1.4 The Help System: your best friend

The command `help` shows the format(s) for using a command and directs you to related commands; without any arguments, it gives you a hyperlinked list of topics to find help on; with a topic as an argument, it gives you a list of subtopics.

```
help plot
help qr
help
```

If you want to see all the commands associated with *elementary matrix manipulation*

```
help matlab/elmat
```

The command `doc` is like `help`, except it comes up in a different window, and may include more details

```
help fft
doc fft
```

The command `lookfor` is used when you do not know what command you want; it does something like a keyword search through the documentation

```
lookfor wavelet
```

Similarly, you can use `docsearch`

```
docsearch fourier
```

The command `which` helps you tell which file a particular command refers to, or whether it is built in

```
which abs
which hadamard
```

`demo` gives video guides and example code

```
demo
```

1.5 Abort Command: Ctrl + C

If MATLAB is running a program and you want to terminate it, type Ctrl + C.

1.6 Basic Operations

In this section you will learn the basic operations in MATLAB and know how to use MATLAB as a calculator. You can try all examples in the command window.

Here are some standard commands to start a session:

```
clear all  % clears all variable definitions
close all  % closes all figures
clc        % clears the screen
```

1.6.1 Scalar Arithmetics

MATLAB has the basic arithmetics for scalars such as `+`, `-`, `*`, `/`, power `^`.

```
a = 3;
b = 5;
a + b
a - b
a * b
a / b
a ^ b
mod(b,a)
```

Fractional power and square root are also available

```
8^(1/3)
sqrt(9)
```

One may also create complex numbers by taking square root to negative number

```
>> z = 1+sqrt(-1)
z =
   1.0000 + 1.0000i
```

or

```
>> z = 1 + 1i
z =
   1.0000 + 1.0000i
```

Its complex conjugation and absolute value are

```
>> conj(z)
ans =
   1.0000 - 1.0000i
>> abs(z)
ans =
    1.4142
```

One may verify the famous *Euler formula*

$$e^{i\pi} = -1$$

```
>> exp(1i*pi)
ans =
  -1.0000 + 0.0000i
```

Note that MATLAB has the math constant π built-in, but not for the natural constant e. To obtain the natural constant e, use `exp(1)`.

```
>> pi
ans =
    3.1416
>> exp(1)
ans =
    2.7183
```

Although MATLAB stores numerical values of variables with double precision (16 decimal digits), the command window displays numerics result only up to 4 digits. To show more digits in display

```
>> format long
>> pi
ans =
   3.141592653589793
>> format short
>> pi
ans =
    3.1416
>>
```

Note that this only changes the number of digits that is displayed. It does not change computation accuracy.

Floating point types have special values "`inf`"(∞) and "`NaN`"(not-a-number). Try these out

```
0/0
1e999^2
isnan(NaN)  % tests if the argument is nan
isinf(Inf)
isfinite(NaN)
isfinite(-Inf)
isfinite(3)
```

1.6.2 Matrix Construction

The square brackets `[ ]` concatenate elements within and create a matrix

```
>> v = [1,2,3]
v =
     1     2     3
>> A = [1,2,3;4,5,6]
A =
     1     2     3
     4     5     6
```

The comma "`,`" is used to separate elements that belong in the same row, while the semicolon "`;`" creates a new row in the array (matrix). It is also common to use the following alternative expressions

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

To see the size of a matrix

```
>> size(A)
ans =
     2     3
```

or

```
>> [m,n] = size(A)
m =
     2
n =
     3
```

The function `zeros` and `ones` are handy to create all-zero and all-one matrices:

```
>> B = ones(3,3)
B =
     1     1     1
     1     1     1
     1     1     1
>> C = zeros(5,2)
C =
     0     0
     0     0
     0     0
     0     0
     0     0
```

where the integer arguments of `zeros` and `ones` are the desired size of the matrix.

Remember how the square brackets `[ ]` concatenate elements enclosed and construct a matrix?

```
>> [[A;B],C]
ans =
     1     2     3     0     0
     4     5     6     0     0
     1     1     1     0     0
     1     1     1     0     0
     1     1     1     0     0
```

The `repmat`, replicating and tiling matrices, is a useful function for generating matrices:

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

The colon "`:`" is one of the most useful operators in MATLAB. One of its usage is to construct row vectors with regularly spaced values.

```
>> u = 2:5
u =
     2     3     4     5
```

Another example:

```
>> u2 = pi:6
u2 =
    3.1416    4.1416    5.1416
```

The spacing does not need to be 1

```
>> u3 = 2 : 0.5 : 4
u3 =
    2.0000    2.5000    3.0000    3.5000    4.0000
>> u4 = 5 : -1 : 2
u3 =
      5     4     3     2
```

The function “`linspace`”, which generates linearly spaced row vector, has a similar functionality

```
>> x = linspace(2,4,5)
x =
    2.0000    2.5000    3.0000    3.5000    4.0000
```

(5 points linearly spaced between 2 and 4).

Other methods for generating matrices:

```
eye(7)     % the identity
ones(5)    % same as ones(5,5)
rand(2,2)  % random numbers distributed uniformly in [0,1]
randn(2,2) % random numbers with standard normal distribution
inf(3,3)
nan(4,1)
magic(5)
```

Summary The basic way to generate matrices is to use the square brackets “`[ ]`” operator, in which one uses symbols such as spaces, commas, semicolons or new lines. The colon operator “`:`” constructs row vectors with equispacing numbers. Functions such as `repmat`, `linspace`, `zeros`, `ones`, `eye`, `rand`, `randn`, etc. are useful to generate elementary matrices.

1.6.3 Matrix Indexing

Let

```
>> A = [1 2 3 4
        5 6 7 8
        9 10 11 12]
A =
     1     2     3     4
     5     6     7     8
     9    10    11    12
```

One may get the value of a matrix entry by

```
>> a = A(3,2)
a =
    10
```

or set value

```
>> A(3,2) = -20
A =
     1     2     3     4
     5     6     7     8
     9   -20    11    12
```

One may get a block from a matrix:

```
>> A([2 3],[2 3 4])
ans =
     6     7     8
   -20    11    12
```

Equivalent expressions include

```
A( 2:3 , 2:4 )
A( 2:3 , 2:end )
```

In parentheses `( )` (array indexing), “end” indicates last array index. A single colon indicates selecting all indexes

```
>> A( : , 2:end)
ans =
     2     3     4
     6     7     8
   -20    11    12
```

so one may permute columns (or rows) simply by

```
>> B = A( : , [4,2,1,3] )
B =
     4     2     1     3
     8     6     5     7
    12   -20     9    11
```

One may assign values to an entire block of a matrix

```
>> B = zeros(5,6)
B =
     0     0     0     0     0     0
     0     0     0     0     0     0
     0     0     0     0     0     0
     0     0     0     0     0     0
     0     0     0     0     0     0
>> B(2:4,2:5) = A
B =
     0     0     0     0     0     0
     0     1     2     3     4     0
     0     5     6     7     8     0
     0     9   -20    11    12     0
     0     0     0     0     0     0
```

An important concept is that MATLAB uses *column major order*. That is, if we view the matrix

```
A =
     1     2     3     4
     5     6     7     8
     9   -20    11    12
```

as a 1D array (vector),

```
>> A(:)
ans =
     1
     5
```

```
     9
     2
     6
   -20
     3
     7
    11
     4
     8
    12
```

One may access `A` by a single index; for example `A(4)` is 2 in this case. The expression `A(4)` is known as the *linear indexing*. To convert linear indices to their corresponding rows and columns, use `ind2sub`:

```
>> [r,c] = ind2sub([3,4],4)
r =
     1
c =
     2
```

where `[3,4]` is the size of `A`. You may check that `A(4)` equals to `A(1,2)`. Conversely,

```
>> ind = sub2ind([3,4],1,2)
ind =
     4
```

Reshaping is frequently used as well:

```
>> reshape(A,4,3)
ans =
     1     6    11
     5   -20     4
     9     3     8
     2     7    12
```

which reshapes `A` to a matrix with size 4×3 so that after postfixed by `(:)` it recovers `A(:)`. Note that the argument 4 and 3 must multiply to 12, the total number of entries in `A`. One may replace one of them by "empty matrix" `[]`

```
>> reshape(A,[],2)
ans =
     1     3
     5     7
     9    11
     2     4
     6     8
   -20    12
```

and MATLAB will do the calculation the only matching numbers of columns or rows for you.

The operators "`.'`" and "`'`" are transpose.

```
>> A'
ans =
     1     5     9
     2     6   -20
     3     7    11
```

```
     4     8    12
>> A.'
ans =
     1     5     9
     2     6   -20
     3     7    11
     4     8    12
```

The "undotted" transpose is the *Hermitian transpose*, which also takes complex conjugation to each elements

```
>> [1+2i,3+4i]
ans =
   1.0000 + 2.0000i   3.0000 + 4.0000i
>> [1+2i,3+4i]'
ans =
   1.0000 - 2.0000i
   3.0000 - 4.0000i
>> [1+2i,3+4i].'
ans =
   1.0000 + 2.0000i
   3.0000 + 4.0000i
```

Summary With parentheses `()` postfixing a matrix `A` one may access the matrix elements using index. In the parentheses, one may use indices which need to be positive integers (1-based indexing); colon "`:`" and "end" notations are allowed. One may either use *subscript indices* `A(r,c)` or *linear indices* `A(ind)`, and one may convert the two using the functions `sub2ind` and `ind2sub`. Reshaping (`reshape`) and transposing (`.'` and `'`) are handy in indexing as well.

1.6.4 Logical Indexing

The comparison operators > (greater than), == (equal to), ~= (not equal to), < (less than), >= (greater or equal to), <= (less or equal to), returns logical matrices. A logical matrix contains entry of value *true* or *false*, displayed as 1 or 0:

```
>> A = magic(4)
A =
    16     2     3    13
     5    11    10     8
     9     7     6    12
     4    14    15     1
>> A > 9
ans =
     1     0     0     1
     0     1     1     0
     0     0     0     1
     0     1     1     0
>> A == 10
ans =
     0     0     0     0
     0     0     1     0
     0     0     0     0
     0     0     0     0
```

Common boolean operations "and", "or", and "not" for logical matrices are "`&`", "|" and "~" respectively:

```
>> (A>5) & (A<10)
ans =
     0     0     0     0
     0     0     0     1
     1     1     1     0
     0     0     0     0
>> (A<=5) | (A>=10)
ans =
     1     1     1     1
     1     1     1     0
     0     0     0     1
     1     1     1     1
>> ~((A==4) | (A==5))
ans =
     1     1     1     1
     0     1     1     1
     1     1     1     1
     0     1     1     1
```

When the matrix is a scalar (1-by-1), MATLAB will suggest you to replace `&` with `&&` and | with ||. These *non-vectorized* "and" and "or" operators for logical scalars are the *short circuit*. For the case of `a||b`, it will return *true* if `a` is true, without looking at `b`; for the case of `a&&b`, it will return *false* if `a` is false without evaluating `b`.

Now let us look at logical indexing. One may access entries of a matrix by "plugging in" a logical matrix of the same size:

```
>> isSmall = A<=5
isSmall =
     0     1     1     0
     1     0     0     0
     0     0     0     0
     1     0     0     1
>> A(isSmall) = 0
A =
    16     0     0    13
     0    11    10     8
     9     7     6    12
     0    14    15     0
```

Logical indexing can be very handy

```
>> A(A>10) = 10
A =
    10     0     0    10
     0    10    10     8
     9     7     6    10
     0    10    10     0
```

Note Logical matrices can also be returned by following functions

```
>> true(3,2)
ans =
```

```
     1     1
     1     1
     1     1
>> false(2)
ans =
     0     0
     0     0
>> isnan([nan,0,1])
ans =
     1     0     0
```

`any` and `all` are useful

```
any([1 0 0 0])  % true if any of the vector entries is true or nonzero
all([1 1 1 1])  % true only if all vector entries are true or nonzero
all([1 0 0 0])  % this case it retuns false
```

1.6.5 Matrix Arithmetics

There is no ambiguity of what $A + B$ means for A and B being matrices of the same size. It adds each counterpart components together. It is also clear that cA for a scalar c and a matrix A is the matrix with each component multiplied by c. In MATLAB these operations are intuitive

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

>> A + B
ans =
     1    -3     5
     5     2     3
>> A+1
ans =
     0     2     3
     5     3     4
>> A*10
ans =
   -10    10    20
    40    20    30
```

However, for matrix-matrix multiplications, there is a distinction between "componentwise multiplication" and "linear-algebraic multiplication". The componentwise multiplication denoted by "`.*`" views matrices as arrays and take products of numbers in the counterpart entries:

$$C = A.*B \quad \text{means} \quad C_{ij} = A_{ij}B_{ij} \quad \text{for each } i, j$$

```
>> A.*B
ans =
```

```
    -2    -4     6
     4     0     0
```

Linear-algebraic multiplication is denoted by "$\star$"

$$C = A \star B \quad \text{means} \quad C_{ij} = \sum_{k=1}^{n} A_{ik} B_{kj}$$

where n = (number of columns of A) = (number of rows of B).

```
>> A = [1,2;3,4]
A =
     1     2
     3     4
>> B = [-1,1;1,2]
B =
    -1     1
     1     2
>> A*B
ans =
     1     5
     1    11
```

There are other operations and functions that other than "multiplications" that take different notations in MATLAB for elementwise operations and linear algebraic operations.

Elementwise arithmetics

Elementwise arithmetics include

```
A + B   % plus
A - B   % minus
A .* B  % times
A ./ B  % (right) division (rdivide)
1 ./ B  % division by scalar
B .\ A  % right-to-left division (ldivide)
A .^ B  % power
```

Common functions listed below also operates elementwise (There are still many functions not listed here)

```
sqrt(A)  % square root
exp(A)   % natural exponential
log(A)   % natural logarithm
log10(A) % base 10 logarithm
abs(A)   % absolute values
sin(A), cos(A), tan(A), cot(A), sec(A), csc(A)  % trigonometric functions
asin(A), acos(A), atan(A), acot(A), asec(A), acsc(A) % inverse trigonometrics
sinh(A), cosh(A), tanh(A), coth(A), sech(A), csch(A) % hyperbolic functions
asinh(A), acosh(A), atanh(A), acoth(A), asech(A), acsch(A) % inverse hyperbolics
```

Linear algebraic arithmetics

In the following, `A` and `B` are matrices, `b` is a column vector, and `c` is a scalar.

```
A * B    % matrix times (mtimes)
A / c, c \ A    % divide by scalar (mrdivide, mldivide)
A \ b    % solves the linear system Ax = b (mldivide)
b'/A     % solves x'A = b', that is A'x = b
A^2      % this case is the same as A*A (mpower)
```

For square matrix A, the matrix exponential is defined to be

$$e^A = \sum_{k=0}^{\infty} \frac{1}{k!} A^k.$$

To evaluate matrix exponential in MATLAB

```
expm(A)  % matrix exponential
```

A square root of a matrix A is a matrix B (may not be unique or even exist for general square matrix A) so that $B^2 = A$:

```
sqrtm(A) % matrix square root
```

Example 1.5. If I is the 3×3 identity matrix

```
>> I = eye(3)
I =
     1     0     0
     0     1     0
     0     0     1
```

what are `exp(I)` and `expm(I)`?

Solution.

$$\texttt{exp(I)} = \begin{bmatrix} e & 1 & 1 \\ 1 & e & 1 \\ 1 & 1 & e \end{bmatrix}, \quad \texttt{expm(I)} = \begin{bmatrix} e & 0 & 0 \\ 0 & e & 0 \\ 0 & 0 & e \end{bmatrix}.$$

■

1.6.6 Strings

Strings are just arrays of character (`char`) values in MATLAB.

```
>> 'hello world'
ans =
hello world
>> ['h','e','llo w','orld']
ans =
hello world
```

Characters are essentially integers

```
>> char(77)
ans =
M
```

```
>> double('s')
ans =
   115
>> double('7')
ans =
    55
```

To convert a string that *represents a number* to a number

```
>> str2double('123')
ans =
   123
```

To convert a number to a string displaying that number

```
>> S = num2str(123)
S =
123
```

"`disp(S)`" displays a string `S`:

```
>> a = 123
a =
   123
>> disp(['My favorite number is ',num2str(a)])
My favorite number is 123
```

"`disp`" also displays numbers or other values

```
>> disp(3)
     3
```

A C-compatible expression

```
>> fprintf('integer: %d, double: %f, string: %s \n',1234,0.999, 'Hello World.')
integer: 1234, double: 0.999000, string: Hello World.
>> S = sprintf('integer: %d, double: %f, string: %s \n',1234,0.999, 'Hello World.')
S =
integer: 1234, double: 0.999000, string: Hello World.
```

To test equality of two strings, use `strcmp` instead of ==

```
strcmp('aaa','bbbb')
strcmp('aaa','aaa')
```

1.6.7 Loops and Controls

We learn four commands here: `for`, `while`, `if` and `switch`.

The basic form of "for loops" in MATLAB takes

```
for i=1:10
   disp(i);
end
```

Here `i` is the *index* or *iterator*, "`1:10`" is the *value array*, and everything between `for` and `end` are program statements that will be executed repeatedly for `i` iterating through each value in the value array.

"If control" in MATLAB

```
    if (1 > 0)
       disp(' 1 is greater than 0');
    else
        disp('this will never happen in this case')
    end
```

and while loops:

```
a = 1;
while a < 100
   disp(a);
   a = a*2; %MATLAB still doesnt't have *= and similar operators
end
```

The *expression* after `while` (here "`a<100`") or `if` is a logical scalar or a real numerics. The statements between `while` and `end` will be repeatedly executed until that *expression* is *false* or 0.

"`break`" and "`continue`" can terminate loops:

```
    % break and continue
    a = 0;
    while (1)
        a = a + 7;
        disp(a)
        if mod(a,5)==0
           break; %breaks the loop
        end
    end

    for i=1:100
       if i > 10
           continue; %skips the rest of the loop
       end
       disp(i)
    end
```

The "`switch`/`case`" switches among cases based on expression

```
a = 'str';
switch a % <-- a scalar or a string
    case 1,
        disp('one');
    case 2,
        disp('two');
    case 'str',
        disp(a);
    otherwise,
        disp('other');
end
```