TOPIC 2
Computer application for manipulating matrix using MATLAB

Definition of Matrices in MATLAB

MATLAB is based on matrix and vector algebra; even scalars are treated as 1x1 matrices. Therefore, vector and matrix operations are as simple as common calculator operations.

Row and Column vectors are special cases of matrices. An m x n matrix is a rectangular array of numbers having m rows and n columns. It is usual in a mathematical setting to include the matrix in either round or square brackets-we shall use square ones.

The best way for you to get started with MATLAB is to learn how to handle matrices. You can enter matrices into MATLAB in several different ways:

- Enter an explicit list of elements.
- Load matrices from external data files.
- Generate matrices using built-in functions.
- Create matrices with your own functions in M-files.

Start by entering a simple matrix as a list of its elements. You only have to follow a few basic conventions:

- Separate the elements of a row with blanks or commas.
- Use a semicolon, ; , to indicate the end of each row.
- Surround the entire list of elements with square brackets, [ ].

To enter a matrix, simply type in the Command Window

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

A=

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

```
>> B = [1:5; 6:10; 11:2:20]
B =
     1     2     3     4     5
     6     7     8     9    10
    11    13    15    17    19
```

So A is 3x3 matrix, and B is 3 x 5. In this context, a row vector is a 1 x n matrix and a column vector a m x 1 matrix.

Notes that each row in B are generated using equally spaced elements. The middle number defines the increment. If only two numbers are given, then the increment is set to a default of 1.

So *1:5* creates 1 2 3 4 5
and
11:2:20 creates 11 13 15 17 19

Size of a matrix
We can get the size (dimensions) of a matrix with the command **size**

```
>> size(A), size(B)
ans =
        3        3
ans =
        3        5
>> size(ans)
ans =
        1        2
```

So A is 3x3 and B is 3x5. The last command size(ans) shows that the value returned by size is itself a 1x2 matrix (a row vector). We can save the results for use in subsequent calculations.

```
>> [r c] = size(A)
r =
        3
c =
        3
```

Transpose of a matrix
Transposing a vector changes it from a row to a column vector and vice versa. The extension of this idea to matrices is that transposing interchanges rows with the corresponding columns: the 1st row becomes the 1st column, and so on.

```
>> B,B'
B =
     1     2     3     4     5
     6     7     8     9    10
    11    13    15    17    19

ans =
     1     6    11
     2     7    13
     3     8    15
     4     9    17
     5    10    19
>> size(B),size(B')
ans =
     3     5

ans =
     5     3
```

Special Matrices

Matlab provides a number of useful built-in matrices of any desired size.
ones(m,n) gives an m _ n matrix of 1's,

```
>> P = ones(2,3)
P =
     1   1   1
     1   1   1
```

zeros(m,n) gives an m x n matrix of 0's,

```
>> Z = zeros(2,3), zeros(size(P'))
Z =
     0   0   0
     0   0   0
ans =
     0   0
     0   0
     0   0
```

The second command illustrates how we can construct a matrix based on the size of an existing one. Try ones(size(B)).

An n x n matrix that has the same number of rows and columns is called a square matrix. A matrix is said to be symmetric if it is equal to its transpose (i.e. it is unchanged by transposition):

```
>> S = [2 -1 0; -1 2 -1; 0 -1 2],
S =
    2   -1    0
   -1    2   -1
    0   -1    2

>> ST=S'
ST =
    2   -1    0
   -1    2   -1
    0   -1    2

>> S-ST
ans =
    0    0    0
    0    0    0
    0    0    0
```

The Identity Matrix

The n x n identity matrix is a matrix of zeros except for having ones along its leading diagonal (top left to bottom right). This is called **eye(n)** in Matlab (since mathematically it is usually denoted by I).

```
>> I = eye(3), x = [8; -4; 1], I*x
I =
     1     0     0
     0     1     0
     0     0     1

x =
     8
    -4
     1

ans =
     8
    -4
     1
```

Notice that multiplying the 3 x 1 vector x by the 3 x 3 identity I has no effect (it is like multiplying a number by 1).

Diagonal Matrices

A diagonal matrix is similar to the identity matrix except that its diagonal entries are not necessarily equal to 1. To construct the matrix in Matlab, we could either type it in directly

```
>> D = [-3 0 0; 0 4 0; 0 0 2]
D =
    -3     0     0
     0     4     0
     0     0     2
```

but this becomes impractical when the dimension is large (e.g. a 100 x 100 diagonal matrix). We then use the **diag** function. We first define a vector **d**, say, containing the values of the diagonal entries (in order) then **diag(d)** gives the required matrix.

```
>> d = [-3 4 2], D = diag(d)
d =

    -3     4     2

D =
    -3     0     0
     0     4     0
     0     0     2
```

On the other hand, if A is any matrix, the command diag(A) extracts its diagonal entries:

```
>> A = [0 1 8 7; 3 -2 -4 2; 4 2 1 1], diag(A)
A =

     0     1     8     7
     3    -2    -4     2
     4     2     1     1
```

ans =

```
     0
    -2
     1
```

Notice that the matrix does not have to be square.

Building Matrices

It is often convenient to build large matrices from smaller ones:

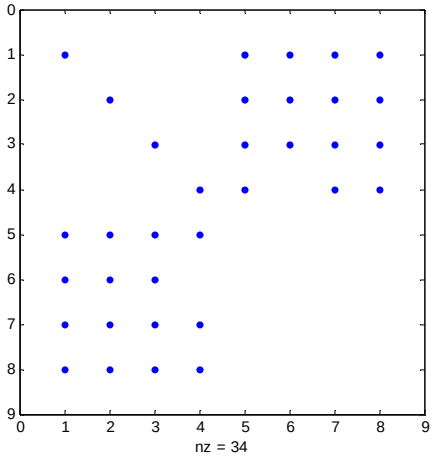
```
>> C=[0 1; 3 -2; 4 2]; x=[8;-4;1];
>> G = [C x]
G =
     0     1     8
     3    -2    -4
     4     2     1

>> C=[-1 2 5; 0 8 6], H=[A;C]
C =
    -1     2     5
     0     8     6
H =
     1     2     3
     4     5     6
     7     8     9
    -1     2     5
     0     8     6
```

so we have added an extra column (x) to C in order to form G and have stacked A and C on top of each other to form H.

```
>> J = [1:4; 5:8; 9:12; 20 0 5 4]
J =
     1     2     3     4
     5     6     7     8
     9    10    11    12
    20     0     5     4

>> K = [ diag(1:4) J; J' zeros(4,4)]
K =
     1     0     0     0     1     2     3     4
     0     2     0     0     5     6     7     8
     0     0     3     0     9    10    11    12
     0     0     0     4    20     0     5     4
     1     5     9    20     0     0     0     0
     2     6    10     0     0     0     0     0
     3     7    11     5     0     0     0     0
     4     8    12     4     0     0     0     0
```



The command spy(K) will produce a graphical display of the location of the nonzero entries in K (it will also give a value for nz-the number of nonzero entries):

```
>> spy(K), grid
```

Extracting Bits of Matrices

We may extract sections from a matrix in much the same way as for a vector. Each element of a matrix is indexed according to which row and column it belongs to. The entry in the ith row and jth column is denoted mathematically by Ai;j and, in Matlab, by A(i,j).

```
>> J
J =
     1     2     3     4
     5     6     7     8
     9    10    11    12
    20     0     5     4
>> J(4,3)
ans =
          5
>> J(4,5)
??? Index exceeds matrix dimensions.

>> J(4,1) = J(1,1) + 6
J =
     1     2     3     4
     5     6     7     8
     9    10    11    12
     7     0     5     4
>> J(1,1) = J(1,1) - 3*J(1,2)
J =
    -5     2     3     4
     5     6     7     8
     9    10    11    12
     7     0     5     4
```

In the following examples we extract i) the 3rd column, ii) the 2nd and 3rd columns, iii) the 4th row, and iv) the “central" 2x2 matrix.

```
>> J(:,3) % 3rd column
>> J(:,2:3) % columns 2 to 3
>> J(4,:) % 4th row
>> J(2:3,2:3) % rows 2 to 3 & cols 2 to 3
```

Thus, **:** on its own refers to the entire column or row depending on whether it is the first or the second index.

Dot product of matrices (.*)

The dot product works as for vectors: corresponding elements are multiplied together-so the matrices involved must have the same size.

```
>> A.*G
```

```
ans =
     0     2    24
    12   -10   -24
    28    16     9
>> A.*C
??? Error using ==> times
Matrix dimensions must agree.
```

Matrix-vector products

We turn next to the definition of the product of a matrix with a vector. This product is only defined for column vectors that have the same number of entries as the matrix has columns. So, if **A** is an mxn matrix and **x** is a column vector of length n, then the matrix-vector **Ax** is legal. An **mxn** matrix times an **nx1** matrix → a m x 1 matrix.

We visualize **A** as being made up of **m** row vectors stacked on top of each other, then the product corresponds to taking the scalar product of each row of **A** with the vector **x**: The result is a column vector with **m** entries.

$$
\begin{aligned}
A\underline{x} &= \begin{bmatrix} \boxed{5 & 7 & 9} \\ \boxed{1 & -3 & -7} \end{bmatrix} \begin{bmatrix} \boxed{8} \\ -4 \\ 1 \end{bmatrix} \\
&= \begin{bmatrix} 5\times 8 + 7\times(-4) + 9\times 1 \\ 1\times 8 + (-3)\times(-4) + (-7)\times 1 \end{bmatrix} \\
&= \begin{bmatrix} 21 \\ 13 \end{bmatrix}
\end{aligned}
$$

It is somewhat easier in Matlab:

```
>> A = [5 7 9; 1 -3 -7]
A =
     5     7     9
     1    -3    -7
>> x = [8; -4; 1]
x =
     8
    -4
     1
>> A*x
ans =
    21
    13
```

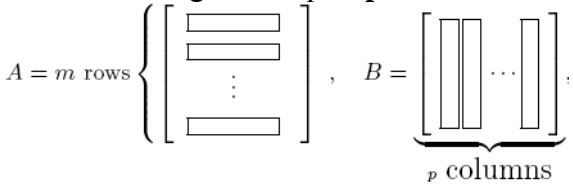
$$(m\times\boxed{n)\ \text{times}\ (n}\times 1) \Rightarrow (m\times 1)$$

```
>> x*A
??? Error using ==> mtimes
Inner matrix dimensions must agree.
```

Unlike multiplication in arithmetic, A*x is not the same as x*A.

Matrix-Matrix Products

To form the product of an **mxn** matrix **A** and a **nxp** matrix **B**, written as **AB**, we visualize the first matrix (**A**) as being composed of **m** row vectors of length **n** stacked on top of each other while the second (**B**) is visualized as being made up of **p** column vectors of length **n**:



The entry in the *i*th row and *j*th column of the product is then the scalar product of the *i*th row of A with the *j*th column of B. The product is an **m x p** matrix:

$$(m \times \boxed{n) \text{ times } (n} \times p) \Rightarrow (m \times p)$$

Check that you understand what is meant by working out the following examples by hand and comparing with the Matlab answers.

```
>> A,B = [0, 1; 3, -2; 4, 2]
A =
     5     7     9
     1    -3    -7
B =
     0     1
     3    -2
     4     2

>> C = A*B
C =
    57     9
   -37    -7

>> D = B*A
D =
     1    -3    -7
    13    27    41
    22    22    22
>> E = B'*A'
E =
    57   -37
     9    -7
```

We see that E = C' suggesting that (A*B)' = B'*A'

Why is B x A a 3 x 3 matrix while A x B is 2 x 2?

Sparse Matrices

Matlab has powerful techniques for handling sparse matrices these are generally large matrices (to make the extra work involved worthwhile) that have only a very small proportion of non{zero entries.

Example: Create a sparse 5 x 4 matrix S having only 3 non-zero values: $S_{1,2} = 10$, $S_{3,3} = 11$ and $S_{5,4} = 12$.

We first create 3 vectors containing the i-index, the j-index and the corresponding values of each term and we then use the **sparse** command.

```
>> i = [1, 3, 5]; j = [2,3,4];
>> v = [10 11 12];
>> S = sparse (i,j,v)
S =
   (1,2)       10
   (3,3)       11
   (5,4)       12
>> T = full(S)
T =
    0    10     0     0
    0     0     0     0
    0     0    11     0
    0     0     0     0
    0     0     0    12
```

Example: Develop Matlab code to create, for any given value of n, the sparse (tridiagonal) matrix

$$B = \begin{bmatrix} 1 & n & & & & \\ -2 & 2 & n-1 & & & \\ & -3 & 3 & n-2 & & \\ & & \ddots & \ddots & \ddots & \\ & & & -n+1 & n-1 & 1 \\ & & & & -n & n \end{bmatrix}$$

We define three column vectors, one for each “diagonal" of non-zeros and then assemble the matrix using **spdiags** (short for sparse diagonals). The vectors are named **k, m** and **u.** They must all have the same length and only the first n-1 terms of **k** are used while the last n - 1 terms of **u** are used. **spdiags** places these vectors in the diagonals labeled -1, 0 and 1 (0 defers to the leading diagonal, negatively numbered diagonals lie below the leading diagonal, etc.)

```
>> n = 5;
>> k = -(2:n+1)', m = (1:n )', u = ((n+1):-1:2)',

k =
    -2
    -3
    -4
    -5
    -6
m =
     1
     2
     3
     4
     5
u =
```

```
     6
     5
     4
     3
     2
>> B = spdiags([k m u],-1:1,n,n)
B =
   (1,1)        1
   (2,1)       -2
   (1,2)        5
   (2,2)        2
   (3,2)       -3
   (2,3)        4
   (3,3)        3
   (4,3)       -4
   (3,4)        3
   (4,4)        4
   (5,4)       -5
   (4,5)        2
   (5,5)        5
>> full(B)
ans =
     1     5     0     0     0
    -2     2     4     0     0
     0    -3     3     3     0
     0     0    -4     4     2
     0     0     0    -5     5
```

Inverse of Matrix

MATLAB function **inv** is used to compute the inverse matrix.
Let the matrix **A** be defined as follows

```
A = [1 2 3;4 5 6;7 8 10]
A =
1 2 3
4 5 6
7 8 10
```

Then

```
B = inv(A)
B =
-0.6667 -1.3333 1.0000
-0.6667 3.6667 -2.0000
1.0000 -2.0000 1.0000
```

Determinant

In some applications of linear algebra knowledge of the determinant of a matrix is required.
MATLAB built-in function **det** is designed for computing determinants.
Let

```
A = magic(3);
```

Determinant of **A** is equal to

```
det(A)
ans =
-360
```

The sum Function

The "sum" applied to a vector adds up its components (as in sum(1:10)) while, for a matrix, it adds up the components in each column and returns a row vector. **sum(sum(A))** then sums all the entries of A.

```
>> A = [1:3; 4:6; 7:9]
A =
     1     2     3
     4     5     6
     7     8     9
>> s = sum(A), ss = sum(sum(A))
s =
    12    15    18


ss =

    45
```

max & min

These functions act in a similar way to sum. If x is a vector, then max(x) returns the largest element in x

```
>> x = [1.3   -2.4   0   2.3], max(x), max(abs(x))
x =
    1.3000   -2.4000         0    2.3000
ans =
    2.3000
ans =
    2.4000
>> [m, j] = max(x)
m =
    2.3000
j =
    4
```

When we ask for two outputs, the first gives us the maximum entry and the second the index of the maximum element.

For a matrix, **A, max(A)** returns a row vector containing the maximum element from each column. Thus to find the largest element in A we have to use **max(max(A))**.

find for matrices

The function "find" returns a list of the positions (indices) of the elements of a matrices satisfying a given condition.

For example,

```
>> A = [ -2 3 4 4; 0 5 -1 6; 6 8 0 1]
A =
    -2     3     4     4
     0     5    -1     6
     6     8     0     1
>> k = find(A==0)
k =
     2
     9
```

Thus, we find that A has elements equal to 0 in positions 2 and 9. To interpret this result we have to recognize that “find" first reshapes A into a column vector-this is equivalent to numbering the elements of **A** by columns as in

```
     1   4   7   10
     2   5   8   11
     3   6   9   12
>> n = find(A <= 0)
n =
    1
    2
    8
    9
>> A(n)
ans =
   -2
    0
   -1
    0
```

Thus, n gives a list of the locations of the entries in A that are ≤ 0 and then A(n) gives us the values of the elements selected.

```
>> m = find( A' == 0)
m =
    5
   11
```

Since we are dealing with A', the entries are numbered by rows.

****