

# Module 01:

# Basic MATLAB Programming

Chul Min Yeum

Assistant Professor

Civil and Environmental Engineering

University of Waterloo, Canada



**UNIVERSITY OF WATERLOO**  
FACULTY OF ENGINEERING



## Module 1: Intended Learning Outcomes

- Write an assignment statement to define a variable and assign a value(s).
- Define a valid variable name acceptable in MATLAB
- Describe data types and a type casting process.
- Explain numeric, logical operator, relational operators and compose expression using these operators
- Evaluate the expression with multiple operators following to operator precedence
- Assess some basic math functions (e.g., `round(x)`, `sign(x)`, or `abs(x)`)
- Generate random number(s) using a built-in random number function

# Variables and Assignments

- To store a value, use a ***variable***
- A way to put a value in a variable is with an ***assignment statement***
- General form:

***variable = expression***

- The order is important:
  - variable name on the left
  - the assignment operator “=”
  - expression on the right

⚠: Here, = is not meant to an equality.

⚠: Make sure that the variable name is always on the left.

```
% variable = expression  
a1 = 3  
a2 = 5
```

Name	Value
a1	3
a2	5

# Variable Names

- Names must begin with a letter of the alphabet.
- After that names can only contain letters, digits, and **the underscore character(\_)**.
- Variable names must not have a space.
- You cannot use other characters except for ‘\_’.
- MATLAB is **case-sensitive**.
- Names should be **mnemonic**: You and others know what are stored in your variable through its name.
- **clear** clears out variables and also functions.

## Example: Variable Names

```
8val = 10; % error: must begin with a letter of the alphabet
_col = 0 ; % error: must begin with a letter of the alphabet
row_3_ = 1; % no error
row@3 = 10; % error: cannot contain characters other than underscore
col-03 = 10; % error: cannot contain characters other than underscore
% Following scripts have no error but the names should be mnemonic
asdf1 = 100; % no error
love = 10; % no error
aaaa3 = 10; % no error
```

**⚠: Recommend mnemonic variable names**

```
% define a variable of 'gal'
gal = 100
```

Q. What value is in 'Ga1' ?

## Example: Question in S19 Midterm



Q. Which of the following scripts have errors?

(1)	<code>Val1 = 10</code>
(2)	<code>4val = 5</code>
(3)	<code>new@data = 8</code>
(4)	<code>val*2 = 3</code>
(5)	<code>Jason = 10+2</code>

1. (1), (2), and (4)
2. (2), (3), and (5)
3. (2), (3), and (4)
4. (1), (2), and (5)

# Constants

- In programming, variables are used for values that **could change**, or **are not known in advance**
- **Constants** are used when the value is known, is **pre-defined** and not updated in the program.
- Examples in MATLAB (these are actually functions that return constant values)
  - **pi**        3.14159....
  - **i, j**      imaginary number
  - **inf**        infinity
  - **NaN**       stands for “not a number”; e.g. the result of 0/0

⚠: You can overwrite values to the constants but I do not recommend the use of constants as variables. For example: `pi = 3 % no error`

# Modifying Variables

```
myvar = 10; % initialize a variable  
myvar = myvar + 3; % increment by 3
```

Name	Value
myvar	13

```
myvar = 10; % initialize a variable  
val = 3; % initialize a variable
```

```
% identical operations  
myvar1 = myvar + val;  
myvar2 = 10 + 3;
```

Name	Value
myvar	10
val	3
myvar1	13
myvar2	13

## Swap two values

```
myvar1 = 10; % initialize a var.  
myvar2 = 5; % initialize a var.
```

```
tmp = myvar1;  
myvar1 = myvar2;  
myvar2 = tmp;
```

Name	Value
myvar1	5
myvar2	10
tmp	10

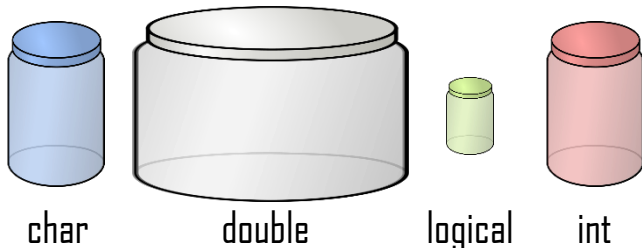


# Data Types

- Every expression and variable has an associated *type* or *class*
  - Real numbers: *single*, *double* (**default type for numbers**)
  - Integer types: numbers in the names are the number of bits used to store a value of that type
    - Signed integers: *int8*, *int16*, *int32*, *int64*
    - Unsigned integers: *uint8*, *uint16*, *uint32*, *uint64*
  - Single characters and character vectors: *char*
  - Strings of characters: *string*
  - **True/false: *logical* (represented as 1 or 0)**

😊: The same value can be defined as a different type.

## Container



## Examples

- 2.145, 0.15893, 3.0, 2.45
- 10, 11, 24, 30, 400
- 'a', 'b', 'A', 'c'
- "sam"
- true, false

## Data Types (Continue)

### Example

- double : 2.145, 0.15893, 3.0,
- integer type: 10, 11, 24, 30
- char: 'a', 'b', 'A', 'c'
- string: "sam"
- logical: true, false

**class** is a function of determining a class of object.

```
>> class("sam")  
  
ans =  
    'string'
```

```
>> class('a')  
  
ans =  
    'char'  
  
>> class(true)  
  
ans =  
    'logical'  
  
>> class(2.145)  
  
ans =  
    'double'  
  
>> class(10)  
  
ans =  
    'double'
```

**⚠:** The default type of numbers is 'double'. Integers like 10, 11, .. can be defined as integer types. It does not mean that integer numbers always become integer types.

# Type Cast

There are many functions that convert values from one type to another. The names of these functions are the same as the names of the types. The names can be used as functions to convert a value to that type. This is called casting the value to a different type or type casting.

```
>> a = logical(1);
>> class(a)

ans =

    'logical'
>> b = double(a);
>> class(b)

ans =

    'double'
```


```
>> a = logical(1);
>> b = a + 2;
>> class(b)

ans =

    'double'
>> c = logical(b)
c =

    logical

    1
```

 In the second line in red, the type of `a` is converted to double and arithmetic operation is conducted with 2. In the fourth line, `b` is double but the logical type only carries 0 or 1 so non-zero double (or other numeric type) values becomes 1, otherwise 0. This is a very important concept for logical operations.

# Operators

- There are in general two kinds of operators: *unary* operators, which operate on a single value and *binary* operators, which operate on two values.
- Operators include:
  - + addition
  - subtraction or negation
  - \* multiplication
  - / division
  - ^ exponentiation (e.g.  $5^2$  is 25)

```
val1 = 5 + 1;  
val2 = 10*2;  
val3 = 10^2;  
val4 = 100/5;  
val5 = 10 + 2 + 3;
```

Name	Value
val1	6
val2	20
val3	100
val4	20
val5	15

# Operator Precedence Rules

- Some operators have precedence over others.
- Within a given precedence, the expressions are evaluated from **left to right**.
- Precedence list (highest to lowest) :
  - ( ) parentheses
  - ^ exponentiation
  - negation
  - \*, / , \ all multiplication and division
  - +, - addition and subtraction
- Nested parentheses: expressions in inner parentheses are evaluated first

```
val1 = (5 + 1)*2;  
val2 = 10^2*3;  
val3 = 10^(2+3);  
val4 = 4-3*2;  
val5 = 3*((4+3)*2);  
val6 = -10^2;
```

☺: A good practice is to use parentheses to clarify your operation:  $(-10)^2$  or  $(-10)^2$ .

Name	Value
val1	12
val2	300
val3	100000
val4	-2
val5	42
val6	-100

## Example: Operator Precedence Rules



Q. What values are assigned to the variables?

```
val1 = 3*2+3-2*3  
val2 = -10^4*3  
val3 = -(10^4*3)  
val4 = 5-3*2;  
val5 = (3+3)^(2-1);
```

Name	Value
val1	3
val2	-30000
val3	-30000
val4	-1
val5	6

# Relational Operator

- The relational operators in MATLAB are:

>	greater than
<	less than
>=	greater than or equals
<=	less than or equals
==	equality
~=	inequality




⚠: Here, == is meant to an equality. Not =.

- It is also called *Boolean* expressions or *logical* expressions.
- The resulting type is logical 1 for true or 0 for false**


📖: “true” is represented by the logical value 1, and “false” is represented by the logical value 0. A logical type only contain two value, 0 or 1.

## Relational Operator (Continue)

```
% relation operator  
ro1 = 3 < 4;  
ro2 = 3 > 5;  
ro3 = 3 == 5;  
ro4 = 3 ~= 7;  
ro5 = 3 <= 3;  
ro6 = 3 >= 3;  
ro7 = 3 > 3;
```

: In the second line, the code is first running the expression ( $3 < 4$ ) and computing its value. This expression means “3 less than 4”, which is a true. Thus, `true` (or logical 1) is assigned to `ro1`.

Name	Value
ro1	1
ro2	0
ro3	0
ro4	1
ro5	1
ro6	1
ro7	0

: Here, the values are all *logical* values, 0 or 1, not *double*.



# Logical Operator

- The logical **operators** are:


||            or for scalars  
&&          and for scalars  
~            not

- Note that the logical operators are **commutative**
  - (e.g., `x || y` is equivalent to `y || x`)
- The resulting type is **logical 1** for true or 0 for false

x	y
true	true
true	false
false	false



~x	x    y	x && y
false	true	true
false	true	false
true	false	false

 For example, logical (1) && logical (1) becomes 1. This is the same with true && true.

# Logical Operator (Continue)

```
% relation operator
lo1 = true && false
lo2 = 1 || 0
lo3 = 0 || 0
lo4 = true && true
lo5 = 0 || ~false
lo6 = ~false && true
lo7 = ~true || false
```

Name	Value
lo1	0
lo2	1
lo3	0
lo4	1
lo5	1
lo6	1
lo7	0

x	y
true	true
true	false
false	false




~x	x    y	x && y
false	true	true
false	true	false
true	false	false

# General Operator Precedence

## Challenging

Parentheses () Highest  
Power: ^  
Unary: negation (-), not (~)  
Multiplication or division: \*, /, \  
Addition, subtraction: +, -  
Relational: <, <=, >, ==, ~=  
And: &&  
Or: || Lowest



☺: I know it's bit hard to wrap your head but, you consider relational and logical operators as common operators like +, - and compute the values with this precedence order.

📖: **Remember ! 0 is false otherwise true.**

```
val1 = 3 < (1 + 3)
val2 = (3 < 1) + 3
val3 = 3 < 1 + 3
```

📖: The type of val2 is double and the types of val1 and val3 are logical.

Name	Value
val1	1
val2	3
val3	1

# Example: Operator Precedence



```
lg1 = (3 < 4) < 4;  
lg2 = 3 < (4 < 5);  
lg3 = (3 > 5) + 3;  
lg4 = (10 > 4) && (4 > 1);  
lg5 = (10 < 4) && (4 < 1);  
lg6 = ~((10 < 4) && (4 < 1));  
lg7 = 2 < 3 + 4;
```

Parentheses ()                      Highest

Power: ^

Unary: negation (-), not (~)

Multiplication or division: \*, / , \

Addition, subtraction: +, -

Relational: <, <=, >, ==, ~=

And: &&

Or: ||

Lowest

Q. What values are assigned to the variables?

Name	Value
lg1	1
lg2	0
lg3	3
lg4	1
lg5	0
lg6	1
lg7	1

😊: Regardless of the operator precedence, you could use parentheses to clarify the operation order

## Example: Operator Precedence



Q. How to write a code to check if  $x$  lies in between 5 and 10. If yes, 1 and otherwise 0.

```
x1 = 6;  
x2 = 11;  
  
lg1 = (5>x1) && (x1<10)  
lg2 = 5 < x1 <10; % incorrect!  
lg3 = (5 < x1) <10; % incorrect!  
  
lg4 = (5 > x2) && (x2 < 10)  
lg5 = 5 < x2 < 10; % incorrect!  
lg6 = (5 < x2) < 10; % incorrect!
```

Name	Value
x1	6
x2	11
lg1	0
lg2	1
lg3	1
lg4	0
lg5	1
lg6	1

**⚠:** For lg2, the first expression  $5 < x$  will be evaluated. It gives a logical value 1. Then, the rest of the expression will be evaluated,  $1 < 10$ . So, the final value to be assigned to lg2 become logical 1, true.

## Using Functions: Terminology

- To use a function, you **call** it
- To call a function, give its name followed by the **argument(s)** that are **passed** to it in parentheses




**out = fun (arg1, arg2, ...)**

- Many functions calculate values and **return** the results
- For example, to find the absolute value of  $-4$

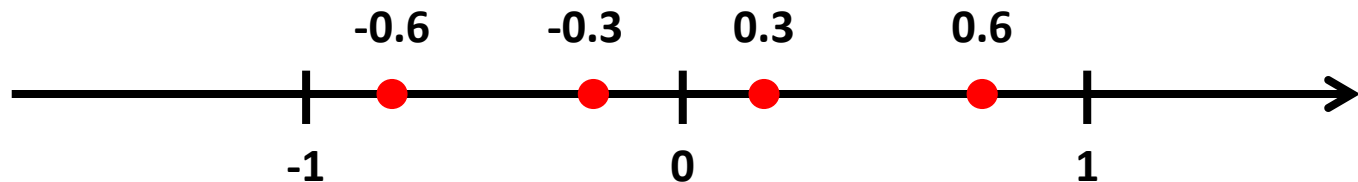
```
>> abs(-4)
ans =
     4
```

- The name of the function is “abs”
- One argument,  $-4$ , is passed to the `abs` function
- The `abs` function finds the absolute value of  $-4$  and returns the result,  $4$ .

# Rounding Functions

Function	Description	Note
<code>round(x)</code>	Rounds <b>x</b> to the nearest integer	
<code>fix(x)</code>	Truncates <b>x</b> to the nearest integer toward zero.	
<code>floor(x)</code>	Rounds <b>x</b> to the nearest integer toward negative infinity.	
<code>ceil(x)</code>	Rounds <b>x</b> to the nearest integer toward positive infinity.	

# Rounding Functions (Continue)



Function	Description
<code>round(x)</code>	Rounds <b>x</b> to the nearest integer
<code>fix(x)</code>	Truncates <b>x</b> to the nearest integer toward zero.
<code>floor(x)</code>	Rounds <b>x</b> to the nearest integer toward negative infinity.
<code>ceil(x)</code>	Rounds <b>x</b> to the nearest integer toward positive infinity.

	round	ceil	fix	floor
0.3	0	1	0	0
0.6	1	1	0	0
-0.3	0	0	0	-1
-0.6	-1	0	0	-1



## Rounding Functions (Continue)

```
x1 = 10.3;  
x2 = 12.7;  
x3 = -1.3;
```

```
x1_ce = ceil(x1);  
x1_fi = fix(x1);  
x1_fl = floor(x1);
```

```
x2_ce = ceil(x2);  
x2_fi = fix(x2);  
x2_fl = floor(x2);
```

```
x3_ce = ceil(x3);  
x3_fi = fix(x3);  
x3_fl = floor(x3);  
x3_ro = round(x3);
```

Name	Value
x1	10.3
x2	12.7
x3	-1.3
x1_ce	11
x1_fi	10
x1_fl	10
x2_ce	13
x2_fi	12
x2_fl	12
x3_ce	-1
x3_fi	-1
x3_fl	-2
x3_ro	-1

Function	Description
<b>round (x)</b>	Rounds <b>x</b> to the nearest integer
<b>fix (x)</b>	Truncates <b>x</b> to the nearest integer toward zero.
<b>floor (x)</b>	Rounds <b>x</b> to the nearest integer toward negative infinity.
<b>ceil (x)</b>	Rounds <b>x</b> to the nearest integer toward positive infinity.

😊: You do not have to memorize the functions and their usage. You can simply search for their usage in google or type `doc round` in command window (`doc fun_name`).

# Common Math Functions

Function	Description	Script	Value
<b>abs (x)</b>	Finds the absolute value of <b>x</b>	<b>abs (-3)</b> <b>abs (2)</b>	3 2
<b>sqrt (x)</b>	Finds the square root of <b>x</b>	<b>sqrt (4)</b> <b>sqrt (1.75)</b>	2 1.5
<b>sign (x)</b>	Return <b>-1</b> if <b>x</b> is less than zero, a value of <b>0</b> if <b>x</b> equals zero, and a value of <b>1</b> if <b>x</b> is greater than zero.	<b>sign (-5)</b> <b>sign (3)</b> <b>sign (0)</b>	-1 1 0
<b>rem (x,y)</b>	Computes the remainder of <b>x/y</b>	<b>rem (25,4)</b> <b>rem (4,2)</b> <b>rem (9,5)</b>	1 0 4

# Common Math Functions (Continue)

Function	Description
<b>abs (x)</b>	Finds the absolute value of <b>x</b>
<b>sqrt (x)</b>	Finds the square root of <b>x</b>
<b>sign (x)</b>	Return <b>-1</b> if <b>x</b> is less than zero, a value of <b>0</b> if <b>x</b> equals zero, and a value of <b>1</b> if <b>x</b> is greater than zero.
<b>rem (x,y)</b>	Computes the remainder of <b>x/y</b>

```
x = -4;  
y = 9;  
z = 2;  
  
x_ab = abs(x)  
x_si = sign(x)  
  
xz_r = rem(x, z)  
  
y_ab = abs(y)  
y_sq = sqrt(y)  
y_si = sign(y)  
  
yz_r = rem(y, z)
```

Name	Value
x	<b>-4</b>
y	<b>9</b>
z	<b>2</b>
x_ab	<b>4</b>
x_si	<b>-1</b>
xz_r	<b>0</b>
y_ab	<b>9</b>
y_sq	<b>3</b>
y_si	<b>1</b>
yz_r	<b>1</b>

# Logical Operation Functions

Function	Operator
<b>and(A, B)</b>	A && B
<b>or(A, B)</b>	A    B
<b>not(A)</b>	~A

☺: These functions improve readability.

For instance:

```
log1 = 3<x && x<10
```

```
log2 = and(3<x, x<10)
```

```
x1 = true;  
x2 = false;  
  
lg1a = and(x1, x2);  
lg1b = x1 && x2;  
  
lg2a = or(x1, x2);  
lg2b = x1 || x2  
  
lg3 = not(x1);
```

Name	Value
x1	1
x2	0
lg1a	0
lg1b	0
lg2a	1
lg2b	1
lg3	0

- MATLAB generates pseudorandom numbers. These numbers are not strictly random and independent in the mathematical sense, but they pass various statistical tests of randomness and independence, and their calculation can be repeated for testing or diagnostic purposes.
- Several built-in functions generate random numbers.
- Random number generators start with a number called the `seed`. This is either a predetermined value or **from the clock**. This is considered as an “index” of a random number lookup table.

Seed: n  
Seed: 2  
Seed: 1

Seed: i

Seed: i

0000	0001	0002	0003	0004	0005	0006	0007	0008	0009	0010	0011	0012	0013	0014	0015	0016	0017	0018	0019	0020	0021	0022	0023	0024	0025	0026	0027	0028	0029	0030	0031	0032	0033	0034	0035	0036	0037	0038	0039	0040	0041	0042	0043	0044	0045	0046	0047	0048	0049	0050	0051	0052	0053	0054	0055	0056	0057	0058	0059	0060	0061	0062	0063	0064	0065	0066	0067	0068	0069	0070	0071	0072	0073	0074	0075	0076	0077	0078	0079	0080	0081	0082	0083	0084	0085	0086	0087	0088	0089	0090	0091	0092	0093	0094	0095	0096	0097	0098	0099	0100	0101	0102	0103	0104	0105	0106	0107	0108	0109	0110	0111	0112	0113	0114	0115	0116	0117	0118	0119	0120	0121	0122	0123	0124	0125	0126	0127	0128	0129	0130	0131	0132	0133	0134	0135	0136	0137	0138	0139	0140	0141	0142	0143	0144	0145	0146	0147	0148	0149	0150	0151	0152	0153	0154	0155	0156	0157	0158	0159	0160	0161	0162	0163	0164	0165	0166	0167	0168	0169	0170	0171	0172	0173	0174	0175	0176	0177	0178	0179	0180	0181	0182	0183	0184	0185	0186	0187	0188	0189	0190	0191	0192	0193	0194	0195	0196	0197	0198	0199	0200	0201	0202	0203	0204	0205	0206	0207	0208	0209	0210	0211	0212	0213	0214	0215	0216	0217	0218	0219	0220	0221	0222	0223	0224	0225	0226	0227	0228	0229	0230	0231	0232	0233	0234	0235	0236	0237	0238	0239	0240	0241	0242	0243	0244	0245	0246	0247	0248	0249	0250	0251	0252	0253	0254	0255	0256	0257	0258	0259	0260	0261	0262	0263	0264	0265	0266	0267	0268	0269	0270	0271	0272	0273	0274	0275	0276	0277	0278	0279	0280	0281	0282	0283	0284	0285	0286	0287	0288	0289	0290	0291	0292	0293	0294	0295	0296	0297	0298	0299	0300	0301	0302	0303	0304	0305	0306	0307	0308	0309	0310	0311	0312	0313	0314	0315	0316	0317	0318	0319	0320	0321	0322	0323	0324	0325	0326	0327	0328	0329	0330	0331	0332	0333	0334	0335	0336	0337	0338	0339	0340	0341	0342	0343	0344	0345	0346	0347	0348	0349	0350	0351	0352	0353	0354	0355	0356	0357	0358	0359	0360	0361	0362	0363	0364	0365	0366	0367	0368	0369	0370	0371	0372	0373	0374	0375	0376	0377	0378	0379	0380	0381	0382	0383	0384	0385	0386	0387	0388	0389	0390	0391	0392	0393	0394	0395	0396	0397	0398	0399	0400	0401	0402	0403	0404	0405	0406	0407	0408	0409	0410	0411	0412	0413	0414	0415	0416	0417	0418	0419	0420	0421	0422	0423	0424	0425	0426	0427	0428	0429	0430	0431	0432	0433	0434	0435	0436	0437	0438	0439	0440	0441	0442	0443	0444	0445	0446	0447	0448	0449	0450	0451	0452	0453
------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------

[illegible]

## Random Number (Continue)

## Challenging

- **rand(n)** creates an  $n \times n$  matrix of random reals
- **rand(n,m)** create an  $n \times m$  matrix of random reals
- **randi([range],n,m)** creates an  $n \times m$  matrix of random integers in the specified range
- **rng(seed)** specifies the seed for the random generator

```
>> rand(1)
```


```
ans =
```

```
0.8147
```

```
>> rand(1)
```

```
ans =
```

```
0.9058
```

: A seed is changed in each run of your script. Thus, different random numbers are generated

```
>> rng(10); rand(1)
```


```
ans =
```

```
0.7713
```

```
>> rng(10); rand(1)
```

```
ans =
```

```
0.7713
```

: You can specify a seed. Then, the same numbers are generated.

# Write a Numerical Expression

Suppose that you want to compute  $y$  when  $x = 10$

$$y = \frac{\{x^2(100x + 10) + x^3(20x^2 + 3)\}}{-x^{-3} + 1}$$

```
x = 10;  
y1 = (x^2*(100*x + 10) + x^3*(20*x^2 + 3))/(-x^(-3)+1)
```

☺: The more terms and parentheses in your equation, the larger the probability that you may be making a mistake. In this case, I usually do it like this

```
x = 10;  
y_nom = (x^2)*(100*x + 10) + (x^3)*(20*(x^2) + 3);  
y_den = -x^(-3)+1;  
y = y_nom/y_den;
```

# Arithmetic Operation in MATLAB

Optional

Suppose that you are solving a problem: If the car has a mass of 300kg and you push the car with an acceleration of 5 inch/s<sup>2</sup>, compute the force that is generated from the car in newton

```
% MATLAB as a calculator  
300*5*0.0254
```

Name	Value
ans	38.1

☺: If you are using MATLAB as a programming tool, I would recommend writing the code below. This improves code readability and allows others to understand your code.

```
% MATLAB as programming tool  
  
inch2m = 0.0254; % inch to m  
mass = 300; % kg  
accel = 5; % inch/s/s  
  
% force = mass(kg) * acceleration (m/s/s)  
force = mass * accel * inch2m;
```

Name	Value
inch2m	0.0254
mass	300
accel	5
force	38.1