Introduction to Applied Scientific Computing using MATLAB

Mohsen Jenadeleh

In this lecture, slides from Mathworks, MIT, Waterloo and Rutgers Universities are used

Arrays and Matrices

arrays and matrices are the most important data objects in MATLAB

We discuss briefly:

a) row and column vectors
b) transposition operator, '
c) colon operator, **:**
d) equally-spaced elements, linspace
e) accessing array elements
f) dynamic allocation & de-allocation
g) pre-allocation

The key to efficient MATLAB programming can be summarized in three words:

vectorize, vectorize, vectorize

and avoid all loops

Compare the two alternative computations:

```
x = [2,-3,4,1,5,8];
y = zeros(size(x));
for n = 1:length(x)
   y(n) = x(n)^2;
end
```

```
x = [2,-3,4,1,5,8];
y = x.^2;
```

element-wise exponentiation .^

ordinary exponentiation ^

answer: y = [4,9,16,1,25,64]

```
>> x = [0 1 2 3 4 5]                % row vector
x =
     0     1     2     3     4     5

>> x = 0:5                          % row vector
x =
     0     1     2     3     4     5

>> x = [0 1 2 3 4 5]'               % column vector, (0:5)'
x =
     0
     1
     2
     3
     4
     5
```

the prime operator, ', or transpose, turns row vectors into column vectors, and vice versa

caveat: ' is actually conjugate transpose,
use dot-prime, .' , for transpose w/o conjugation

```
>> z = [i; 1+2i; 1-i]      % column vector
z =
        0 + 1.0000i
   1.0000 + 2.0000i
   1.0000 - 1.0000i

>> z.'              % transpose without conjugation
ans =
        0 + 1.0000i   1.0000 + 2.0000i   1.0000 - 1.0000i

>> z'               % transpose with conjugation
ans =
        0 - 1.0000i   1.0000 - 2.0000i   1.0000 + 1.0000i

>> (z.')'           % same as (z').' , or, conj(z)
ans =
        0 - 1.0000i
   1.0000 - 2.0000i
   1.0000 + 1.0000i
```

about linspace:

```
x = linspace(a,b,N+1);
```

is equivalent to:

```
x = a : (b-a)/N : b;
```

i.e., N+1 equally-spaced points in the interval [a,b]
or, dividing [a,b] into N equal sub-intervals

$$x(n) = a + \left(\frac{b-a}{N}\right)(n-1), \quad n = 1, 2, \ldots, N+1$$

step increment

```
>> x = 0 : 0.2 : 1              % in general, x = a:s:b
>> x = linspace(0,1,6)          % see also logspace
x =
      0   0.2000   0.4000   0.6000   0.8000   1.0000
```

6 points, 5 subintervals

```
>> x = 0 : 0.3 : 1
x =
    0    0.3    0.6    0.9

>> x = 0 : 0.4 : 1
x =
    0    0.4    0.8

>> x = 0 : 0.7 : 1
x =
    0    0.7

% before rounding, (b-a)/s was in the three cases:
% 1/0.3 = 3.3333,  1/0.4 = 2.5,  1/0.7 = 1.4286
```

step increment

`x = a : s : b;`

the number of subintervals within [a,b] is obtained by rounding **`(b-a)/s`**, down to the nearest integer,

```
N = floor((b-a)/s);
```

`length(x)` is equal to **`N+1`**

```
x(n) = a + s*(n-1),
n = 1,2,...,N+1
```

Note: MATLAB array indices always start with 1
and may not be 0 or negative

exception:
logical indexing,
discussed later

```
>> x = [ 2,    5,   -6,   10,    3,    4 ];
      x(1), x(2), x(3), x(4), x(5), x(6)
```

Other languages, such as C/C++ and Fortran, allow indices to start at 0. For example, the same array would be declared/defined in C as follows:

```
double x[6] = { 2,    5,   -6,   10,    3,    4 };
              x[0], x[1], x[2], x[3], x[4], x[5]
```

accessing array entries:

```
>> x = [2, 5, -6, 10, 3, 4]
x =
     2     5    -6    10     3     4

>> length(x)     % length of x, see also size(x)
ans =
     6

>> x(1)          % first entry
ans =
     2

>> x(3)          % third entry
ans =
    -6

>> x(end)        % last entry - need not know length
ans =
     4
```

accessing array entries:

```
>> x(end-3:end)            % x = [2, 5, -6, 10, 3, 4]
ans =
     -6    10     3     4        % last four

>> x(3:5)                  % list third-to-fifth entries
ans =
    -6    10     3

>> x(1:3:end)              % every third entry
ans =
     2    10

>> x(1:2:end)              % every second entry
ans =
     2    -6     3
```

accessing array entries:

```
>> x = [2, 5, -6, 10, 3, 4];

>> x(end:-1:1)     % list backwards, same as fliplr(x)
ans =
     4    3    10    -6     5     2

>> x([3,1,5])        % list [x(3),x(1),x(5)]
ans =
     -6     2     3

>> x(end+3) = 8
x =
    2     5    -6    10     3     4     0     0     8
```

automatic memory re-allocation

automatic memory allocation and de-allocation:

```
>> clear x

>> x(3) = -6
x =
     0     0    -6

>> x(6) = 4
x =
     0     0    -6     0     0     4

>> x(end) = []                        % delete last entry
x =
     0     0    -6     0     0

>> x = [2, 5, -6, 10, 3, 4];
>> x(3)=[]                            % delete third entry
x =
     2     5     10     3     4
```

pre-allocation

```
>> clear x
>> x = zeros(1,6)          % 1x6 array of zeros
x =
     0     0     0     0     0     0

>> x = zeros(6,1)          % 6x1 array of zeros
x =
     0
     0
     0
     0
     0
     0
```

```
>> help zeros
>> help ones
```

illustrating dynamic allocation & pre-allocation

```
clear x;
for k=[3,7,10]
   x(k) = 3 + 0.1*k;
   disp(x);
end
```

```
% k runs successively through
% the values of [3,7,10]
% diplay current vector x
```

```
    0.0   0.0   3.3
    0.0   0.0   3.3   0.0   0.0   0.0   3.7
    0.0   0.0   3.3   0.0   0.0   0.0   3.7   0.0   0.0   4.0
```

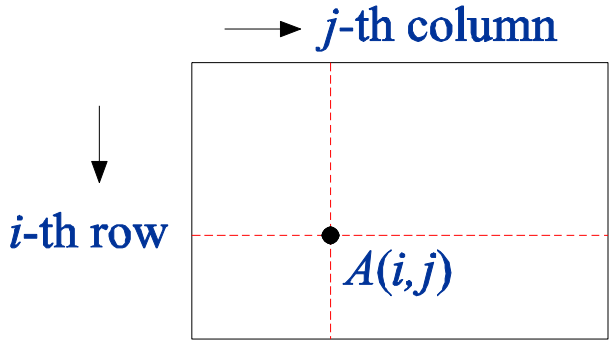
```
x = zeros(1,10);
for k=[3,7,10]
   x(k) = 3 + 0.1*k;
   disp(x);
end
```

```
% pre-allocate x to length 10
```

```
    0.0   0.0   3.3   0.0   0.0   0.0   0.0   0.0   0.0   0.0
    0.0   0.0   3.3   0.0   0.0   0.0   3.7   0.0   0.0   0.0
    0.0   0.0   3.3   0.0   0.0   0.0   3.7   0.0   0.0   4.0
```

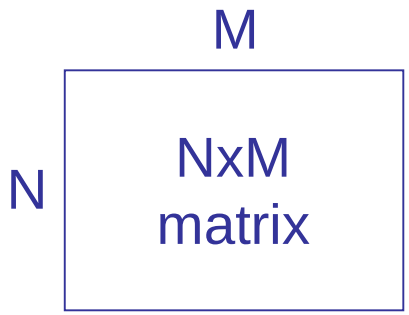
defining matrices

$$A = \begin{bmatrix} A_{11} & A_{12} & A_{13} \\ A_{21} & A_{22} & A_{23} \\ A_{31} & A_{32} & A_{33} \end{bmatrix}$$



matrix indexing convention

```
>> A = [1 2 3; 2 0 4; 0 8 5]
A =
     1     2     3
     2     0     4
     0     8     5
```




```
>> size(A)                % [N,M] = size(A), NxM matrix
ans =
     3     3
```

column dimension = no. rows

row dimension = no. columns

accessing matrix elements

```
>> A(1,1)        % 11 matrix element
ans =
     1
```

$$A = \begin{bmatrix} \textcircled{1} & 2 & 3 \\ 2 & 0 & 4 \\ 0 & 8 & 5 \end{bmatrix}$$

```
>> A(2,3)        % 23 matrix element
ans =
     4
```

$$A = \begin{bmatrix} 1 & 2 & 3 \\ 2 & 0 & \textcircled{4} \\ 0 & 8 & 5 \end{bmatrix}$$

```
>> A(:,2)        % second column
ans =
     2
     0
     8
```

$$A = \begin{bmatrix} 1 & 2 & 3 \\ 2 & 0 & 4 \\ 0 & 8 & 5 \end{bmatrix}$$

```
>> A(3,:)         % third row
ans =
     0     8     5
```

$$A = \begin{bmatrix} 1 & 2 & 3 \\ 2 & 0 & 4 \\ 0 & 8 & 5 \end{bmatrix}$$

concatenating columns

```
>> A = [1 2 3; 2 0 4; 0 8 5]
A =
     1     2     3
     2     0     4
     0     8     5
```

$$A = \begin{bmatrix} 1 & 2 & 3 \\ 2 & 0 & 4 \\ 0 & 8 & 5 \end{bmatrix}$$

```
>> A(:)      % concatenate columns
ans =
     1
     2
     0
     2
     0
     8
     3
     4
     5
```

$$A = \begin{bmatrix} 1 & 2 & 3 \\ 2 & 0 & 4 \\ 0 & 8 & 5 \end{bmatrix}$$

column-wise indexing

A(6) ← 8

A(9) ← 5

concatenating rows

```
B = A'; B(:)
```

see also the built-in functions:

sub2ind, ind2sub

building a matrix column-wise

```
>> A = zeros(3,3);

>> A(:) = [1 2 0  2 0 8  3 4 5]

A =
     1     2     3
     2     0     4
     0     8     5
```

define desired size

enter elements in a row (or column)

elements are re-arranged column-wise

sub-matrices

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

```
A(3:4, 2:4)
ans =
     2     5     2
     6     1     8
```

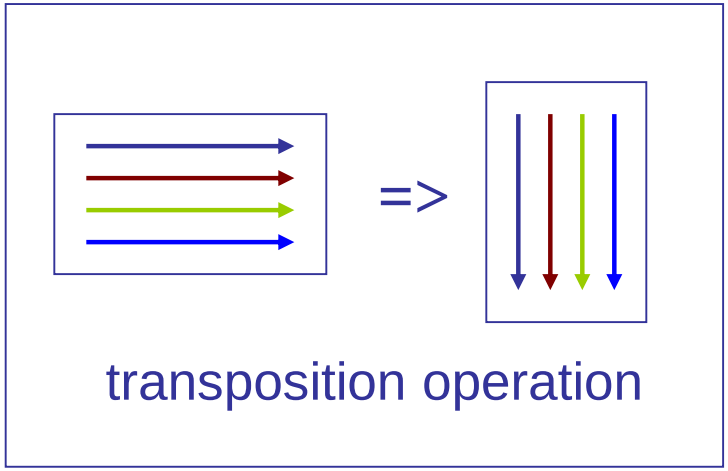
$$A = \begin{bmatrix} 2 & 4 & 1 & 3 & 5 \\ 8 & 6 & 7 & 4 & 9 \\ 3 & 2 & 5 & 2 & 1 \\ 5 & 6 & 1 & 8 & 4 \end{bmatrix}$$

```
A(1:3, [1,5])
ans =
     2     5
     8     9
     3     1
```

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

transposing a matrix

```
>> A = [1 2 3 4; 2 0 5 6; 0 8 7 9]    % size 3x4
A =
     1     2     3     4
     2     0     5     6
     0     8     7     9

>> A'                                  % size 4x3
ans =
     1     2     0
     2     0     8
     3     5     7
     4     6     9
```



adding / deleting rows or columns

```
>> A = [1 2 3; 2 0 4; 0 8 5]

>> A(5,:) = [7 8 9]      % add a fifth row
A =
     1     2     3
     2     0     4
     0     8     5
     0     0     0
     7     8     9
```

4th row is automatically allocated

```
>> A(:,2) = []           % delete second column
A =
     1     3
     2     4
     0     5
     0     0
     7     9
```

[ ] denotes an empty 0x0 matrix

alternatively, redefine A by omitting its second column:

```
>> A = A(:,[1,3]);
```

replacing rows or columns

```
>> A = [1 2 3; 2 0 4; 0 8 5]
A =
     1     2     3
     2     0     4
     0     8     5

>> A(:,2) = [20 30 40]'    % replace second column
A =
     1    20     3
     2    30     4
     0    40     5

>> A(3,:) = [50 60 70]     % replace third row
A =
     1    20     3
     2    30     4
    50    60    70
```

inserting rows or columns

```
>> A = [1 2 3; 2 0 4; 0 8 5]
A =
     1     2     3
     2     0     4
     0     8     5

% insert new column between columns 2 & 3
A = [A(:,1:2), [10 20 30]', A(:,3)]
A =
     1     2    10     3
     2     0    20     4
     0     8    30     5

% insert new row between rows 1 & 2
A = [A(1,:); [60 70 80 90]; A(2:3,:)]
A =
     1     2    10     3
    60    70    80    90
     2     0    20     4
     0     8    30     5
```

concatenating matrices

$$A = \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix}$$

$$B = \begin{bmatrix} 5 & 6 \\ 7 & 8 \end{bmatrix}$$

```
>> A = [1 2; 3 4];
>> B = [5 6; 7 8];

>> C = [A, B]
C =
     1     2     5     6
     3     4     7     8
```

A,B must have same number of rows

```
>> C = [A; B]
C =
     1     2
     3     4
     5     6
     7     8
```

A,B must have same number of columns

appending columns or rows

```
>> A = [1 2; 3 4; 5 6];
>> b = [7; 7; 7];
>> c = [8 8 8]';

>> B = [A,b,c]
B =
     1     2     7     8
     3     4     7     8
     5     6     7     8

>> C = [b,A,c]
C =
     7     1     2     8
     7     3     4     8
     7     5     6     8
```

$$A = \begin{bmatrix} 1 & 2 \\ 3 & 4 \\ 5 & 6 \end{bmatrix}, \quad \mathbf{b} = \begin{bmatrix} 7 \\ 7 \\ 7 \end{bmatrix}, \quad \mathbf{c} = \begin{bmatrix} 8 \\ 8 \\ 8 \end{bmatrix}$$

```
>> D = [A; [7 7];]
>> E = [[8 8]; A]
D =
     1     2
     3     4
     5     6
     7     7
E =
     8     8
     1     2
     3     4
     5     6
```

special matrices

```
eye(3)    % 3x3 identity matrix
ans =
     1     0     0
     0     1     0
     0     0     1
```

```
>> help eye
>> help zeros
>> help ones
```

```
zeros(3)   % 3x3 matrix of zeros
ans =
     0     0     0
     0     0     0
     0     0     0
```

```
ones(3)    % 3x3 matrix of ones
ans =
     1     1     1
     1     1     1
     1     1     1
```

general usage:

eye(N,M)
zeros(N,M)
ones(N,M)

see also:

rand(N,M)
randn(N,M)
randi(I,N,M)

for more information on elementary matrices see:

```
>> help elmat

  Elementary matrices and matrix manipulation.

  Elementary matrices.
    zeros         - Zeros array.
    ones          - Ones array.
    eye           - Identity matrix.
    repmat        - Replicate and tile array.
    linspace      - Linearly spaced vector.
    logspace      - Logarithmically spaced vector.

    etc.
```

diagonals

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

>> d = diag(A)          % main diagonal
d =
     1
     5
     9

>> d = diag(A,-1)       % first sub-diagonal
d =
     4
     8
```

```
>> help diag
```

`diag(diag(A))` = what does it do?

how to make a diagonal matrix

```
>> d = [4 5 6];          % or, column d = [4 5 6]';

A = diag(d)              % d is main diagonal
A =
     4     0     0
     0     5     0
     0     0     6

>> d = [4 5];

A = diag(d,1)            % d is first upper-diagonal
A =
     0     4     0
     0     0     5
     0     0     0
     0     0     0
```

```
>> A = [1 2; 3 4]; B = [5 6; 7 8];
>> C = [9 8 7; 6 5 4; 3 2 1];

>> blkdiag(A,B)
ans =
     1     2     0     0
     3     4     0     0
     0     0     5     6
     0     0     7     8
```

how to make block-diagonal matrices

matrix dimensions expand as necessary

```
>> blkdiag(A,B,C)
ans =
     1     2     0     0     0     0     0
     3     4     0     0     0     0     0
     0     0     5     6     0     0     0
     0     0     7     8     0     0     0
     0     0     0     0     9     8     7
     0     0     0     0     6     5     4
     0     0     0     0     3     2     1
```

replicating matrices – using repmat

```
>> A=[1 2; 3 4]
A =
     1     2
     3     4
```

repmat works also with other data types, such as strings or cell arrays

```
>> repmat(A,3,4)
ans =
     1     2     1     2     1     2     1     2
     3     4     3     4     3     4     3     4
     1     2     1     2     1     2     1     2
     3     4     3     4     3     4     3     4
     1     2     1     2     1     2     1     2
     3     4     3     4     3     4     3     4
```

```
>> s = repmat('%7.4f   ', 1, 4)
s =
%7.4f   %7.4f   %7.4f   %7.4f
```

← replicated string

reshaping a matrix or a vector

B = reshape(A,P,Q)

reshapes an NxM matrix into a PxQ matrix (must have PxQ=NxM)

B is formed column-wise from the elements of A

```
>> a = [1 2 3 4 5 6];

>> reshape(a,2,3)
ans =
     1     3     5
     2     4     6

>> reshape(a,3,2)
ans =
     1     4
     2     5
     3     6
```

```
>> reshape(a,6,1)
ans =
     1
     2
     3
     4
     5
     6
```

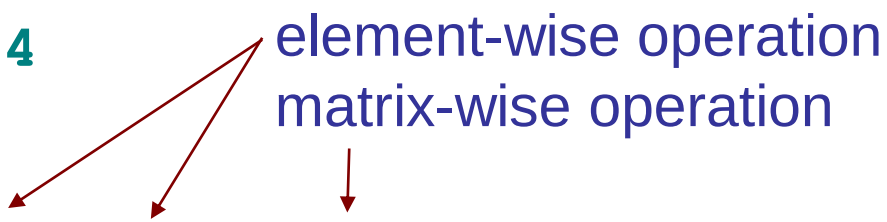
reshaping a matrix or a vector

```
A = [1    5    9
     2    6    5
     3    7    0
     4    8    4];

>> reshape(A,3,4)
ans =
     1    4    7    5
     2    5    8    0
     3    6    9    4


>> reshape(A,2,6)
ans =
     1    3    5    7    9    0
     2    4    6    8    5    4
```

```
>> reshape(A,6,2)
ans =
     1    7
     2    8
     3    9
     4    5
     5    0
     6    4
```

element-wise
matrix operations

```
>> A = [1 2; 3 4]
A =
     1     2
     3     4
```




```
>> [A, A.^2; 2.^A, A^2]          % form sub-blocks
ans =
     1     2     1     4
     3     4     9    16
     2     4     7    10          % note A.^2 ~= A^2
     8    16    15    22

>> B = 10.^A;
>> [B, log10(B)]
ans =
       10      100        1        2
     1000    10000        3        4
```

element-wise matrix operations

```
>> A=[1 4; 8 2], B=[1 2; 2 1]
A =
     1     4
     8     2
B =
     1     2
     2     1

>> A./B
ans =
     1     2
     4     2

>> A.\B
ans =
    1.0000    0.5000
    0.2500    0.5000
```

But note the matrix operations:

```
>> sym(A/B)   % A*inv(B)
ans =
   [  7/3, -2/3]
   [ -4/3, 14/3]

>> A\B        % inv(A)*B
ans =
    0.2   0.0
    0.2   0.5
```

element-wise matrix operations

```
>> A=[1 4; 8 2], B=[1 2; 2 1]
A =
     1     4
     8     2
B =
     1     2
     2     1

>> A.*B
ans =
     1     8
    16     2

>> A.^B
ans =
     1    16
    64     2

>> B.^A
ans =
     1    16
   256     1
```

functions of matrices

```
>> X = [pi/2, pi/3; pi/4, pi/8]
X =
    1.5708    1.0472
    0.7854    0.3927

>> sin(X)
ans =
    1.0000    0.8660
    0.7071    0.3827

>> sin(sym(X))
ans =
[          1,                3^(1/2)/2]
[ 2^(1/2)/2, (2 - 2^(1/2))^(1/2)/2]
```

many functions operate element-wise on matrices
e.g., trig, exp, log functions

others operate column-wise
e.g., min, max, sort, diff, mean, std, median, sum, cumsum, prod, cumprod

functions of matrices

functions that operate column-wise can also operate row-wise by using a second argument

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

>> sum(A)
ans =
    25    12    18

>> cumsum(A)% comulative sum
ans =
     8     5     8
    17     6    11
    19    10    16
    25    12    18
```

```
>> sum(A,2)
ans =
    21
    13
    11
    10

>> cumsum(A,2)
ans =
     8    13    21
     9    10    13
     2     6    11
     6     8    10
```

functions of matrices

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

```
>> mean(A), mean(A,2)
ans =
    6.25   3.00   4.50
ans =
    7.0000
    4.3333
    3.6667
    3.3333
```

means computed down each column

means computed across rows

```
>> [m,i]=min(A)
m =
     2     1     2
i =
     3     2     4
```

```
>> [m,i]=min(A,[],2);
>> [m,i]
ans =
     5     2
     1     2
     2     1
     2     2
```

min, max require a sightly different syntax for row-wise operation, similarly for **diff,std**

functions of matrices

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

```
>> fliplr(A)
ans =
     8     5     8
     3     1     9
     5     4     2
     2     2     6
```

```
>> flipud(A)
ans =
     6     2     2
     2     4     5
     9     1     3
     8     5     8
```

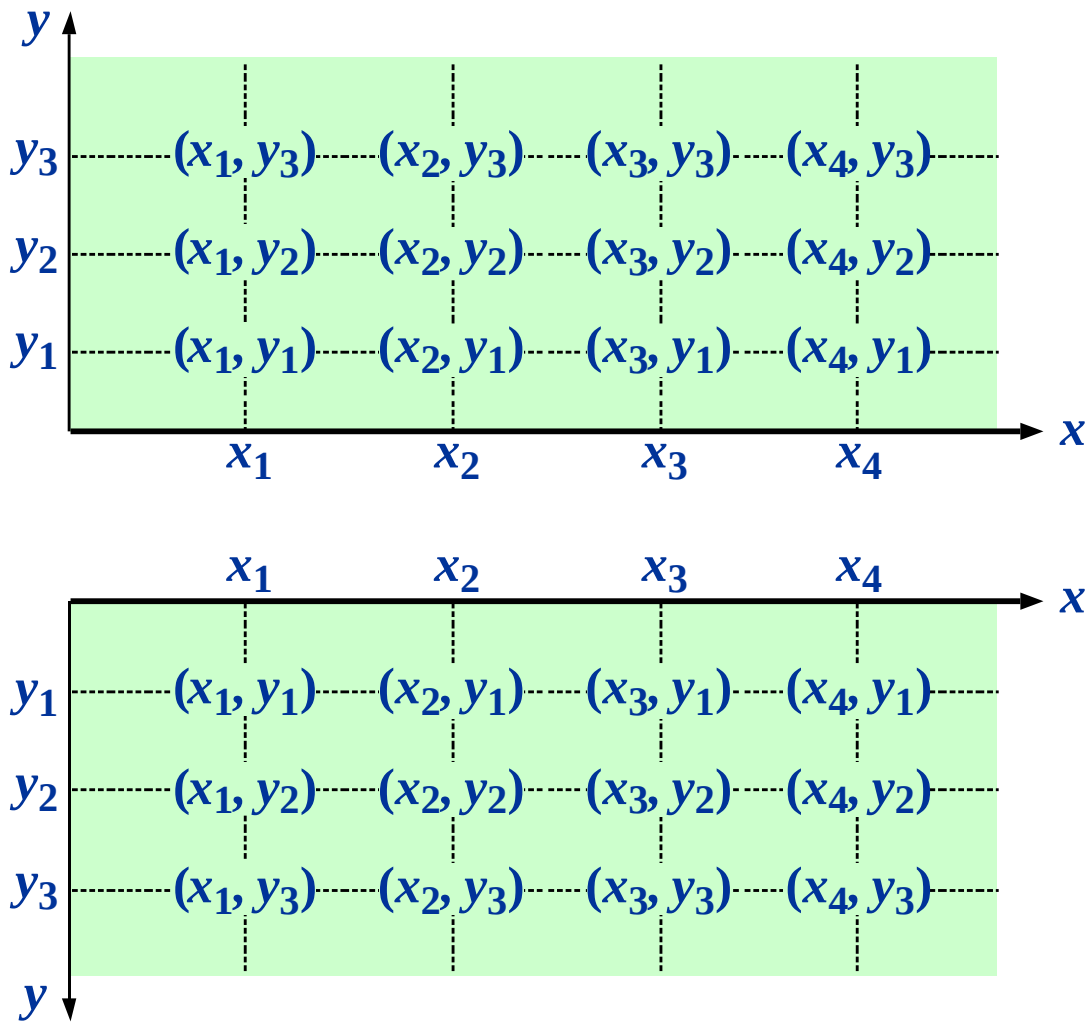
```
>> rot90(A)
ans =
     8     3     5     2
     5     1     4     2
     8     9     2     6
```

rotate by 90 degrees

reverse each row

reverse each column

`flipud(rot90(A))` and **`rot90(fliplr(A))`** are the same as **`A'`**



meshgrid function

a function of two variables,
$z = f(x,y)$
defines a surface

meshgrid

```
x = [x1,  x2,  x3,  x4]
y = [y1,  y2,  y3]

[X,Y] = meshgrid(x,y)
```

$$X = \begin{bmatrix} x_1 & x_2 & x_3 & x_4 \\ x_1 & x_2 & x_3 & x_4 \\ x_1 & x_2 & x_3 & x_4 \end{bmatrix}, \quad Y = \begin{bmatrix} y_1 & y_1 & y_1 & y_1 \\ y_2 & y_2 & y_2 & y_2 \\ y_3 & y_3 & y_3 & y_3 \end{bmatrix}$$

meshgrid
ndgrid

```
>> x = [1 2 3 4];   % N-dim
>> y = [5 6 7];     % M-dim

>> [X,Y] = meshgrid(x,y)
X =
   1   2   3   4
   1   2   3   4
   1   2   3   4
Y =
   5   5   5   5
   6   6   6   6
   7   7   7   7
```

x is replicated row-wise

y is replicated column-wise

X,Y have size MxN

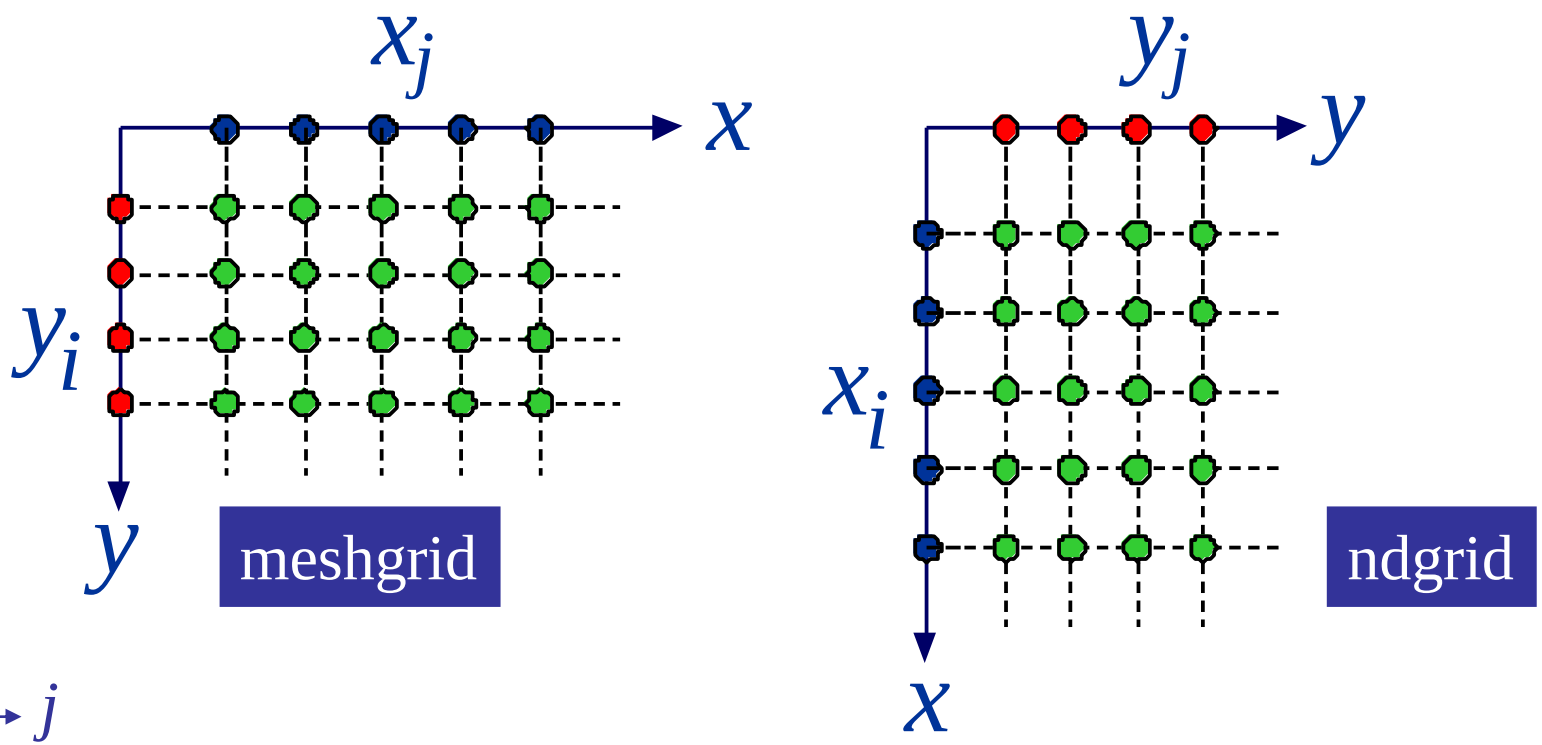
```
[Y,X] = meshgrid(y,x)
```

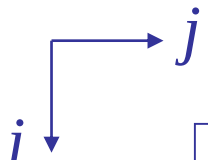
equivalent

```
>> [X,Y] = ndgrid(x,y)
X =
    1   1   1
    2   2   2
    3   3   3
    4   4   4
Y =
    5   6   7
    5   6   7
    5   6   7
    5   6   7
```

x is replicated column-wise, and **y**, row-wise

X,Y have size NxM





[X,Y] = meshgrid(x,y)

x = rows
y = columns

[X,Y] = ndgrid(x,y)

x = columns
y = rows

equivalent

[Y,X] = meshgrid(y,x)

[Y,X] = ndgrid(y,x)