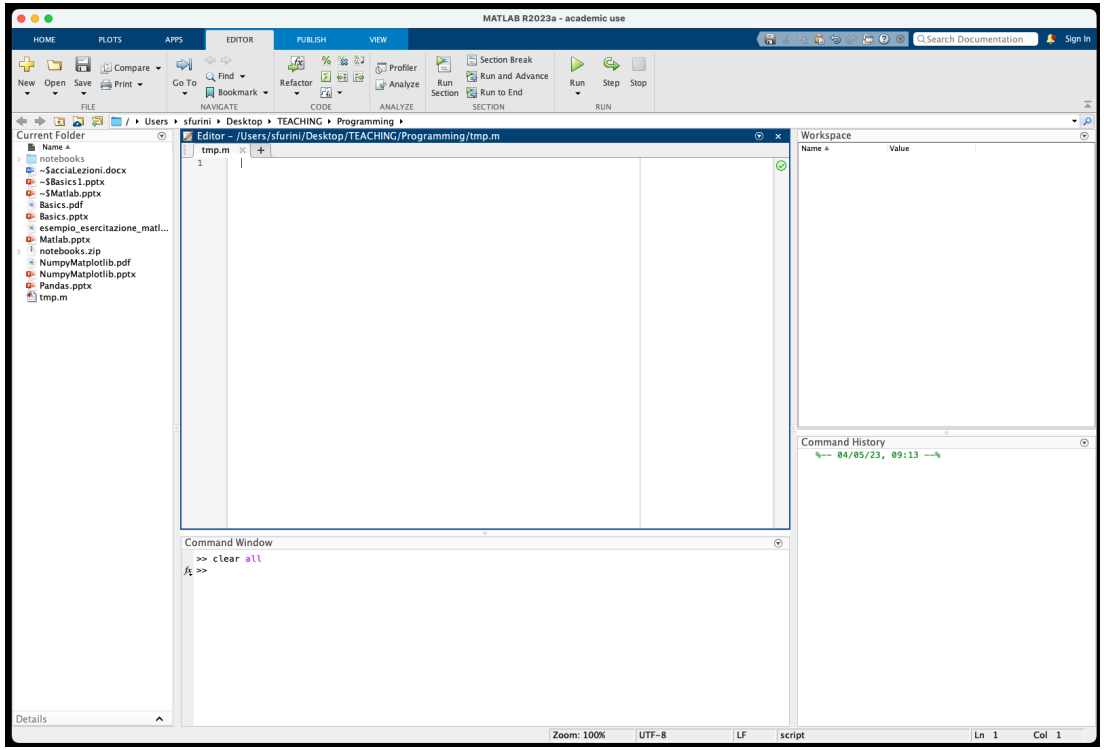
MATrix LABoratory



- High-level language for numerical computation
- Interpreted (but there's also a compiler)
- Written in C/Java
- Widely used in science and engineering
  - many toolboxes

(some) useful commands to check/modify the workspace and the command window

clc	clear command window
clear	remove all variables from workspace
clear X	remove variable X from workspace
who	list variables in the workspace
whos	list variables in the workspace with more infos
close	close figure windows

(some) useful predefined symbols

eps	accuracy of floating point values
i,j	imaginary unit
Inf	infinity
NaN	not a number
pi	the number pi

Comments & End of Instructions

- one line = one instruction
- ; terminates the instruction and prevents printing the results to stdout
- … for going to the next line
- % for comments

```
>> a = 1; b = 2; c = 3;
```

```
>> a = 1 % this is a comment
a =
     1
```

```
>> a = 1;
```

```
>> a = ...
1;
```

Naming objects

- Object names are made of letters, digits and underscores
- The first character must be a letter
- Names are case-sensitive
- Length up to the environment variable **namelengthmax** (63)

```
>> this_is_a_perfectly_file_name_of_a_variable = 1;
```

help <COMMAND>

```
>> help disp
 disp Display array.
    disp(X) displays array X without printing the array name or
    additional description information such as the size and class name.
    In all other ways it is the same as leaving the semicolon off an
    expression except that nothing is shown for empty arrays.

    If X is a string or character array, the text is displayed.

    See also formattedDisplayText, sprintf, num2str, format, details.

    Documentation for disp
    Other uses of disp
```

functions – client side

[OUT1, OUT2, …] = function_name(IN1, IN2, …)

- anything in blue is not compulsory
- any number of return values is possible (in between square brackets)
- any number of input parameters is possible
- there are no parameters with default values

Input/output functions

disp(OBJ): shows OBJ to stdout

fprintf

- fprintf(FORMAT_STRING, VAR1, VAR2, ….)
  - with FORMAT_STRING defined as in c
  - example: fprintf('x = %f\n', 5.1);
- fprint(FID, FORMAT_STRING, VAR1, VAR2, …)
  - with FID being an integer file identifier
    - 1 for stdout
    - 2 for stderr

var = input(PROMPT)

Shows PROMPT and wait input from keyboard

The class of var is guessed automatically

Use '' for strings

```
>> x = input('Insert a floating number: ');
Insert a floating number: 3.14
>> fprintf('x = %4.3f\n', x)
x =  3.140
```

```
>> x = input('Insert a string: ');
Insert a string: 'Hello !'
>> fprintf('x = %s\n', x)
x = Hello !
```

Elementary data types

int[8,16,32,64]	[8,16,32,64]-bit integer
uint[8,16,32,64]	[8,16,32,64]-bit unsigned integer
single	single precision floating number
double	double precision floating number (the default)
logical	Booleans: true, false
char	s = 'this is a char variable' it's an array of characters
string	s = "this is a string variable" container for text, with functions for manipulating strings

- *intmax*(int_type), *intmin*(int_type) to know the maximum/minimum integer value
- *realmax*(real_type), *realmin*(real_type) to know the maximum/minimum real value
- the name of the elementary data type is also the name of the function that can be used for converting, e.g. x = *int8*(3)
- *class*(object): return the class of the object

Every variable is an Array

```
>> x = 1;
>> whos
  Name   Size        Bytes     Class     Attributes
  x      1x1          8        double
```

Definition of row vectors

- x = [12 23 45 7] % separated by space/tab
- x = [12, 23, 45, 7] % or by comma

Definition of matrixes (2D arrays)

- A = [1 2 3; 4 5 6; 7 8 9] % 3x3 matrix
- A = [1, 2, 3; 4, 5, 6; 7, 8, 9] % 3x3 matrix

Functions to create arrays

zeros(N)	N x N matrix of zeros
zeros(N, M)	N x M matrix of zeros
ones(N)	N x N matrix of ones
ones(N, M)	N x M matrix of ones
eye(N)	identity matrix of order N
linspace(start, stop, N)	N equally spaced point from start to stop (included)
logspace(start, stop, N)	N points from 10^start to 10^end (included) equally spaced in logscale
rand(N)	N x N matrix of uniformly distributed random number on [0,1]
rand(N, M)	N x M matrix of uniformly distributed random number on [0,1]

Creating array with the : operator

START:END → array from START to END with step 1

a = 1:10 → [1,2,3,4,5,6,7,8,9,10] % a = [1:10] is the same

a = 1:10.5 → [1,2,3,4,5,6,7,8,9,10]

START:STRIDE:END → array from START to END with step STRIDE

a = 1:2:10 → [1,3,5,7,9]

a = 10:-1:1 → [10,9,8,7,6,5,4,3,2,1]

a = 10:1:-1 → empty row vector

a = 1:0.1:1.5 → [1,1.1,1.2,1.3,1.4,1.5]

Concatenating Arrays 1/2

- Creating an array made of other arrays

```
a = [4 6 7];
b = [9 0 2];
C = [a; b];
```

```
C =

   4   6   7
   9   0   2
```

```
a = [4 6 7];
b = [9 0 2];
c = [a b];
```

```
c =

   4   6   7   9   0   2
```

Concatenating Arrays 2/2

- Using the function cat as: *array = cat(dim, array1, array2, …)*, where *dim* is the direction of concatenation (1 = 1st axis, the rows)

```
a = [4 6 7];
b = [9 0 2];
C = cat(1, a, b);
```

```
C =

   4   6   7
   9   0   2
```

```
a = [4 6 7];
b = [9 0 2];
c = cat(2, a, b)
```

```
c =

   4   6   7   9   0   2
```

Multidimensional (>2D) arrays

- Arrays with more than 2 dimensions can be created with the same functions used for 1D and 2D arrays

```
>> x = ones(4, 3, 2);
```

- Or concatenating lower dimensional arrays

```
a = [4 6 7; 9 0 2];
b = [1 5 8; 3 1 7];
C = cat(3, a, b)
```

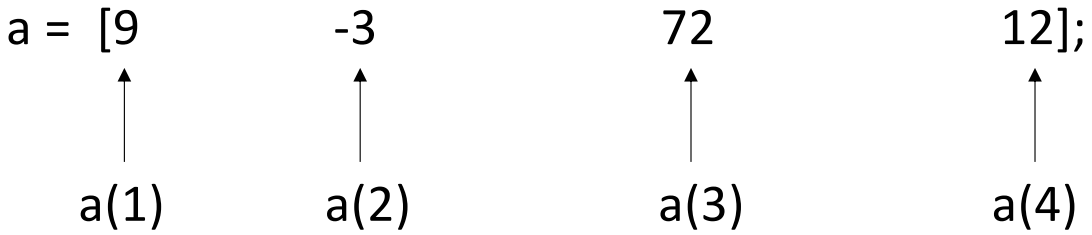
```
C(:,:,1) =

     4     6     7
     9     0     2


C(:,:,2) =

     1     5     8
     3     1     7
```

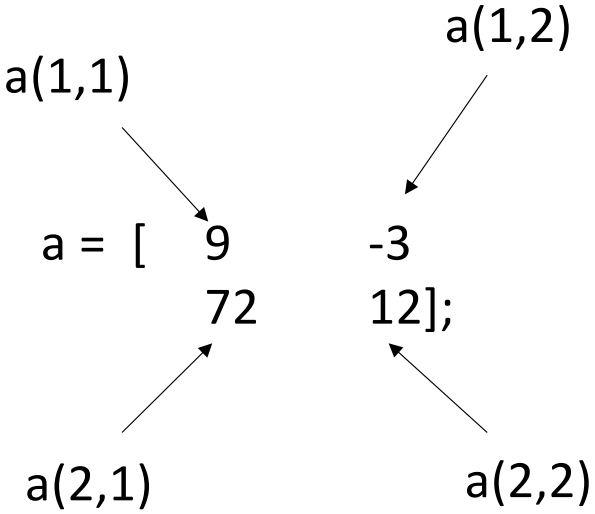
Functions to know the array structure

length	number of elements along the longest dimension
ndims	number of dimensions
numel	number of elements
size	shape of the array (number of elements along each dimension)

Indexing

- Arrays are indexed using parenthesis (not [] !)
- Indexes start from 1
- Dimensions are separated by commas





Slicing

- Use START:END to select contiguous parts of an array from START to END (included)
- Use just : to select an entire dimension
- But remember that for 1D array : transform to a column vector

a(2:4)

```
a = [9   12   43   31   89];
```



```
a(:)
 9
12
43
31
89
```

A(1:2,2:4)

```
A = [
   6   7   9   12
   3   4   11  -3
   6   32   0  -8
   9   -1   4   3
];
```

```
A = [
   6   7   9   12
   3   4   11  -3
   6   32   0  -8
   9   -1   4   3
];
```

A(3:4,:)

Slicing

- Use BEGIN:STRIDE:END to select elements from BEGIN to END with period STRIDE
- Use keyword **end** for the index of the last element

A(:,1:2:4) A(:,1:2:end)

```
A = [
    6    7    9   12
    3    4   11   -3
    6   32    0   -8
    9   -1    4    3
];
```

```
A = [
    6    7    9   12
    3    4   11   -3
    6   32    0   -8
    9   -1    4    3
];
```

A(end,end:-1:1)

[3 4 -1 9]

Using vectors as indexes

A([1, 4], 2:4)

```
A = [
    6    7    9   12
    3    4   11   -3
    6   32    0   -8
    9   -1    4    3
];
```

A([1, 4], [2,4])

```
A = [
    6    7    9   12
    3    4   11   -3
    6   32    0   -8
    9   -1    4    3
];
```

Using Booleans as indexes

A(A > 10)

```
A = [
    6    7    9   12
    3    4   11   -3
    6   32    0   -8
    9   -1    4    3
];
```

as column vector

A(A(:,1) > 5, :)

```
A = [
    6    7    9   12
    3    4   11   -3
    6   32    0   -8
    9   -1    4    3
];
```

Cell Arrays

A cell array is a data type with indexed data containers called cells, where each cell can contain any type of data

```
c = {12, [1,2], 'abcd'} % commas are not needed

c =
  1×3 cell array
    {[12]}    {[1 2]}    {'abcd'}
```

c{1} c{2} c{3}

Use normal parenthesis to get back an object of class cell

c(1) → it's an object of class cell containing only the 1st element of c
c(1:2) → it's an object of class cell containing the first two elements of c

Multidimensional cell arrays

Cell arrays can have any number of dimensions

```
c = cell(2,3) %2d cell arrays with 2 rows and 3 columns

c{1,1} = ‘red’
c{1,2} = ‘green’
c{1,3} = ‘blue’
c{2,1} = [1,0,0]
c{2,2} = [0,1,0]
c{2,3} = [0,0,1]
```

```
c =

  2×3 cell array

    {'red'  }    {'green'}    {'blue' }
    {[1 0 0]}    {[0 1 0]}    {[0 0 1]}
```

Relational / Logical operators

<	
<=	
>	
>=	
==	equal
~=	not equal

& &&	and + only for scalar inputs with short-circuiting behaviour
| ||	or + only for scalar inputs with short-circuiting behaviour
~	not

Conditional statements

```
if <expression>
   <block of code>
end
```

```
if <expression>
   <block of code>
else
   <block of code>
end
```

- indentation is not part of the language
- but use it !

```
if <expression>
   <block of code>
elseif <expression>
   <block of code>
…
else
   <block of code>
end
```

- only the first true statement is executed
- as many elseif blocks as needed
- else might be missing

switch statement

```
switch <expression_s>
  case <expression_c1>
     <block of code>
  …
  otherwise
     <block of code>
end
```

- executed if expression_s is equal to expression_c1
- as many case blocks as needed
- executed when no case block is true

```
switch a
       case 1
              disp('a == 1’)
       case 2
              disp('a == 2’)
       otherwise
              disp('a ~= 1 AND a ~= 2')
end
```

```
if a == 1
       disp('a == 1’)
elseif a == 2
       disp('a == 2’)
otherwise
       disp('a ~= 1 AND a ~= 2')
end
```

Cycles

```
while <expression>
   <block of code>
end
```

```
for var = values
   <block of code>
end
```

```
a = [9 8 7 6];
i = 1;
while i <= length(a)
      disp(a(i));
      i = i + 1;
end
```

```
a = [9 8 7 6];
for x = a
      disp(x);
end
```

for loops with indexes

```
for i = start:end
      <block of code>
end
```

i ranges from start to end (included)

```
for i = start:step:end
      <block of code>
end
```

i ranges from start to end (included) with period step

loop control statement

- break: end the loop immediately
- continue: go to the next iteration (first evaluate the condition)

```
while true
       x = input('Write a positive number:’)
       if x > 0
              break
       end
end
```

```
a = [-1 2 3 -4];
for x = a
       if x <= 0
              continue
       end
       disp(log10(x))
end
```

Function

Functions are defined inside .m files having the same name of the function

pitagora.m

```
function z = pitagora(x, y)
        % Return the hypotenuse of a right triangle
        % given the other two sides
        z = (x*x + y*y)^0.5;
end
```

help pitagora

```
Return the hypotenuse of a right triangle
given the other two sides
```

```
>> pitagora(2,4)

ans =

    4.4721
```

Functions can have multiple outputs

solve_quadratic.m

```
function [x1, x2] = solve_quadratic(a, b, c)
      % Return the solutions of the quadratic equation:
      % a*x^2 + b*x + c = 0
      d = sqrt(b^2 - 4*a*c);
      x1 = (-b + d) / (2*a);
      x2 = (-b - d) / (2*a);
end
```

```
>>[x, y] = solve_quadratic(1, -1, -6);
```

More than one function can be included in each .m file

But only the primary function is accessible from outside

solve_quadratic.m

```
function [x1, x2] = solve_quadratic(a, b, c)
        d = delta(a, b, c);
        x1 = (-b + d) / (2*a);
        x2 = (-b - d) / (2*a);
end

function d = delta(a, b, c)
        d = sqrt(b^2 - 4*a*c);
end
```

```
>>[x, y] = solve_quadratic(1, -1, -6);
```

```
>>delta (1, -1, -6); → ERROR
```

The workspace of the function is separate from the main workspace

Use global to access variables from the main workspace

counter.m

```
function counter()
        n = n + 1;
end
```

```
>> n = 1;
>> counter();
Unrecognized function or variable 'n'.

Error in counter (line 2)
   n = n + 1;
```

counter.m

```
function counter()
        global n;
        n = n + 1;
end
```

```
>> global n; % needs to be defined also here !
>> n = 1;
>> counter();
>> n

n =

    2
```

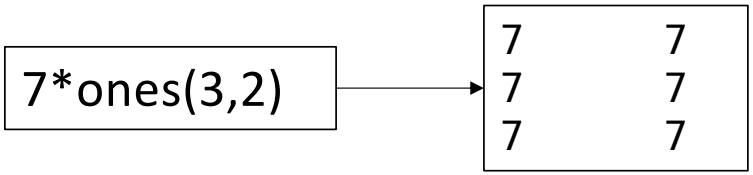
Operators

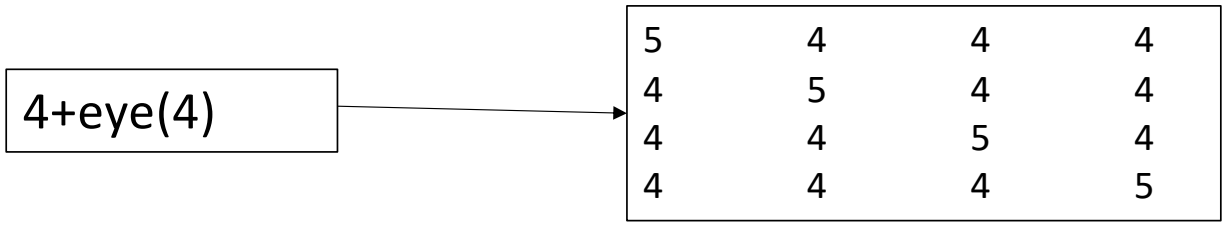
- All operators work on matrixes (as in linear algebra)
- Use the dot-prefix for the element-wise operators
- Functions mod(), rem() for module and reminder of division

+	plus, addition
-	minus, subtraction
*	scalar and matrix multiplication operator
.*	element-wise multiplication
^	scalar and matrix multiplication operator
.^	element-wise exponentiation operator
/, \	right-/left-division operator
./, .\	element wise right-/left-division operator
‘	matrix transpose (for complex matrixes is the conjugate transpose)

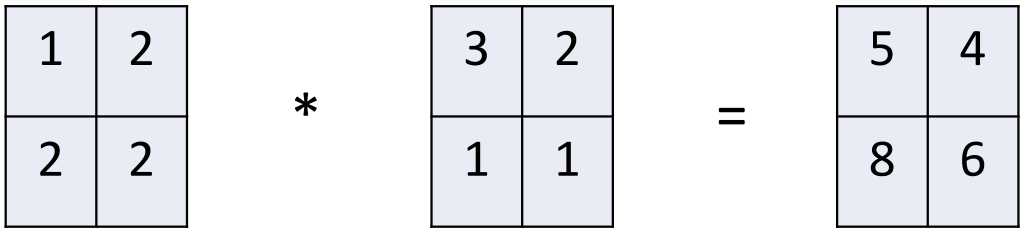
Operations with scalar

- The operation is performed between the scalar and all the elements of the array

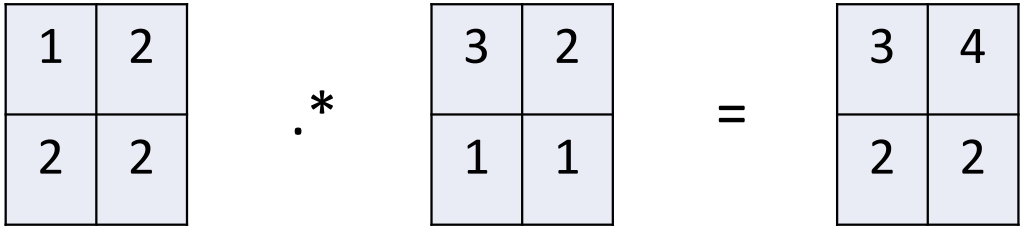




Matrix Vs element-wise operation



It’s standard matrix multiplication



It’s standard the element-wise multiplication

Incorrect Dimensions → ERROR (but see next slide)

```
a = [1, 2, 3];
b = [0, -1, 2];
a*b → ERROR
```

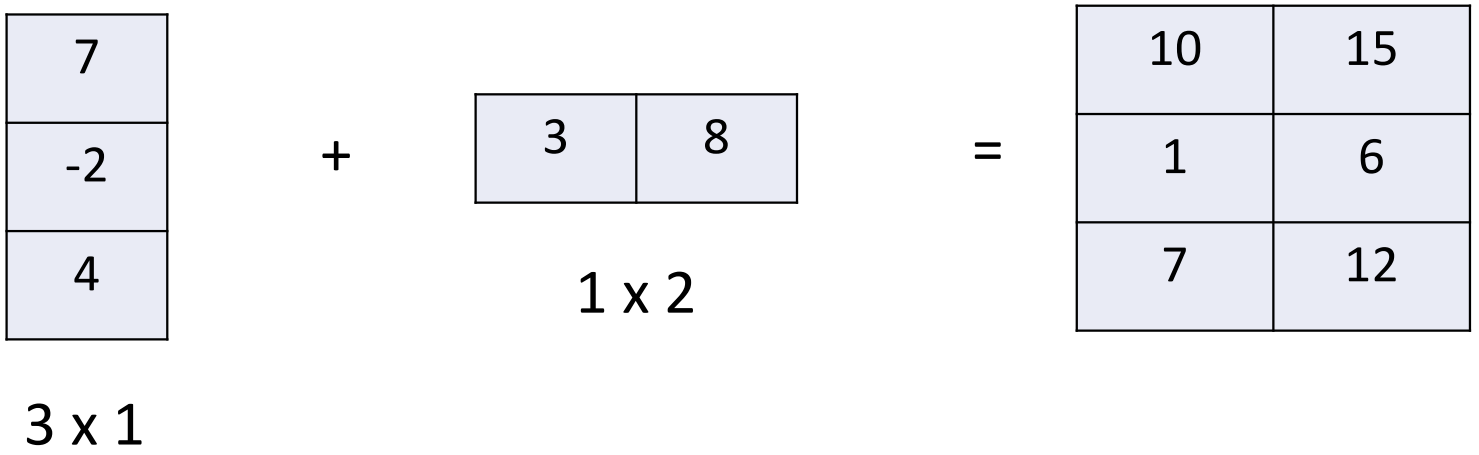
```
a = [1, 2, 3];
b = [0; -1; 2];
a*b → 4
a*a' → 14
```

In this case it's a 1x3 multiplied by a 3x1, it's fine !

```
a = [1, 2, 3];
b = [0, -1, 2, 4];
a.*b → ERROR
```

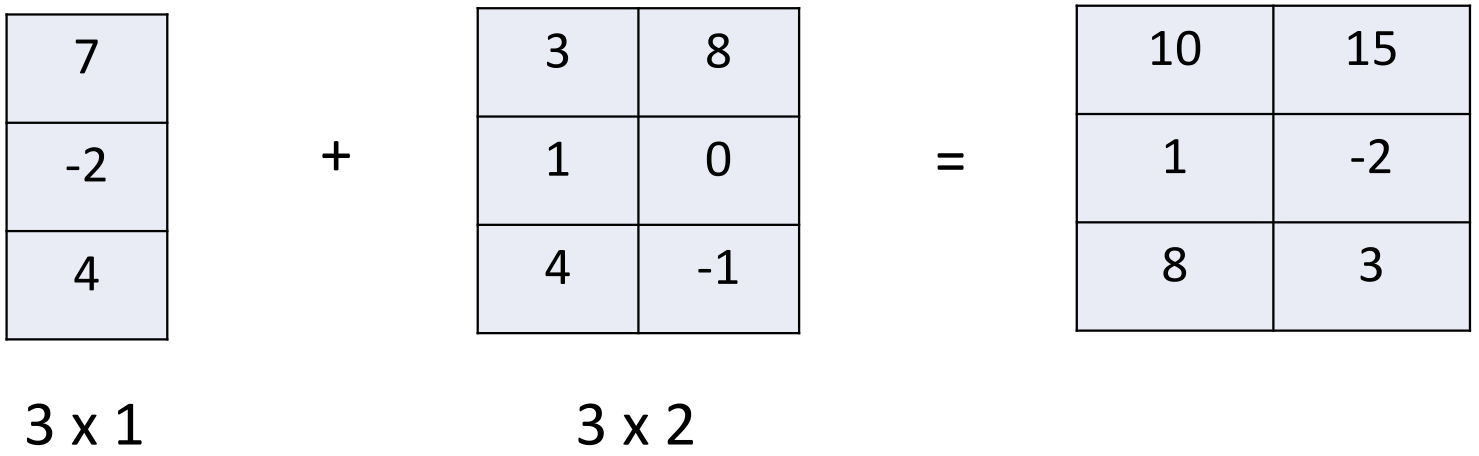
MATLAB broadcasting rules (1)

column vector / row vector: They are expanded to the same shape
The order of the operands is not important



MATLAB broadcasting rules (2)

column vector / matrix: The column vector is expended to match the number of columns in the matrix. It works only if the number of elements in the column vector is equal to the number or row. The order of the operands is not important



7
-2
4

3 x 1

*

3	8
1	0
4	-1

3 x 2

=

ERROR: this is an undefined matrix multiplication

7
-2
4

3 x 1

.*

3	8
1	0
4	-1

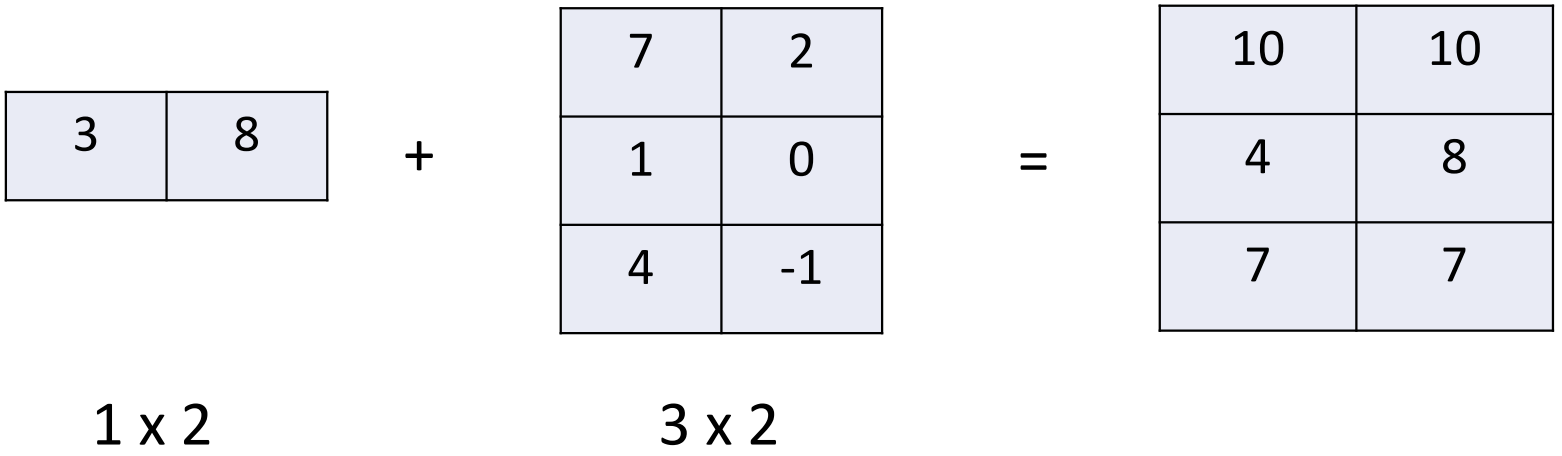
3 x 2

=

21	56
-2	0
16	-4

MATLAB broadcasting rules (3)

row vector / matrix: The row vector is expended to match the number of rows in the matrix. It works only if the number of elements in the row vector is equal to the number or columns. The order of the operands is not important



Right division

- for scalar is the classical division (4 / 2 → 2)

$$A/B = AB^{-1}$$

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

```
A / B
    -1     4
     0     2
```

```
A * inv(B)
    -1     4
     0     2
```

Left division

- for scalar is the reversed division (4 \ 2 = 2 / 4 → 0.5)

$$A \backslash B = A^{-1}B$$

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

```
A \ B
    -2    -1
     2.5   1.5
```

- it can be used to compute the solution of linear systems

$$Ax = B$$
$$x = A^{-1}B = A \backslash B$$

A

```
1   2
2   2
```

*

x

```
-2    -1
2.5   1.5
```

=

B

```
3   2
1   1
```

Writing/Reading data

- save myfile[.mat]
  →save all variables in the workspace to myfile.mat
- save myfile[.mat] AR1 AR2 …
  →save AR1, AR2,… to myfile.mat
- save myfile.txt AR1 AR2 … -ascii
  → save AR1, AR2, … to myfile.txt as a text file
- save myfile.txt -ascii
  → save all variables in the workspace to myfile.txt as a text file

- load myfile[.mat]
  →load all variables in myfile.mat to the workspace

min(), max()

A =

4	7	8
2	5	7
7	7	6
10	6	10

min(A) → [2 5 6]

min(A, [], “all”) → 2

min(A, [], 2) → [4 2 6 6]’

mean(), median(), mode(), std(), var(), sum(), prod()

mean(A) → [5.75 6.25 7.75]

mean(A, ”all”) → 6.5833

mean(A, 2) →

[6.3333
4.6667
6.6667
8.6667]

any() , all()

A =

0	1	0
0	1	1
0	1	0
0	1	1

any(A) → [0 1 1]

any(A, 2) → [1 1 1 1]’

all(A) → [0 1 0]

all(A,2) → [0 0 0 0]’

find()

```
A =
    0    1    0
    0    1    1
    0    1    0
    0    1    1
```

```
find(A)
→
     5
     6
     7
     8
    10
    12
```

```
[r,c] = find(A) →
r =                 c =
     1                   2
     2                   2
     3                   2
     4                   2
     2                   3
     4                   3
```

- linear indexes of the non-zero elements
- matlab is column-major

sort()

```
b =
         6     4     7     5     1     7


sort(b)
→1     4     5     6     7     7

[bs, i] = sort(b)
bs =
     1     4     5     6     7     7
i =
     5     2     4     1     3     6
```

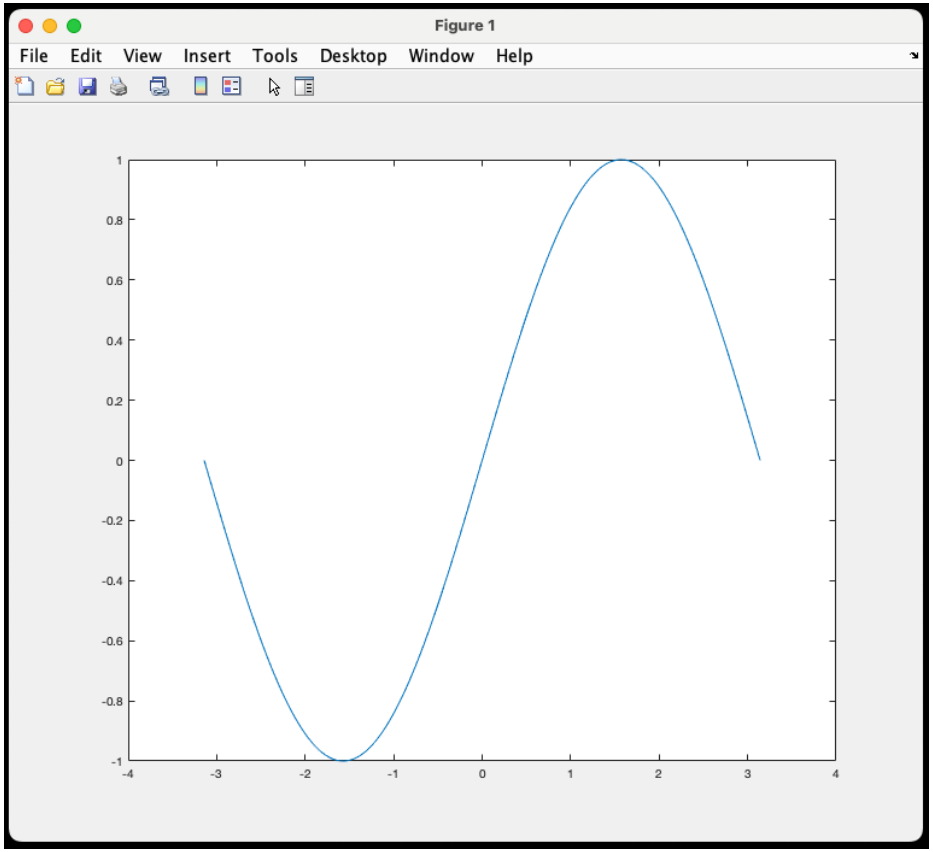
```
A =
    10     6
     5     2
     6     7

sort(A)
     5     2
     6     6
    10     7

sort(A, 2)
     6    10
     2     5
     6     7
```

```
[As, i] = sort(A)

As =

     5     2
     6     6
    10     7

i =
     2     2
     3     1
     1     3
```

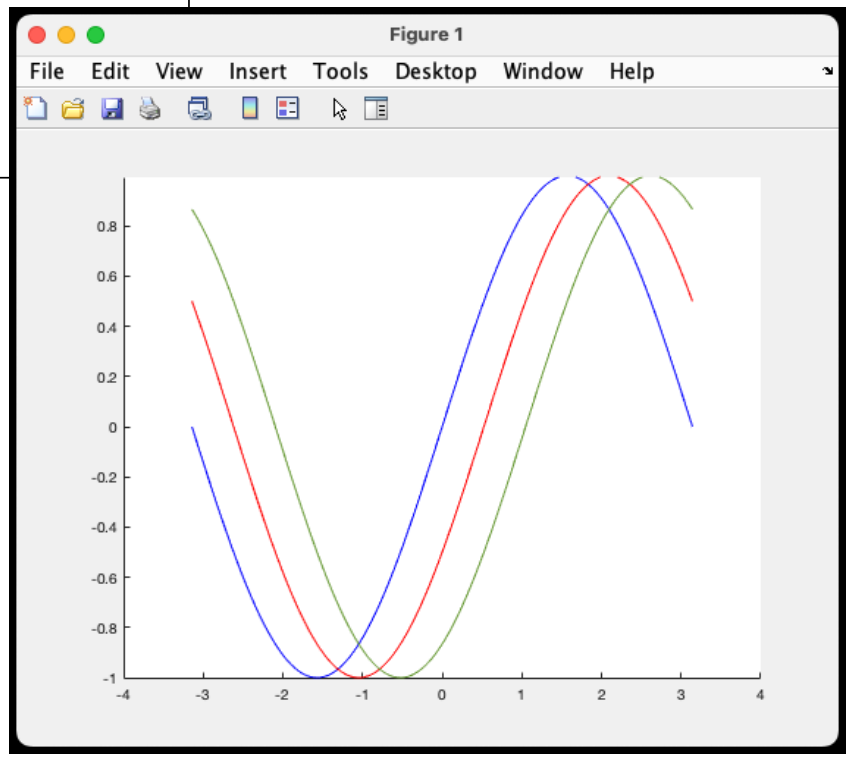
```
close all;

x = linspace(-pi,pi,1000);
y = sin(x);

f = figure();
ax = subplot(1,1,1);
plot(x, y);

exportgraphics(f,'figure1.png')
% close f
```

for saving the figure (many options available)

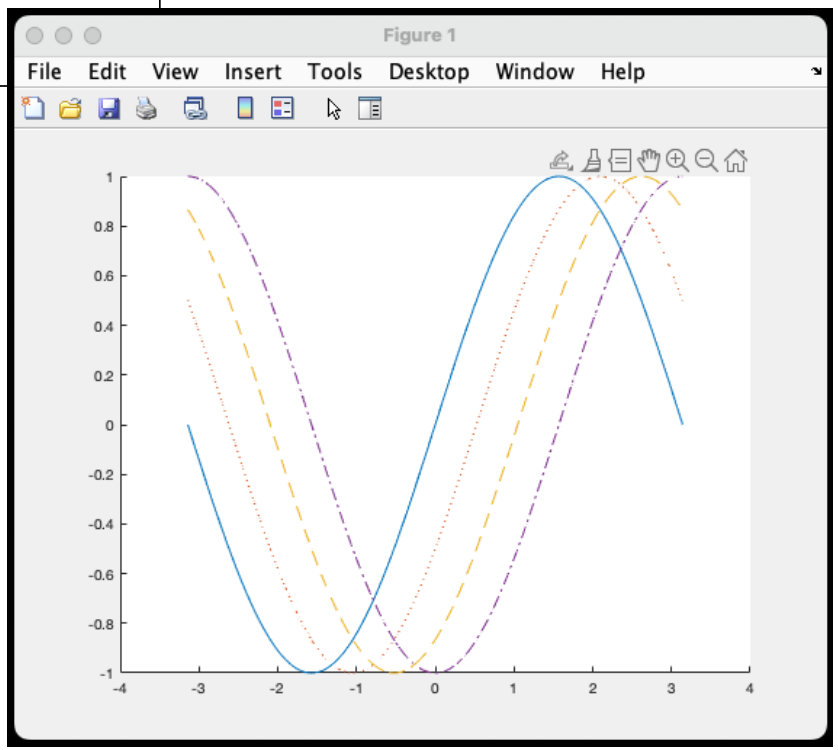
for closing the figure's window

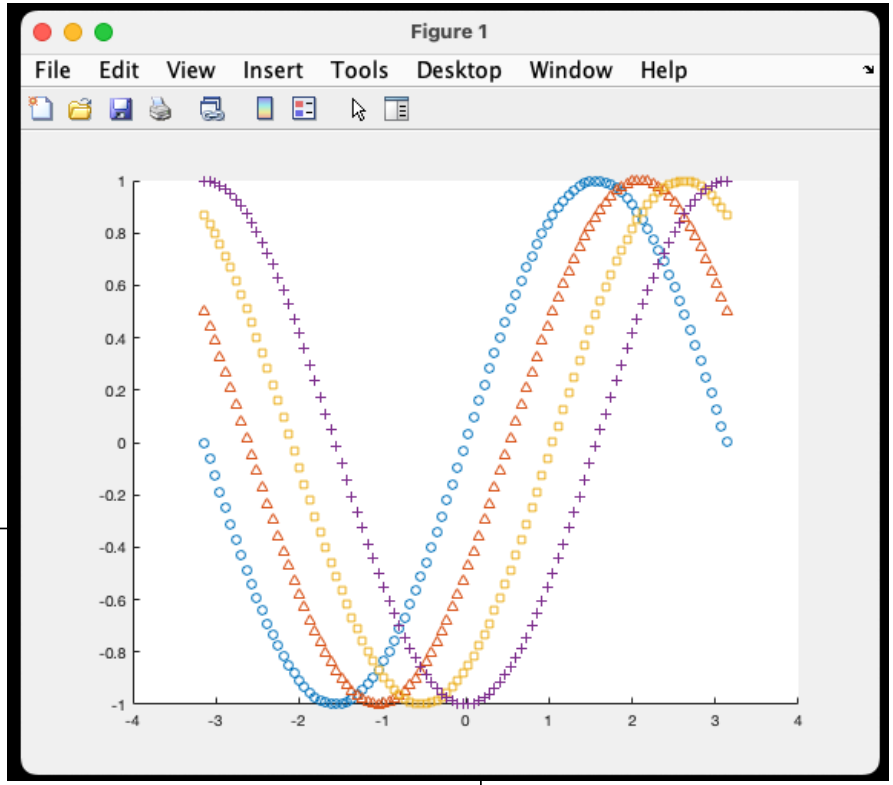


```
x = linspace(-pi,pi,1000);

f = figure();
ax = subplot(1,1,1);
hold on; %needed to have more plots on the same figure
% name of the color
plot(x, sin(x), color = 'blue');
% any of  r,g,b,c,m,y,k
plot(x, sin(x-pi/6), color = 'r');
% RGB code
plot(x,sin(x-pi/3), color = [0.4,0.6,0.2])
```



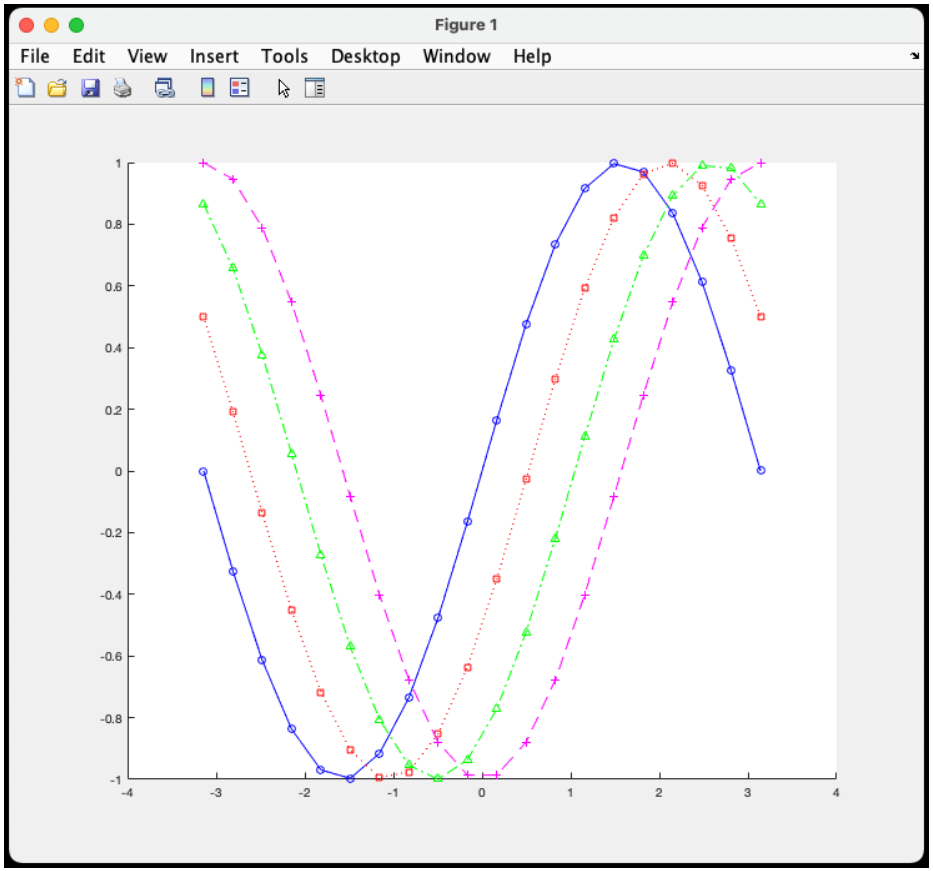
```
x = linspace(-pi,pi,1000);

f = figure();
ax = subplot(1,1,1);
hold on;
plot(x,sin(x), linestyle = '-')
plot(x,sin(x-pi/6), linestyle = ':')
plot(x,sin(x-pi/3), linestyle = '--')
plot(x,sin(x-pi/2), linestyle = '-.')
```



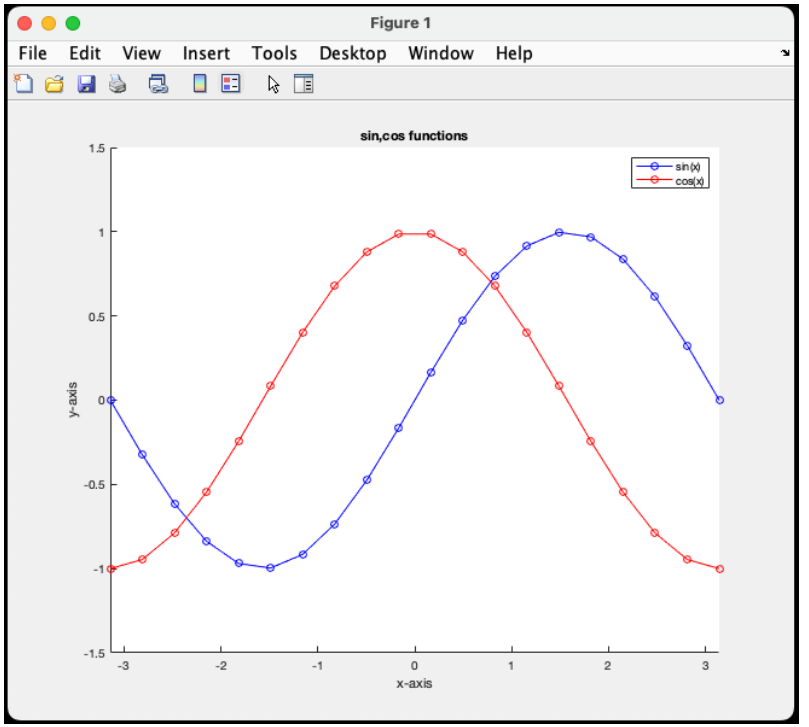



```
x = linspace(-pi,pi,100);

f = figure();
ax = subplot(1,1,1);
hold on;
plot(x,sin(x), marker = 'o', linestyle = 'none')
plot(x,sin(x-pi/6), marker = '^', linestyle = 'none')
plot(x,sin(x-pi/3), marker = 's', linestyle = 'none')
plot(x,sin(x-pi/2), marker = '+', linestyle = 'none')
```

```
x = linspace(-pi,pi,20);

f = figure();
ax = subplot(1,1,1);
hold on;
plot(x,sin(x), '-ob');
plot(x,sin(x-pi/6), 's:r');
plot(x,sin(x-pi/3), '^-.g');
plot(x,sin(x-pi/2), '+--m');
```



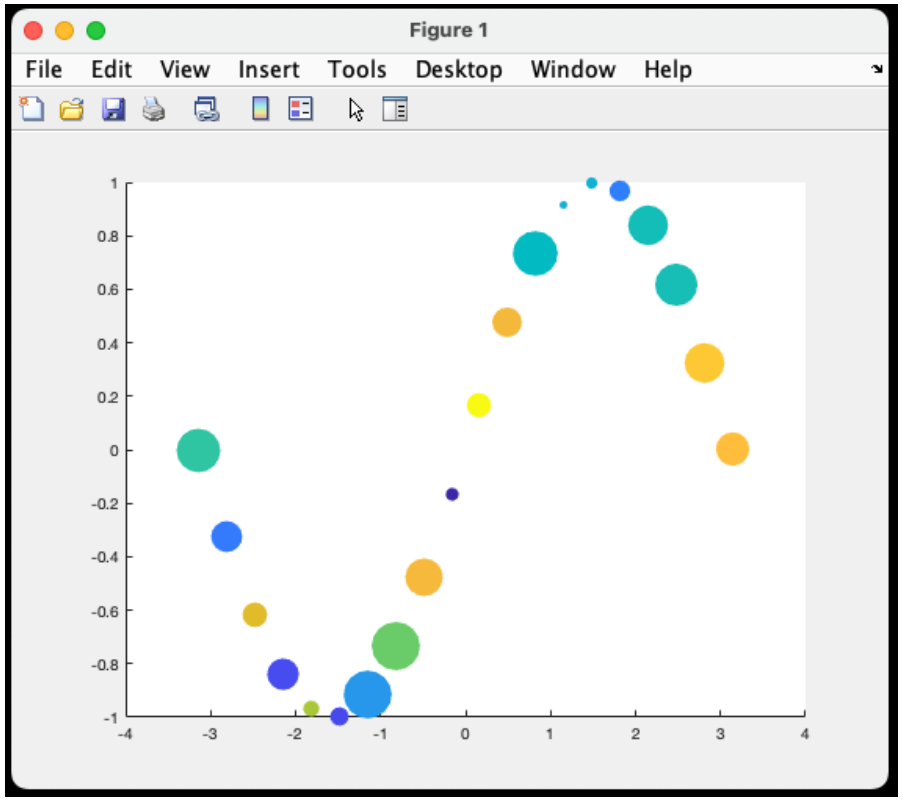
```
x = linspace(-pi,pi,20);

f = figure();
ax = subplot(1,1,1);
hold on;
plot(x,sin(x), '-ob');
plot(x,cos(x), '-or');
xlim(ax, [-pi,pi]);
ylim(ax, [-1.5,1.5]);
title(ax, 'sin,cos functions')
xlabel(ax, 'x-axis')
ylabel(ax, 'y-axis')
legend(ax, 'sin(x)', 'cos(x)')
```



not strictly needed in this case, by default it's the current axis

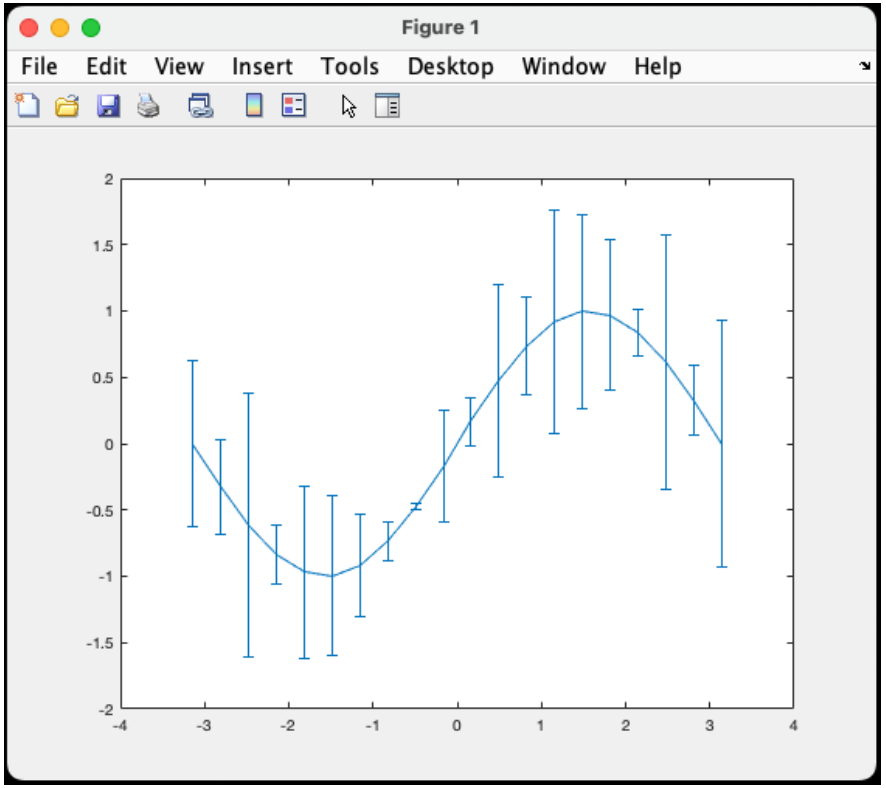
Use *scatter* to control size/color of individual points

```
f = figure();
ax = subplot(1,1,1);
sz = randi([10, 1000], 1, 20);
c = 10*rand(1,20);
scatter(x, sin(x), sz, c, 'filled', marker = 'o')
```



Use *errorbar* to show uncertain ranges

```
x = linspace(-pi,pi,20);

f = figure();
ax = subplot(1,1,1);
yerr = rand(1,20);
errorbar(x, sin(x), yerr);
```



```
x = [0,1,2,3];
y = randi([100,300], 1, 4);
f = figure();
ax = subplot(1,1,1);
bar(x,y);
```