Quick primer on Matlab for IACV

MATlab : Matrix Laboratory can be used as a scripting language any variable is a matrix (a scalar is a special case of a multidimensional array)

Credits:

- Alessandro Giusti alessandrog@idsia.ch
- Giacomo Boracchi giacomo.boracchi@polimi.it - September 28th, 2020

Edits: Luca Magri - September 19th, 2023

Clear the workspace and close all the figure and clear the command window

```
close all % close all the figures
clear % clear the workspace
clc % clear the command window
```

Print to the command window

```
disp('For enquiry, please send an email to luca.magri@polimi.it');
```

```
For enquiry, please send an email to luca.magri@polimi.it
```

```
fprintf('there is also a C-like printing function\n%s %dth, %d', 'September', 20, 2018
```

```
there is also a C-like printing function
September 20th, 2018
```

Dealing with variables:

all the variables are stored in matrix with proper dimensions. Scalar are treated as 1x1 matrix.

Variables are created by assignements, the size of a variable is the dimension of the matrix representing it.

```
v = 12;
c = 'c';
size(v)
```

```
ans = 1×2
     1     1
```

```
size(c)
```

```
ans = 1×2
     1     1
```

```
% use ; at the end of instruction not to display results
% in command line
v = 7;
```

Data types are automatically defined depending on the assigned value and can be checked using whos

```
whos v
```

```
  Name      Size            Bytes  Class     Attributes

  v         1x1                 8  double
```

Variables have datatypes (usually you can forget, but not when dealing with images).

Most common types we will consider are:

- *double*,(i.e., double-precision floating point),
- *uint8* (i.e., unsigned integers with 8 bits, [0 255] range),
- *logical* (i.e., boolean)

We can explicilty cast a double to a uint8 type

```
v = uint8(v)
```

```
v = uint8

    7
```

```
whos v
```

```
  Name      Size            Bytes  Class    Attributes

  v         1x1                 1  uint8
```

Arrays

can be row or column vector.

```
% a row vector (commas can be omitted)
r = [1, 2, 3, 4]
```

```
r = 1×4
     1     2     3     4
```

```
size(r)
```

```
ans = 1×2
     1     4
```

It is important to remember the index start from 1.

```
r(1)
```

```
ans = 1
```

```
%r(0) %raises an error! Array indices must be positive integers or logical values.
r(2) % returns the first element of the array r
```

```
ans = 2
```

```
% a column vector
c = [1; 2; 3; 4]
```

```
c = 4×1
     1
     2
     3
     4
```

```
size(c)
```

```
ans = 1×2
     4     1
```

You can define vectors by regular increment operator: [start : step : end]

```
a = [1 : 2 : 10]
```

```
a = 1×5
     1     3     5     7     9
```

when omitted, step is equal to 1

```
a2 = [1 : 10]
```

```
a2 = 1×10
     1     2     3     4     5     6     7     8     9    10
```

Matrices

```
v=[1 2; 3 4]
```

```
v = 2×2
     1     2
     3     4
```

```
size(v)
```

```
ans = 1×2
     2     2
```

Array concatenation

you can concatenate matrices or vectors as far as their sizes are consistent. ' denotes the transpose operation

```
B = [v', v']
```

```
B = 2×4
     1     3     1     3
     2     4     2     4
```

```
C = [v ; v]
```

```
C = 4×2
     1     2
     3     4
     1     2
     3     4
```

```
% the dimension of the data, visible also typing whos
dim = size(C)
```

```
dim = 1×2
     4     2
```

```
num_rows = size(C,1)
```

```
num_rows = 4
```

```
num_cols = size(C,2)
```

```
num_cols = 2
```

```
% this is not allowed
disp('K = [r,c]: this is not allowed')
```

```
K = [r,c]: this is not allowed
```

```
%K = [r,c]
```

Other examples of array concatenation:

```
my_vec1 = [1 2 3];
my_vec2 = 4:6;
```

```
my_matrix = [my_vec1; my_vec2]
```

```
my_matrix = 2×3
     1     2     3
     4     5     6
```

```
size(my_matrix)
```

```
ans = 1×2
     2     3
```

```
my_long_vector = [my_vec1 my_vec2]
```

```
my_long_vector = 1×6
     1     2     3     4     5     6
```

```
size(my_long_vector)
```

```
ans = 1×2
     1     6
```

C = cat(dim,A,B) concatenates B to the end of A along dimension dim when A and B have compatible sizes (the lengths of the dimensions match except for the operating dimension dim).

```
my_matrix = cat(1,my_vec1,my_vec2)
```

```
my_matrix = 2×3
     1     2     3
     4     5     6
```

```
my_long_vector = cat(2,my_vec1,my_vec2)
```

```
my_long_vector = 1×6
     1     2     3     4     5     6
```

```
my_3d_matrix = cat(3,my_vec1,my_vec2)
```

```
my_3d_matrix =
my_3d_matrix(:,:,1) =

     1     2     3


my_3d_matrix(:,:,2) =

     4     5     6
```

```
size(my_3d_matrix)
```

```
ans = 1×3
     1     3     2
```

```
%% indexing (starts from 1)
my_vec = (1:10)';
my_vec(1)
```

```
ans = 1
```

```
my_matrix = [1:4;5:8;9:12]
```

```
my_matrix = 3×4
     1     2     3     4
     5     6     7     8
     9    10    11    12
```

```
my_matrix(3,2)
```

```
ans = 10
```

```
% coumn-wise linear indexing for matrices
my_matrix(6)
```

```
ans = 10
```

Subarray

v(indexes) returns a vector of all the elements of v at locations in array indexes

```
% you can reference single or multiple values in an array:
my_matrix(2,3) % first row and second column (row and column indices are 1-based)
```

```
ans = 7
```

```
my_matrix(:,2) % the second column of my_matrix
```

```
ans = 3×1
     2
     6
    10
```

```
my_matrix(:,2) % the first row of my_matrix
```

```
ans = 3×1
     2
     6
    10
```

```
B = my_matrix(:,[1,3]) % some of the columns
```

```
B = 3×2
     1     3
     5     7
     9    11
```

You can extend vectors / matrices by assigning some element out of the range

```
my_matrix(:,5)=1
```

```
my_matrix = 3×5
     1     2     3     4     1
     5     6     7     8     1
     9    10    11    12     1
```

You can use end to specify the last row/columns

```
my_matrix(1:3,[2 4])
```

```
ans = 3×2
     2     4
     6     8
    10    12
```

```
my_matrix(1:end,[2 4])
```

```
ans = 3×2
     2     4
     6     8
    10    12
```

```
my_matrix(:,[2 4])
```

```
ans = 3×2
     2     4
     6     8
    10    12
```

```
my_matrix(1:2,[2 4])
```

```
ans = 2×2
     2     4
     6     8
```

```
my_matrix(1:end-1,[2 4])
```

```
ans = 2×2
     2     4
     6     8
```

You can unfold a matrix columnwise

```
my_vector = my_matrix(:)
```

```
my_vector = 15×1
     1
     5
     9
     2
     6
    10
     3
     7
    11
     4
     ⋮
```

Operations

on vectors are from linear algebra common operations work on matrices (careful about multiplication and division)

```
v = [1 2 3]
```

```
v = 1×3
     1     2     3
```

```
v*v'
```

```
ans = 14
```

```
v'*v
```

```
ans = 3×3
     1     2     3
     2     4     6
     3     6     9
```

There are also element-wise operators

```
[1 2 3].*[4 5 6]
```

```
ans = 1×3
     4    10    18
```

```
[1 2 3] + 5
```

```
ans = 1×3
     6     7     8
```

```
[1 2 3] * 2 % no need to use element-wise operator with scalars
```

```
ans = 1×3
     2     4     6
```

```
[1 2 3] .* 2 %explicit element-wise multiplication
```

```
ans = 1×3
     2     4     6
```

```
[1 2 3] / 2
```

```
ans = 1×3
    0.5000    1.0000    1.5000
```

```
[1 2 3] ./ 2 %explicit element-wise division
```

```
ans = 1×3
    0.5000    1.0000    1.5000
```

```
[1 2 3] .^ 2 %explicit element-wise power
```

```
ans = 1×3
     1     4     9
```

```
[1 2 3] * [4 5 6]' % this matrix product, returns a scalar (it is the inner product)
```

```
ans = 32
```

```
[1 2 3]' * [4 5 6] % this is the matrix product, returns a 3 x 3 matrix
```

```
ans = 3×3
     4     5     6
     8    10    12
    12    15    18
```

```
% rounding functions
ceil(10.56)
```

```
ans = 11
```

```
floor(10.56)
```

```
ans = 10
```

```
round(10.56)
```

```
ans = 11
```

```
ceil(0:0.1:1)
```

```
ans = 1×11
     0     1     1     1     1     1     1     1     1     1     1
```

```
% arithmetic functions
sum([1 2 3 4])
```

```
ans = 10
```

```
sum([1:4;5:8])
```

```
ans = 1×4
     6     8    10    12
```

```
sum([1:4;5:8],2)
```

```
ans = 2×1
    10
    26
```

```
sum(sum([1:4;5:8]))
```

```
ans = 36
```

```
my_matrix = [1:4;5:8];
sum(my_matrix(:))
```

```
ans = 36
```

Printing

```
disp('Hello World!');
```

```
Hello World!
```

```
disp(['Hello ', 'World!']);
```

```
Hello World!
```

```
% string concatenation in printf.
disp(['number of columns in the matrix: ', num2str(size(my_matrix))]);
```

```
number of columns in the matrix: 2  4
```

```
fprintf('number of columns in the matrix: %f\n',size(my_matrix,2));
```

```
number of columns in the matrix: 4.000000
```

Logicals

you can compare a vector via the customary relational operators and obtain a vector of logicals (i.e. b)

```
b = v>2
```

```
b = 1×3 logical array
   0   0   1
```

```
whos b
```

```
  Name      Size            Bytes  Class     Attributes

  b         1x3                 3  logical
```