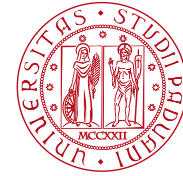


Introduction to MATLAB

Riccardo Antonello
(riccardo.antonello@unipd.it)

Giulia Michieletto
(giulia.michieletto@unipd.it)

Dipartimento di Tecnica e Gestione dei Sistemi Industriali

Università degli Studi di Padova

4 Marzo 2024

This work is licensed under a
Creative Commons Attribution-NonCommercial-ShareAlike 4.0
International License

MATLAB Scripts

MATLAB is primarily an *interpreted* language[1].

MATLAB programs (**MATLAB scripts**) are basic text files, with extension `.m`, containing multiple sequential lines of MATLAB *commands*, *function calls* and *comments*.

(1) Compilation of MATLAB scripts is also possible by using the **MATLAB Compiler**.

MATLAB Scripts

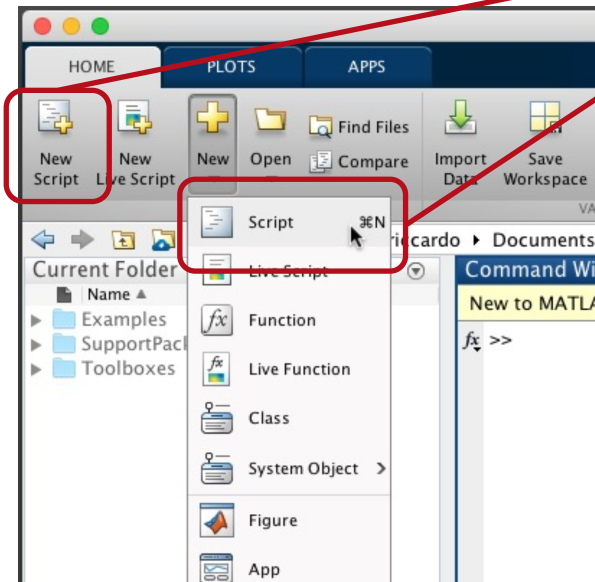
MATLAB scripts can be prepared with any conventional text-editor, or with the MATLAB built-in editor (**`edit`**).

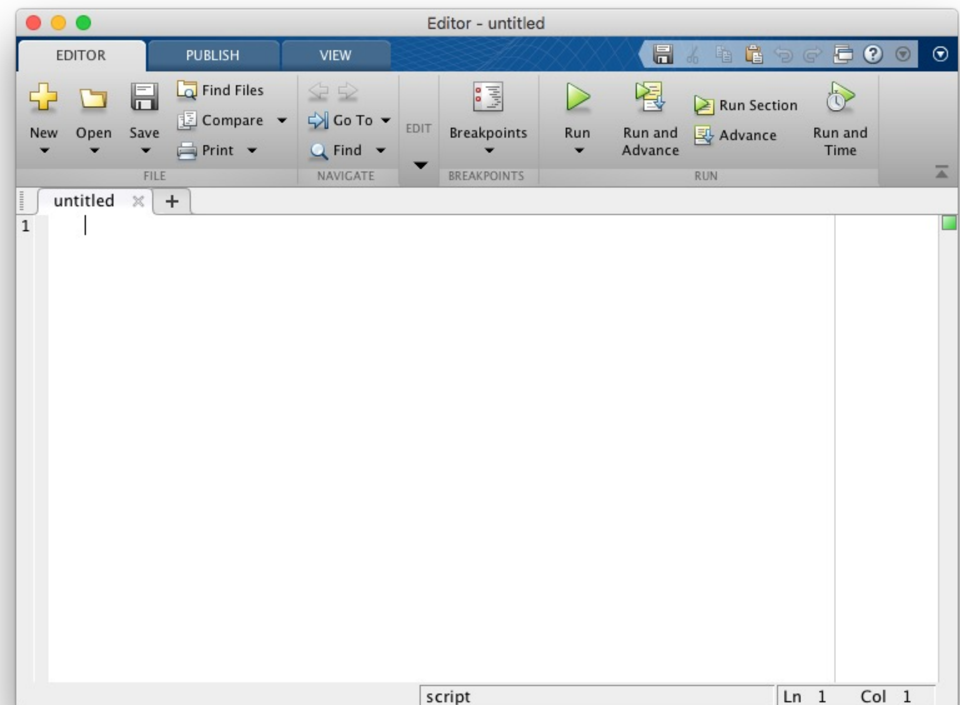
MATLAB Editor Window

```
>> edit
```

or ...

or ...





MATLAB Scripts

Save MATLAB script to specified folder
(e.g. `intersect.m` in current working directory – `pwd`)

```
%% Find intersection of two lines

% lines coefficients
a1 = 1.5; b1 = 1; c1 = 6;   % line 1: a1*x + b1*y = c1
a2 = -0.5; b2 = 1; c2 = 2;  % line 2: a2*x + b2*y = c2

% compute intersection
A = [a1, b1; a2, b2];
C = [c1; c2];
X = A\C;

% display results
disp('x = ');
X(1)
disp('y = ');
X(2)
```

Code lines

Comments start with `%`

The first comment lines of a script are printed as help when using the command:

```
>> help <script_name>
```

MATLAB Scripts

To run a MATLAB script, either push the *Run* button on the editor toolbar, or type its name at the command line, i.e.

```
>> intersect.m
```

Script outputs are displayed on the Command Window

Editor - /Users/riccardo/Documents/MATLAB/intersect.m

EDITOR | PUBLISH | VIEW

New | Open | Save | Find Files | Compare | Print | Go To | Find | EDIT | Breakpoints | Run | Run and Advance | Run Section | Advance | Run and Time

FILE | NAVIGATE | BREAKPOINTS | RUN

intersect.m

```
%%  Find intersection of two lines

%   lines coefficients
a1 = 1.5; b1 = 1; c1 = 6;   %   line 1:  a1*x + b1*y = c1
a2 = -0.5; b2 = 1; c2 = 2;  %   line 2:  a2*x + b2*y = c2

%  compute intersection
A = [a1, b1; a2, b2];
C = [c1; c2];
X = A\C;

%   display results
disp('x = ');
X(1)
disp('y = ');
X(2)
```

script | Ln 16 | Col 5

Import Data | Save Workspace | New Variable | Open Variable | Clear Workspace | Favorites | VARIABLE | CODE

ardo ▸ Documents ▸ MATLAB ▸

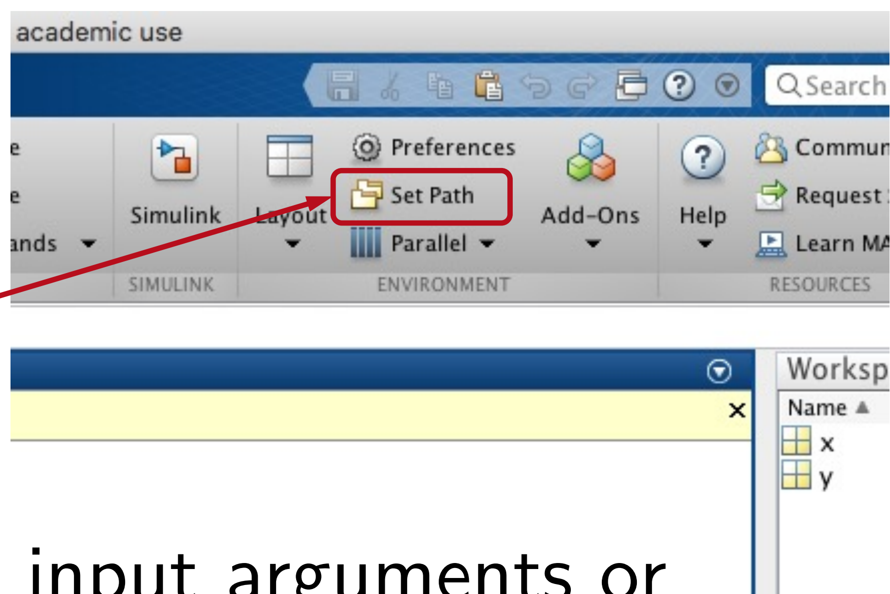
Command Window

New to MATLAB? See resources for Getting Started.

```
>> intersect
x =

ans =

     2

y =

ans =

     3

>>
```

MATLAB Scripts

To run a script, the file must be in the current folder (see: **`pwd`**) or in a folder on the *search path* (see: **`path`**).



Configure the default search path.

MATLAB scripts have no input arguments or output parameters; they operate on variables defined in the workspace.

User-defined MATLAB Functions

User-defined **MATLAB Functions** are similar to scripts, but:

- they can accept *input arguments*, and return *output parameters.*
- they have their own *local scope*, different from the main workspace: the variables ...
  - ... defined within the function are not visible outside;
  - ... in the main workspace are not visible in the function.

User-defined MATLAB Functions

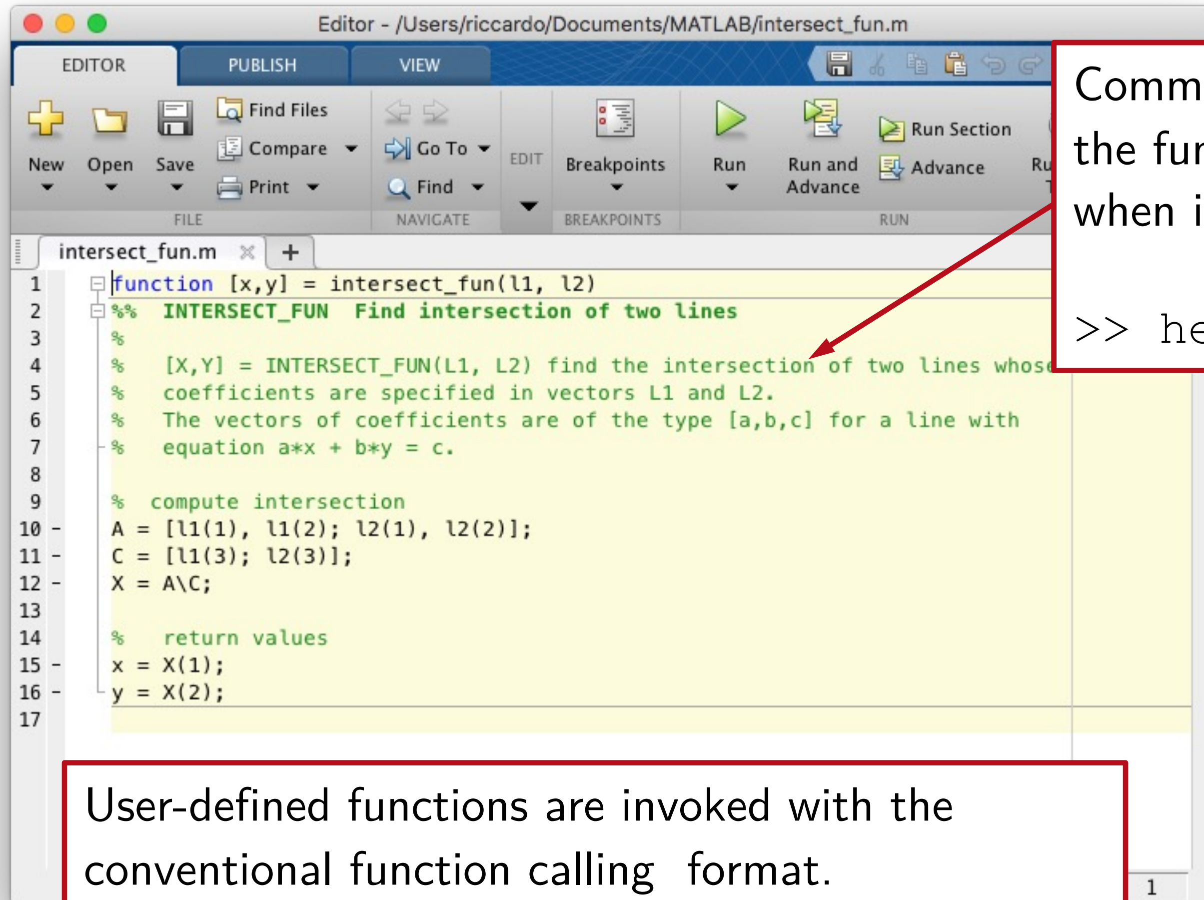
User-defined MATLAB Functions:

- must be saved on a *M-file* (text file with extension `.m`) with the same name of the function.
- the format of 1st non-empty line in the M-file must be:

```
function [<out1>, …, <outN>] = <fcn_name>(<in1>, …, <inM>)
```

- they can be invoked from the Command Window or other scripts/functions with the conventional function calling format.

User-defined MATLAB Functions



Comment lines at the beginning of the function are printed as help when invoking the command:

```
>> help <fcn_name>
```

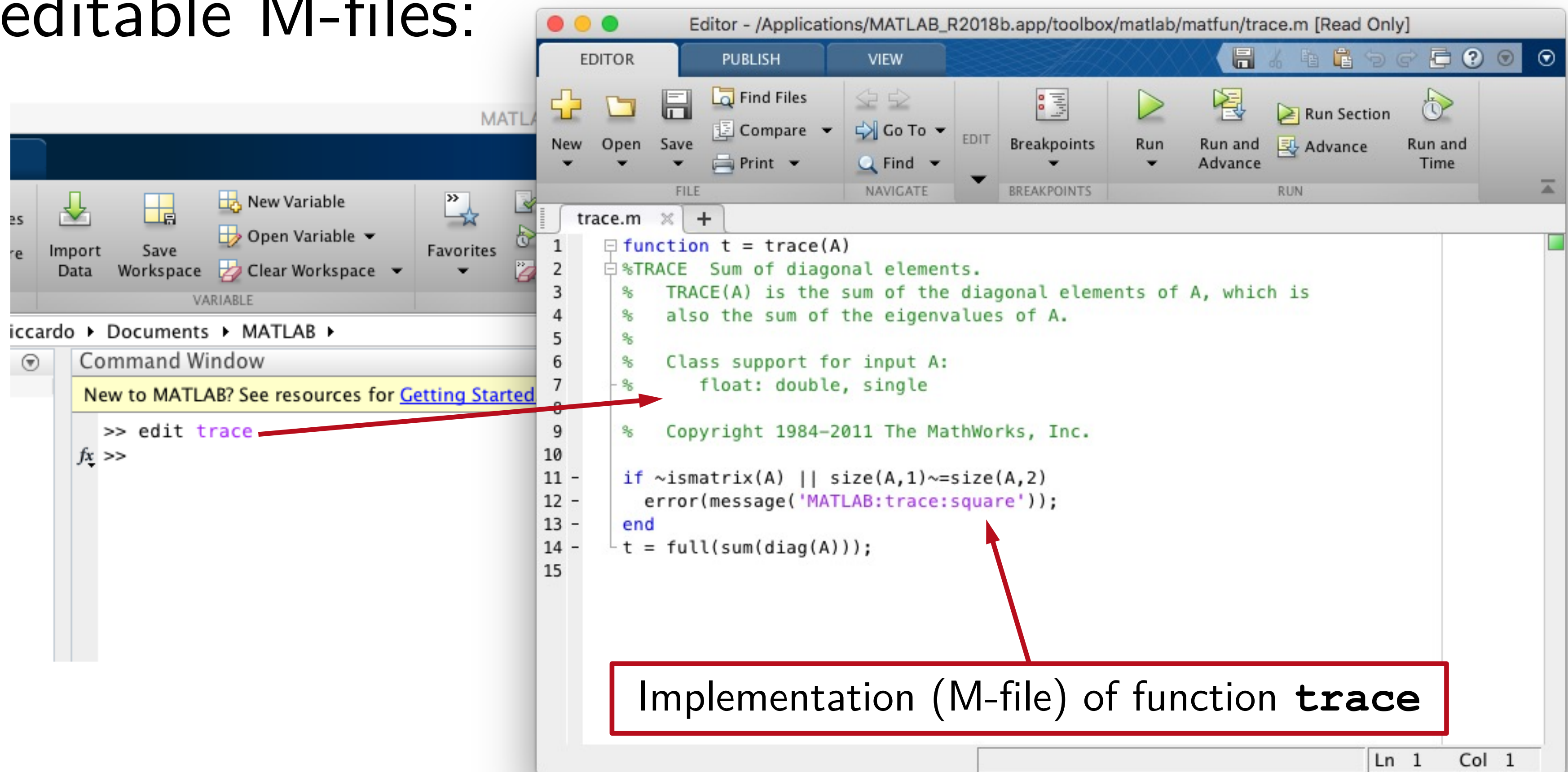
User-defined functions are invoked with the conventional function calling format.

<u>Note</u>: the *Run* button on the editor toolbar does not work with functions (because a function must be invoked by specifying the input arguments).

```
>> [x,y] = intersect_fun([1.5, 1, 6], [-0.5, 1, 2])

x =

     2


y =

     3

>>
```

User-defined MATLAB Functions

Most of the MATLAB Functions[1] are regularly editable M-files:



(1) Exceptions include *built-in*, *compiled* (e.g. *mex*) and *protected* (*p-code*) functions.

Precedence Order for Names

Names are resolved by MATLAB with the following precedence order (*simplified*):

1. Variable.
2. Built-in function.
3. Function in the current folder.
4. Script in the current folder.
5. Function or script elsewhere on the path, in order of appearance.

Structured Programming

Sequence: ordered statements or subroutines executed in sequence.

Selection: one or a number of statements executed depending on the state of the program:

↳ **`if`** … **`else`** … **`elseif`** … **`end`**

↳ **`switch`** … **`case`** … **`otherwise`** … **`end`**

Iteration: one or a number of statements repeatedly executed until the program reaches a certain state:

↳ **`for`** … **`end`**

↳ **`while`** … **`end`**

if, elseif, else

```
if expression
    statements
elseif expression
    statements
else
    statements
end
```

Typically includes relational operators (e.g. **<**, **<=**, **==**, ...).

Optional parts.

<u>Example</u>

```
if rem(x,2)==0
    disp('x is even');
elseif rem(x,2)==1
    disp('x is odd');
else
    disp('x is not a natural number');
end
```

switch, case, otherwise

```
switch switch_expression
    case case_expression
        statements
    case case_expression
        statements
        ...
    otherwise
        statements
end
```

The switch block tests each case until one of the case expressions is true. When:

`switch_expression == case_expression`

MATLAB executes the corresponding statements and exits the switch block immediatley.

Optional part.

Example

```
switch rem(x,2)
    case 0
        disp('x is even');
    case 1
        disp('x is odd');
    otherwise
        disp('x is not a natural number');
end
```

`for` loop

```
for index = values
    statements
end
```

`values` has one of the following forms:

1) `init_value:end_value`

2) `init_value:step:end_value`

3) `array_of_values`

- Can contains other for-loops (i.e. for-loops can be *nested*).
- The current loop can be terminated with the command **`break`**.

Example

```
s = 0;
v = rand(1,20);
for k = 1:length(v)
    s = s + v(k);
end
disp(['sum = ', s]);
```

`while` loop

```
while expression
    statements
end
```

Typically includes relational operators (e.g. **<**, **<=**, **==**, ...).

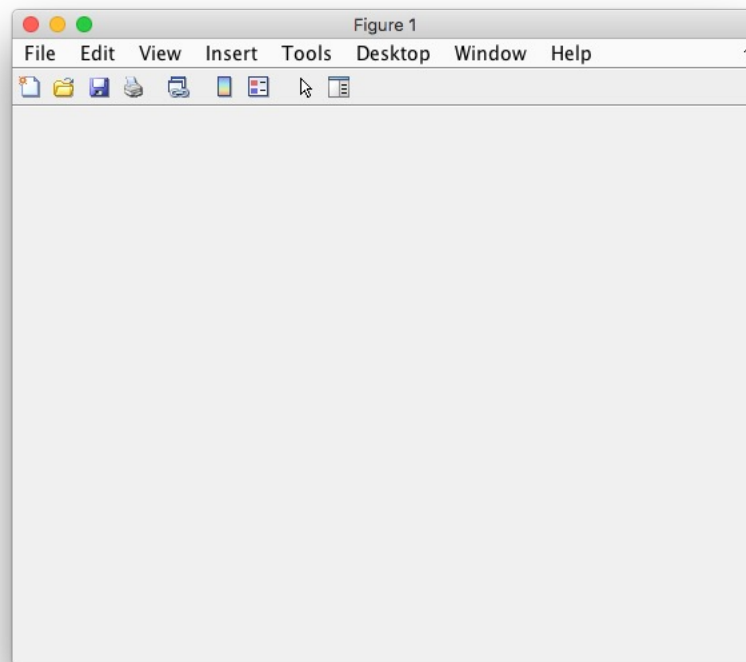
- Can contains other while-loops (i.e. while-loops can be nested).
- The current loop can be terminated with the command **`break`**.

Example

```
s = 0;
v = rand(1,20);
k = 1;
while k < length(v)
    s = s + v(k);
    k = k + 1;
end
disp(['sum = ', s]);
```

Data plotting

Plots are displayed on dedicated windows, called MATLAB **figures**.



`figure``(n)`	Finds the open figure with id-number n, and makes it the current figure. If no figure exists with that number, creates a new figure `n`.
`close``(n)`	Close the figure with id-number `n`.
`close all`	Close all open figures.

Data plotting

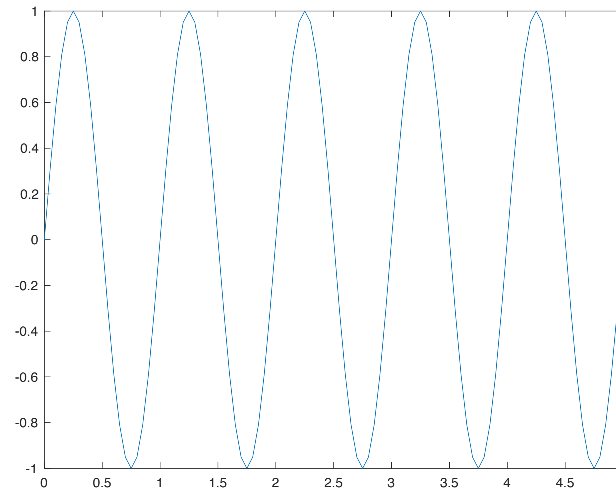
`plot(X, Y, LineSpec)` Creates a 2-D line plot of the data in `Y` vs the corresponding values in `X`. If `X` is omitted, data are plotted vs the corresponding vector index value.

`LineSpec` is a string that specifies the *line style*, *marker symbol*, and *color*.

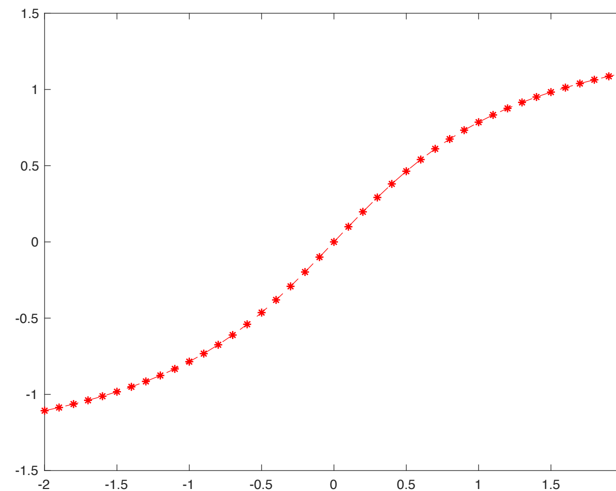
Line Style		Marker Symbol		Color	
`-`	solid	`.`	point	`y`	yellow
`:`	dotted	`o`	circle	`m`	magenta
`-.`	dash-dotted	`x`	x-mark	`c`	cyan
`--`	dashed	`+`	plus	`r`	red
		`*`	star	`g`	green
		`s`	square	`b`	blue
		`d`	diamond	`w`	white
		`v`	triangle (down)	`k`	black
		`^`	triangle (up)		
		`<`	triangle (left)		
		`>`	triangle (right)		
		`p`	pentagram		
		`h`	hexagram		

Data plotting

```
>> t = 0:0.05:5;
>> y = sin(2*pi*t);
>> plot(t, y);
```




```
>> x = -2:0.1:2;
>> y = atan(x);
>> plot(x, y, 'r*--');
```

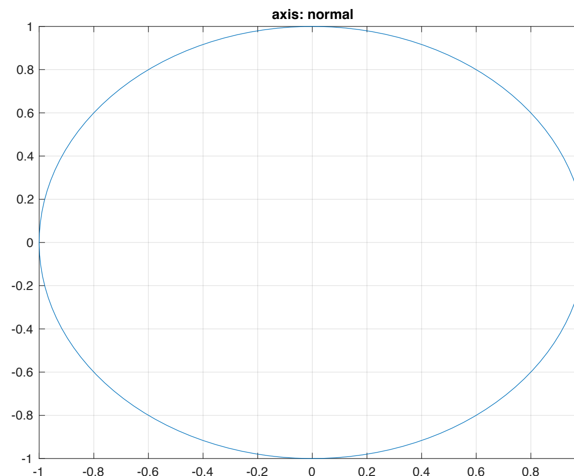


Data plotting

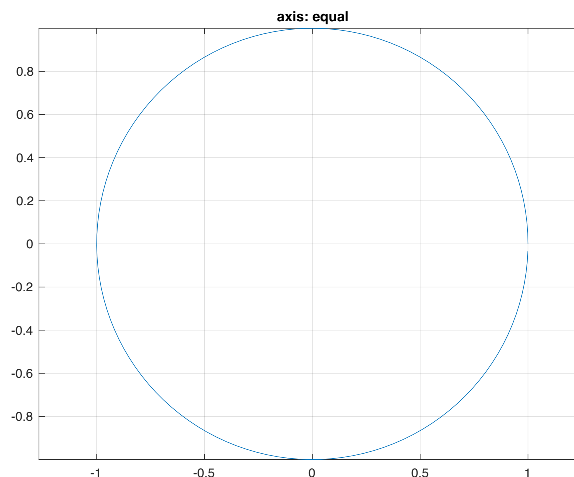
title(s)	Adds title s to current axes.
xlabel(s), **ylabel**(s)	Labels x-axis/y-axis of current axes.
grid on/**off**	Displays/hides grid lines on current axes.
axis(limits)	Specifies the limits for the current axes ([xmin, xmax, ymin, ymax]). Use **xlim**/**ylim** to set limits only for the x-axis/y-axis.
axis style	Uses a predefined style to set the limits and scaling: • **equal**: same length for the data units along each axis. • **square**: axis lines with equal lengths. • **normal**: restore the default behavior.
axis mode	Sets whether MATLAB automatically chooses the limits or not (**manual**/**auto**).

Data plotting

```
>> theta = 0:0.05:2*pi;
>> x = cos(theta);
>> y = sin(theta);
>> figure;
>> plot(x, y);
>> grid on;
>> title('axis: normal');
```



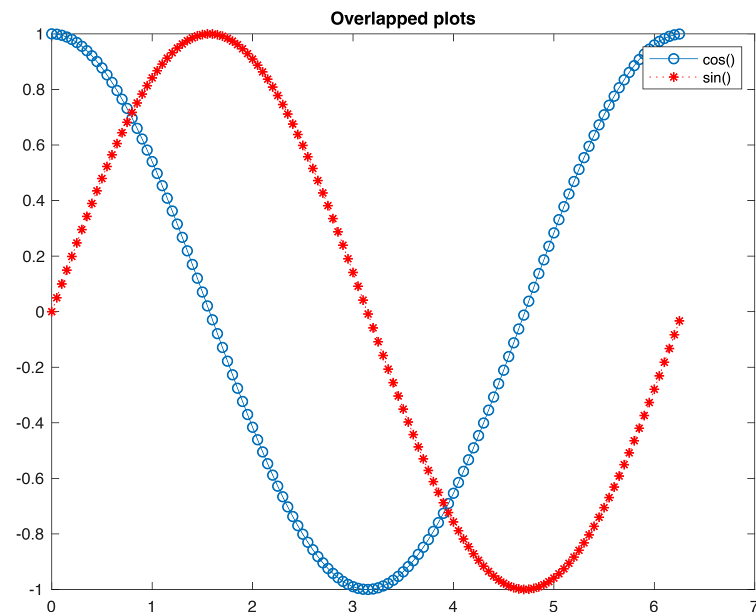

```
>> figure;
>> plot(x, y);
>> grid on;
>> axis equal;
>> title('axis: equal');
```



Data plotting

hold on/off	Retains current plot when adding new plots on same axes.
legend(s1,...,sN)	Creates a legend with labels s1,...,sN for the plotted data series.

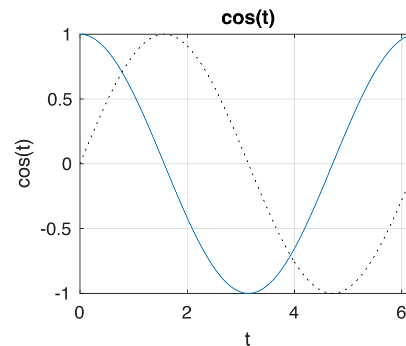
```
>> theta = 0:0.05:2*pi;
>> x = cos(theta);
>> y = sin(theta);
>> figure;
>> plot(theta, x, 'o-');
>> hold on;
>> plot(theta, y, 'r*:');
>> legend('cos()', 'sin()');
>> title('Overlapped plots');
```

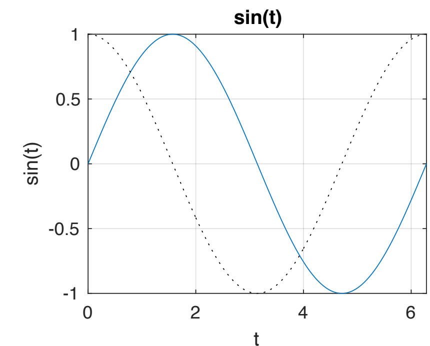


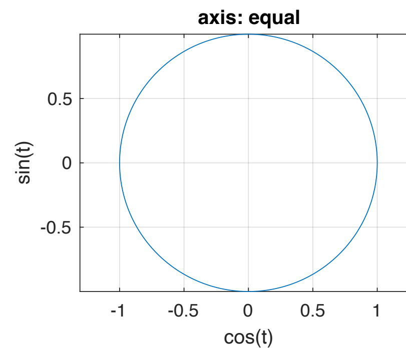
Data plotting

subplot(m,n,p) Divides the current figure into an m-by-n grid, and creates axes in the position specified by p.

```
>> t = linspace(0,2*pi,400);
>> x = cos(t); y = sin(t);
>> subplot(2,2,1); % Top-left plot
>> plot(t,x); hold on;
>> plot(t,y,'k:');
>> grid on;
>> title('cos(t)');
>> xlabel('t'); ylabel('cos(t)');

>> subplot(2,2,2); % Top-right plot
>> plot(t,y); hold on;
>> plot(t,x,'k:');
>> grid on;
>> title('sin(t)');
>> xlabel('t'); ylabel(sin(t)');

>> subplot(2,2,3); % Bottom-left plot
>> plot(x,y); grid on;
>> axis equal;
>> title('axis: equal');
>> xlabel('cos(t)'); ylabel('sin(t)');
```





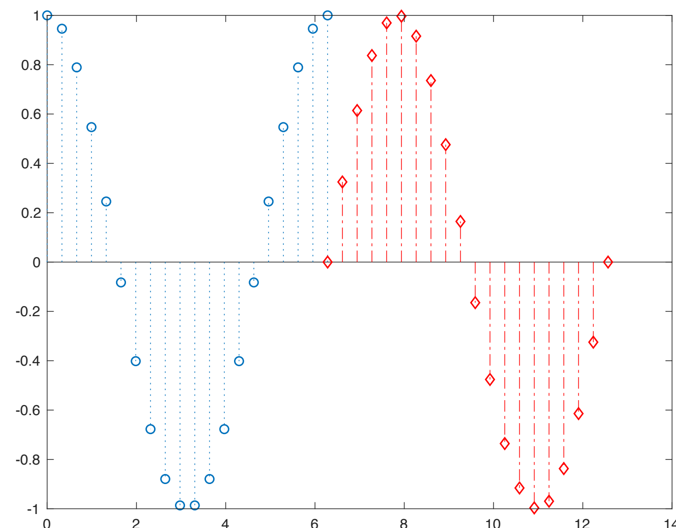


Data plotting

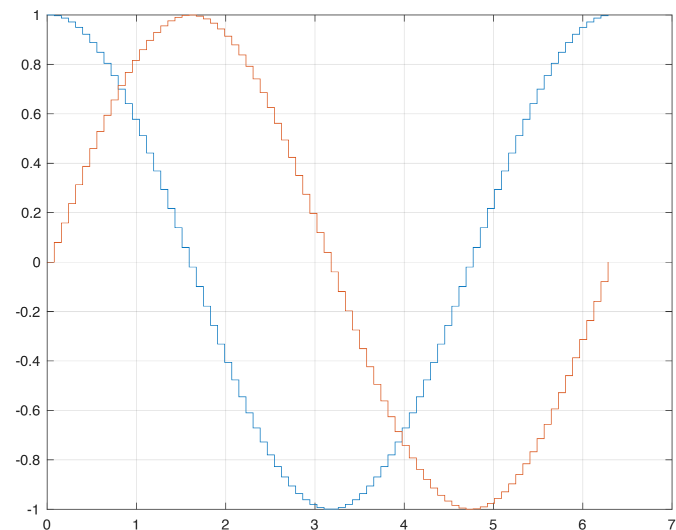
semilogx(X,Y)	plot data as logarithmic scales for the *x*-axis.
semilogy(X,Y)	plots data with logarithmic scale for the *y*-axis.
loglog(X,Y)	log-log scale plot.
stem(X,Y)	plots the data sequence `Y`, at values specified by `X`, as stems that extend from a baseline along the *x*-axis.
stairs(X,Y)	draws a stair-step graph of the elements in `Y`, at the locations specified by `X`.
polarplot(theta,rho)	plots a line in polar coordinates, with `theta` indicating the angle in [rad], and `rho` indicating the radius value for each point.
bar(x,y)	creates a bar graph with one bar for each element in `y`, at locations specified by `x`.
histogram(X,nbins)	creates a histogram plot of data `X`, using `nbins` number of bins (uniform subintervals in `[min(X),max(X)]`).

Data plotting

```
>> tx = linspace(0,2*pi,20);
>> ty = linspace(2*pi,4*pi,20);
>> x = cos(tx); y = sin(ty);
>> figure;
>> stem(tx, x, ':');
>> hold on;
>> stem(ty, y, 'rd-.');
```

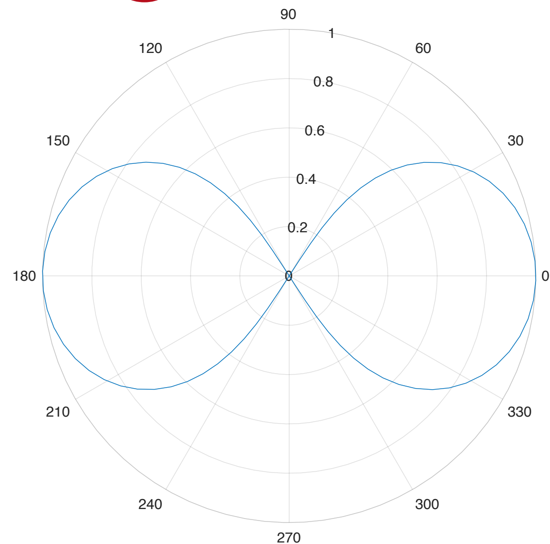



```
>> t = linspace(0,2*pi,80);
>> x = cos(t);
>> y = sin(t);
>> figure;
>> stairs(t,[x', y']);
>> grid on;
```

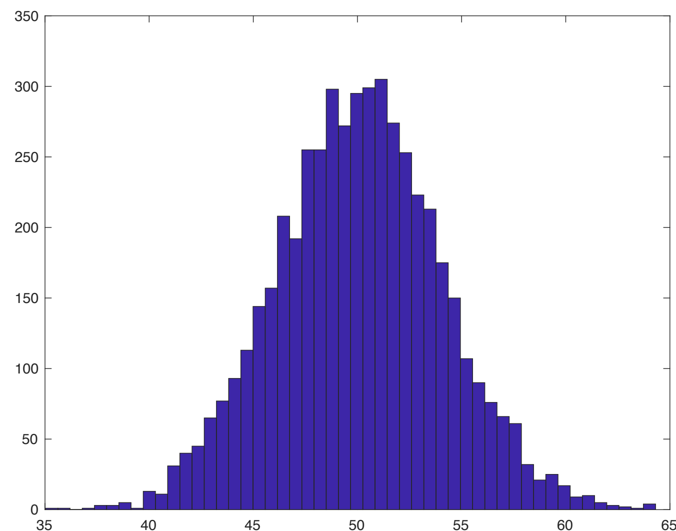


Data plotting

```
>> t = linspace(0,2*pi,80);
>> x = cos(t);
>> y = sin(t);
>> figure;
>> polar(x,y);
>> grid on;
```




```
>> figure;
>> hist(50+4*randn(1,5000),50)
```

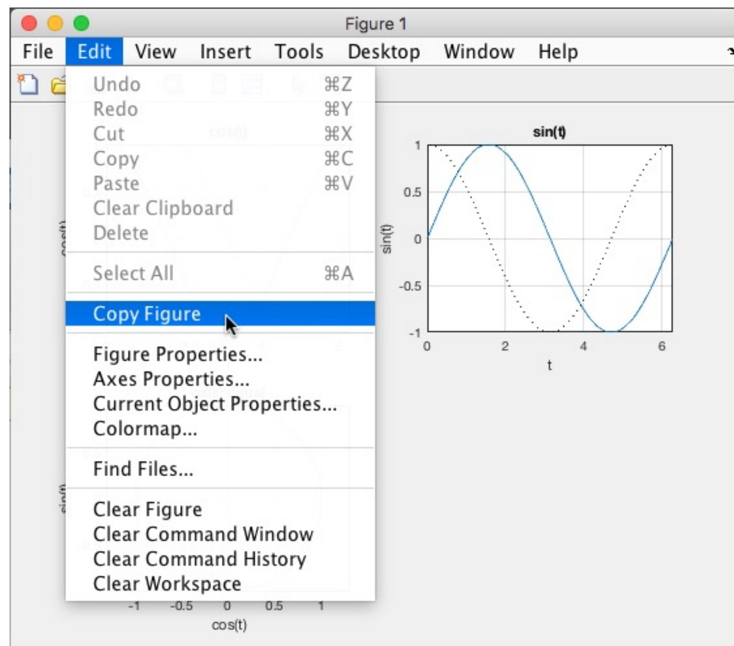


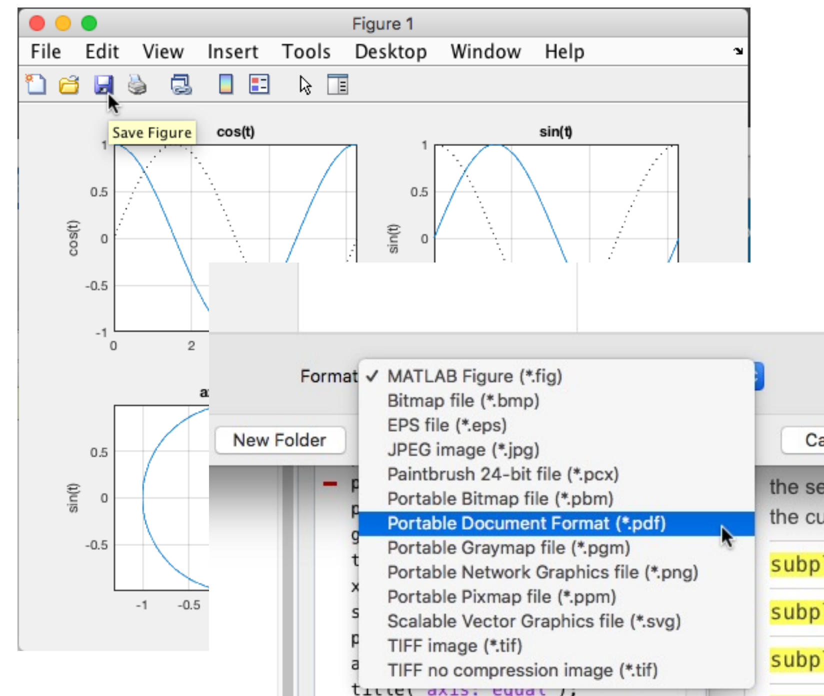
Exporting figures

To export a figure into other applications:

Option 1: select *Edit* $\rightarrow$ *Copy Figure* to copy the Figure into Clipboard; then, paste figure on the other application.

Option 2: save figure with one of the supported formats; then, import the figure (as a file) on the other application.





General recommendations

- Remind that MATLAB is *case-sensitive*.
- Remind that array indexing starts from 1 (and not from 0).
- Perform array indexing compatibly with the array dimensions.
- Do not define variables with names of predefined keywords, variables, functions, etc.
- Do not confuse matrix operators (`*`, `/`, `^`, ...) with their *element-wise* versions (`.*`, `./`, `.^`, ...).

General recommendations

For performance improvements:

- Avoid *for-loops* for processing arrays; try to *vectorize* operations on arrays whenever possible.
- Avoid dynamic array resizing whenever possible; use array pre-allocation instead (e.g. initialize a matrix by using `zeros` with the requested size).