MATLAB Tutorial

Chul Min Yeum

Assistant Professor
Civil and Environmental Engineering
University of Waterloo, Canada

UNIVERSITY OF WATERLOO
FACULTY OF ENGINEERING

CIVE 497 – CIVE 700: Smart Structure Technology

Last updated: 2020-01-13

Variables and Assignments

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

`variable = expression`

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

Name	Value
`a1`	**3**
`a2`	**5**

- The order is important
    - Variable name on the left
    - Assignment operator “=” (**Note: this does NOT mean equality**)
    - Expression on the right

Variable names

- Names **must** begin with a letter of the alphabet
- After that names can contain letters, digits, and the underscore character(_)
- You cannot use other character except for '_'
- MATLAB is **case-sensitive**
- Names should be mnemonic (they should make sense!)
- Use a workspace browser rather than type these command
- **`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
```

```
% define a variable of 'ga1'
ga1 = 100
```

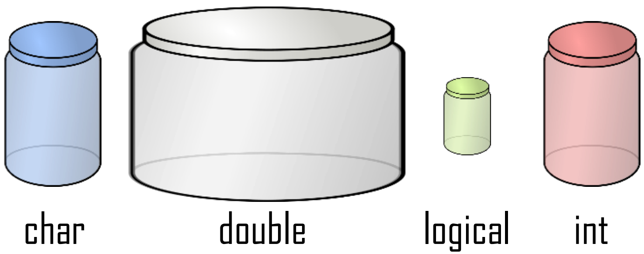
Q: What value is in 'Ga1' ?

Data Types

- Every expression and variable has an associated *type*, or *class*
  - Real numbers: **single**, **double**
  - 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**

The default type for numbers in MATLAB is **double**

Container



Example

- 2.145, 0.15893, 3.0, 2.45
- 10, 11, 24, 30, 400
- ‘a’, ‘b’, ‘A’, ‘c’
- 0, 1

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 and is 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

Example: Arithmetic Operation in MATLAB

If the car has a mass of 300kg and you push the car with an acceleration of 5 , compute the force that is generated from the car in newton

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

Name	Value
ans	**38.1**

```
% 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**

Operator Precedence

- A symbol that perform specific mathematical or logical manipulations.
- Precedence list (highest to lowest) so far:
  - ( ) 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);
```

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

Relational Operator

- The relational operators in MATLAB are:

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

- The resulting type is **logical** 1 for true or 0 for false

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

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

Logical Operator

- The logical operators are:

  `||` or for scalars

  `&&` and for scalars

  `~` not

- Also, **`xor`** function which returns true if only one of the arguments is true
- Note that the logical operators are commutative
  - (e.g., `x|| y` is equivalent to `y || x`)
- <u>The resulting type is **logical** 1 for true or 0 for false</u>

x	y
true	true
true	false
false	false

~x	x || y	x && y	xor(x,y)
false	true	true	false
false	true	false	true
true	false	false	false

Operator Precedence

Parentheses `()`

Transpose `(.')`, power `(.^)`, complex conjugate transpose `(')`, matrix power `(^)`

Multiplication `(.*)`, right division `(./)`, left division `(.\)`, matrix multiplication `(*)`, matrix right division `(/)`, matrix left division `(\)`

Addition `(+)`, subtraction `(-)`

Colon operator `(:)`

Less than `(<)`, less than or equal to `(<=)`, greater than `(>)`, greater than or equal to `(>=)`, equal to `(==)`, not equal to `(~=)`

Short-circuit AND `(&&)`

Short-circuit OR `(||)`

```
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;
```

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

Example2: Operator Precedence

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;
lg3 = (5 <= x1) <=10;


lg4 = (5<=x2) && (x2<=10)
lg5 = 5<= x2 <=10;
lg6 = (5 <= x2) <=10;
```

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

Array Operations

Array operations on two matrices A and B:

- These are applied term-by-term, or element-by-element
- The matrices must have the same dimensions (no! after R2016)
- In MATLAB:
  - addition/subtraction: A + B, A – B
  - array multiplication: A .* B
  - array division: A ./ B, A .\ B
  - array exponentiation A .^ 2
- Matrix multiplication: NOT an array operation

```
A = [2 2 2; 4 4 4; 6 6 6];
B = [1 1 1; 2 2 2; 3 3 3];

AmB = A.*B;
AdB = A./B;
```

2	2	2
4	4	4
6	6	6

A

1	1	1
2	2	2
3	3	3

B

2	2	2
8	8	8
18	18	18

AmB

2	2	2
2	2	2
2	2	2

AdB

Matrix Multiplication: Dimensions

- ***Matrix multiplication*** is not an array operation
  - It does not mean multiplying term by term
- In MATLAB, the multiplication operator * performs matrix multiplication
- In order to be able to multiply a matrix A by a matrix B, the number of columns of A must be the same as the number of rows of B
- If the matrix A has dimensions *m x n*, that means that matrix B must have dimensions *n x something*; we will call it *p*
  - In mathematical notation, [A]*m x n* [B]*n x p*
  - We say that the ***inner dimensions*** must be the same
- The resulting matrix C has the same number of rows as A and the same number of columns as B
  - in other words, the ***outer dimensions*** *m x p*
  - In mathematical notation, [A]*m x n* [B]*n x p* = [C]*m x p*.
  - This only defines the size of C

Matrix Times a Vector

A linear system of equations with coefficient matrix A, variable vector $\vec{x}$ and constant term vector $\vec{b}$, can be expressed as

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} & \cdots & a_{1n} \\ a_{21} & a_{22} & a_{23} & \cdots & a_{2n} \\ \vdots & \vdots & \vdots & & \vdots \\ a_{m1} & a_{m2} & \cdots & \cdots & a_{mn} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ \vdots \\ x_n \end{bmatrix} = \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_m \end{bmatrix}$$

$$\underset{(m \times n)}{\text{matrix}} \; \underset{(n \times 1)}{\text{vector}} = \underset{(m \times 1)}{\text{vector}}$$

Compatibility – the number of columns in the matrix **must** equal the number of rows in the vector.

Example: Matrix Times a Vector

```
m1 = [1 1 1; 2 2 2; 3 3 3];
v1 = [2 2 2]';

c1 = m1*v1;

c21 = m1(1,:)*v1;
c22 = m1(2,:)*v1;
c23 = m1(3,:)*v1;

c2 = [c21;c22;c23];
```

$$\begin{bmatrix} 1 & 1 & 1 \\ 2 & 2 & 2 \\ 3 & 3 & 3 \end{bmatrix} * \begin{bmatrix} 2 \\ 2 \\ 2 \end{bmatrix} = \begin{bmatrix} 6 \\ 12 \\ 18 \end{bmatrix}$$

`m1 * v1 = c1`

$$\begin{bmatrix} 1 & 1 & 1 \end{bmatrix} * \begin{bmatrix} 2 \\ 2 \\ 2 \end{bmatrix} = \begin{bmatrix} 6 \end{bmatrix}$$

`m1(1,:) * v1 = c21`

$$\begin{bmatrix} 2 & 2 & 2 \end{bmatrix} * \begin{bmatrix} 2 \\ 2 \\ 2 \end{bmatrix} = \begin{bmatrix} 12 \end{bmatrix}$$

`m1(2,:) * v1 = c22`

$$\begin{bmatrix} 3 & 3 & 3 \end{bmatrix} * \begin{bmatrix} 2 \\ 2 \\ 2 \end{bmatrix} = \begin{bmatrix} 18 \end{bmatrix}$$

`m1(3,:) * v1 = c23`

Matrix Times a Matrix

Matrix multiplication is an extension of matrix and vector multiplication.

Consider the product of an $m \times n$ matrix A, and an $n \times p$ matrix B:

$$AB = \begin{bmatrix} a_{11} & a_{12} & a_{13} & \cdots & a_{1n} \\ a_{21} & a_{22} & a_{23} & \cdots & a_{2n} \\ \vdots & \vdots & \vdots & & \vdots \\ a_{m1} & a_{m2} & \cdots & \cdots & a_{mn} \end{bmatrix} \begin{bmatrix} b_{11} & b_{12} & \cdots & b_{1p} \\ b_{21} & b_{22} & \cdots & b_{2p} \\ b_{31} & b_{32} & \cdots & b_{3p} \\ \vdots & \vdots & & \vdots \\ b_{n1} & b_{n2} & \cdots & b_{np} \end{bmatrix}$$

$$\overset{\text{matrix}}{(m \times n)}\overset{\text{matrix}}{(n \times p)} = \overset{\text{matrix}}{(m \times p)}$$

Compatibility – the number of columns in the first matrix **must** equal the number of rows in the second matrix.

Example: Matrix Times a Matrix

```
m1 = [1 2;3 4];
m2 = [1 2;2 1];

m3 = m1*m2;

m411 = m1(1,:)*m2(:,1);
m421 = m1(2,:)*m2(:,1);
m412 = m1(1,:)*m2(:,2);
m422 = m1(2,:)*m2(:,2);

m4 = [m411 m412;m421 m422];
```

$$\begin{array}{|c|c|}\hline 3 & 4 \\ \hline \end{array} * \begin{array}{|c|}\hline 2 \\ \hline 1 \\ \hline \end{array} = \begin{array}{|c|}\hline 11 \\ \hline \end{array}$$

`m1(1,:) * m2(:,2) = m421`

$$\begin{array}{|c|c|}\hline 1 & 2 \\ \hline 3 & 4 \\ \hline \end{array} * \begin{array}{|c|c|}\hline 1 & 2 \\ \hline 2 & 1 \\ \hline \end{array} = \begin{array}{|c|c|}\hline 5 & 4 \\ \hline 11 & 10 \\ \hline \end{array}$$

`m1 * m2 = m3`

$$\begin{array}{|c|c|}\hline 1 & 2 \\ \hline \end{array} * \begin{array}{|c|}\hline 1 \\ \hline 2 \\ \hline \end{array} = \begin{array}{|c|}\hline 5 \\ \hline \end{array}$$

`m1(1,:) * m2(:,1) = m411`

$$\begin{array}{|c|c|}\hline 1 & 2 \\ \hline \end{array} * \begin{array}{|c|}\hline 2 \\ \hline 1 \\ \hline \end{array} = \begin{array}{|c|}\hline 4 \\ \hline \end{array}$$

`m1(1,:) * m2(:,2) = m412`

$$\begin{array}{|c|c|}\hline 3 & 4 \\ \hline \end{array} * \begin{array}{|c|}\hline 2 \\ \hline 1 \\ \hline \end{array} = \begin{array}{|c|}\hline 10 \\ \hline \end{array}$$

`m1(2,:) * m2(:,2) = m422`

Example: Swapping Columns

Swap the 2nd and 3rd columns in `mat1`

1	2	3
4	5	6
7	8	9

1	3	2
4	6	5
7	9	8

```
mat1 = [1 2 3; 4 5 6; 7 8 9];
```

```
mat1(:, [3 2]) = ...
              mat1(:, [2 3]);
```

Quiz

$$\underset{\mathbf{A}}{\begin{pmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \\ a_{41} & a_{42} & a_{43} \end{pmatrix}} \quad \underset{\mathbf{B}}{\begin{pmatrix} b_{11} & b_{12} & b_{13} & b_{14} \\ b_{21} & b_{22} & b_{23} & b_{24} \\ b_{31} & b_{32} & b_{33} & b_{34} \end{pmatrix}} \quad \underset{\mathbf{r}}{\begin{pmatrix} r_1 & r_2 & r_3 & r_4 \end{pmatrix}} \quad \underset{\mathbf{c}}{\begin{pmatrix} c_1 \\ c_2 \\ c_3 \\ c_4 \end{pmatrix}}$$

`vec_c =`

$$\begin{pmatrix} a_{11}\, r_1 + a_{21}\, r_2 + a_{31}\, r_3 + a_{41}\, r_4 \\ a_{12}\, r_1 + a_{22}\, r_2 + a_{32}\, r_3 + a_{42}\, r_4 \\ a_{13}\, r_1 + a_{23}\, r_2 + a_{33}\, r_3 + a_{43}\, r_4 \end{pmatrix}$$

`mat_f =`

$$\begin{pmatrix} a_{11}\, c_1 + a_{12}\, c_2 + a_{13}\, c_3 \\ a_{21}\, c_1 + a_{22}\, c_2 + a_{23}\, c_3 \\ a_{31}\, c_1 + a_{32}\, c_2 + a_{33}\, c_3 \\ a_{41}\, c_1 + a_{42}\, c_2 + a_{43}\, c_3 \end{pmatrix}$$

`mat_g =`

$$\begin{pmatrix} a_{11}\, b_{11} + a_{12}\, b_{21} & a_{11}\, b_{12} + a_{12}\, b_{22} \\ a_{21}\, b_{11} + a_{22}\, b_{21} & a_{21}\, b_{12} + a_{22}\, b_{22} \end{pmatrix}$$

Linear Indexing

Linear indexing: only using one index into a matrix (MATLAB will unwind it column-by column)

A

1	5	9	5
12	3	4	14
3	17	6	1
4	1	7	2

A(1)	1
A(2,3)	4
A(10)	3
A([10 12])	[4 7]
A(1:4)	[1 12 3 4]’
A([1:4:13])	[1 5 9 5]

Linear index of each cell in A

1	5	9	13
2	6	10	14
3	7	11	15
4	8	12	16

```
m1 = [1 2;3 4];
m2 = [1 2;3 4];

m1(1,1) = 3; % m2(1) = 3;
m1(1,:) = [3 4 5 6];
% m1([1 5 9 13]) = [3 4 5 6];
```

If-Statement

- The **if** statement is used to determine whether or not a statement or group of statements is to be executed
- General form:

```
if condition
    action
end
```

- the *condition* is any relational expression (True or False)
- the *action* is any number of valid statements (including, possibly, just one)
- If the condition is true, the action is executed – <u>otherwise, it is skipped entirely</u>

Example 1: If-Statement

abs(x)	Finds the absolute value of **x**	**abs(-3)** **abs(2)**	3 2

```
x1 = -3
xsign = 1;

if x1<0
   xsign = -1;
end

x1_abs = xsign*x1;
```

Name	Value
x1	**-3**
x1_abs	**3**
xsign	**-1**

```
x1 = 3
xsign = 1;

if x1<0
   xsign = -1;
end

x1_abs = xsign*x1;
```

Name	Value
x1	**3**
x1_abs	**3**
xsign	**1**

Example 2: If-Statement

sign(x)	Return **-1** if **x** is less than zero, a value of **0** if **x** equals zero, and a value of **1** if **x** is greater than zero.	sign(-5) sign(3) sign(0)	-1 1 0

```
x1 = -3
xsign = 0;

if x1<0
   xsign = -1;
end

if x1>0
   xsign = 1;
end
```

Name	Value
x1	**-3**
xsign	**-1**

```
x1 = 3
xsign = 0;

if x1<0
   xsign = -1;
end

if x1>0
   xsign = 1;
end
```

Name	Value
x1	**3**
x1_abs	**1**

If-else Statement

- The **if-else** statement chooses between two actions
- General form:

```
if condition
    action1
else
    action2
end
```

- Only one action is executed; which one depends on the value of the condition (`action1` if it is logical true or action2 if it is false)

Example 1: If-else Statement

abs(x)	Finds the absolute value of **x**	**abs(-3)** **abs(2)**	3 2

```
x1 = -3;
xsign = 1;

if x1<0
   xsign = -1;
end

x1_abs = xsign*x1;
```

Name	Value
x1	**-3**
x1_abs	**3**
xsign	**-1**

```
x1=-3;

if x1 < 3
   x1_abs = -1*x1;
else
   x1_abs = x1;
end
```

Name	Value
x1	**-3**
x1_abs	**3**

Nested if-else Statements

- To choose from more than two actions, ***nested*** **if-else** statements can be used (an **if** or **if-else** statement as the action of another)

- General form:

```
if condition1
    action1
else
    if condition2
       action2
    else
       action3
    end
end
```

Example: Nested if-else Statements

Q: If ‘scalar1’ is larger than 0 and less than 50, assign 10 to ‘out1’. Otherwise, assign 5 to ‘out1’.

```
scalar1 = 20;

if scalar1 > 0
   if scalar1<50
      out1 = 10;
   else
      out1 = 5;
   end
end
```

```
scalar1 = 20;

if (scalar1 > 0) && (scalar1 < 50)
   out1 = 10;
else
   out1 = 5;
end
```

if, if-else, if-elseif, if-elseif-else

```
if condition1
    action1
end
```

```
if condition1
    action1
else
    action2
end
```

```
if condition1
    action1
elseif condition2
    action2
end
```

```
if condition1
    action1
elseif condition2
    action2
else
    action3
end
```

For-Loop Statement

- Used as a <u>counted</u> loop (We know how many times are repeated)
- Repeats an action a specified number of times
- An iterator or loop variable specifies how many times to repeat the action
- General form:

```
for loopvar = range
    action
end
```

- The range is specified by a vector.
- The action is repeated for every value of `loopvar` in the specified vector
- If it is desired to repeat the process of prompting the user and reading input a specified number of times (N), a for loop is used:

```
for ii = 1:N
     % do something with it!
end
```

Example1: How For-Loop Works

```
1  sumv = 0;
2  for ii=1:3
3      sumv = sumv + ii;
4  end
5
```

Step	Operation	Workspace
1	line1: assign `0` to `sumv`	`sumv` → `0`
2	line2: `ii` becomes `1`	`sumv` → `0, ii` → `1`
3	line3: add `sumv` and `ii`, and assign the value to `sumv`	`sumv` → `1, ii` → `1`
4	line4: `end`	`sumv` → `1, ii` → `1`
5	go to line2 and `ii` becomes `2`	`sumv` → `1, ii` → `2`
6	line3: add `sum` and `ii`, and assign the value to `sumv`	`sumv` → `3, ii` → `2`
7	line4: `end`	`sumv` → `3, ii` → `2`
8	go to line2 and `ii` becomes `3`	`sumv` → `3, ii` → `3`
9	line3: add `sum` and `ii`, and assign the value to `sumv`	`sumv` → `6, ii` → `3`
10	line4: `end`	`sumv` → `6, ii` → `3`
11	no more value in range, and go to line5	`sumv` → `6, ii` → `3`

Example2: Summation Using For-Loop (continue)

Sum all values in a vector named ‘`vec`’ and assign the value to ‘`sumv`’

```
vec = [1 3 7 11];
sumv = 0;
for ii=1:4
    sumv = sumv + vec(ii);
end
```

```
vec = [1 3 7 11];
sumv = 0;
for ii=vec
    sumv = sumv + ii;
end
```

```
for ii = 1:N
    % do something!
end
```

```
for loopvar = range
    action
end
```

Q. Do we need `ii`?

Nested For-Loop Statement

- A nested **for** loop is one inside of (as the action of) another **for** loop
- General form of a nested **for** loop:

```
for loopvar1 = range1
    action1
    for loopvar2 = range2
      action2
    end
end
```

- The inner loop action is executed in its entirety for every value of the outer loop variable

Example1: How Nested For-Loop Works

```
1  mat1 = [1 2 3;4 5 6];
2  sumv = 0;
3  for ii=1:2
4      for jj=1:3
5          sumv = sumv + mat1(ii,jj);
6      end
7  end
```

mat1

1	2	3
4	5	6

Step	Operation	Workspace
1	line1: assign values to `mat1`	`mat1 → [1 2 3;4 5 6]`
2	line2: assign `0` to `sumv`	`mat1 → [1 2 3;4 5 6], sumv → 0`
3	line3: `ii` becomes 1	`ii → 1, mat1 → [1 2 3;4 5 6], sumv → 0`
4	line4: `jj` becomes 1	`ii → 1, jj → 1, mat1 → [1 2 3;4 5 6], sumv → 0`
5	line5: read a value at `1` row and `1` column in mat1, and add the value to `sumv`	`ii → 1, jj → 1, mat1 → [1 2 3;4 5 6], sumv → 1`
6	line6: `end`	`ii → 1, jj → 1, mat1 → [1 2 3;4 5 6], sumv → 1`
7	line4: `jj` becomes 2	`ii → 1, jj → 2, mat1 → [1 2 3;4 5 6], sumv → 1`

Example1: How Nested For-Loop Works

```
1  mat1 = [1 2 3;4 5 6];
2  sumv = 0;
3  for ii=1:2
4      for jj=1:3
5          sumv = sumv + mat1(ii,jj);
6      end
7  end
```

mat1

1	2	3
4	5	6

Step	Operation	Workspace
7	line5: read a value at 1 row and 2 column in mat1, and add the value to `sumv`	`ii → 1, jj → 2, mat1 → [1 2 3;4 5 6], sumv → 3`
8	line6: `end`	`ii → 1, jj → 2, mat1 → [1 2 3;4 5 6], sumv → 3`
9	line4: `jj` becomes 3	`ii → 1, jj → 3, mat1 → [1 2 3;4 5 6], sumv → 3`
10	line5: read a value at 1 row and 3 column in mat1, and add the value to `sumv`	`ii → 1, jj → 3, mat1 → [1 2 3;4 5 6], sumv → 6`
11	line6: `end` and line7: `end`	`ii → 1, jj → 3, mat1 → [1 2 3;4 5 6], sumv → 6`
12	line3: `ii` becomes 2	`ii → 2, jj → 3, mat1 → [1 2 3;4 5 6], sumv → 6`
13	line4: `jj` becomes 1	`ii → 2, jj → 1, mat1 → [1 2 3;4 5 6], sumv → 6`

Example1: How Nested For-Loop Works

```
1  mat1 = [1 2 3;4 5 6];
2  sumv = 0;
3  for ii=1:2
4      for jj=1:3
5          sumv = sumv + mat1(ii,jj);
6      end
7  end
```

mat1

1	2	3
4	5	6

Step	Operation	Workspace
14	line5: read a value at `1` row and `2` column in mat1, and add the value to `sumv`	`ii → 2, jj → 1, mat1 → [1 2 3;4 5 6], sumv → 10`
15	line6: `end`	`ii → 2, jj → 1, mat1 → [1 2 3;4 5 6], sumv → 10`
16	line4: `jj` becomes 2	`ii → 2, jj → 2, mat1 → [1 2 3;4 5 6], sumv → 10`
17	line5: read a value at `1` row and `3` column in mat1, and add the value to `sumv`	`ii → 2, jj → 2, mat1 → [1 2 3;4 5 6], sumv → 15`
18	line6: `end`	`ii → 2, jj → 2, mat1 → [1 2 3;4 5 6], sumv → 15`
19	line4: `jj` becomes 3	`ii → 2, jj → 3, mat1 → [1 2 3;4 5 6], sumv → 15`

MATLAB Operator

Symbol	Role
+	Addition
–	Subtraction
*	Multiplication
/	Division
^	Matrix power
./	Element-wise right division
.*	Element-wise multiplication
.^	Element-wise power
`	Transpose

<table>
<tr><th>Symbol</th><th>Role</th></tr>
<tr><td>==</td><td>Equal to</td></tr>
<tr><td>~=</td><td>Not equal to</td></tr>
<tr><td>></td><td>Greater than</td></tr>
<tr><td>>=</td><td>Greater than or equal to</td></tr>
<tr><td><</td><td>Less than</td></tr>
<tr><td><=</td><td>Less than or equal to</td></tr>
<tr><td>&&</td><td>Logical AND (scalar logical)</td></tr>
<tr><td>||</td><td>Logical OR (scalar logical)</td></tr>
<tr><td>~</td><td>Logical NOT</td></tr>
<tr><td>&</td><td>Logical AND (array)</td></tr>
<tr><td>|</td><td>Logical OR (array)</td></tr>
</table>

Assign Value(s) to a Variable

variable = expression

```
val2 = 2; % scalar value

vec1 = [1 2 3 4]; % vector

vec3  =  vec1-val2
```

- Conduct a ‘minus’ operation between a vector of ‘vec1’ and a scalar value ‘val2’. This operation subtract 2 from each element in [1 2 3 4]. A resulting vector is [-1 0 1 2].

- Assign a resulting vector to ‘vec3’. ‘vec3’ contains [-1 0 1 2].

Q: Matrix multiplication ?

MATLAB Operator

```
val1 = 10; % scalar value
val2 = 2; % scalar value

vec1 = [1 2 3 4]; % vector

mat1 = [1 2; 3 4]; % matrix
mat2 = [5 6; 7 8]; % matrix
```

1	2
3	4

mat1

5	6
7	8

mat2

1	2	3	4

vec1

Operator
+
-
*
/
^
./

Scalar ★ Scalar
Vector ★ Scalar
Matrix ★ Scalar
Vector ★ Vector
Matrix ★ Matrix

★: Operator

val1 + val2	12
val1 * val2	20
val1 / val2	5
val1^val2	100
vec1*val1	[10 20 30 40]
vec1/val2	[0.5 1 1.5 2]
vec1 + val2	[3 4 5 6]
vec1 - val2	[-1 0 1 2]
mat1*val2	[2 4; 6 8]
mat1/val2	[0.5 1; 1.5 2]
mat1 + val2	[3 4; 5 6]
mat1 - val2	[-1 0;1 2]
mat1 + mat2	[6 8; 10 12]
mat1 - mat2	[-4 -4;-4 -4]

MATLAB Operator

```
vec1 = [1 2]; % vector
vec2 = [1; 0]; % vector

mat1 = [1 2; 3 4]; % matrix
mat2 = [1 0; 0 1]; % matrix
```

1	**2**
3	**4**

mat1

1	**0**
0	**1**

mat2

1	**2**

vec1

1
0

vec2

Operator
+
-
*
/
^
./
.*
.^
`

Scalar ★ Scalar

Vector ★ Scalar

Matrix ★ Scalar

Vector ★ Vector

Matrix ★ Matrix

★: Operator

mat1*vec1	*error*
mat1*vec2	[1; 3]
vec1*mat1	[7 10]
vec2*mat1	*error*
mat1*mat2	[1 2;3 4]
mat1*mat1	[7 10;15 22]
mat1^2	[7 10;15 22]
vec1.* vec2’	[1 0]
mat1.*mat2	[1 0;0 4]
mat2./mat1	[1 0;0 0.25]
mat1.^2	[1 4;9 16]
mat1.*vec2	[1 2;0 0]
mat1.*vec1	[1 4;3 8]

Q. How to Change the Matrix A

1	2	3
4	5	6
7	8	9

A

8	2	3
4	8	6
7	8	8

A

1	2	8
4	8	6
8	8	9

?

1	2	3
4	5	6
7	8	9

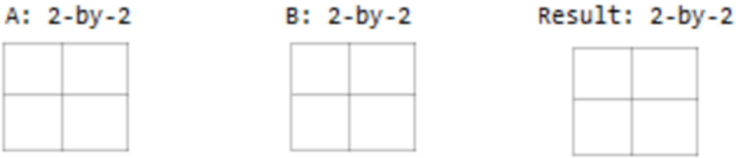
A

5	4	3
20	10	6
35	16	9

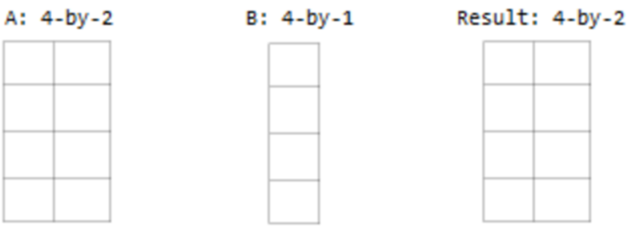
A

MATLAB Implicitly Expands Element-wise Operations (Changed After R2016b)

Two inputs which are exactly the same size.



One input is a matrix, and the other is a column vector with the same number of rows.



$$\begin{bmatrix} 1 & 2 \\ 3 & 4 \\ 5 & 6 \end{bmatrix} .* \begin{bmatrix} 1 \\ 1 \\ 2 \end{bmatrix} \quad ?$$

MATLAB Operator

```
val1 = 1; % scalar value
val2 = 2; % scalar value

vec1 = [1 2 3 4]; % vector
vec2 = [1 0 1 4]; % vector

mat1 = [1 2; 3 4]; % matrix
mat2 = [1 0; 5 4]; % matrix
```

Symbol	Role
==	Equal to
~=	Not equal to
>	Greater than
>=	Greater than or equal to
<	Less than
<=	Less than or equal to

1	**2**
3	**4**

mat1

1	**0**
0	**1**

mat2

1	**2**	**3**	**4**	vec1
1	**0**	**1**	**4**	vec2

`vec1== val1`	`[1 0 0 0]`
`vec1<=val2`	`[1 1 0 0]`
`vec1 ~= val2`	`[1 0 1 1]`
`vec1 == vec2`	`[1 0 0 1]`
`vec1 >= vec2`	`[1 1 1 1]`
`vec1 < vec2`	`[0 0 0 0]`
`mat1 == val1`	`[1 0;0 0]`
`mat1 ~= val2`	`[1 0;1 1]`
`mat1 == mat2`	`[1 0;0 1]`
`mat1 ~= mat2`	`[0 1;1 0]`
`mat1 >= mat2`	`[1 1;0 1]`
`mat1 == [1 2]`	`[1 1;0 0]`
`mat1 == [1;2]`	`[1 0;0 0]`
`mat1 > [1 2]`	`[0 0;1 1]`

MATLAB Operator

```
val1 = 1; val2 = 0;

vec1 = [1 0]; vec2 = [0 0];

mat1 = [1 1; 0 0];
mat2 = [1 0; 0 1];
```

Symbol	Role
`&&`	Logical AND (scalar logical)
`||`	Logical OR (scalar logical)
`~`	Logical NOT
`&`	Logical AND (array)
`|`	Logical OR (array)

1	**1**
0	**0**

`mat1`

1	**0**
0	**1**

`mat2`

1	**0**	`vec1`
0	**0**	`vec2`

`val1 && val1`	1
`val2 && val2`	0
`val1 || val2`	1
`val2 || val2`	0
`~ val1`	0
`~ val2`	1
`vec1 && val1`	*error*
`vec1 & val1`	[1 0]
`vec1 & val2`	[0 0]
`vec1 | val2`	[1 0]
`vec2 | val1`	[1 1]
`vec2 | vec1`	[1 0]
`vec2 & vec1`	[0 0]
`mat1 | mat2`	[1 1;0 1]
`mat1 & mat2`	[1 0;0 0]
`mat1 & vec1`	[1 0;0 0]
`mat1 | vec2`	[1 1;0 0]

Value Replacement

If values in ‘vec1' are larger than 0 and less than 50, replace the values to 10. Otherwise, replace them to 5.

```
vec1 = [1 10 70 80 2]
```

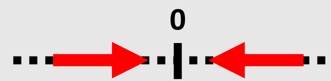
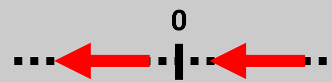
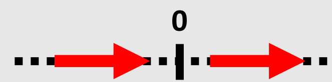
```
vec1 = [1 10 70 80 2];

for ii=1:numel(vec1)
   testv = vec1(ii);
   if testv > 0
       if testv < 50
           replv = 10;
       else
           replv = 5;
       end
   else
       replv = 5;
   end
   vec1(ii) = replv;
end
```

```
vec1 = [1 10 70 80 2];

logi10 = and(vec1>0, vec1<50);

vec1(logi10) = 10;
vec1(~logi10) = 5;
```

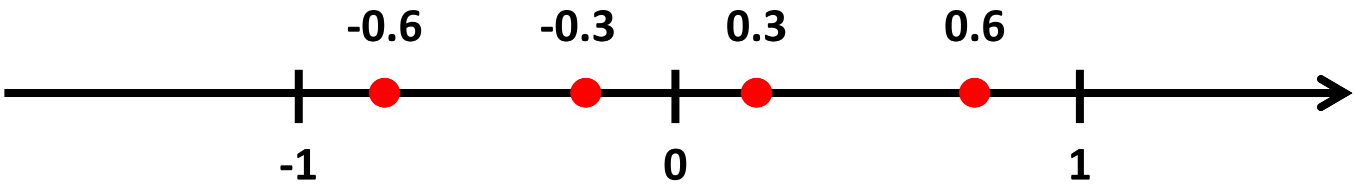
‘`vec1`’ becomes

```
vec1 = [10 10 5 5 10]
```

Rounding Functions

Function	Description	Note
`round(x)`	Rounds **x** to the nearest integer	
`fix (x)`	Truncates **x** to the nearest integer toward zero.	
`floor(x)`	Rounds **x** to the nearest integer toward negative infinity.	
`ceil(x)`	Rounds **x** to the nearest integer toward positive infinity.	

Rounding Functions (Example 1)



	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 (Example 2)

```
x1 = -0.6;
x2 = -0.3;
x3 = 0.3;
x4 = 0.6;

x = [x1 x2 x3 x4];

x_ce = ceil(x);
x_fi = fix(x);
x_fl = floor(x);
x_ro = round(x);
```

Name	Value
x1	-0.6
x2	-0.3
x3	0.3
x4	0.3
x	[-0.6 -0.3 0.3 0.6]
x_ce	[0 0 1 1]
x_fi	[0 0 0 0]
x_fl	[-1 -1 0 0]
x_ro	[-1 0 0 1]

Array Operations

- **reshape** changes dimensions of a matrix to any matrix with the same number of elements
- **diag** create diagonal matrix or get diagonal elements of matrix
- **rot90** rotates a matrix 90 degrees counter-clockwise
- **fliplr** flips columns of a matrix from left to right
- **flipud** flips rows of a matrix up to down
- **flip** flips a row vector left to right, column vector or matrix up to down
- **repmat** replicates an entire matrix; it creates *m x n* copies of the matrix
- **repelem** replicates each element from a matrix in the dimensions specified

Example: Create and Index Arrays

```
mat1 = zeros(3,3)
mat2 = ones(3,3)

mat4 = ones(6,6);
mat5 = repmat(mat2, 2, 2);

mat6 = eye(3,3);
mat7 = diag(ones(3,1));
```

```
mat1 = 3×3
     0     0     0
     0     0     0
     0     0     0

mat2 = 3×3
     1     1     1
     1     1     1
     1     1     1

mat4 = 6×6
     1     1     1     1     1     1
     1     1     1     1     1     1
     1     1     1     1     1     1
     1     1     1     1     1     1
     1     1     1     1     1     1
     1     1     1     1     1     1
```

```
mat5 = 6×6
     1     1     1     1     1     1
     1     1     1     1     1     1
     1     1     1     1     1     1
     1     1     1     1     1     1
     1     1     1     1     1     1
     1     1     1     1     1     1

mat6 = 3×3
     1     0     0
     0     1     0
     0     0     1

mat7 = 3×3
     1     0     0
     0     1     0
     0     0     1
```

Example: Combine and Transform Array (horcat, vertcat, cat)

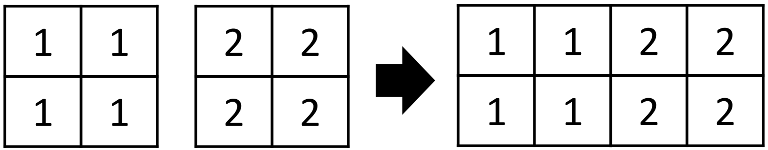
```
mat01 = reshape(1:9, 3, 3);
mat02 = horzcat(mat01, mat01);
mat03 = cat(2, mat01, mat01);

mat05 = vertcat(mat01, mat01);
mat06 = cat(1, mat01, mat01);
```

```
mat01 = 3×3
     1     4     7
     2     5     8
     3     6     9

mat02 = 3×6
     1     4     7     1     4     7
     2     5     8     2     5     8
     3     6     9     3     6     9

mat03 = 3×6
     1     4     7     1     4     7
     2     5     8     2     5     8
     3     6     9     3     6     9
```

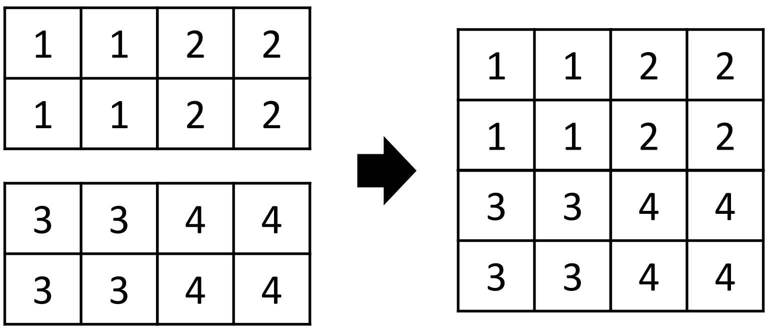
```
mat05 = 6×3
     1     4     7
     2     5     8
     3     6     9
     1     4     7
     2     5     8
     3     6     9

mat06 = 6×3
     1     4     7
     2     5     8
     3     6     9
     1     4     7
     2     5     8
     3     6     9
```

Example: Combine and Transform Array (repelem)

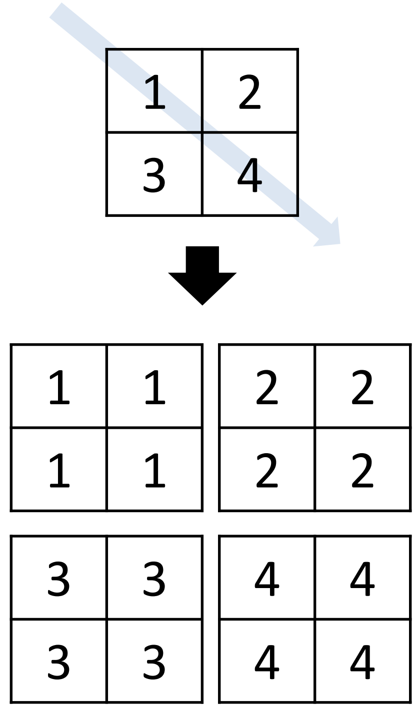
```
mat08 = cat(2, ones(2,2), ones(2,2)+1);
mat09 = cat(1, mat08, mat08+2);
mat10 = repelem([1 2;3 4], 2, 2);
```



mat08



mat09



mat10

Example: Combine and Transform Array

```
mat1 = reshape(1:9, 3, 3)
mat2 = flip(mat1, 1)
mat3 = flipud(mat1)

mat4 = flip(mat1, 2)
mat5 = fliplr(mat1)

mat6 = transpose(mat1)
mat7 = mat1’
mat8 = flipud(rot90(mat1))
```

1	4	7
2	5	8
3	6	9

mat1

3	6	9
2	5	8
1	4	7

mat2

3	6	9
2	5	8
1	4	7

mat3

7	4	1
8	5	2
9	6	3

mat4

7	4	1
8	5	2
9	6	3

mat5

1	2	3
4	5	6
7	8	9

mat6

1	2	3
4	5	6
7	8	9

mat7

1	2	3
4	5	6
7	8	9

mat8

Cell Array

- A **cell array** is a type of data structure that can **store different types of values** in its elements
- A cell array could be a vector (row or column) or a matrix
- It is an array, so indices are used to refer to the elements

e.g., Store multiple values in different types or matrix in a single variable

Cell Array

- The syntax used to create a cell array is curly braces { } instead of [ ]
- The direct method is to put values in the row(s) separated by commas or spaces, and to separate the rows with semicolons (so, same as other arrays) – the difference is using { } instead of [ ]
- The cell function can also be used to preallocate by passing the dimensions of the cell array, e.g.
- `cell(4,2)`

Example: Containing Multiple Data Types

```
temp_st1 = [10 11; 12 15; 9 7];
temp_st2 = [12 13];
temp_st3 = [8 9; 11 12];

num_st1 = 3;
num_st2 = 1;
num_st3 = 2;

name_st1 = ‘Kitchener’;
name_st2 = ‘Waterloo’;
name_st3 = ‘Guelph’;
```

```
data_st{1,1} = [10 11; 12 15; 9 7];
data_st{2,1} = [12 13];
data_st{3,1} = [8 9; 11 12];

data_st{1,2} = 3;
data_st{2,2} = 1;
data_st{3,2} = 2;

data_st{1,3} = ‘Kitchener’;
data_st{2,3} = ‘Waterloo’;
data_st{3,3} = ‘Guelph’;
```

`find([data_st{:,2}] == 2)`	3
`find(strcmp(data_st(:,3), 'Waterloo'))`	2

Other benefit?

Structure Variables

- Structures store values of different types, in fields
- Fields are given names; they are referred to as
- **structurename.fieldname** using the dot operator
- Structure variables can be initialized using the struct function, which takes pairs of arguments (field name as a string followed by the value for that field)
- To print, `disp` will display all fields; `fprintf` can only print individual fields

Example: Containing Multiple Data Types

```
temp_st1 = [10 11; 12 15; 9 7];
temp_st2 = [12 13];
temp_st3 = [8 9; 11 12];

num_st1 = 3;
num_st2 = 1;
num_st3 = 2;

name_st1 = ‘Kitchener’;
name_st2 = ‘Waterloo’;
name_st3 = ‘Guelph’;
```

```
temp_st{1} = [10 11; 12 15; 9 7];
temp_st{2} = [12 13];
temp_st{3} = [8 9; 11 12];

num_st = [3 1 2];

name_st{1} = ‘Kitchener’;
name_st{2} = ‘Waterloo’;
name_st{3} = ‘Guelph’;

data_st.temp_st = temp_st;
data_st.num_st = num_st;
data_st.name_st = name_st;
```

Cell Arrays vs Structures

- Cell arrays are arrays, so they are indexed
  - That means that you can loop though the elements in a cell array – or have MATLAB do that for you by using a vectorized code
- Structs are not indexed, so you cannot loop
  - However, the field names are mnemonic so it is more clear what is being stored in a struct
- For example:
  - variable{1} vs. variable.weight: which is more mnemonic?

Cell Arrays vs Structures

- Cell arrays are arrays, so they are indexed
  - That means that you can loop though the elements in a cell array – or have MATLAB do that for you by using vectorized code
- Structs are not indexed, so you cannot loop
  - However, the field names are mnemonic so it is more clear what is being stored in a struct
- For example:
  - variable{1} vs. variable.weight: which is more mnemonic?

Numerical Technique

- Algorithms that are used to obtain numerical solutions of a mathematical problem
- It is useful when no analytical solution exists or analytical solution is difficult to obtain.

$$f(x) = 2x + 3$$

$$f'(3)$$

$$f(x) = \log(x) * 2x + e^{3x} * x^{2/3}$$

$$f'(3)$$

Theory: Differentiation

The derivative is a measure of the rate at which a function is changing

The derivative of $f(x)$ at $x = a$ can be defined using limits as:

$$f'(a) = \lim_{x \to a} \frac{f(x) - f(a)}{x - a}$$

With a small change in notation, we can write that:

$$f'(a) = \lim_{h \to 0} \frac{f(a+h) - f(a)}{h}$$

For one point

Instead of finding a derivate at every point in a function, we can find the derivative for a function $f(x)$.

$$f'(x) = \lim_{h \to 0} \frac{f(x+h) - f(x)}{h} \text{ can be written as } f'(x) = \frac{d}{dx} f(x) = \frac{dy}{dx}$$

For a function

Theory: Tangent Lines

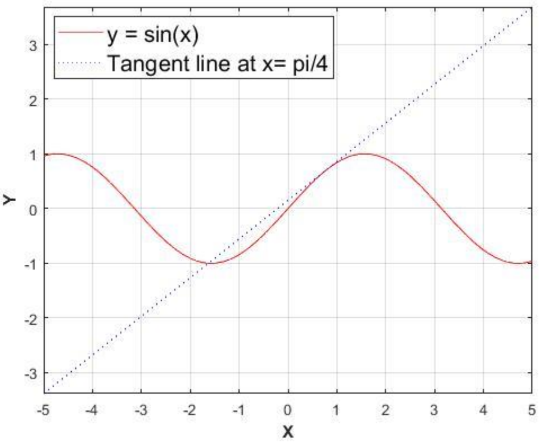
We can find the equation of a tangent line to $f(x)$ at point (a, $f(x)$).
Recall the general equation of a line $y = mx + b$
where m is the slope and b is the y-intercept. The slope of the tangent line found by $m = f'(a)$

Given two points on a line, the slope can be found using:

$$Slope = \frac{y_2 - y_1}{x_2 - x_1}$$

From there, the equation of a line can be found using:

$$y - y_1 = m(x - x_1)$$



Theory: Differentiation 1 – Symbolic Way

Example: Find the derivative of:

$$f(x) = \sqrt{x^2 + 1}$$

- $f(x) = \sqrt{u} \rightarrow u = x^2 + 1$
- $f'(x) = \frac{1}{2} u^{\frac{1}{2}} * (2x)$
- $f'(x) = \frac{2x}{2\sqrt{u}} = \frac{x}{\sqrt{x^2+1}}$

```
syms x
fx = sqrt(x^2 + 1);
fxp = diff(fx);

fxp
fxp_2 = subs(fxp, x, 2)
double(fxp_2)
```

```
fxp =
```

$$\frac{x}{\sqrt{x^2+1}}$$

```
fxp_2 =
```

$$\frac{2\sqrt{5}}{5}$$

```
ans = 0.8944
```

$$f'(x) = \lim_{h \to 0} \frac{f(x+h) - f(x)}{h}$$

Theory: Differentiation 1 – Numerical Way

Example: Find the derivative of:

$$f(x) = \sqrt{x^2 + 1}$$

- $f(x) = \sqrt{u} \rightarrow u = x^2 + 1$
- $f'(x) = \frac{1}{2} u^{\frac{1}{2}} * (2x)$
- $f'(x) = \frac{2x}{2\sqrt{u}} = \frac{x}{\sqrt{x^2+1}}$

```
h = 0.000000001;
x = 2;
(diff_ex1(x+h) - diff_ex1(x))/h
```

```
ans = 0.8944
```

```
function fx = diff_ex1(x)

fx = sqrt(x^2 + 1);

end
```

$$f'(x) = \lim_{h \to 0} \frac{f(x+h) - f(x)}{h}$$

Theory: L'Hôpital's Rule

L'Hôpital's Rule is a way of solving certain limits of an **indeterminant** form.

In order to apply L'Hôpitals rule the limit must be:

1. A ratio, like $\frac{f(x)}{g(x)}$
2. Indeterminate

So, L'Hôpitals rule can be applied to limits of the form $\frac{0}{0}$ or $\frac{\infty}{\infty}$.

If $f(x)$ and $g(x)$ are different functions and if $\lim_{x \to a} \frac{f(x)}{g(x)}$ is indeterminate, then

$$\lim_{x \to a} \frac{f(x)}{g(x)} = \lim_{x \to a} \frac{f'(x)}{g'(x)}$$

Example: L’Hôpital’s Rule

Example: Find the following using L’Hôpital’s rule

1. $\lim\limits_{x \to 1} \frac{\ln(x)}{x-1} \Rightarrow \frac{0}{0}$

Using L’Hôpital’s rule $= \lim\limits_{x \to 1} \frac{\frac{1}{x}}{1} = 1$

2. $\lim\limits_{x \to \infty} \frac{e^x}{x^2} \Rightarrow \frac{0}{0}$

Using L’Hôpitlals rule $= \lim\limits_{x \to \infty} \frac{e^x}{2x} = \lim\limits_{x \to \infty} \frac{e^x}{2} = \infty$

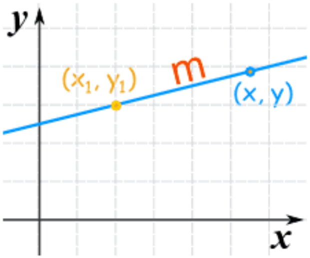
Example: L'Hopitals Rule (Script)

```
tol = 10^-4;
x = 10;
val1 = sin(x)/x

x = 10;
if abs(x)<tol
    x = tol;
end
val2 = sin(x)/x

x = 0;
if abs(x)<tol
    x = tol;
end
val3 = sin(x)/x

x = 0;
val4 = cos(x)
```

```
val1 = -0.0544

val2 = -0.0544

val3 = 1.0000

val4 = 1
```

Theory: Point-Slope Equation of a Line

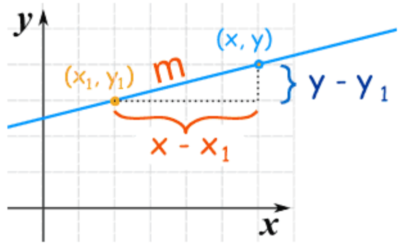
$$y - y_1 = m(x - x_1)$$



(x_1, y_1) is a **known** point

m is the **slope** of the line

(x, y) is any other point on the line



[Slope](https://www.mathsisfun.com/algebra/line-equation-point-slope.html) $m = \dfrac{\text{change in y}}{\text{change in x}} = \dfrac{y - y_1}{x - x_1}$

Starting with the slope: $\dfrac{y - y_1}{x - x_1} = m$

we rearrange it like this: $\dfrac{y - y_1}{\boxed{x - x_1}} = m(x - x_1)$

to get this: $y - y_1 = m(x - x_1)$

https://www.mathsisfun.com/algebra/line-equation-point-slope.html

Theory: Newton's Method

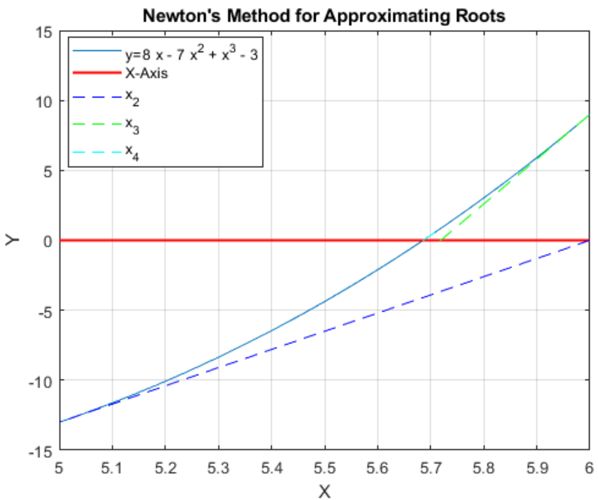
Newton's Method is a way successively finding better and better approximations to the roots of a function.
The slope of tangent line L is $f'(x)$so its equation is:

$$y - f(x_1) = f'(x_1)(x - x_1)$$

To find the roots, we need to find the x-intercepts where $y = 0$,
so we assume x_L is where $y = 0$,
$0 - f(x_1) = f'(x_1)(x_2 - x_1)$

$$x_2 = x_1 - \frac{f(x_1)}{f'(x_1)}$$

We call x_2 the second approximation to r, but what if we want x_2 to be even more accurate?



Initial Estimation: x = 5

This process can be repeated for x_1, x_2, x_3…
In general, if the n^{th} approximation is x_n and $f'(x_n) \neq 0$, then the next approximation x_{n+1} is:

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}$$

Example: Newton's Method 1

Example: Use Newton's Method to find a root:

$$f(x) = x^6 - 2$$

$$f'(x) = 6x^5$$

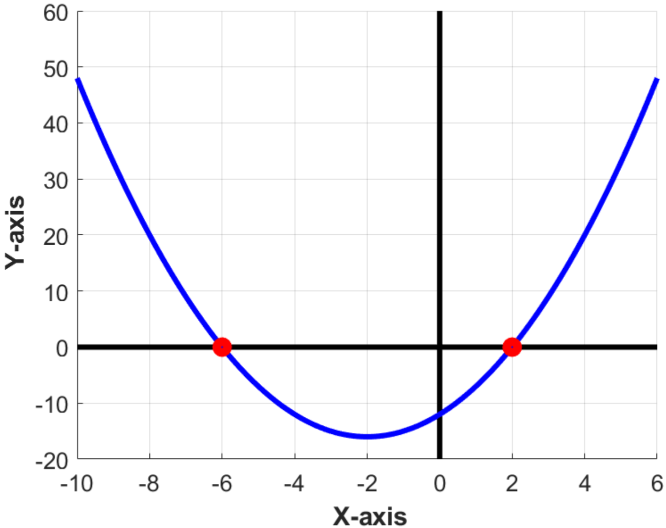
We can apply Newton's method to solve for the root $y = 0$. $x_1 = 1$ (Initial Guess)

- $x_2 = 1 - \frac{f(1)}{f'(1)} = 1 - \frac{1^6 - 2}{6(1^5)} \approx 1.16666667$
- $x_3 = x_2 - \frac{f(x_2)}{f'(x_2)} \approx 1.12644368$
- $x_4 = x_3 - \frac{f(x_3)}{f'(x_3)} \approx 1.12249707$
- $x_5 = x_4 - \frac{f(x_4)}{f'(x_4)} \approx 1.12246205$
- $x_6 = x_5 - \frac{f(x_5)}{f'(x_5)} \approx 1.12246205$

$$\boxed{x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}}$$

Example: Root Finding 1

$$f(x) = (x-2)(x-6)$$
$$= x^2 - 4x - 12$$




```
% plot a graph
x = -10:0.01:6;
y = (x-2) .* (x + 6);

figure(1);
line([0 0 ], [-20 60], 'color', ...
    'k', 'Linewidth', 3); hold on; % y-axis
line([min(x) max(x)], [0 0 ], 'color', ...
    'k', 'Linewidth', 3); % x-axis
plot(x, y, 'b', 'LineWidth',3); % graph
plot(2, 0, 'or', 'LineWidth',5);
plot(-6, 0, 'or', 'LineWidth',5); hold off
xlabel('\bf X-axis')
ylabel('\bf Y-axis')
xticks(-10:2:6);
xticklabels({'-10', '-8', '-6', '-4', ...
    '-2','0','2', '4','6'})
set(gca, 'fontsize', 13)
xlim([-10 6])
grid on;
```

Example: Root Finding 1 (Simulation)

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}$$

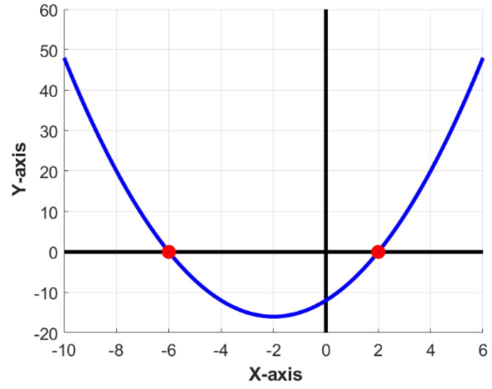
```
x1 = 5;
x2 = x1 - myfun(x1)/myfunp(x1)
x3 = x2 - myfun(x2)/myfunp(x2)
x4 = x3 - myfun(x3)/myfunp(x3)
x5 = x4 - myfun(x4)/myfunp(x4)
```

```
x2 = 2.6429
x3 = 2.0445
x4 = 2.0002
x5 = 2.0000
```

```
function fx = myfun(x)

fx = (x-2) .* (x + 6);

end

function fxp = myfunp(x)

fxp = 2*x + 4;

end
```

$$f(x) = x^2 - 4x - 12$$

$$f'(x) = 2x - 4$$



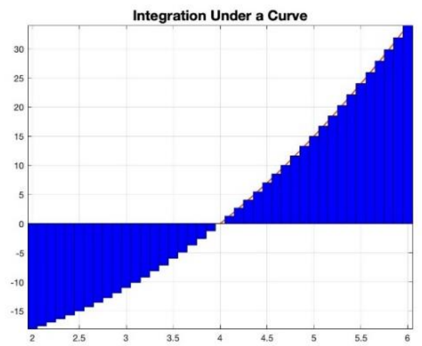
Theory: Integral

Integration is the technique of determining the **area under a curve**. This process is **opposite the process of differentiation**.

To find the area under a curve, we divide the curve into many equal segments of equal width. Each rectangle is multiplied by it's corresponding y-value to get the area of that rectangle. The rectangle's areas are summed for the total area under that curve segment:

- Left endpoints can be used so that the rectangular segments give an underestimation
- Right endpoints can be used so that the rectangular segments give an overestimation
- Center endpoints can be used to try and balance the error from overestimations and underestimations

How can you make your estimation even more accurate?
Take smaller segments!



Theory: Integral

The Definite Integral

If f is defined for $a \leq x \leq b$, we divide the interval [a,b] into n segments of equal width $\Delta x = \frac{b-a}{n}$.

We let $x_o = a, x_1, x_2, \dots, x_n = b$ be the endpoints of these segments and let $x_1^*, x_2^*, x_3^*, \dots, x_n^*$ be sample points in these segments such that x_i^* lies in the i^{th} segment. This means that the definite integral from a to b is:

$$\boxed{\int_a^b f(x)dx = \lim_{n\to\infty} \sum_{i=1}^{n} f(x_i^*)\Delta x}$$

Recall that,

$$\sum_{i=1}^{n} i = \frac{n(n+1)}{2} \text{ and } \sum_{i=1}^{n} i^2 = \frac{n(n+1)(2n+1)}{4} \quad and \sum_{i=1}^{n} i^3 = \frac{n^2(n+1)^2}{4}$$

Example: Definite Integral

Solve the following definite integral using the definition of the definite integral:

$$\int_0^3 (x^3 - 6x)\,dx$$

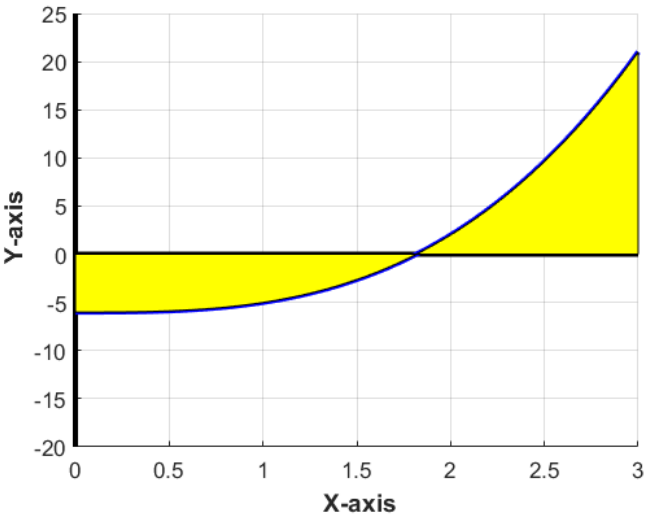
$$\int_0^3 (x^3 - 6x)\,dx = \lim_{n\to\infty} \sum_{i=1}^{n} f(x_i)\Delta x$$

$$\Delta x = \frac{3}{n}, x_i = \frac{3i}{n}$$

- $= \lim_{n\to\infty} \sum_{i=1}^{n} f\left(\frac{3i}{n}\right)\frac{3}{n}$
- $= \lim_{n\to\infty} \frac{3}{n}\sum_{i=1}^{n} \left[\left(\frac{3i}{n}\right)^3 - 6\left(\frac{3i}{n}\right)\right]$
- $= \lim_{n\to\infty} \frac{3}{n}\sum_{i=1}^{n} \left[\frac{27i^3}{n^3} - \frac{18i}{n}\right]$
- $= \lim_{n\to\infty} \left[\left(\frac{81}{n^4}\right)\sum_{i=1}^{n} i^3 - \frac{54}{n^2}\sum_{i=1}^{n} i\right]$
- $= \lim_{n\to\infty} \left[\left(\frac{81}{n^4}\right)\left(\frac{n(n+1)}{2}\right)^2 - \frac{54}{n^2}\left(\frac{n(n+1)}{2}\right)\right]$
- $= \lim_{n\to\infty} \left[\left(\frac{81}{n^4}\right)\left(\frac{n^2+n}{2}\right)^2 - \frac{54}{n^2}\left(\frac{n^2+n}{2}\right)\right]$
- $= \lim_{n\to\infty} \left[\left(\frac{81}{n^4}\right)\left(\frac{n^4+2n^3+n^2}{4}\right) - \frac{54}{n^2}\left(\frac{n^2+n}{2}\right)\right]$
- $= \frac{81}{4} - \frac{54}{2} = \frac{-27}{4}$

Example: Integral (graph)

$$f(x) = x^3 - 6x$$




```
% plot a graph
x = 0:0.01:3;
y = x.^3 - 6;

figure(1);
line([0 0 ], [-20 25], 'color', ...
    'k', 'Linewidth', 3); hold on; % y-axis
line([min(x) max(x)], [0 0 ], 'color', ...
    'k', 'Linewidth', 3); % x-axis
plot(x, y, 'b', 'LineWidth',3); % graph
area(x, y, 'FaceColor','y'); % filled area
xlabel('\bf X-axis')
ylabel('\bf Y-axis')
xticks(0:0.5:3);
xticklabels({'0','0.5','1','1.5', '2', '2.5', '3'})
set(gca, 'fontsize', 13)
xlim([0 3]);
ylim([-20 25])
grid on;
```

Example: Integral (Symbolic)

```
syms x
y = x^3 -6*x;
int_y_ab = int(y, 0, 3)
double(int_y_ab)
```

```
int_y_ab =
```

$$-\frac{27}{4}$$

```
ans = -6.7500
```

Example: Integral (Numeric 1)

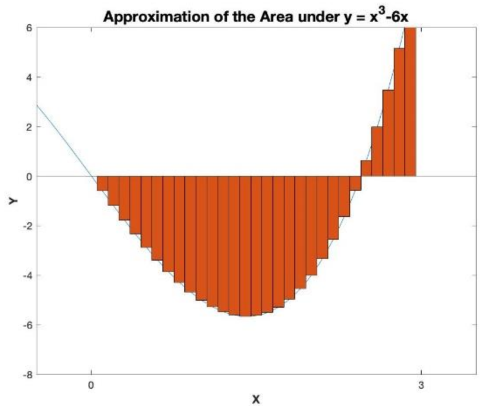
```
n = 10000;
a = 0;
b = 3;
del_x = (b-a)/n;

area_fx = 0;
for ii=1:n
    x_star = a + del_x*ii;
    area_fx = area_fx + myfun(x_star)*del_x;
end
area_fx

error_est = area_fx - (-27/4)
```

```
function fx = myfun(x)

fx = x^3 - 6*x;

end
```

$$f(x) = x^3 - 6x$$

```
area_fx = -6.7486

error_est = 0.0014
```

$$\int_a^b f(x)dx = \lim_{n\to\infty} \sum_{i=1}^{n} f(x_i^*)\Delta x$$

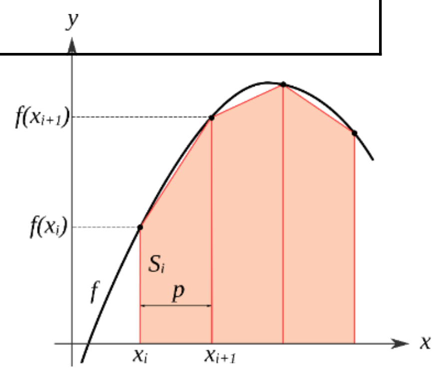


Example: Integral (Numeric 2)

```
n = 10000;
a = 0;
b = 3;
del_x = (b-a)/n;

area_fx = 0;
for ii=1:n-1
    x_star1 = a + del_x*ii;
    x_star2 = a + del_x*(ii+1);
    area_fx = area_fx + (myfun(x_star2)+myfun(x_star1))/2*del_x;
end
area_fx

error_est = area_fx - (-27/4)
```

```
function fx = myfun(x)

fx = x^3 - 6*x;

end
```

```
area_fx = -6.7500


error_est = 4.7250e-07
```



Example: Integral (Numeric 1 vs Numeric 2)

```
n = 10000;
a = 0;
b = 3;
del_x = (b-a)/n;

area_fx1 = 0;
for ii=1:n
    x_star = a + del_x*ii;
    area_fx1 = area_fx1 + myfun(x_star)*del_x;
end

area_fx2 = 0;
for ii=1:n-1
    x_star1 = a + del_x*ii;
    x_star2 = a + del_x*(ii+1);
    area_fx2 = area_fx2 + (myfun(x_star2)+myfun(x_star1))/2*del_x;
end

area_fx1
area_fx2

error_est1 = area_fx1 - (-27/4)
error_est2 = area_fx2 - (-27/4)

```

```
function fx = myfun(x)

fx = x^3 - 6*x;

end
```

```
area_fx1 = -6.7486
area_fx2 = -6.7500

error_est1 = 0.0014
error_est2 = 4.7250e-07
```