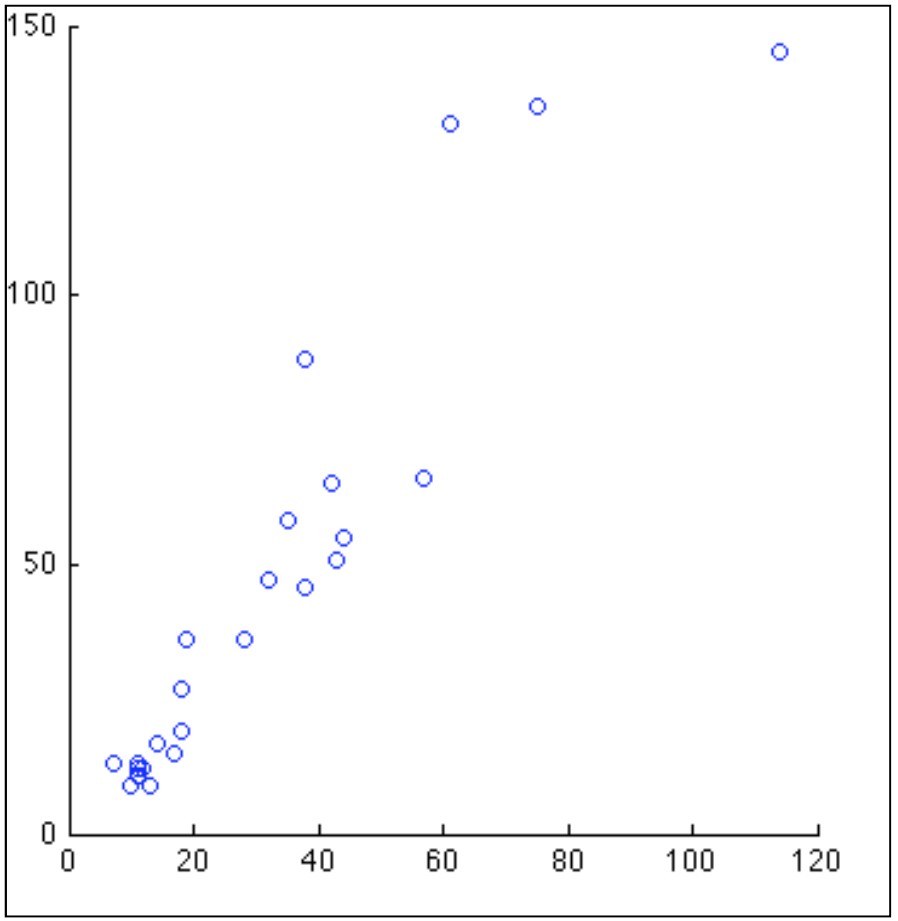
- `hist(x,5)`
- 5 groupes of values (on the abscissa)
- on the ordinate are the frequencies (how many values fall in each class)

Histograms /2

- `[f, r]=hist(x,c)` with two output arguments
  - groups the data `x` as specified before
  - returns the number `f` of elements in each bin (frequency) and the central value `r` of each class
  - no histogram plot is produced this time!

Scatter plot

- A scatter plot displays a set of data as a collection of points represented by two variables
- `scatter(x,y)` displays the data (`x`, `y`) as a collection of single points
- Scatter diagrams are useful to show the kind of correlation existing between two variables *x* and *y*



Missing data /1

- When working with series of data we may find inconsistent data (special characters or missing values) due to:
  - black-out of the sensor
  - errors in the transfer or in the import of data
  - processing errors (such as division by 0)
- Matlab represents missing data or inconsistent data using the character *NaN* (**N**ot **a** **N**umber)
- *NaN* are not ignored in Matlab and may produce processing errors!

Missing data /2

- How to identify missing data
  - `isnan(x)` if applied to a matrix, shows which elements of `x` are NaN elements (value 1 means true)
  - `f=find(isnan(x))` returns the indices that satisfy the condition on NaN values in `x`
- How to solve missing data
  - Several solutions are available:
  - `x(f)=val` replacement of the NaN value with the correct value `val` if it is known
  - `x(f)=[]` removal of the missing value
  - interpolation of the missing value

Outliers

- Outliers are values that deeply differ from the other values in the rest of data and can distort the analysis of data:
  - a measurement error or processing error
  - a real (actually significant) data but anomalous (due to an extraordinary event)
- How to identify outliers (in data having normal distribution)
  - *rule of thumb*: potential *outliers* are the data far away from the mean *M* for more than 3 times the standard deviation *s*
    - `abs(data-M)>3*s`

Other useful commands /1

- `who` lists the variables currently defined in the *Workspace*
- `whos variable_name` shows only the informations about the specified variable
- `clear` removes all the variables from the *Workspace*
- `clear variable_name` removes only the specified variable
- `clc` clears the *Command Window*

Other useful commands /2

- `disp('a text')` displays the text string between quotes
- Anything following a percent sign **%** is ignored as it is seen as a comment
  - **`% this is a comment`**
- File system-related commands
  - `cd` allows to change the current working directory and to move in the file system
  - `pwd` shows the name of current directory
  - `ls` shows the content of current directory
- `help function_name` displays information about the command or function specified (syntax and examples of use)
  - The Help is also available in the Help Browser (in the graphical format with `doc function_name`)

Load and save data /1

- When quitting Matlab all variables defined in the *Workspace* are lost
- To save or load the *Workspace*, use:
  - `save('mysession.mat')` saves the Workspace in the specified file mysession.mat (binary MAT-file)
    - MAT-file Matlab file to store data
  - `save('mysession.mat',p,q)` saves only the specified variables `p` and `q`
  - `load ('mysession.mat')` loads in the Workspace the content of the specified binary file, restoring the former state of the Workspace
    - the *load* function reads binary files (such as **.mat** files or **.dat** files)
    - the *load* function reads text files (**.txt** files) well organized as a rectangular regular table of numbers

Load and save data /2

- To load data from external files
- Excel files
  - `[txt, num]=xslread('filename.xls')` reads data from an Excel file and puts data in `num` and textual data in `txt`
  - `xslwrite('filename.xls',A)` writes data (contained in matrix `A`) in the specified Excel file
- Comma-separated-value files
  - `M=csvread('filename.csv')` reads data from an csv file
  - `csvwrite('filename.csv',A)` writes data (contained in matrix `A`) in the specified .csv file
- for more complex data structure, use the Import… tool