Built-in Functions

Chul Min Yeum
Assistant Professor
Civil and Environmental Engineering
University of Waterloo, Canada

AE121: Computational Method

UNIVERSITY OF WATERLOO
FACULTY OF ENGINEERING

Last updated: 2019-04-30

Common Math Functions

`abs(x)`	Finds the absolute value of **x**.	`abs(-3)` `ans = 3`
`sqrt(x)`	Finds the square root of **x**.	`sqrt(85)` `ans = 9.2195`
`sign(x)`	Return – is 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(-8)` `ans = -1`
`rem(x,y)`	Computes the remainder of **x/y**.	`rem(25,4)` `ans = 1`
`exp(x)`	Computes the value of e^x, where e is the base for natural logarithms, approximately 2.7183.	`exp(10)` `ans = 2.2026e+004`
`log(x)`	Computes ln(**x**), the natural logarithm of **x** (to the base e).	`log(10)` `ans = 2.3026`
`log10(x)`	Computes $\log_{10}$(**x**), the common logarithm of **x** (to the base 10).	`log10(10)` `ans = 1`

Example: Common Math Functions

```
val_pos = 3;
val_neg = -5;

val1 = abs(val_pos);
val2 = sign(val_pos) * val_pos; % same result

sign_val1 = -(val_pos<0) + (val_pos>0);
val3 = sign_val1 * val_pos;

val4 = abs(val_neg);
val5 = sign(val_neg) * val_neg;

sign_val2 = -(val_neg<0) + (val_neg>0);
val6 = sign_val2 * val_neg;
```

```
val1 = 3
val2 = 3
val3 = 3
val4 = 5
val5 = 5
val6 = 5
```

```
s1 = exp(1);
s2 = log(exp(1));
s3 = log(1);
s4 = log10(10^3);
```

```
s1 = 2.7183
s2 = 1
s3 = 0
s4 = 3
```

```
val_pos = 10;
val_neg = -10;
b = 4;

rem1 = rem(val_pos, b);
rem2 = val_pos - b*fix(val_pos/b); % doc rem
rem3 = rem(val_neg, b);
rem4 = val_neg - b*fix(val_neg/b);
```

```
rem1 = 2
rem2 = 2
rem3 = -2
rem4 = -2
```

Rounding Functions

`round(x)`	Rounds **x** to the nearest integer.	`round(8.6)` `ans = 9`
`round(x,N)`	Rounds **x** to a specified decimal digit	`round(8.6436, 3)` `ans = 8.644`

`fix(x)`	Truncates **x** to the nearest integer toward zero. Notice that 8.6 truncates to 8, not 9, with this function.	`fix(8.6)`
		`ans = 8`
		`fix(-8.6)`
		`ans = -8`

`floor(x)`	Rounds **x** to the nearest integer toward negative infinity.	`floor(-8.6)` `ans = -9`
`ceil(x)`	Rounds **x** to the nearest integer toward positive infinity.	`ceil(-8.6)` `ans = -8`

Example: Rounding Functions

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

```
fprintf('round(0.3): %d \n', round(0.3));
fprintf('ceil(0.3): %d \n', ceil(0.3));
fprintf('fix(0.3): %d \n', fix(0.3));
fprintf('floor(0.3): %d \n', floor(0.3));

fprintf('round(-0.3): %d \n', round(-0.3));
fprintf('ceil(-0.3): %d \n', ceil(-0.3));
fprintf('fix(-0.3): %d \n', fix(-0.3));
fprintf('floor(-0.3): %d \n', floor(-0.3));
```

```
fprintf('round(0.6): %d \n', round(0.6));
fprintf('ceil(0.6): %d \n', ceil(0.6));
fprintf('fix(0.6): %d \n', fix(0.6));
fprintf('floor(0.6): %d \n', floor(0.6));

fprintf('round(-0.6): %d \n', round(-0.6));
fprintf('ceil(-0.6): %d \n', ceil(-0.6));
fprintf('fix(-0.6): %d \n', fix(-0.6));
fprintf('floor(-0.6): %d \n', floor(-0.6));
```

Function Used in Discrete Mathematics

`factor(x)`	Finds the prime factors of **x**.	`factor(12)` `ans = 2 2 3`
`gcd(x,y)`	Finds the greatest common denominator of **x** and **y**.	`gcd(10,15)` `ans = 5`

`lcm(x,y)`	Finds the least common multiple of **x** and **y**.	`lcm(2,5)`
		`ans = 10`
		`lcm(2,10)`
		`ans = 10`

`factorial(x)`	Finds the value of **x** factorial (**x**!). A factorial is the product of all the integers less than **x**. For example, $6! = 6 \times 5 \times 4 \times 3 \times 2 \times 1 = 720$	`factorial(6)` `ans = 720`
`nchoosek(n,k)`	Finds the number of possible combinations of *k* items from a group of *n* items. For example, use this function to determine the number of possible subgroups of 3 chosen from a group of 10.	`nchoosek(10,3)` `ans = 120`

Example: factor(x), gcd(x,y), and lcm(x,y)

```
val1 = 120;
f = factor(val1);
val2 = prod(f);

val1_val2_equal = (val1==val2);

n_f = numel(f);
idx = randperm(n_f, 2); % doc randperm
% comment: please check the difference between randi(n_f,1,2) and randperm(n_f,2)

p_num1 = f(idx(1));
p_num2 = f(idx(2));
val3 = lcm(p_num1, p_num2);
val4 = p_num1*(p_num1 == p_num2) + (p_num1*p_num2)*(p_num1 ~= p_num2);
% comment: we will learn a selection statement.
val3_val4_equal = (val3==val4);

val5 = gcd(p_num1, p_num2);
val6 = p_num1*(p_num1 == p_num2) + (p_num1 ~= p_num2);
val5_val6_equal = (val5==val6);
```

```
val1_val2_equal = logical
   1
val3_val4_equal = logical
   1
val5_val6_equal = logical
   1
```

Example: factorial(x) and nchoosek(n,k)

```
x = 6;
val1 = factorial(x);
val2 = prod(1:x);
tmp = cumprod(1:x); % doc cumprod
val3 = tmp(end);

val1
val2
val3

n = 5;
k = 2;
val4 = nchoosek(n,k);
```

```
val1 = 720
val2 = 720
val3 = 720
```

nchoosek(n,k) = $\frac{n!}{(n-k)!k!}$

```
val5 = factorial(n)/factorial(n-k)/factorial(k);
val4_val5_equal = (val4==val5);
val4_val5_equal

pair_id = nchoosek(1:n,k); % all pair
pair_id % take a look at the result
```

```
val4_val5_equal = logical
   1

pair_id = 10×2
     1     2
     1     3
     1     4
     1     5
     2     3
     2     4
     2     5
     3     4
     3     5
     4     5
```

Some of the Available Trigonometric Functions

`deg2rad`	Converts degrees to radians.	`deg2rad(90)` `ans = 1.5708`
`rad2deg`	Converts radians to degrees.	`rad2deg(pi)` `ans = 180`
`sin(x)`	Finds the sine of **x** when **x** is expressed in radians.	`sin(0)` `ans = 0`
`cos(x)`	Finds the cosine of **x** when **x** is expressed in radians.	`cos(pi)` `ans = -1`
`tan(x)`	Finds the tangent of **x** when **x** is expressed in radians.	`tan(pi)` `ans = -1.2246e-016`
`asin(x)`	Finds the arcsine, or inverse sine, of **x**, where **x** must be between -1 and 1. The function returns an angle in radians between $\pi/2$ and $-\pi/2$.	`asin(-1)` `ans = -1.5708`
`sind(x)`	Finds the sin of **x** when **x** is expressed in degrees.	`sind(90)` `ans = 1`
`asind(x)`	Finds the inverse sin of **x** and reports the result in degrees.	`asind(1)` `ans = 90`

Example: Trigonometric Functions

```
% You always check if your input angle is expressed in radians or degrees.

ang_rad = pi/6;
ang_deg = rad2deg(ang_rad);

val1 = sin(ang_rad); % sine (radian)
val2 = sind(ang_deg); % sine (degree)
val3 = cos(ang_rad); % cosine (radian)
val4 = cosd(ang_deg) % cosine (degree)
val5 = tan(ang_rad); % tagent (radian)

val6 = val1/val3; % relation for tangent
val7 = val1^2  + val3^2; % relation for cosine and sine
```

```
val4 =
   0.866025403784439
val1 =
   0.500000000000000
val2 =
   0.500000000000000
val3 =
   0.866025403784439
val4 =
   0.866025403784439
val5 =
   0.577350269189626
val6 =
   0.577350269189626
val7 =
     1
```

```
ang_deg = rad2deg(pi/6);
val1 = sind(ang_deg); % sine (degree)
val2 = asind(val1); % inverse sine
val3 = cosd(ang_deg); % cosine (degree)
val4 = acosd(val3); % inverse cosine
```

```
ang_deg =
  29.999999999999996
val1 =
   0.500000000000000
val2 =
  29.999999999999996
val3 =
   0.866025403784439
val4 =
  29.999999999999993
```

Why isn't 30 degrees?

Maxima and Minima

`max(x)`

Finds the largest value in a **vector x**. For example, if $x = [1\ 5\ 3]$, the maximum value is 5.

```
x = [1, 5, 3];
max(x)
ans = 5
```

Creates a row vector containing the maximum element from each column "of a **matrix x**. For example, if $x = \begin{bmatrix} 1 & 5 & 3 \\ 2 & 4 & 6 \end{bmatrix}$, then the maximum value in column 1 is 2, the maximum value in column 2 is 5, and the maximum value in column 3 is 6.

```
x = [1, 5, 3; 2, 4, 6];
max(x)
ans = 2    5    6
```

`[a,b] = max(x)`

Finds both the largest value in a **vector x** and its location in vector **x**. For $x = [1\ \ 5\ \ 3]$ the maximum value is named **a** and is equal to 5. The location of the maximum value is element 2 and is named **b**.

```
x = [1, 5, 3];
[a,b] = max(x)
a = 5
b = 2
```

Creates a row vector containing the maximum element from each column of a matrix **x** and returns a row vector with the location of the maximum in each column of matrix **x**. For example, if $= \begin{bmatrix} 1 & 5 & 3 \\ 2 & 4 & 6 \end{bmatrix}$, then the maximum value in column 1 is 2, the maximum value in column 2 is 5, and the maximum value in column 3 is 6. These maxima occur in row 2, row 1, and row 2, respectively.

```
x = [1, 5, 3; 2, 4, 6];
[a,b] = max(x)
a = 2    5    6
b = 2    1    2
```

Maxima and Minima (Continue)

`max(x,y)`

Creates a matrix the same size as **x** and **y**. (Both **x** and **y** must have the same number of rows and columns.) Each element in the resulting matrix contains the maximum value from the corresponding positions in **x** and **y**. For example,

if $x = \begin{bmatrix} 1 & 5 & 3 \\ 2 & 4 & 6 \end{bmatrix}$ and $y = \begin{bmatrix} 10 & 2 & 4 \\ 1 & 8 & 7 \end{bmatrix}$, then the resulting matrix will be $x = \begin{bmatrix} 10 & 5 & 4 \\ 2 & 8 & 7 \end{bmatrix}$.

`min(x)`

The syntax for the `min` function is the same as that for the `max` function, except that minimums are returned.

```
x = [1, 5, 3; 2, 4, 6];
y = [10,2,4; 1, 8, 7];
max(x,y)
ans = 10 5 4
      2 8 7
```

Example: Maxima and Minima

```
% doc max, doc min
% Here are only codes related to 'max'
% The syntax for 'min' is the same as that for the 'max'
vec1 = [1:22 24 23];

val1 = max(vec1);
[val2, val2_loc] = max(vec1);
val3 = max(10,20); % compare only 10 and 20
val4 = max(1:10, 8); % compare 1:10 to 8

mat1 = reshape(vec1, 4, 6);
[val5, loc_val5] = max(mat1); % find the max value from each column (1)
[val6, loc_val6] = max(mat1, [], 1); % find the max value from each column (1)
[val7, loc_val7] = max(mat1, [], 2); % find the max value from each row (2)
val8 = max(mat1, [], 'all'); % not support two argouts
[val9, loc_val9] = max(mat1(:));
```

```
val1 = 24
val2 = 24
val2_loc = 23
val3 = 20
val4 = 1×10
     8     8     8     8     8     8     8     8     9    10
```

```
mat1 = 4×6
     1     5     9    13    17    21
     2     6    10    14    18    22
     3     7    11    15    19    24
     4     8    12    16    20    23
```

```
val5 = 1×6
     4     8    12    16    20    24

loc_val5 = 1×6
     4     4     4     4     4     3

val6 = 1×6
     4     8    12    16    20    24

loc_val6 = 1×6
     4     4     4     4     4     3

val7 = 4×1
    21
    22
    24
    23

loc_val7 = 4×1
     6
     6
     6
     6

val8 = 24
val9 = 24

loc_val9 = 23
```

Sums and Products

`sum(x)`

Sums the elements in **vector x**. For example, if $x = [1\ 5\ 3]$, the sum is 9.

```
x = [1, 5, 3];
sum(x)
ans = 9
```

Computes a row vector containing the sum of the elements in each column of a **matrix x**. For example, if $x = \begin{bmatrix} 1 & 5 & 3 \\ 2 & 4 & 6 \end{bmatrix}$, then the sum of column 1 is 3, the sum of column 2 is 9, and the sum of column 3 is 9.

```
x = [1, 5, 3; 2, 4, 6];
sum(x)
ans = 3 9 9
```

`prod(x)`

Computes the product of the elements of a **vector x**. For example, if $x = [1\ 5\ 3]$, the product is 15.

```
x = [1, 5, 3];
prod(x)
ans = 15
```

Computes a row vector containing the product of the elements in each column of a **matrix x**. For example, if $x = \begin{bmatrix} 1 & 5 & 3 \\ 2 & 4 & 6 \end{bmatrix}$, then the product of column 1 is 2, the product of column 2 is 20, and the product of column 3 is 18.

```
x = [1, 5, 3; 2, 4, 6];
prod(x)
ans = 2 20 18
```

Example: Sum and Product

```
vec1= 1:6;
val1= vec1(1) + vec1(2) + vec1(3) ...
    + vec1(4) + vec1(5) + vec1(6);
val2= sum(vec1);

val3= vec1(1) * vec1(2) * vec1(3)...
    * vec1(4) * vec1(5) + vec1(6);
val4 = prod(vec1);

mat1 = reshape(vec1, 2, 3);
val5 = sum(mat1);
val6 = sum(mat1,1);
val7 = sum(mat1,2);
val8 = sum(mat1,'all');
val9 = sum(mat1(:));
% This syntax is similar to that for 'max' and 'min'.
% The syntax for 'prod' is the same as that for the 'sum'
```

```
vec1 = 1×6
     1     2     3     4     5     6

mat1 = 2×3
     1     3     5
     2     4     6
```

```
val1 = 21
val2 = 21
val3 = 126
val4 = 720
val5 = 1×3
     3     7    11

val6 = 1×3
     3     7    11

val7 = 2×1
     9
    12

val8 = 21
val9 = 21
```

Averages

mean(x)

Computes the mean value (or average value) of a **vector x**. For example, if $x = [1\ 5\ 3]$, the mean value is 3.

```
x = [1, 5, 3];
mean(x)
ans = 3.0000
```

Returns a row vector containing the mean value from each column of a **matrix x**. For example, if $x = \begin{bmatrix} 1 & 5 & 3 \\ 2 & 4 & 6 \end{bmatrix}$, then the mean value of column 1 is 1.5, the mean value of column 2 is 4.5, and the mean value of column 3 is 4.5.

```
x = [1, 5, 3; 2, 4, 6];
mean(x)
ans = 1.5   4.5   4.5
```

median(x)

Finds the median of the elements of a **vector x**. For example, if $x = [1\ 5\ 3]$, the median value is 3.

```
x = [1, 5, 3];
median(x)
ans = 3
```

Returns a row vector containing the median value from each column of a **matrix x**. For example, if $x = \begin{bmatrix} 1 & 5 & 3 \\ 2 & 4 & 6 \\ 3 & 8 & 4 \end{bmatrix}$, then the median value from column 1 is 2, the median value from column 2 is 5, and the median value from column 3 is 4.

```
x = [1, 5, 3;
2, 4, 6;
3, 8, 4];
median(x)
ans = 2   5   4
```

mode(x)

Finds the value that occurs most often in an array. Thus, for the array $\underline{x} = [1, 2, 3, 3]$ the mode is 3.

```
x = [1, 2, 3, 3]
mode(x)
ans = 3
```

Example: Averages

```
rng(200)
vec1 = randi(5,1,20)
mat1 = reshape(vec1, 4, 5)

% mean
val11 = mean(vec1)
val12 = mean(mat1)
val13 = mean(mat1,1)
val14 = mean(mat1,2)
val15 = mean(mat1,'all')
val16 = mean(mat1(:))

% median
val21 = median(vec1)
val22 = median(mat1)
val23 = median(mat1,1)
val24 = median(mat1,2)
val25 = median(mat1,'all')
val26 = median(mat1(:))

% mode
val31 = mode(vec1)
val32 = mode(mat1)
val33 = mode(mat1,1)
val34 = mode(mat1,2)
val35 = mode(mat1,'all')
val36 = mode(mat1(:))
```

```
vec1 = 1×20
     5     2     3     3     4     1     2     5     3     5     5     5     5     2     5     1     4     2     1     5

mat1 = 4×5
     5     4     3     5     4
     2     1     5     2     2
     3     2     5     5     1
     3     5     5     1     5
```

```
val31 = 5
val32 = 1×5
     3     1     5     5     1

val33 = 1×5
     3     1     5     5     1

val34 = 4×1
     4
     2
     5
     5

val35 = 5
val36 = 5
```

```
val11 = 3.4000
val12 = 1×5
    3.2500    3.0000    4.5000    3.2500    3.0000

val13 = 1×5
    3.2500    3.0000    4.5000    3.2500    3.0000

val14 = 4×1
    4.2000
    2.4000
    3.2000
    3.8000

val15 = 3.4000
val16 = 3.4000
```

```
val21 = 3.5000
val22 = 1×5
    3.0000    3.0000    5.0000    3.5000    3.0000

val23 = 1×5
    3.0000    3.0000    5.0000    3.5000    3.0000

val24 = 4×1
     4
     2
     3
     5

val25 = 3.5000
val26 = 3.5000
```

similar structure

Sorting Functions

`sort(x)`

Sorts the elements of a vector **x** in ascending order. For example, if $x = [1 \quad 5 \quad 3]$, the resulting vector is $x = [1 \quad 3 \quad 5]$.

```
x = [1,5,3];
sort(x)
ans = 1 3 5
```

Sorts the elements in each column of a matrix **x** in ascending order. For example, if $x = \begin{bmatrix} 1 & 5 & 3 \\ 2 & 4 & 6 \end{bmatrix}$, the resulting matrix is

$$x = \begin{bmatrix} 1 & 4 & 3 \\ 2 & 5 & 6 \end{bmatrix}$$

```
x = [1,5,3; 2,4,6];
sort(x)
ans = 1 4 3
      2 5 6
```

`sort(x,'descend')`

Sorts the elements in each column in descending order.

```
x = [1,5,3; 2,4,6];
sort(x,'descend')
ans = 2 5 6
      1 4 3
```

Sorting Functions (Continue)

`sortrows(x)`	Sorts the rows in a matrix in ascending order on the basis of the values in the first column, and keeps each row intact.	`x = [3,1,3; 1,9,3; 4, 3, 6]`
		`sortrows(x)`
		`ans = 1 9 3`
		`3 1 2`
		`4 3 6`

`sortrows(x,n)`	Sorts the rows in a matrix on the basis of the values in column *n*. If *n* is negative, the values are sorted in descending order. If *n* is not specified, the default column used as the basis for sorting is column 1.	`sortrows(x,2)`
		`ans = 3 1 2`
		`4 3 6`
		`1 9 3`

Example: Sort

```
rng(10);

num = 15;
vec1 = 1:num;
rand_idx = randperm(num,num);
vec1 = vec1(rand_idx);

out01 = sort(vec1, 'ascend');
out02 = sort(vec1, 'descend');
out03 = sort(vec1, 'ascend');

out04 = out03(1);
out05 = min(vec1);

out06 = out03(end);
out07 = max(vec1);

out08 = out03(round(num/2));
out09 = median(vec1);

mat1 = reshape(vec1, 3, 5);
out10 = sort(mat1, 'ascend');
out11 = sort(mat1, 'descend');
out12 = sort(mat1, 1, 'ascend');
out13 = sort(mat1, 2, 'ascend');
```

```
vec1 = 1×15
    14    10     6     5     1     4     9    15    12     8     3     2    13    11     7

out01 = 1×15
     1     2     3     4     5     6     7     8     9    10    11    12    13    14    15

out02 = 1×15
    15    14    13    12    11    10     9     8     7     6     5     4     3     2     1

out03 = 1×15
     1     2     3     4     5     6     7     8     9    10    11    12    13    14    15

out04 = 1
out05 = 1
out06 = 15
out07 = 15
out08 = 8
out09 = 8
```

```
mat1 = 3×5
    14     5     9     8    13
    10     1    15     3    11
     6     4    12     2     7
```

```
out10 = 3×5
     6     1     9     2     7
    10     4    12     3    11
    14     5    15     8    13

out11 = 3×5
    14     5    15     8    13
    10     4    12     3    11
     6     1     9     2     7

out12 = 3×5
     6     1     9     2     7
    10     4    12     3    11
    14     5    15     8    13

out13 = 3×5
     5     8     9    13    14
     1     3    10    11    15
     2     4     6     7    12
```

Example: sortrows

```
rng(10);

num = 15;
vec1 = 1:num;
rand_idx = randperm(num,num);
vec1 = vec1(rand_idx);
mat1 = reshape(vec1, 3, 5);

mat2 = sortrows(mat1);

col1 = mat1(:,1);
[~, col1_idx] = sort(col1);
mat3 = mat1(col1_idx,:);

mat4 = sortrows(mat1, 3);

col3 = mat1(:,3);
[~, col3_idx] = sort(col3);
mat5 = mat1(col3_idx,:);
```

```
mat1 = 3×5
    14     5     9     8    13
    10     1    15     3    11
     6     4    12     2     7
```

```
mat2 = 3×5
     6     4    12     2     7
    10     1    15     3    11
    14     5     9     8    13

mat3 = 3×5
     6     4    12     2     7
    10     1    15     3    11
    14     5     9     8    13

mat4 = 3×5
    14     5     9     8    13
     6     4    12     2     7
    10     1    15     3    11

mat5 = 3×5
    14     5     9     8    13
     6     4    12     2     7
    10     1    15     3    11
```

Functions Used with Complex Numbers

`abs(x)`	Computes the absolute value of a complex number, using the Pythagorean theorem. This is equivalent to the radius if the complex number is represented in polar coordinates.	`x = 3 + 4i;`
		`abs(x)`
		`ans =`
		`5`

`angle(x)`	For example, if $x = 3 + 4i$, the absolute value is $\sqrt{3^2 + 4^2} = 5$. Computes the angle from the horizontal in radians when a complex number is represented in polar coordinates.	`x = 3 + 4i;`
		`angle(x)`
		`ans =`
		`0.9273`

`complex(x,y)`	Generates a complex number with a real component x and an imaginary component y.	`x = 3;`
		`y = 4;`
		`complex(x,y)`
		`ans =`
		`3.0000+`
		`4.0000i`

Functions Used with Complex Numbers (Continue)

`real(x)`	Extracts the real component from a complex number.	`x = 3 + 4i;`
		`real(x)`
		`ans =`
		`3`

`imag(x)`	Extracts the imaginary component from a complex number.	`x = 3 + 4i;`
		`imag(x)`
		`ans =`
		`4`

`isreal(x)`	Determines whether the values in an array are real. If they are real, the function returns a 1; if they are complex, it returns a 0.	`x = 3 = 4i;`
		`isreal(x)`
		`ans =`
		`0`

`conj(x)`	Generates the complex conjugate of a complex number.	`x = 3 + 4i;`
		`conj(x)`
		`ans =`
		`3.0000 -`
		`4.0000i`

Example: Complex Numbers

```
val0 = 3 + 4i
val01 = 3 + 4*sqrt(-1)
val02 = complex(3, 4)
val03 = (i == sqrt(-1))

val04 = angle(val0)
val05 = atan(imag(val0)/real(val0))

val06 = isreal(val0)
val07 = isreal(real(val0))
val08 = isreal(imag(val0))
val09 = isreal(imag(val0)*i)

val10 = conj(val0)
val11 = complex(real(val0), -imag(val0))
val12 = sqrt(val0*val10)
val13 = abs(val0)
```

```
val0 = 3.0000 + 4.0000i

val01 = 3.0000 + 4.0000i
val02 = 3.0000 + 4.0000i
val03 = logical
   1
```

```
val04 = 0.9273
val05 = 0.9273
val06 = logical
   0
val07 = logical
   1
val08 = logical
   1
val09 = logical
   0
val10 = 3.0000 - 4.0000i
val11 = 3.0000 - 4.0000i
val12 = 5
val13 = 5
```

Computational Limits

`realmax`	Returns the largest possible floating-point number used in MATLAB®.	`realmax`
		`ans =`
		`1.7977e + 308`

`realmin`	Returns the smallest possible floating-point number used in MATLAB®.	`realmin`
		`ans =`
		`2.2251e − 308`

`intmax`	Returns the largest possible integer number used in MATLAB®.	`intmax`
		`ans =`
		`2147483647`

`intmin`	Returns the smallest possible integer number used in MATLAB®.	`intmin`
		`ans =`
		`−2147483648`

eps : Floating-point relative accuracy

https://www.mathworks.com/matlabcentral/answers/57444-faq-why-is-0-3-0-2-0-1-not-equal-to-zero

Example: Computational Limit

```
val1 = intmax;
val2 = intmax + 1;
val3 = intmin;
val4 = intmin - 1;
val5 = val3*2;
% comment: Similarly, we can compute a value mo

% precision
val6 = 0.3-0.2-0.1;
val7 = 0;
is_equal_val6_val7 = (val6==val7);

val8 = tand(30)-sind(30)/cosd(30);
val9 = 0;
is_equal_val8_val9 = (val8==val9);

tol = eps * 10^4; % small number
is_equal_val6_val7_tol = (abs(val6-val7)<tol);
is_equal_val8_val9_tol = (abs(val8-val9)<tol);
```

Precision!

```
val1 = int32
   2147483647
val2 = int32
   2147483647
val3 = int32
   -2147483648
val4 = int32
   -2147483648
val5 = int32
   -2147483648
val6 =
    -2.775557561562891e-17
val7 =
     0
val8 =
     1.110223024625157e-16
val9 =
     0
```

```
is_equal_val6_val7 = logical
   0
is_equal_val6_val7_tol = logical
   1
is_equal_val8_val9 = logical
   0
is_equal_val8_val9_tol = logical
   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)
mat3 = mat1 + 1;

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
```

```
mat1 = eye(5,5);
ind_mat1 = sub2ind(size(mat1), 1:5, 1:5); % print indM7

mat2 = zeros(size(mat1));
mat2(ind_mat1) = 1;

% ind2sub
[row2, col2] = ind2sub(size(mat2), find(mat2));
% equivalent row and column subscripts corresding to each linear index

[row3, col3] = find(mat2); % doc find

mat1
mat2
[row2 col2]
[row3 col3]
```

```
mat1 = 5×5
     1     0     0     0     0
     0     1     0     0     0
     0     0     1     0     0
     0     0     0     1     0
     0     0     0     0     1

mat2 = 5×5
     1     0     0     0     0
     0     1     0     0     0
     0     0     1     0     0
     0     0     0     1     0
     0     0     0     0     1

ans = 5×2
     1     1
     2     2
     3     3
     4     4
     5     5

ans = 5×2
     1     1
     2     2
     3     3
     4     4
     5     5
```

Example: Combine and Transform Array

```
% cat, horzcat, vertcat
mat01 = reshape(1:9, 3, 3)
mat02 = horzcat(mat01, mat01)
mat03 = cat(2,mat01,mat01)
mat04 = repmat(mat01, 1,2)

mat05 = vertcat(mat01, mat01)
mat06 = cat(1,mat01,mat01)
mat07 = repmat(mat01,2,1)

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

%flip, flipud, fliplr
mat10 = flip(mat01, 1)
mat11 = flipud(mat01)

mat12 = flip(mat01, 2)
mat13 = fliplr(mat01)

mat14 = transpose(mat01)
mat15 = mat01'
mat16 = flipud(rot90(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

mat04 = 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

mat07 = 6×3
     1     4     7
     2     5     8
     3     6     9
     1     4     7
     2     5     8
     3     6     9
```

```
mat08 = 4×4
     1     1     2     2
     1     1     2     2
     3     3     4     4
     3     3     4     4

mat09 = 4×4
     1     1     2     2
     1     1     2     2
     3     3     4     4
     3     3     4     4

mat10 = 3×3
     3     6     9
     2     5     8
     1     4     7
```

```
mat11 = 3×3
     3     6     9
     2     5     8
     1     4     7

mat12 = 3×3
     7     4     1
     8     5     2
     9     6     3

mat13 = 3×3
     7     4     1
     8     5     2
     9     6     3

mat14 = 3×3
     1     2     3
     4     5     6
     7     8     9

mat15 = 3×3
     1     2     3
     4     5     6
     7     8     9

mat16 = 3×3
     1     2     3
     4     5     6
     7     8     9
```

Slide Credits and References

- Stormy Attaway, 2018, Matlab: A Practical Introduction to Programming and Problem Solving, 5th edition
- Lecture slides for “Matlab: A Practical Introduction to Programming and Problem Solving”
- Holly Moore, 2018, MATLAB for Engineers, 5th edition