

# MATLAB Tutorial

Mohammad Motamed<sup>1</sup>

<sup>1</sup>*Department of Mathematics and Statistics, The University of New Mexico, Albuquerque, NM 87131*

August 28, 2016

---

## Contents:

1. Scalars, Vectors, Matrices .....	1
2. Built-in variables, functions, and commands .....	2
3. Operations on vectors and matrices .....	3
4. Logical operators .....	4
6. Graphics .....	4
6. MATLAB programing (scripts, conditionals, loops, functions) .....	6
7. Efficient MATLAB programming .....	8

---

## 1 Scalars, Vectors, Matrices

- **Scalars** (or numbers) are zero-dimensional arrays.

`a = 2;` generates a scalar (number 2) and assigns it to variable `a`.

- **Vectors** are one-dimensional arrays.

`x = [1; 2; 3; 4; 5];` generates a column vector.

`y = [1 2 3 4 5];` generates a row vector.

- **Matrices** are two-dimensional arrays.

`A = [1 2 3; 4 5 6; 7 8 9];` generates a  $3 \times 3$  matrix.

Rows are separated by semicolons (`;`)

The entries on each row are separated by empty spaces or commas (`,`)

The output is

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

## 2. Built-in variables, functions, and commands

- **Built-in variables:**

- `i`     complex unit ( $i = \sqrt{-1}$ )

It is better to type `1i` instead of `i` when you need complex unit. Because sometimes we may assign another number to variable `i`, and if we do so, then `i` will no longer be complex unit.

- `pi`      $\pi = 3.14159\dots$

If you type `pi`, MATLAB displays 3.1416. It is important to note that the number `pi` is not 3.1416. It has many many more decimals. You can observe this by typing `pi - 3.1416` in the command window in MATLAB which will not return 0. Indeed, if you first type `format long` and then type `pi`, you will see more decimal digits.

- **Built-in functions:**

- `size`     returns the size of a variable. For example `size(A)` returns  $3 \times 3$ .
- `length`     returns the maximum dimension of a variable
- `max`:     returns the largest entry of a vector
- `min`:     returns the smallest entry of a vector
- Mathematical functions: `sin`   `cos`   `tan`   `exp`   `log`   `log10`
- `disp(.)`     displays its argument
- `find(.)`     For example let `x=(0:0.5:3)`. Then `find(x>1.5)` returns indices corresponding to the entries of `x` which are greater than 1.5, that is `[5 6 7]`.

► Try `a = 5; disp(['a=' num2str(a)])`;

► Try `length(x)` and compare it with `max(size(x))`.

► Try `max(A)`.

► Try `sin(pi/3) / cos(pi/3)` and compare the result with `tan(pi/3)`.

- **Built-in commands** to create special vectors and matrices:

- `colon (:)`     creates a row vector
  - `x1 = (1 : 5)`     components of the vector increase by 1
  - `x2 = (1 : 0.5 : 5)`     components can change by non-unit steps
  - `x3 = (5 : -1 : 1)`     components can change by negative steps

- `linspace(a,b,n)` creates a vector with linearly spaced entries from **a** to **b** with length **n**  
`y1 = linspace(1,5,5)` is the same as `y1 = (1:5)`  
`y2 = linspace(1,5,9)` is the same as `y2 = (1:0.5:5)`
- `zeros(m,n)` generates an  $m \times n$  matrix with all entries zero.
- `ones(m,n)` generates an  $m \times n$  matrix with all entries one.
- `eye(n)` generates the  $n \times n$  identity matrix.
- `diag(x)` generates an  $n \times n$  diagonal matrix with **x** on the diagonal, where *x* is a vector with length **n**.
- `rand(m,n)` generates a random matrix of size  $m \times n$  with uniformly distributed random numbers between 0 and 1.

► Try `2*ones(3,3)`.

► Using the commands `diag` and `ones` generate a  $5 \times 5$  matrix *A* whose diagonal elements are (1,2,3,4,5) and off-diagonal elements are 2.

Answer: `x = (1:5);`

`A = 2*ones(5,5) + diag(x-2);`

• **Other useful commands:**

- `save yourfilename.mat A x`  
 The above command will save the variables **A** and **x** in a mat-file.
- `load yourfilename.mat`  
 The above command will bring all variables saved in `yourfilename.mat`.
- `help min`  
 The `help` command gives information about the command written after it (`min`).
- `clear all` clears all variables from memory.
- `clc` clears the command window.

### 3. Operations on vectors and matrices

- Standard arithmetic operators: `+` `-` `*` `/` `^`
- If we put a dot (`.`) before the operator, we obtain a component-wise operator.

Example: Let  $A = \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix}$  be a  $2 \times 2$  matrix.

Then, if we write `B1 = A * A` we will obtain  $B1 = \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix} * \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix} = \begin{bmatrix} 7 & 10 \\ 15 & 22 \end{bmatrix}$ .

But if we write `B2 = A .* A` we will obtain  $B2 = \begin{bmatrix} 1*1 & 2*2 \\ 3*3 & 4*4 \end{bmatrix} = \begin{bmatrix} 1 & 4 \\ 9 & 16 \end{bmatrix}$

- `inv(A)` computes the inverse of square matrix  $A$ .
- `det(A)` computes the determinant of square matrix  $A$ .
- `A'` computes the conjugate transpose of matrix  $A$ .

#### 4. Logical operators

- The logical operators `<` `>` `=` `<=` `>=` `==` `~=` are binary operators which return 0 (false) or 1 (true) for scalar arguments. If their arguments are vectors, they will return vectors with entries 0 and/or 1.
- Example: `5 == 3` returns 0.
- Example: `5 == 3+2` returns 1.
- Example: `x = (0:0.5:3);` this will return `x = [0 0.5 1 1.5 2 2.5 3]`  
`y = x>1.5;` this will return `y = [0 0 0 0 1 1 1]`  
`z = x(y);` this will return `z = [2 2.5 3]`, that is the elements of `x` which are greater than 1.5

#### 5. Graphics

- A few useful commands:  
`figure plot xlabel ylabel legend subplot loglog semilogx semilogy`  
`set axis hold on`
- **Example 1:** Plot the curves  $\sin x$  and  $\cos x$  for  $x \in [0, 2\pi]$  in the same figure.

```
Answer:  N = 1000;
         x = linspace(0,2*pi,N);
         y1 = sin(x);
         y2 = cos(x);
         figure(1);
         set(gca,'fontsize', 20);
         plot(x,y1,'b-','Linewidth',2);
         hold on;
         plot(x,y2,'r-','Linewidth',2);
         xlabel('x');
         ylabel('y');
         legend('y = sin x', 'y = cos x');
         axis([0 2*pi -1 1]);
```

This code will generate the following plot, depicted in Figure 1.

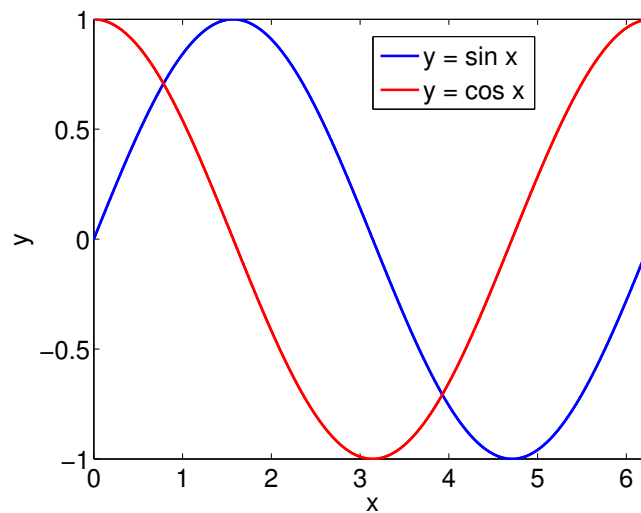


Figure 1: The figure generated by MATLAB code in Example 1.

- **Example 2:** Plot the curves  $\sin x$  and  $\cos x$  for  $x \in [0, 2\pi]$  in two separate figures.

Answer:

```
N = 1000;  
x = linspace(0,2*pi,N);  
y1 = sin(x);  
y2 = cos(x);  
figure(2);  
subplot(1,2,1); %it divides the figure window into a 1×2 matrix  
                  of subplots and makes subplot no.1 active  
set(gca,'fontsize', 20);  
plot(x,y1,'b-','Linewidth',2);  
xlabel('x');  
ylabel('y = sin x');  
axis([0 2*pi -1 1]);  
subplot(1,2,2); %it divides the figure window into a 1×2 matrix  
                  of subplots and makes subplot no.2 active  
set(gca,'fontsize', 20);  
plot(x,y2,'r-','Linewidth',2);  
xlabel('x');  
ylabel('y = cos x');  
axis([0 2*pi -1 1]);
```

This code will generate the following plot, depicted in Figure 2.

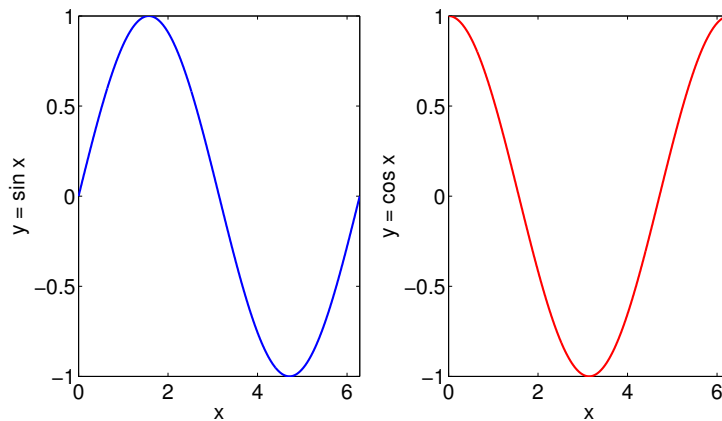


Figure 2: The figure generated by MATLAB code in Example 2.

## 6. MATLAB programming

### • 1. Scripts

A script is a collection of MATLAB commands written in the script window and saved as an m-file.

Example: Open a new script window. In the new window type:

```
x = linspace(0, 2*pi, 1000);  
y = sin(x);  
figure(3);  
set(gca, 'fontsize', 20);  
plot(x, y, 'b-', 'Linewidth', 2);  
xlabel('x');  
ylabel('y = sin x');  
axis([0 2*pi -1 1]);
```

Then save it as `myplot1.m` in a proper folder on your computer. Finally, type `myplot1` in the command window to run your code.

### • 2. Conditionals

Conditional statements enable you to select which block of code to execute. The simplest conditional statement is an `if` statement, with the following general form:

```
if a condition is satisfied  
    do these calculations  
else if another condition is satisfied  
    do these calculations  
else  
    do these calculations  
end
```

In the above format, `elseif` and `else` are optional.

Example: In a new script window type the following simple code:

```
a = rand(1);    % generates a uniform random number between 0 and 1
if a > 2/3
    disp('a > 2/3');
elseif a < 1/3
    disp('a < 1/3');
else
    disp('1/3 <= a <= 2/3');
end
```

- **3. Loops**

MATLAB provides two types of loops:

<pre>for i = I     <u>do these calculations</u> end</pre>
---

<pre>while <u>underlinea statement is true</u>     <u>do these calculations</u> end</pre>
---

- A **for**-loop in MATLAB is comparable to a Fortran **do**-loop or a C **for**-loop. A **for**-loop repeats the statements in the loop as the loop index ( $i$ ) takes on the values in a given row vector (**I**). **I** is called an index vector.

Example: The following **for**-loop repeats the statement inside the loop as the loop index  $i$  takes on values 1 to 5:

```
I = (1:5);
for i = I
    disp(2 * i)
end
```

Alternatively, we can write

```
for i = 1:5
    disp(2 * i)
end
```

- A **while**-loop repeats as long as the given expression in front of **while** is true (non-zero). As soon as the expression becomes false, the calculations stop, and MATLAB exits the loop.

Example: The following **while**-loop repeats the statement inside the loop as long as number  $a$  is less than or equal to 10:

```
a = 1;
while a <= 10
    disp(a)
    a = a+1
end
```

► Consider a vector  $\mathbf{x} = \text{linspace}(0,10,100)$ . Using **for**-loop, write a MATLAB program to compute

- 1) the sum of the elements of  $\mathbf{x}$ , that is  $S = \sum_{i=1}^{100} x_i$
- 2) a vector  $\mathbf{c} = [c_1, c_2, \dots, c_{100}]$  containing the cumulative sum of the elements of  $\mathbf{x}$ , that is  $c_j = \sum_{i=1}^j x_i$ , where  $j = 1, 2, \dots, 100$ .

Answer:  $\mathbf{x} = \text{linspace}(0,10,100);$

$\mathbf{S} = 0;$

$\mathbf{c} = [];$

for  $i = 1:\text{length}(\mathbf{x})$

$\mathbf{S} = \mathbf{S} + \mathbf{x}(i);$

$\mathbf{c}(i) = \mathbf{S};$

end

► Compare the results of the above code with the following built-in commands in MATLAB:

1)  $\text{sum}(\mathbf{x})$

2)  $\text{cumsum}(\mathbf{x})$

#### • 4. Functions

A MATLAB function is a script which takes one or more *inputs* and generates one or more *outputs*. The first line of a MATLAB function reads:

`function [output1, output2, ...] = FunctionName(input1, input2, ...)`

Example: The following MATLAB function takes a vector  $\mathbf{x}$  as input and generates a vector  $\mathbf{y}$  where  $y = \sin^2 x$

`function y = sin2fun(x)`

`y = (sin(x)).^2;`

To use and run the function, in the command window we type:

`x = linspace(0, pi, 1000);`

`y = sin2fun(x);`

Note that instead of `y = sin2fun(x);` we can also type `y = feval('sin2fun',x);`

## 7. Efficient MATLAB programming

See the next four *hand-written* pages.



## 7. Efficient MATLAB Programming

- It is very easy to write a MATLAB program that runs very slowly.
- This is not MATLAB's fault. MATLAB is designed for a particular type of task, which is matrix computations. It is computationally efficient for matrix and vector operations. It should not be regarded as a general purpose language, like C++. Of course, a poor MATLAB code will run slowly.

### • A few efficient techniques:

#### ① inline vs. anonymous functions

```
f = inline('exp(-t)*(x(1)+x(2))/1000', 'x', 't');  
t = 0;  
x = [1; 2];  
N = 1e5;  
for i = 1:N  
    x = x + f(x, t);  
    t = t + 0.1;  
end
```

Now replace the 1st line by:  
and see the difference!

```
f = @(x, t) [exp(-t)*(x(1)+x(2))/1000];
```

⇒ always use anonymous functions instead of inline functions.

## ② preallocation of arrays

```
N = 1e5;  
  
t = 0  
T = t;  
  
for i = 1:N  
    t = t + 0.1;  
    T = [T; t];  
end
```

In each iteration, the array  $T$  grows. MATLAB needs to allocate memory for a larger array, which is very costly.

It is better to pre-allocate an array that will be filled in a loop. Now replace the above code with the following and see the difference!

```
N = 1e5;  
  
t = 0;  
T(1) = t;  
  
for i = 1:N  
    t = t + 0.1;  
    T(i+1) = t;  
end
```

### ③ loop vs. vectorized implementation

Ex.1

```
N = 1e7;
x = linspace(0, pi, N);

f = sin(x);

for i = 1:N-1
    df(i) = f(i+1) - f(i);
end
```

replace the loop by the following command:

$\Rightarrow df = f(2:N) - f(1:N-1);$

and see the difference!

Try to avoid loops and replace them by vectorized commands.

Ex.2 Evaluate a piecewise function  $f(t) = \begin{cases} 0 & t < 0 \\ 1 & 0 \leq t < 2 \\ 2 & t \geq 2 \end{cases}$

for a vector  $t = \text{linspace}(-1, 3, N)$  with  $N = 1e4$ .

Option 1

```
f = zeros(N, 1);
for i = 1:N
    if t(i) >= 0 && t(i) < 2
        f(i) = 1;
    elseif t(i) >= 2
        f(i) = 2;
    end
end
```

Option 2

```
f = zeros(N, 1);
f(find(t >= 0 & t < 2)) = 1;
f(find(t >= 2)) = 2;
```

Option 3

```
f = zeros(N, 1);
f(t >= 0 & t < 2) = 1;
f(t >= 2) = 2;
```

⊗ The 1st option, although very natural for a C programmer, is a poor and slow MATLAB code. We need to write vectorized implementations, instead of computations based on individual elements.

⊗ The 2nd option uses the built-in function "find", which returns a list of indices for which the logical statement in its argument is true.

Ex. `find( (5:9) > 7 )` returns a vector `[4 5]`.

Moreover, the code is not based on individual elements, and hence it will be faster than the code in option 1.

⊗ the 3rd option uses logical referencing instead of "find" and is the fastest among the three codes.