MATLAB

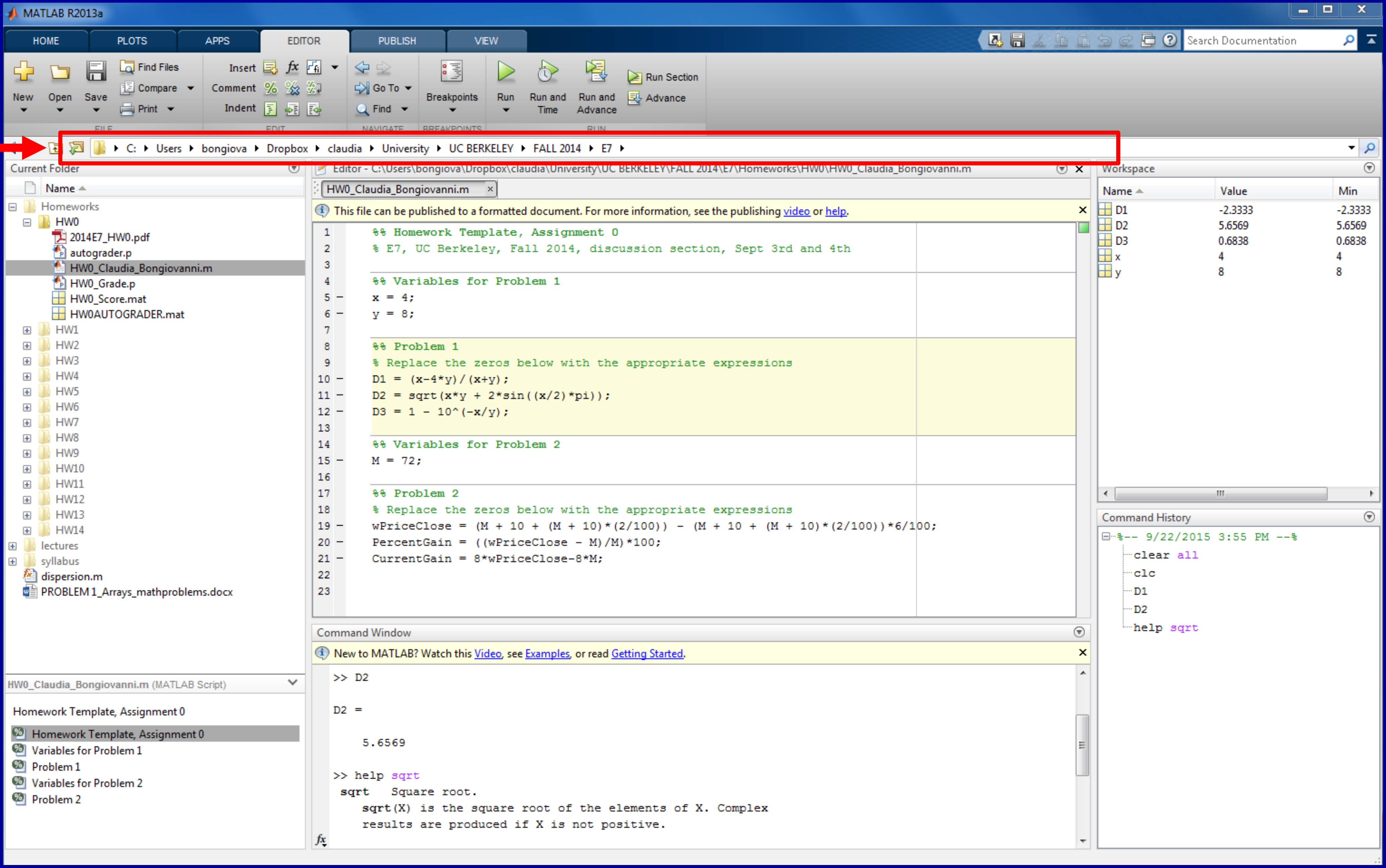
- You can download MatLab from Distrilog at http://distrilog.epfl.ch/main.aspx

- Q&A forum on MatLab can be found at http://ch.mathworks.com/matlabcentral/answers/

- You can find a video tutorial on MatLab at http://ch.mathworks.com/academia/student_center/tutorials/launchpad.html http://www.tutorialspoint.com/matlab/index.htm

The Default MATLAB Desktop (2013a)

Set the path!



Scalar Arithmetic Operations

Symbol	Operation	MATLAB form
^	exponentiation: a^b	`a^b`
*	multiplication: ab	`a*b`
/	right division: a/b	`a/b`
\	left division: b/a	`a\b`
+	addition: $a + b$	`a + b`
-	subtraction: $a - b$	`a - b`

Order of Precedence

- What do I really mean when I type

```
8 + 3 * 5 ^ 3
```

?

- Many possibilities:

```
(8 + 3) * (5 ^ 3)

8 + (3 * (5 ^ 3))

8 + ((3 * 5) ^ 3)
```

correct

Order of Precedence

PEMDAS

Precedence	Operation
First	what is inside Parentheses, evaluated starting with the innermost pair.
Second	Exponentiation, evaluated from ***left to right***.
Third	Multiplication and division with equal precedence, evaluated from ***left to right***.
Fourth	Addition and subtraction with equal precedence, evaluated from ***left to right***.

Examples of Variables and Assignment

- Writing these two line produces

```
>> A = sqrt(4);

>> A = tan(pi/4) + A

    A = 3
```

$$A \leftarrow \sqrt{4}$$

$$A \leftarrow \tan\left(\frac{\pi}{4}\right) + 2$$

- A semicolon at the end of the RHS expression suppresses the display.

- However, the assignment still takes place.

Relational Operators

Compare two expressions or variables

`==`	equal to	`a == b`
`>`	greater than	`a > b`
`<`	less than	`a < b`
`>=`	greater or equal	`a >= b`
`<=`	less than or equal	`a <= b`
`~=`	not equal	`a ~= b`

Relational operators return:

- `logical 1` **if expression is true**
- `logical 0` **if expression is false**

Relational Operators

Relational operators return:
- **`1` if expression is true**
- **`0` if expression is false**

```
>> 8 == 5
ans =
     0
```

- The result of the comparison is a value that can be used in an assignment.

- The precedence of relational operators is **<u>lower</u>** than that of addition and subtraction (***PEMDAS***).

Saving the Workspace

When you “quit” Matlab, the variables in the workspace are erased from memory.

If you need them for later use, you must save them.

```
>> save
```

saves all of the variables in the workspace into a file called **`matlab.mat`** (it is saved in the current directory)

Saving the Workspace

```
>> save Claudia
```

saves all of the variables in the workspace into a file called **`Claudia.mat`**

```
>> save Important A B C D*
```

saves the variables **A**, **B**, **C** and *any variable beginning with* **D** into a file called **`Important.mat`**

Loading from a `.mat` file

```
>> load Claudia
```

loads all of the variables from the file **`Claudia.mat`**

There are no known security problems with **`load`**.

Hence, you can safely send (as attachment), receive and use **`.mat`** files from others.

Loading Excel Files

```
>> xlsread('Claudia.xls')
>> xlsread('Claudia.xls','Sheet1','B10:F28')
```

Loading text Files

Create a .dat file with the following format

```
85.5, 54.0, 74.7, 34.2
63.0, 75.6, 46.8, 80.1
85.5, 39.6, 2.7, 38.7
```

```
>> dlmwrite('Claudia.dat',A,',')
>> csvread('Claudia.dat')
>> csvread('Claudia.dat',0,2)
```

Another option for reading a text file is

```
>> textread('Claudia.dat')
```

Introduction to arrays

- An array is an ordered collection of real numbers
- Arrays are the primary building blocks in MatLab

- Scalar

`a=[1]` (1 row, 1 column)

- Vector

`a=[1,-5, 3, 2]` (1 row, 4 columns)

- General 2D arrays

`a=[1.2, -3.2, 1.0; 3.1, 92, 0.0]` (2-by-3)

Defining arrays

Construction:

- Manual
- Incremental
- **`linspace`**
- transpose: “’ ”
- **`zeros`**
- **`ones`**
- **`rand/randn`**

Row vectors – incremental construction

```
>> r = 3 : 2 : 10

r =

    3    5    7    9
```

$9 + 2 > 10$

[3,5,7,9]

Syntax:

first element : increment : limit

Linspace command

- also creates a linearly spaced row vector
- ***number of elements*** are specified instead of increment

Syntax: `linspace(xf,xl,n)`

- **xf** – first element
- **xl** – last element
- **n** - number of *evenly-spaced* elements

```
>> A = linspace(3,9,4)

A =

     3     5     7     9
```

zeros

- Syntax: `zeros(n,m)`
- Create an array of zeros that has
  - **n** – rows
  - **m** - columns

```
>> r = zeros(1,3)
r =
     0     0     0
```

```
>> c = zeros(3,2)
c =
     0     0
     0     0
     0     0
```

ones

- Syntax: `ones(n,m)`
- Create an array of ***ones*** that has
- **n** – rows
- **m** - columns

```
>> r = ones(1,3)
r =
     1     1     1
```

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

`rand`

- Syntax: **`rand(n,m)`**
- Create an array of random numbers
- **`n`** – rows
- **`m`** - columns

uniform random distribution between 0 and 1

```
>> r = rand(2,3)
r =
        0.9501    0.2311    0.6068
        0.8147    0.1270    0.6324
```

Array *column* concatenation

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

```
>> B = zeros(2,2)
B =
    0     0
    0     0
```

```
>> C = [ A  B ]
C =
    1     1     1     0     0
    1     1     1     0     0
```

A and B ***must*** have the same number of rows

Array *row* concatenation

```
C =

     1     1     1     0     0
     1     1     1     0     0

r =

     3     3     3     3     3

>> D = [ C ; r ]

D =

     1     1     1     0     0
     1     1     1     0     0
     3     3     3     3     3
```

C and r ***must*** have the same number of columns

The transpose operator '

- The transpose operator converts
  - (row vector)' → (column vector)
  - (column vector)' → (row vector)
  - $\begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix}' \longrightarrow \begin{bmatrix} 1 & 3 \\ 2 & 4 \end{bmatrix}$

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

A =
    1    2
    3    4
```

```
>> B = A'

B =

    1    3

    2    4
```

size command

size(A) returns a 1 x 2 array that contains:

– number of rows of **A**

– number of columns **A**

Example:

```
>> A = rand(5,6);

>> size(A)

ans =

     5     6
```

```
>> d = size(A)

d =

     5     6
```

↑ # rows ↑ # columns

size command

- Syntax: `n = size(A,1)`
- **n** – number of rows of **A**
- Syntax: `m = size(A,2)`
- **m** – number of columns **A**

Example:

```
>> A = rand(5,6);
```

```
>> n = size(A,1)

n =

      5
```

```
>> m = size(A,2)

m =

      6
```

numel command – getting number of elements

- Syntax: `n = numel(A)`
- **n** – number of elements of **A**

Example:

```
>> r = ones(1,4)
r =
  1    1    1    1
>> n = numel(r)
n =
     4
```

```
>> A = ones(2,4)
A =
  1    1    1    1
  1    1    1    1
>> n = numel(A)
n =
     8
```

Accessing elements or parts of an array

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

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

`A(3,2)` is

– the element in the (3^{rd} row , 2^{th} column) of **A**

```
>> A(3,2)

ans =

     5
```

```
>> A(end,1)

ans =

     4
```

Find function

If **A** is an array,

- **find(A) returns a row vector containing the <u>linear indexes</u> of the <u>non-zero</u> elements of A**

- Example: (row vector)

```
>> A = [-3 0 1 5];

>> find(A)

ans =

     1  3  4
```

Using the find function

Example: Set all negative elements of array **A** to zero

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

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

```
>> Lindx = find(A < 0)
Lindx =
     2
     5
     6
>> A(Lindx) = 0
A =
     2     1
     0     0
     4     0
```

```
>> A = [2 1 ; -2 -3 ; 4 -5 ];
>> [rI,cI] = find(A < 0);
>> A(rI(i),cI(i)) = 0
>>>> A =
     2     1
     0     0
     4     0
```

Do 3X for i = 1, 2, 3

Question Indexing

B =

1	0	0
0	1	0
0	0	1

```
>> B(2,3)

       ans
               0

>> B(:, 2:3)

       ans
```

B =

0	0
1	0
0	1

Question

```
>> A = 13:-3:2 – 3; B = linspace(13,1,5);
>> isequal(A,B)
```

A. ans =

    1

B. ans =

    0

```
A =
    13    10     7     4     1

B =
    13    10     7     4     1
```

Review: Relational operators

Relational operators are used to compare variables.

There are 6 comparisons:

- “equal to”, using **==**
- “not equal to”, using **~=**
- “less than”, using **<**
- “less than or equal to”, using **<=**
- “greater than”, using **>**
- “greater than or equal to”, using **>=**

The result of a comparison is of class **`logical`**

Two values: true (1) or false (0),

Relational operations on vectors

- Example

```
>> A = [-3 2 1 5];
>> B = [ 1 2 5 1];
```

-3 = 1?   5 ≤1?

```
>> A == B

ans =

     0   1   0   0
```

```
>> A <= B

ans =

     1   1   1   0
```

Note: the result of relational operations are logical variables
1-true, 0-false

Logical Operators

If **A** and **B** are scalars (**double** or **logical**), then

A&B is TRUE (1) if **A** and **B** are both nonzero, otherwise it is FALSE (0). This is the logical **AND** operator.
A|B is TRUE (1) if either **A** or **B** are nonzero, otherwise it is FALSE (0). This is the logical OR operator.
xor(A,B) is TRUE (1) if one argument is 0 and the other is nonzero, otherwise it is FALSE (0). This is the logical EXCLUSIVE OR operator.
~A is TRUE if **A** is 0, and FALSE if **A** is nonzero. This is the logical NEGATION operator.

For arrays, the operations are applied element-wise

Indexing with logical arrays

$$-5 \le A(i,j) \le -2$$

Example

Set all elements in array **A** with values between **-5** and **-2** to zero

$$A = \begin{bmatrix} 2 & -3 \\ -5 & -1.9 \end{bmatrix}$$

```
>> A = [2 -3 ; -5 1.9];

>> Indx = A <= -2 & A >= -5
Indx =
     0     1
     1     0

>> A(Indx) = 0
A =
     2     0
     0    -1.9
```

(A <= -2) & (A >= -5)

Logical function any

```
>> L = [ 1 0 1 0 ; 1 0 0 0 ];
```

$$L = \begin{bmatrix} 1 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 \end{bmatrix}$$

```
>> A =  any(L)
A =
     1     0     1     0
```

`any(L)`: determines if *any* element in the column is nonzero

```
>> B =  any(L,2)
B =
     1
     1
```

`any(L,2)`: determines if any element in the row is nonzero

Logical function all

```
>> L = [ 1 0 1 0 ; 1 0 0 0 ];

>> C =  all(L)
C =
     1     0     0     0
```

$$L = \begin{bmatrix} 1 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 \end{bmatrix}$$

`all(L)`: determines if *all* elements in the column are <u>nonzero</u>

```
>> D =  all(L,2)
D =
     0
     0
```

`all(L,2)`: determines if all elements in the row are nonzero

Scalar and short circuit && (and) and || (or)

```
% Variables b and a must be scalars
% and defined

x = (b ~= 0) && (a/b > 18.5)
```

left expression is evaluated first

if `(b ~= 0)` is false (i.e. b = 0)

`x = false`

without evaluating right expression

otherwise

`(a/b > 18.5)` is evaluated and

the result is assigned to `x`

if and end statements

To conditionally control the execution of statements

```
if condition
     statements
end
```

- If **condition** is TRUE (or nonzero), the statements between the **if** and **end** are executed.
- Otherwise, they are not executed.
- Execution continues with any statements after the **end**.

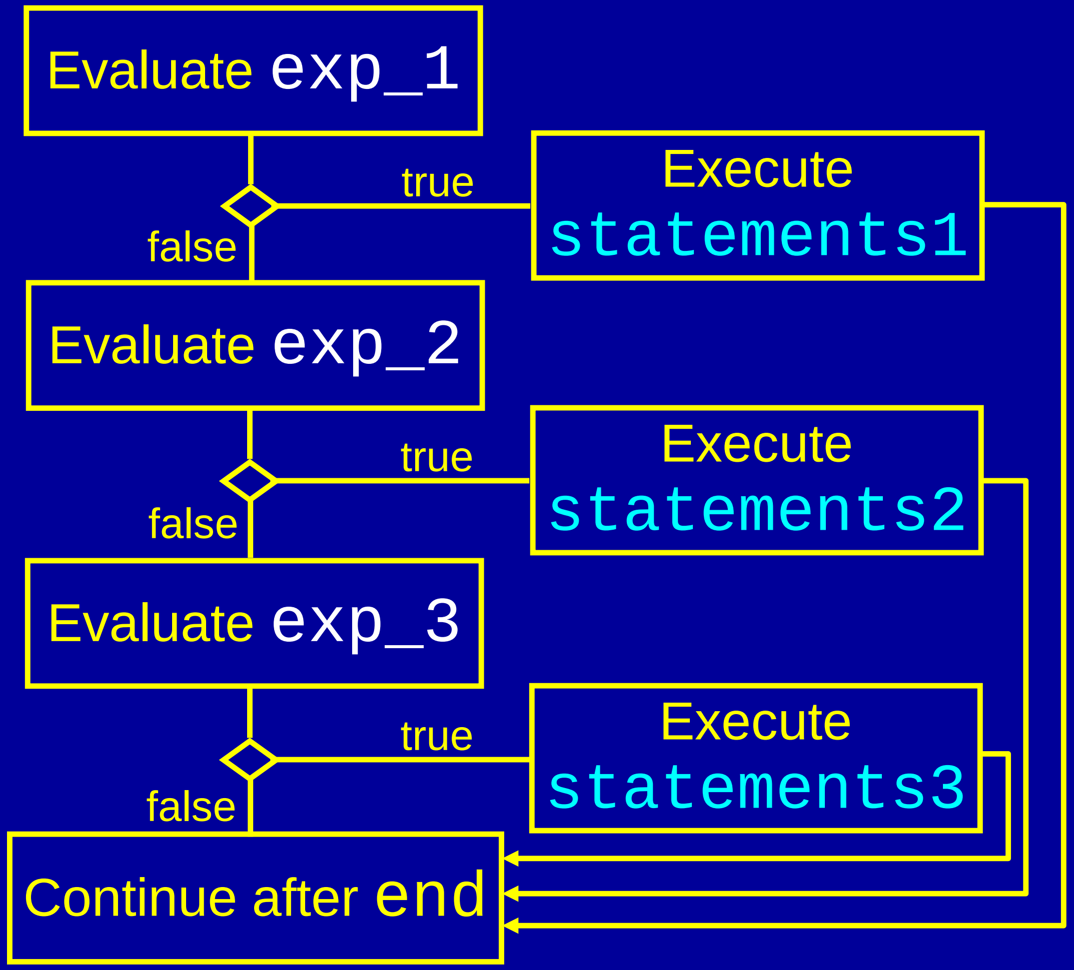
if, else and end statements

```
if condition
    statements1
else
    statements2
end
```

- If **condition** is TRUE, statements1 are executed.
- Otherwise, statements2 are executed.
- Execution continues with any statements after the **end**.

if, elseif and end statements

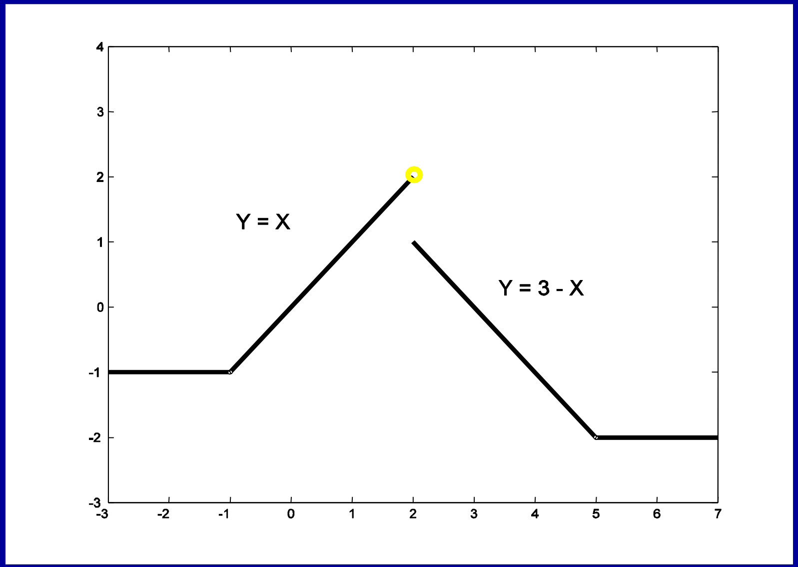
```
...
if exp_1
    statements1
elseif exp_2
    statements2
elseif exp_3
    statements3
end
more statements
```



Could have also used a single **else** before the **end**

Piecewise linear function

```
% check if x is a scalar and a double and a
% real number
if isscalar(x) && isa(x,'double') && isreal(x)
    if x < -1
        y = -1;
    elseif x < 2
        y = x;
    elseif x < 5
        y = 3 - x;
    else
        y = -2;
    end
else
    error('x should be a real scalar');
end
```



For Loops

The most common value for **array** is a row vector of integers, starting at , and increasing to a limit

```
for x = 1:n        array
   statements
end
```

The **array** is simply the row vector

[1 2 3 4 … n]

Hence, the **statements** are executed **n** times.

- The first time through, the value of **x** is set equal to **1**;
- the ’th time through, the value of **x** is set equal to **k**.

For Loops Example

```
for x = 1:3
    x
end
```

```
x = 1
x = 2
x = 3
```

```
for x = 1:3
   disp(['x is ' num2str(x)])
```

```
x is 1
x is 2
x is 3
```

```
A = [3 2 1];
for x = A
   disp(['x is ' num2str(x)])
end
```

```
x is 3
x is 2
x is 1
```

nested for loops

```
A = zeros(4,3)
for m = 1:4
    for n = [1 3]
        A(m,n) = m*n;
    end
end
```

outer

inner

A =

	n=1		n=3
m=1	1.00	0	3.00
m=2	2.00	0	6.00
m=3	3.00	0	9.00
m=4	4.00	0	12.00

Plot Function

If `x` and `y` are vectors (i.e., a row or column vector), of the same length, then

– ***plot***`(x,y)` plots the elements of y versus the elements of x

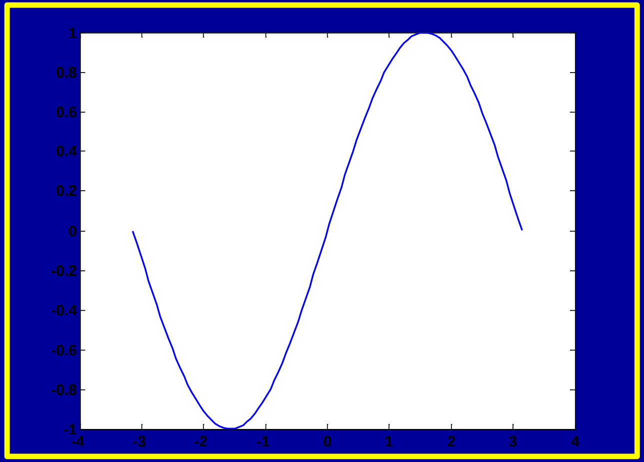
- `plot(y)` plots the elements of y versus its indexes

- (more later – see help for options)

Plot examples

Plot $sin(w)$ for w between $-pi$ and pi

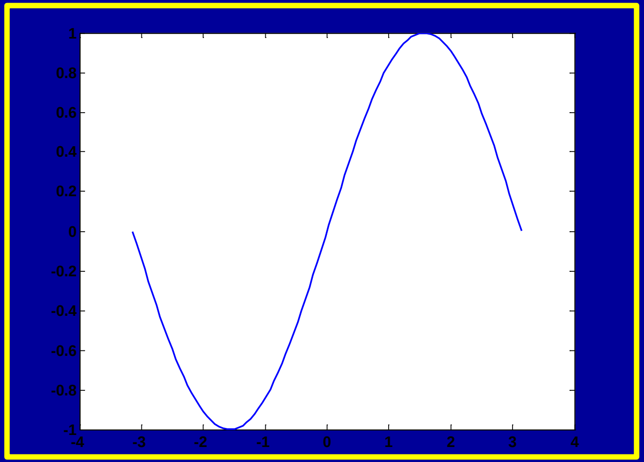
```
x = linspace(-pi,pi);
plot(x,sin(x))
```



Using **`for`** loop

```
x = linspace(-pi,pi);
y = zeros(size(x));

for k = 1 : size(x,2);
    y(k) = sin(x(k));
end
plot(x,y)
```

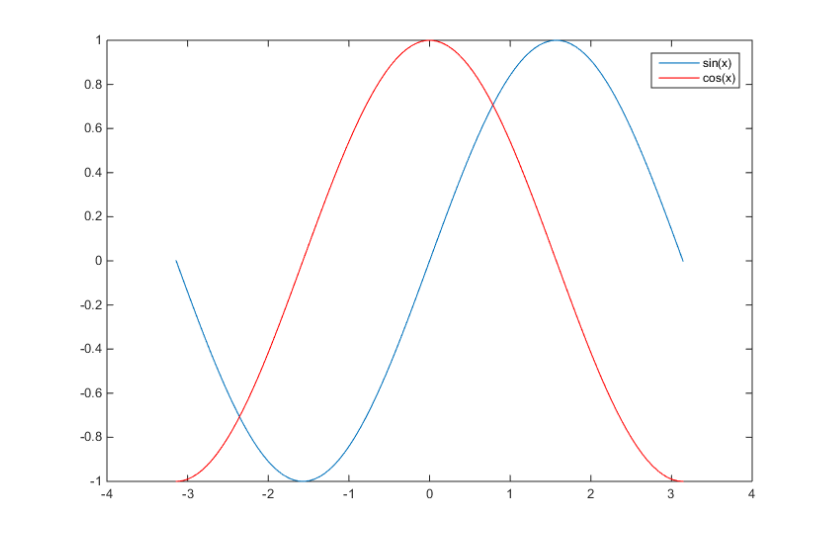


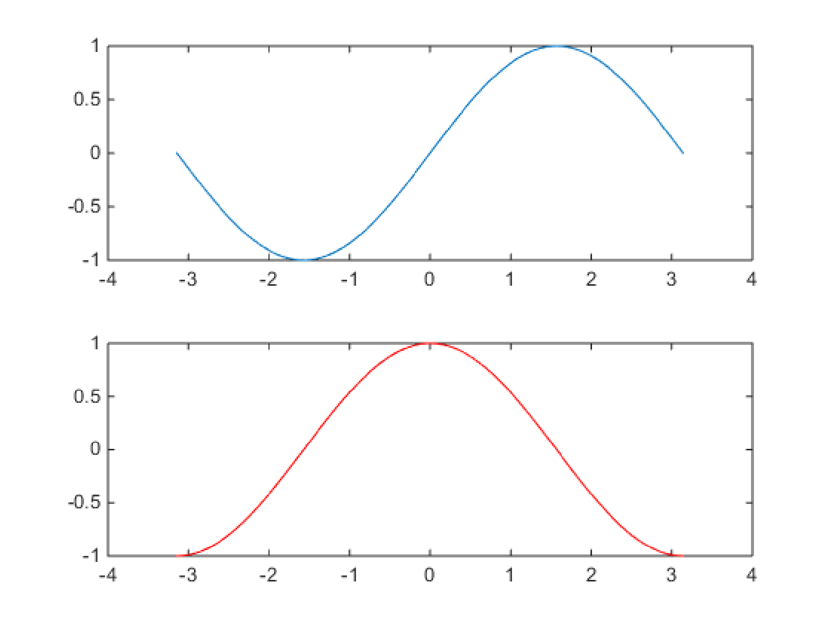
Plot multiple diagrams

```
x = linspace(-pi,pi);

plot(x,sin(x),'b');
hold on;
plot(x,cos(x),'r');

figure();
subplot(2,1,1);
plot(x,sin(x),'b');
subplot(2,1,2);
plot(x,cos(x),'r');
```



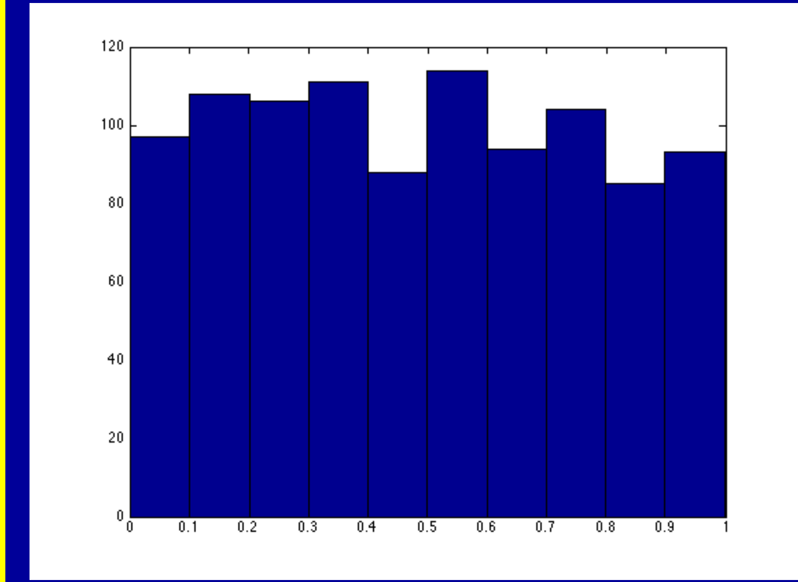


Question 5

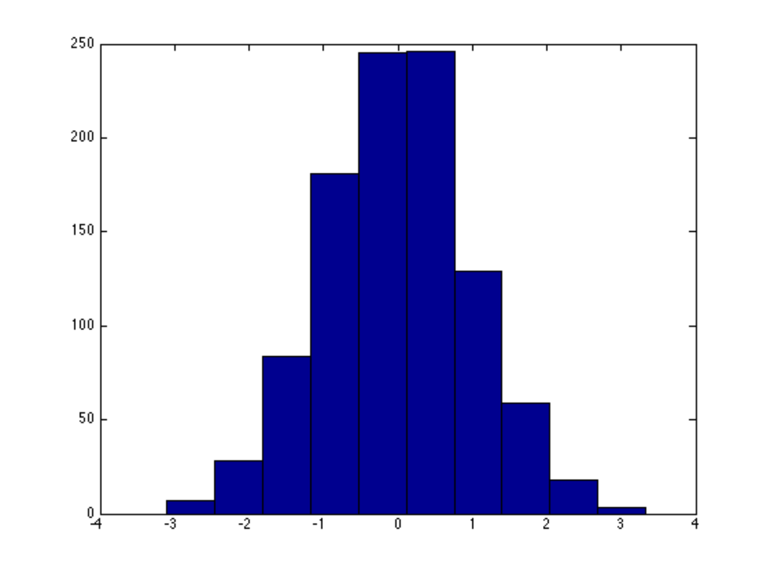
The function 'hist' bins your data and plots it as a histogram. What would the output be for the following code?

```
>> A = rand(1,1000);  hist(A)
```

A



B



Statistical Functions

If `A` is a matrix with size `n*m`, then

- ***mean***`(A,k)` **returns the average of matrix A in dimension k (i.e. if k=1 we get the average of all the columns)**
- ***min***`(A,[],k)`/***max***`(A,[],k)` **returns the minimum/maximum of matrix A in dimension k (i.e. if k=1 we get the minimum/maximum of all the columns)**
- ***var***`(A)` **returns a row vector with the variance of all the columns of matrix A**