MATLAB and its application in Engineering

Assoc. Prof. Kirin Shi

Shanghai Jiao Tong University

Review on Lecture 1

MATLAB Basics

Window	Purpose
Command Window	Main window, enters variables, runs programs.
Figure Window	Contains output from graphic commands.
Editor Window	Creates and debugs script and function files.
Help Window	Provides help information.
Command History Window	Logs commands entered in the Command Window.
Workspace Window	Provides information about the variables that are used.
Current Folder Window	Shows the files in the current folder.

Variable Assignment
&
Arithmetic Operations
(Scalar)

Variable Names-Rules

- Always start a variable name with a letter
- Can be up to 63 characters long
- Can contain letters, digits, and the underscore character
- MATLAB is case sensitive: it distinguishes between uppercase and lowercase letters

Variable Names-Rules

- Cannot contain punctuation characters (e.g., period, comma, semicolon)
- No spaces are allowed between characters
- Do **not** use MATLAB reserved words (such as **length**, **sum**, **end**, **pi**, **i** , **j** , **eps**, **sin**, **cos** and **size**).
- Names should convey some meaning (e.g., a diameter might be stored in a variable named **dia**).

Symbols for arithmetic operations

Operation	Symbol	Example
Addition	+	5 + 3
Subtraction	–	5 – 3
Multiplication	*	5 * 3
Right division	/	5 / 3
Left division	\	5 \ 3 = 3 / 5
Exponentiation	^	5 ^ 3 (means $5^3 = 125$)

Precedence of Operators

1. All bracketed expressions are evaluated first, starting from the **innermost** brackets and working out.
2. All **exponentiations** are evaluated, working **from left to right**.
3. All **multiplications** and **divisions** are evaluated, working from left to right.
4. All **additions** and **subtractions** are evaluated, working from left to right.

Vectors in MATLAB

Horizontal (Row) Vectors: Colon Notation

Creating a vector with constant spacing by specifying the first term, the spacing, and the last term:

```
variable_name = [m:q:n]
```

or

```
variable_name = m:q:n
```

(The brackets are optional.)

- The shorthand **m:q:n** means **m** to **n** in steps of **q**.
- **1:1:5** is equivalent to t **[1,2,3,4,5]**
- **1:5** is equivalent to t **[1,2,3,4,5]**
- **2:3:12** is equivalent to t **[2,5,8,11]**
- **8:-1:3** is equivalent to **[8,7,6,5,4,3]**.

Function linspace

- **linspace** is a built-in MATLAB function. **linspace(X1, X2, N)** generates **N** equally-spaced points between **X1** and **X**2, starting with **X1** and ending with **X2**.

>> x = linspace(0,6,5)

x =

0 1.5000 3.0000 4.5000 6.0000

Alternatively, >> x = 0:1.5:6

x =

0 1.5000 3.0000 4.5000 6.0000

Accessing elements of an array

- *va*(:) Refers to all the elements of the vector *va* (either a row or a column vector).
- *va*(*m*:*n*) Refers to elements *m* through *n* of the vector *va*.
- *va*(*m*:*q*:*n*) Refers to elements from *m to n with a step of q* .

```
>> v=[4 15 8 12 34 2 50 23 11]
v =
     4    15     8    12    34     2    50    23    11
>> u=v(3:7)
u =
     8    12    34     2    50
>>
```

A vector v is created.

A vector u is created from the elements 3 through 7 of vector v.

MATRIX

DEFINITION

The magic *colon* notation and *linspace*

```
>> A=[1:2:11; 0:5:25; linspace(10,60,6); 67 2 43 68 4 13]
A =
     1     3     5     7     9    11
     0     5    10    15    20    25
    10    20    30    40    50    60
    67     2    43    68     4    13
>>
```

Addressing a matrix via colon notation

- A(:,n) Refers to the elements in all the rows of column n of the matrix A.
- A(n,:) Refers to the elements in all the columns of row n of the matrix A.
- A(:,m:n) Refers to the elements in all the rows between columns m and n of the
  matrix A.
- A(m:n,:) Refers to the elements in all the columns between rows m and n of the
  matrix A.
- A(m:n,p:q) Refers to the elements in rows m through n and columns p through
  q of the matrix A.

Addressing a matrix via colon notation

```
>> A=[1 3 5 7 9 11; 2 4 6 8 10 12; 3 6 9 12 15 18; 4 8 12 16
20 24; 5 10 15 20 25 30]

A =
     1     3     5     7     9    11
     2     4     6     8    10    12
     3     6     9    12    15    18
     4     8    12    16    20    24
     5    10    15    20    25    30
>> B=A(:,3)
```

Define a matrix A with 5 rows and 6 columns.

Define a column vector B from the elements in all of the rows of column 3 in matrix A.

Addressing a matrix via colon notation

```
B =
     5
     6
     9
    12
    15
```

```
>> C=A(2,:)
```

```
C =
     2     4     6     8    10    12
```

Define a row vector C from the elements in all of the columns of row 2 in matrix A.

```
>> E=A(2:4,:)
```

```
E =
     2     4     6     8    10    12
     3     6     9    12    15    18
     4     8    12    16    20    24
```

Define a matrix E from the elements in rows 2 through 4 and all the columns in matrix A.

```
>> F=A(1:3,2:4)
```

```
F =
     3     5     7
     4     6     8
     6     9    12
```

Create a matrix F from the elements in rows 1 through 3 and columns 2 through 4 in matrix A.

Examples on manipulating matrixes

Given are a 5×6 matrix A, a 3×6 matrix B, and a 9-element vector v.

$$A = \begin{bmatrix} 2 & 5 & 8 & 11 & 14 & 17 \\ 3 & 6 & 9 & 12 & 15 & 18 \\ 4 & 7 & 10 & 13 & 16 & 19 \\ 5 & 8 & 11 & 14 & 17 & 20 \\ 6 & 9 & 12 & 15 & 18 & 21 \end{bmatrix}$$

$$B = \begin{bmatrix} 5 & 10 & 15 & 20 & 25 & 30 \\ 30 & 35 & 40 & 45 & 50 & 55 \\ 55 & 60 & 65 & 70 & 75 & 80 \end{bmatrix}$$

$$v = \begin{bmatrix} 99 & 98 & 97 & 96 & 95 & 94 & 93 & 92 & 91 \end{bmatrix}$$

Create the three arrays in the Command Window,
by writing **one command**:

1.replace the last four columns of the first and third rows of *A* with the first four columns of the first two rows of *B*,

2.replace the last four columns of the fourth row of *A* with the elements 5 through 8 of *v*,

3.replace the last four columns of the fifth row of *A* with columns 2 through 5 of the third row of *B*.

Step 1

```
>> A=[2:3:17; 3:3:18; 4:3:19; 5:3:20; 6:3:21]

A =

     2     5     8    11    14    17
     3     6     9    12    15    18
     4     7    10    13    16    19
     5     8    11    14    17    20
     6     9    12    15    18    21

>> B=[5:5:30; 30:5:55; 55:5:80]

B =

     5    10    15    20    25    30
    30    35    40    45    50    55
    55    60    65    70    75    80
```

A([1 3], 3:6)=B([1 2], 1:4)

Step 2

```
>> A=[2:3:17; 3:3:18; 4:3:19; 5:3:20; 6:3:21]
A =
     2     5     8    11    14    17
     3     6     9    12    15    18
     4     7    10    13    16    19
     5     8    11    14    17    20
     6     9    12    15    18    21
>> v=[99:-1:91]
v =
    99    98    97    96    95    94    93    92    91
```

A(4, 3:6)=v(5:8)

Step 3

```
>> A=[2:3:17; 3:3:18; 4:3:19; 5:3:20; 6:3:21]
A =
     2     5     8    11    14    17
     3     6     9    12    15    18
     4     7    10    13    16    19
     5     8    11    14    17    20
     6     9    12    15    18    21
>> B=[5:5:30; 30:5:55; 55:5:80]
B =
     5    10    15    20    25    30
    30    35    40    45    50    55
    55    60    65    70    75    80
```

A(5, 3:6)=B(3, 2:5)

Result

A([1 3], 3:6)=B([1 2], 1:4)

A(4, 3:6)=v(5:8)

A(5, 3:6)=B(3, 2:5)

A([1 3 4 5],3:6)=[B([1 2],1:4); v(5:8); B(3,2:5)]

A =

2	5	5	10	15	20
3	6	9	12	15	18
4	7	30	35	40	45
5	8	95	94	93	92
6	9	60	65	70	75

A =

2	5	8	11	14	17
3	6	9	12	15	18
4	7	10	13	16	19
5	8	11	14	17	20
6	9	12	15	18	21

Elementary Matrix Operations

ADDITION AND SUBTRACTION

- The operations + (addition) and – (subtraction) can be used to add (subtract) arrays of identical size (the same numbers of rows and columns) and to add (subtract) a scalar to an array.
- When two arrays are involved the sum, or the difference, of the arrays is obtained by adding, or subtracting, their corresponding elements.

$$A = \begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix} \qquad B = \begin{bmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{bmatrix}$$

$$C = A * B = \begin{bmatrix} a_{11}b_{11} + a_{12}b_{21} & a_{11}b_{12} + a_{12}b_{22} \\ a_{21}b_{11} + a_{22}b_{21} & a_{21}b_{12} + a_{22}b_{22} \end{bmatrix}$$

```
>> A=[5 -3 8; 9 2 10]

A =

     5    -3     8
     9     2    10

>> B=[10 7 4; -11 15 1]

B =

    10     7     4
   -11    15     1

>> A-B

ans =

     -5   -10     4
     20   -13     9

>> C=A+B

C =

    15     4    12
    -2    17    11

>> VectA+A

??? Error using ==> plus
Matrix dimensions must agree.
```

Define two 2×3 matrices A and B.

Subtracting matrix B from matrix A.

Define a matrix C that is equal to A + B.

Trying to add arrays of different size.

An error message is displayed.

ARRAY MULTIPLICATION

- The multiplication operation * is executed by MATLAB according to the rules of linear algebra.
- This means that if A and B are two matrices, the operation A*B can be carried out only if the number of columns in matrix A is equal to the number of rows in matrix B.

$$A = \begin{bmatrix} A_{11} & A_{12} & A_{13} \\ A_{21} & A_{22} & A_{23} \\ A_{31} & A_{32} & A_{33} \\ A_{41} & A_{42} & A_{43} \end{bmatrix} \quad \text{and} \quad B = \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \\ B_{31} & B_{32} \end{bmatrix}$$

$$\begin{bmatrix} (A_{11}B_{11} + A_{12}B_{21} + A_{13}B_{31}) & (A_{11}B_{12} + A_{12}B_{22} + A_{13}B_{32}) \\ (A_{21}B_{11} + A_{22}B_{21} + A_{23}B_{31}) & (A_{21}B_{12} + A_{22}B_{22} + A_{23}B_{32}) \\ (A_{31}B_{11} + A_{32}B_{21} + A_{33}B_{31}) & (A_{31}B_{12} + A_{32}B_{22} + A_{33}B_{32}) \\ (A_{41}B_{11} + A_{42}B_{21} + A_{43}B_{31}) & (A_{41}B_{12} + A_{42}B_{22} + A_{43}B_{32}) \end{bmatrix}$$

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

Define a 4 × 3 matrix A.

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

Define a 3 × 2 matrix B.

```
>> C=A*B
C =
    28    27
    65    49
    98    32
    84    38
```

Multiply matrix A by matrix B and assign the result to variable C.

```
>> D=B*A
??? Error using ==> *
Inner matrix dimensions must agree.
```

Trying to multiply B by A, B*A, gives an error since the number of columns in B is 2 and the number of rows in A is 4.

```
>> F=[1 3; 5 7]
F =
     1     3
     5     7
>> G=[4 2; 1 6]
```

Define two 2 × 2 matrices F and G.

```
G =
     4     2
     1     6
>> F*G
ans =
     7    20
    27    52
>> G*F
ans =
    14    26
    31    45
>> AV=[2 5 1]
AV =
     2     5     1
>> BV=[3; 1; 4]
BV =
     3
     1
     4
>> AV*BV
ans =
    15
>> BV*AV
ans =
     6    15     3
     2     5     1
     8    20     4
```

Multiply F*G

Multiply G*F

Note that the answer for G*F is not the same as the answer for F*G.

Define a three-element row vector AV.

Define a three-element column vector BV.

Multiply AV by BV. The answer is a scalar. (Dot product of two vectors.)

Multiply BV by AV. The answer is a 3 × 3 matrix.

ARRAY MULTIPLICATION

```
>> A = [1 2 3; 3 4 5];
>> A+5
ans =
     6     7     8
     8     9    10
```

>> 5*A (Alternatively, you can use **5.*A**)

```
ans =
     5    10    15
    15    20    25
```

Systems of Linear Equations

$$\begin{aligned} 2X_1 + 3X_2 + X_3 &= 4 \\ X_2 + 2X_3 &= 6 \\ 4X_1 + 2X_2 &= 12 \end{aligned}$$

can be represented as

$$\begin{bmatrix} 2 & 3 & 1 \\ 0 & 1 & 2 \\ 4 & 2 & 0 \end{bmatrix} \begin{bmatrix} X_1 \\ X_2 \\ X_3 \end{bmatrix} = \begin{bmatrix} 4 \\ 6 \\ 12 \end{bmatrix}$$

Systems of Linear Equations

$$
\begin{aligned}
2X_1 &+ 3X_2 &+ X_3 &= 4 \\
&\quad X_2 &+ 2X_3 &= 6 \\
4X_1 &+ 2X_2 & &= 12
\end{aligned}
$$

or, in a more compact form as

$$A\,X = B$$

$$A = \begin{bmatrix} 2 & 3 & 1 \\ 0 & 1 & 2 \\ 4 & 2 & 0 \end{bmatrix}; \quad X = \begin{bmatrix} X_1 \\ X_2 \\ X_3 \end{bmatrix}; \quad B = \begin{bmatrix} 4 \\ 6 \\ 12 \end{bmatrix}$$

ARRAY DIVISION

- The division operation is also associated with the rules of linear algebra.
- Identity matrix
- Inverse of a matrix
- Determinants
- Identity matrix:
  e.g.

```
>> eye(4)
I=
     1     0     0     0
     0     1     0     0
     0     0     1     0
     0     0     0     1
```

Matrix Inversion

A is a n-by-n square matrix.

If there is another n-by-n matrix **B** such that

$$\mathbf{A}\,\mathbf{B} = \mathbf{B}\,\mathbf{A} = \mathbf{I}$$

where I is the n-by-n identity matrix

非奇异矩阵

then **A** is called non-singular and **B** is called the inverse of A and is usually denoted by $\mathbf{A}^{-1}$.

Matrix Inversion

```
>> A = [2 3 1; 0 1 2; 4 2 0];
>> A_inverse = inv(A)
```

```
A_inverse =

   -0.3333    0.1667    0.4167
    0.6667   -0.3333   -0.3333
   -0.3333    0.6667    0.1667
```

Matrix Inversion

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

>> A_inverse = inv(A)

>> C = A*A_inverse

C =

    1.0000   -0.0000   -0.0000

         0    1.0000         0

         0         0    1.0000
```

Try:

```
>> A^-1
```

Matrix Inversion

```
>> A=[2 1 4; 4 1 8; 2 -1 3]
A =
     2     1     4
     4     1     8
     2    -1     3
>> B=inv(A)
B =
    5.5000   -3.5000    2.0000
    2.0000   -1.0000         0
   -3.0000    2.0000   -1.0000
>> A*B
ans =
     1     0     0
     0     1     0
     0     0     1
>> A*A^-1
ans =
     1     0     0
     0     1     0
     0     0     1
```

Creating the matrix A.

Use the inv function to find the inverse of A and assign it to B.

Multiplication of A and B gives the identity matrix.

Use the power –1 to find the inverse of A. Multiplying it by A gives the identity matrix.

Determinants

行列式

$$|A| = \begin{vmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{vmatrix} = a_{11}a_{22} - a_{12}a_{21}$$

$$\begin{vmatrix} 6 & 5 \\ 3 & 9 \end{vmatrix} = 6 \cdot 9 - 5 \cdot 3 = 39$$

```
>> A=[6 5;3 9];
>> det(A)
ans =

    39
```

Array division

Left division, \ :

Left division is used to solve the matrix equation AX=B

$$A^{-1}AX = A^{-1}B \qquad X = A^{-1}B$$

$$A^{-1}AX = IX = X \qquad X = A\backslash B$$

Systems of Linear Equations

- Left division “\” revoke a technique called **Gaussian Elimination**（高斯消去法）.
- **Gaussian Elimination** is a faster and **more stable** way of solving a set of linear equations.

Systems of Linear Equations

```
>> A = [2 3 1; 0 1 2; 4 2 0];
>> B = [4; 6; 12];
>> X = A\B
X =
    4.6667
   -3.3333
    4.6667
```

```
>> A*X
ans =
    4.0000
    6.0000
   12.0000
```

Systems of Linear Equations

If matrix A is **non-singular,** there would be an unique solution (**$X = A^{-1} * B$**)

```
>> A = [2 3 1; 0 1 2; 4 2 0];
>> B = [4; 6; 12];
>> A_inverse = inv(A);
>> X = A_inverse * B
```

$$\begin{bmatrix} 2 & 3 & 1 \\ 0 & 1 & 2 \\ 4 & 2 & 0 \end{bmatrix} \begin{bmatrix} X_1 \\ X_2 \\ X_3 \end{bmatrix} = \begin{bmatrix} 4 \\ 6 \\ 12 \end{bmatrix}$$

The left division method is recommended for solving a set of linear equations because the calculation of the inverse may be less accurate than the Gauss elimination method when large matrices are involved

Array division

Right division, / :

Right division is used to solve the matrix equation XC=D

$$X \cdot CC^{-1} = D \cdot C^{-1}$$

$$X = D \cdot C^{-1} \qquad X = D/C$$

Array division- Example

Use matrix operations to solve the following system of linear equations

$$4x - 2y + 6z = 8$$
$$2x + 8y + 2z = 4$$
$$6x + 10y + 3z = 0$$

Left division method, \ :

$$AX = B$$

$$\begin{bmatrix} 4 & -2 & 6 \\ 2 & 8 & 2 \\ 6 & 10 & 3 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \end{bmatrix} = \begin{bmatrix} 8 \\ 4 \\ 0 \end{bmatrix}$$

Array division- Example

Use matrix operations to solve the following system of linear equations

$$4x - 2y + 6z = 8$$
$$2x + 8y + 2z = 4$$
$$6x + 10y + 3z = 0$$

```
>> A=[4 -2 6; 2 8 2; 6 10 3];
>> B=[8; 4; 0];
>> X=A\B
X =
   -1.8049
    0.2927
    2.6341
>> Xb=inv(A)*B
Xb =
   -1.8049
    0.2927
    2.6341
```

Solving the form $AX = B$.

Solving by using left division: $X = A \setminus B$.

Solving by using the inverse of A: $X = A^{-1}B$.

Array division- Example

Use matrix operations to solve the following system of linear equations

$$4x - 2y + 6z = 8$$
$$2x + 8y + 2z = 4$$
$$6x + 10y + 3z = 0$$

Right division method, /:

$$XC = D \qquad [x\ y\ z]\begin{bmatrix} 4 & 2 & 6 \\ -2 & 8 & 10 \\ 6 & 2 & 3 \end{bmatrix} = [8\ 4\ 0]$$

Array division- Example

Use matrix operations to solve the following system of linear equations

$$4x - 2y + 6z = 8$$
$$2x + 8y + 2z = 4$$
$$6x + 10y + 3z = 0$$

```
>> C=[4 2 6; -2 8 10; 6 2 3];
>> D=[8 4 0];
>> Xc=D/C
Xc =
   -1.8049    0.2927    2.6341
>> Xd=D*inv(C)
Xd =
   -1.8049    0.2927    2.6341
```

Solving the form $XC = D$.

Solving by using right division: $X = D/C$.

Solving by using the inverse of C: $X = D \cdot C^{-1}$.

Element-by-element operations

- When the regular symbols for **“*” “\”** are used with arrays, the mathematical operations follow the **rules of linear algebra**.
- **“+” “–”** are element-by-element operations since when two arrays Element-by-element operations can be done only with arrays of the same size.
- Element-by-element **“*” “\” or “^”** of two vectors or matrices is entered by typing a period in front of the arithmetic operator.

Symbol	Description		Symbol	Description
.*	Multiplication		./	Right division
.^	Exponentiation		.\	Left Division

Element-by-element operations

E.g. $a = \begin{bmatrix} a_1 & a_2 & a_3 & a_4 \end{bmatrix} \quad b = \begin{bmatrix} b_1 & b_2 & b_3 & b_4 \end{bmatrix}$

$$a \,.*\, b = \begin{bmatrix} a_1 b_1 & a_2 b_2 & a_3 b_3 & a_4 b_4 \end{bmatrix}$$

$$a \,./\, b = \begin{bmatrix} a_1 / b_1 & a_2 / b_2 & a_3 / b_3 & a_4 / b_4 \end{bmatrix}$$

$$a \,.\hat{}\, b = \begin{bmatrix} (a_1)^{b_1} & (a_2)^{b_2} & (a_3)^{b_3} & (a_4)^{b_4} \end{bmatrix}$$

Element-by-element operations

E.g.

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

$$A .* B = \begin{bmatrix} A_{11}B_{11} & A_{12}B_{12} & A_{13}B_{13} \\ A_{21}B_{21} & A_{22}B_{22} & A_{23}B_{23} \\ A_{31}B_{31} & A_{32}B_{32} & A_{33}B_{33} \end{bmatrix} \qquad \mathrm{A} .\hat{} \, n = \begin{bmatrix} (A_{11})^n & (A_{12})^n & (A_{13})^n \\ (A_{21})^n & (A_{22})^n & (A_{23})^n \\ (A_{31})^n & (A_{32})^n & (A_{33})^n \end{bmatrix}$$

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

Element-by-element operations

E.g.

```
>> A=[2 6 3; 5 8 4]
A =
     2     6     3
     5     8     4
>> B=[1 4 10; 3 2 7]
B =
     1     4    10
     3     2     7
>> A.*B
ans =
     2    24    30
    15    16    28
>> C=A./B
C =
    2.0000    1.5000    0.3000
    1.6667    4.0000    0.5714
```

Define a 2×3 array A.

Define a 2×3 array B.

Element-by-element multiplication of array A by B.

Element-by-element division of array A by B. The result is assigned to variable C.

Element-by-element operations

E.g.

```
>> A=[2 6 3; 5 8 4]
A =
     2     6     3
     5     8     4
>> B=[1 4 10; 3 2 7]
B =
     1     4    10
     3     2     7
>> B.^3
ans =
     1    64  1000
    27     8   343
>> A*B
??? Error using ==> *
Inner matrix dimensions must agree.
```

Define a 2×3 array A.

Define a 2×3 array B.

Element-by-element exponentiation of array B. The result is an array in which each term is the corresponding term in B raised to the power of 3.

Trying to multiply A*B gives an error since A and B cannot be multiplied according to linear algebra rules. (The number of columns in A is not equal to the number of rows in B.)

Applications

- Element-by-element calculations are very useful for calculating the value of a function at many values of its argument.
- This is done by first defining a vector that contains values of the independent variable
- then using this vector in element-by-element computations to create a vector in which each element is the corresponding value of the function.

$$y = x^2 + 4x \quad x = 1,2 \ldots 8$$

```
>> x=[1:8]

x =
   1   2   3   4   5   6   7   8

>> y=x.^2-4*x

y =
  -3  -4  -3   0   5  12  21  32
>>
```

Create a vector x with eight elements.

Vector x is used in element-by-element calculations of the elements of vector y.

Applications

$$y = \frac{z^3 + 5z}{4z^2 - 10} \quad z = 1,3,5 \ldots 11$$

```
>> z=[1:2:11]
z =
     1     3     5     7     9    11
>> y=(z.^3 + 5*z)./(4*z.^2 - 10)

 y =
   -1.0000    1.6154    1.6667    2.0323    2.4650    2.9241
```

Create a vector z with six elements.

Vector z is used in element-by-element calculations of the elements of vector y.

Applications-with build-in functions

```
>> x=[0:pi/6:pi]
x =
    0    0.5236    1.0472    1.5708    2.0944    2.6180    3.1416
>>y=cos(x)
y =
  1.0000    0.8660    0.5000    0.0000   -0.5000   -0.8660   -1.0000
```

```
>> d=[1 4 9; 16 25 36; 49 64 81]
d =
     1     4     9
    16    25    36
    49    64    81
>> h=sqrt(d)
h =
     1     2     3
     4     5     6
     7     8     9
```

Creating a 3×3 array.

h is a 3×3 array in which each element is the square root of the corresponding element in array d.

Build-in functions

Function	Description	Example

`mean(A)`	If `A` is a vector, returns the mean value of the elements of the vector.	`>> A=[5 9 2 4];`
		`>> mean(A)`
		`ans =`
		`5`

`C=max(A)`	If `A` is a vector, `C` is the largest element in `A`. If `A` is a matrix, `C` is a row vector containing the largest element of each column of `A`.	`>> A=[5 9 2 4 11 6 11 1];`
		`>> C=max(A)`
		`C =`
		`11`

`[d,n]=max(A)`	If `A` is a vector, `d` is the largest element in `A`, and `n` is the position of the element (the first if several have the max value).	`>> [d,n]=max(A)`
		`d =`
		`11`
		`n =`
		`5`

Build-in functions

`min(A)`
 	
`[d,n]=min(A)`
	The same as `max(A)`, but for the smallest element.
	
	The same as `[d,n]= max(A)`, but for the smallest element.
		`>> A=[5 9 2 4];`
		`>> min(A)`
		`ans =`
		`2`

`sum(A)`	If `A` is a vector, returns the sum of the elements of the vector.	`>> A=[5 9 2 4];`
		`>> sum(A)`
		`ans =`
		`20`

`sort(A)`	If `A` is a vector, arranges the elements of the vector in ascending order.	`>> A=[5 9 2 4];`
		`>> sort(A)`
		`ans =`
		`2 4 5 9`

`median(A)`	If `A` is a vector, returns the median value of the elements of the vector.	`>> A=[5 9 2 4];`
		`>> median(A)`
		`ans =`
		`4.5000`

Build-in functions

Function	Description	Example

`std(A)`	If `A` is a vector, returns the standard deviation of the elements of the vector.	`>> A=[5 9 2 4];`
		`>> std(A)`
		`ans =`
		`2.9439`

`det(A)`	Returns the determinant of a square matrix `A`.	`>> A=[2 4; 3 5];`
		`>> det(A)`
		`ans =`
		`-2`

`dot(a,b)`	Calculates the scalar (dot) product of two vectors `a` and `b`. The vectors can each be row or column vectors.	`>> a=[1 2 3];`
		`>> b=[3 4 5];`
		`>> dot(a,b)`
		`ans =`
		`26`

Build-in functions

`cross(a,b)`	Calculates the cross product of two vectors `a` and `b`, (`axb`). The two vectors must have each three elements.	`>> a=[1 3 2];`
		`>> b=[2 4 1];`
		`>> cross(a,b)`
		`ans =`
		`-5 3 -2`

`inv(A)`	Returns the inverse of a square matrix `A`.	`>> A=[2 -2 1; 3 2 -1; 2 -3 2];`
		`>> inv(A)`
		`ans =`
		`0.2000 0.2000 0`
		`-1.6000 0.4000 1.0000`
		`-2.6000 0.4000 2.0000`

Build-in functions

Command	Description	Example

`rand`	Generates a single random number between 0 and 1.	`>> rand`
		`ans =`
		`0.2311`

`rand(1,n)`	Generates an n-element row vector of random numbers between 0 and 1.	`>> a=rand(1,4)`
		`a =`
		`0.6068 0.4860 0.8913 0.7621`

`rand(n)`	Generates an n × n matrix with random numbers between 0 and 1.	`>> b=rand(3)`
		`b =`
		`0.4565 0.4447 0.9218`
		`0.0185 0.6154 0.7382`
		`0.8214 0.7919 0.1763`

`rand(m,n)`	Generates an m × n matrix with random numbers between 0 and 1.	`>> c=rand(2,4)`
		`c =`
		`0.4057 0.9169 0.8936 0.3529`
		`0.9355 0.4103 0.0579 0.8132`

`randperm(n)`	Generates a row vector with n elements that are random permutation of integers 1 through n.	`>> randperm(8)`
		`ans =`
		`8 2 7 4 3 6 5 1`

Random Samples from a Uniform Distribution between zero and one 均匀分布

```
>> xx = rand(3,5)
```

xx =

0.6721	0.6813	0.5028	0.3046	0.6822
0.8381	0.3795	0.7095	0.1897	0.3028
0.0196	0.8318	0.4289	0.1934	0.5417

Random Samples from a Uniform Distribution between a and b

```
>> a = 10;

>> b = 85;

>> xx = rand(3,5);

>> yy = (b-a)*xx + a
```

yy =

60.4103	61.0958	47.7110	32.8463	61.1667
72.8589	38.4611	63.2104	24.2240	32.7073
11.4730	72.3847	42.1669	24.5073	50.6255

Random Integers between -20 & 150

- First simulate random numbers between -20 and 150

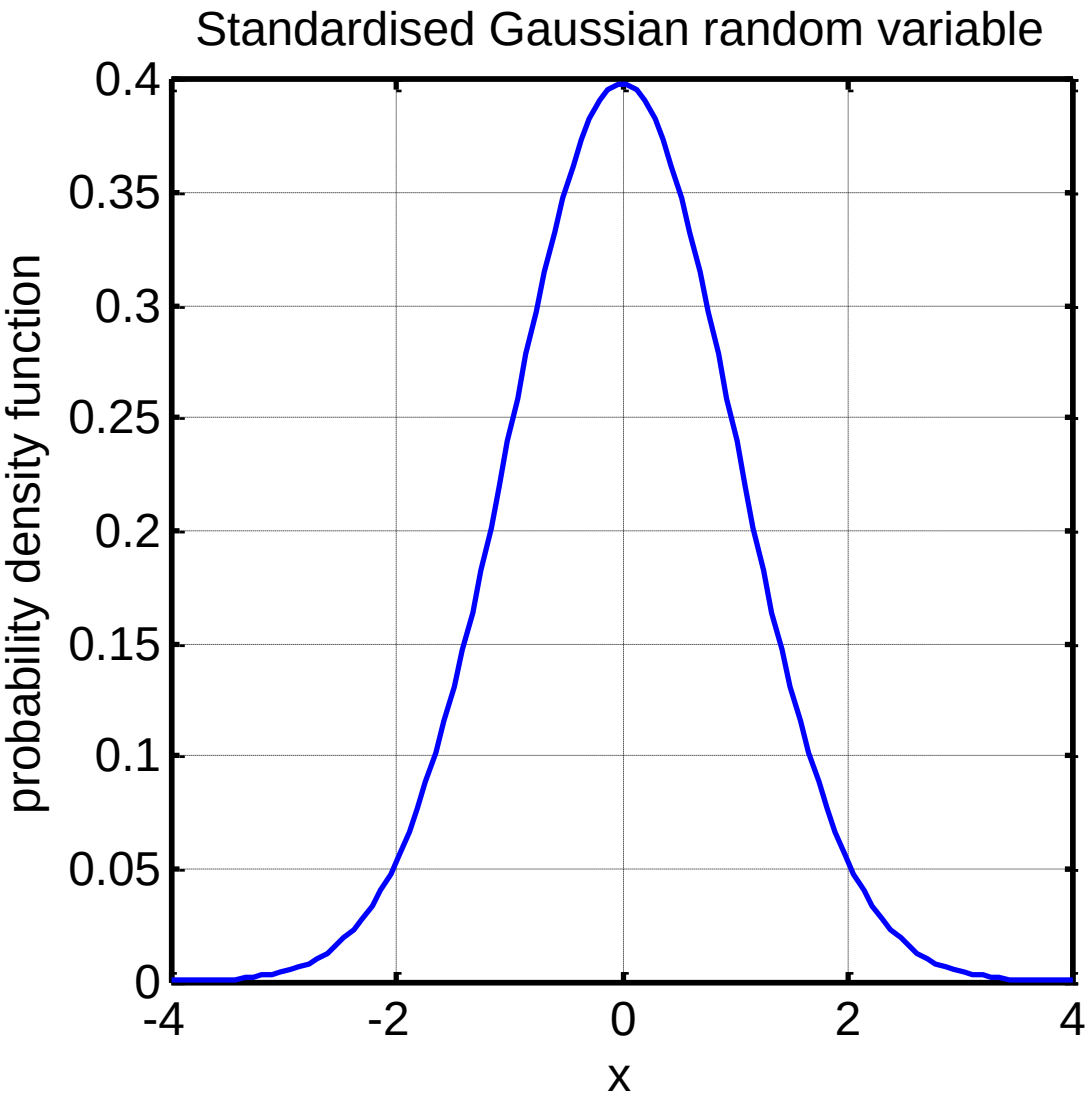
```
>> a = -20;

>> b = 150;

>> xx = rand(3,5);

>> yy = (b-a)*xx+a
```

```
yy =

 72.6372   126.7859  100.3727   15.7428   91.0830

121.2067  123.1127  102.5977   88.2388  136.0464

109.0900   -12.1738   23.8156   -4.1077   77.3836
```

- Then use the function round to convert the real numbers into integers.

```
>> zz = round(yy)

zz =

  73   127   100    16    91
 121   123   103    88   136
 109   -12    24    -4    77
```

Standardised Gaussian （标准正态分布）(Normal) random variable



E(X) = 0;

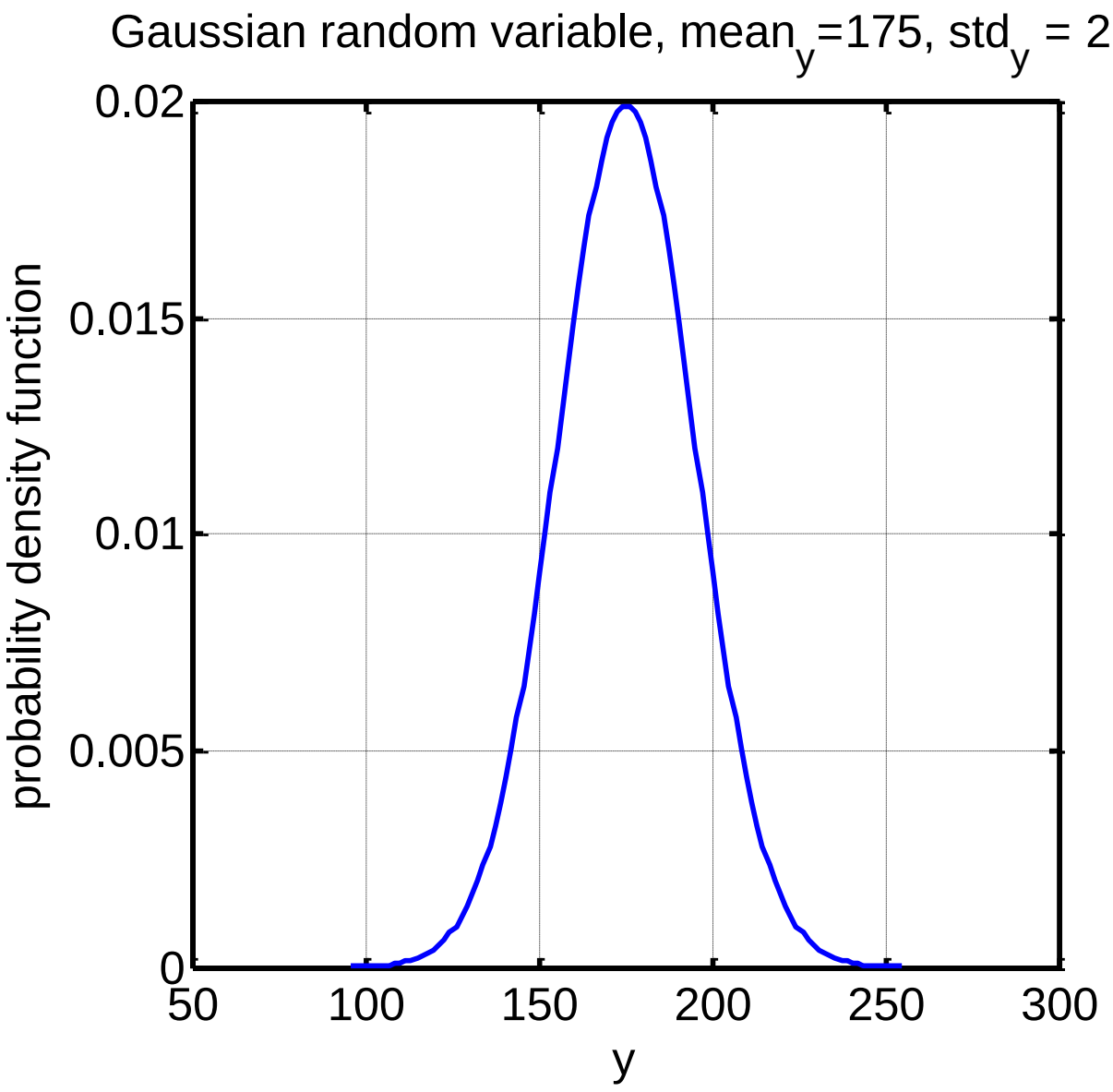
σ(X) = 1;

Random Samples from a Standardised Gaussian Distribution

```
>> xx = randn(3,5)
```

```
xx =

    2.0211    0.2723   -1.6106   -2.0889   -0.1624
    0.5018    0.3368   -1.0075    1.0461   -0.1758
   -1.9983    0.1378   -0.5144    0.2153    1.1022
```

Gaussian random variable



mean_y = 175;

sigma_y = 20;

Random Samples from a Gaussian Distribution

>> mean_y = 175;

>> sigma_y = 20;

- First simulate random numbers from a standardised Gaussian random variable.

>> x = randn(3,5)

x =

0.8979 -0.0901 -0.0858 -1.4038 0.2215

0.5058 0.6481 -0.3083 -0.7092 -1.3326

0.3299 1.7347 1.4596 0.6330 0.7305

Random Samples from a Gaussian Distribution

- Then use the following relationship to change the standard deviation and the mean of the random variable.

>> y = sigma_y*x + mean_y

$$\sigma = \sqrt{\frac{1}{N}\sum_{i=1}^{N}(x_i - \mu)^2}$$

y =

192.9581 173.1973 173.2833 146.9244 179.4303

185.1164 187.9621 168.8337 160.8168 148.3489

181.5986 209.6938 204.1930 187.6603 189.6097

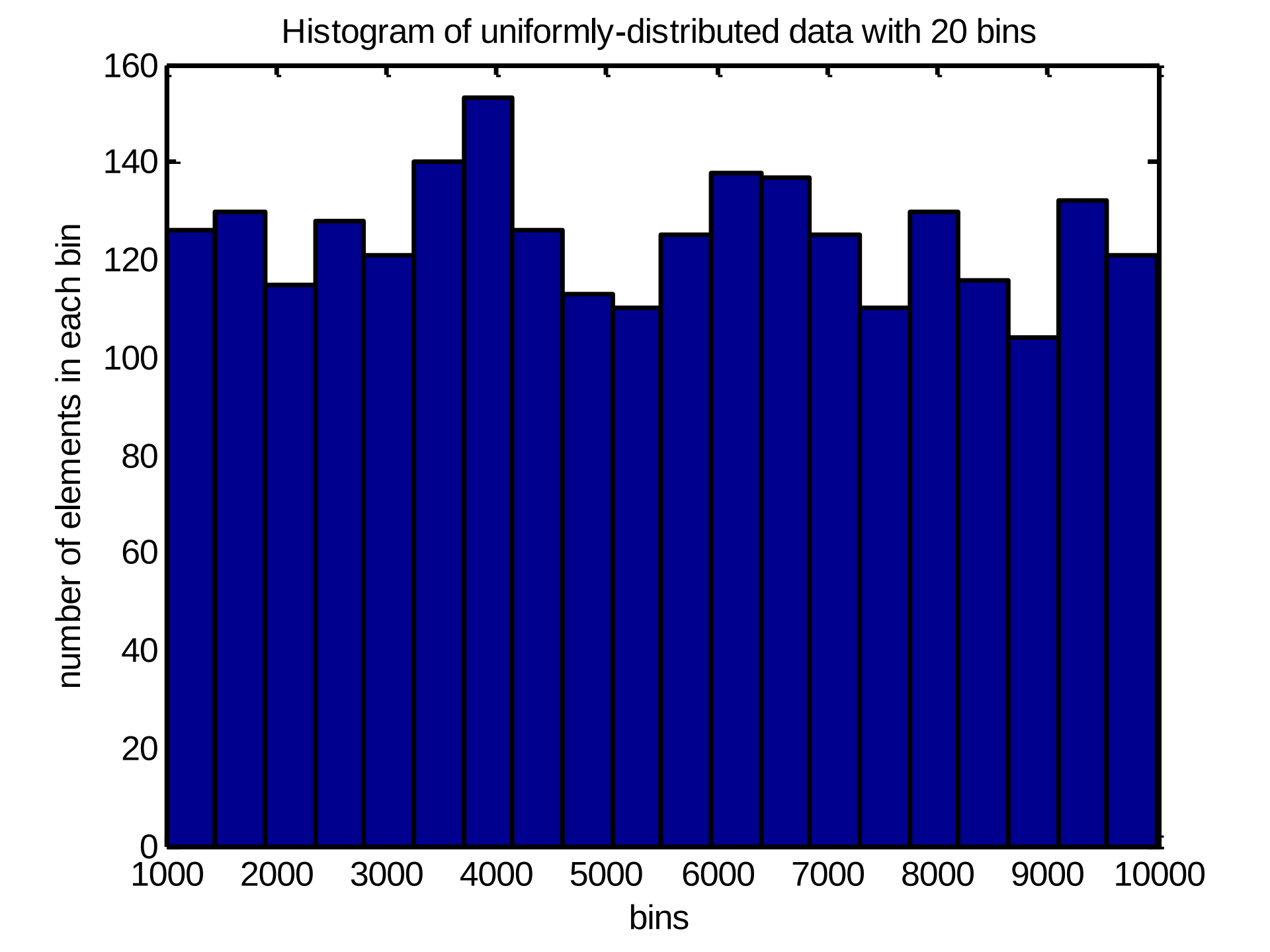
Sample mean and standard deviation

```
>> mean_y = 175;

>> sigma_y = 20;

>> x = randn(1,1000);

>> y = sigma_y*x + mean_y;

>> mean_y = mean(y)

>> std_y = std(y)
```

```
mean_y =

   174.7924

std_y =

   19.6464
```

Uniform distribution

```
>> a=1000;
>> b=10000;
>> xx = rand(1,2500);
>> yy = (b-a)*xx + a;
>> hist(yy,20)
>> xlabel('bins')
>> ylabel('number of elements in each bin')
>> title('Histogram of uniformly-distributed data with 20
   bins')
```


Histogram of uniformly-distributed data with 20 bins
number of elements in each bin
0
20
40
60
80
100
120
140
160
1000
2000
3000
4000
5000
6000
7000
8000
9000
10000
bins

Build-in functions

Command	Description	Example

`randi(imax)`
(imax is an integer)
	Generates a single random number between 1 and imax.	`>> a=randi(15)`
		`a =`
		`9`

`randi(imax, n)`	Generates an n × n matrix with random integers between 1 and imax.	`>> b=randi(15,3)`
		`b =`
		`4 8 11`
		`14 3 8`
		`1 15 8`

`randi(imax, m,n)`	Generates an m × n matrix with random integers between 1 and imax.	`>> c=randi(15,2,4)`
		`c =`
		`1 1 8 13`
		`11 2 2 13`

```
>> d=randi([50 90],3,4)
d =
    57    82    71    75
    66    52    67    61
    84    66    76    67
```

Eigenvalues & Eigenvectors

特征值、特征向量

Let's assume **A** is a **square** matrix. If there is a scalar λ and a vector **v** such that

$$Av = \lambda v$$

then λ is called an eigenvalue of matrix **A** and **v** would be its corresponding eigenvector.

Eigenvalues & Eigenvectors

```
>> A = [1 2 3; 2 4 5; 3 5 6];
>> [V,D] = eig(A)
V =
    0.7370    0.5910    0.3280
    0.3280   -0.7370    0.5910
   -0.5910    0.3280    0.7370
D =
   -0.5157         0         0
         0    0.1709         0
         0         0   11.3448
```

Eigenvalues & Eigenvectors

>> A*V(:,1)

ans =

-0.3801

-0.1692

0.3048

>> D(1,1)*V(:,1)

ans =

-0.3801

-0.1692

0.3048

which is equal to

Functions length, size, sum, mean, max, min

```
>> A = [1 2 3; 3 4 5; 6 7 9;
   12 13 14];
```

```
>> B = ones(size(A))
```

```
B =

     1     1     1
     1     1     1
     1     1     1
     1     1     1
```

```
>> C = zeros(size(A))
```

```
C =

     0     0     0
     0     0     0
     0     0     0
     0     0     0
```

Functions length, size, sum, mean, max, min

```
>> x=[1 3 5 7];
>> sum_x = sum(x)
sum_x =
    16
```

```
>> A = [1 2 3; 3 4 5; 6 7 9;
   12 13 14]
A =
     1     2     3
     3     4     5
     6     7     9
    12    13    14
>> sum_A = sum(A)
sum_A =
    22    26    31
```

Functions length, size, sum, mean, max, min

```
>> A = [1 2 3; 3 4 5; 6 7 9; 12 13 14]

A =

     1     2     3
     3     4     5
     6     7     9
    12    13    14

>> sum_A = sum(sum(A))

sum_A =

    79
```

```
>> sum_A = sum(A,1)

sum_A =

    22    26    31

>> sum_A = sum(A,2)

sum_A =

     6
    12
    22
    39
```

Functions length, size, sum, mean, max, min

```
>> A = [1 2 3; 3 4 5; 6 7 9;
   12 13 14]
```

```
A =

     1     2     3
     3     4     5
     6     7     9
    12    13    14
```

```
>> mean_A = mean(A)

mean_A =

    5.5000    6.5000    7.7500
```

```
>> max_A = max(A)

max_A =

    12    13    14
```

Try: mean(A,1) mean(A,2)

mean(A(:,1)) mean(A(:,2))

max(A,1) max(A,2)

max(A(:,1)) max(A(:,2))

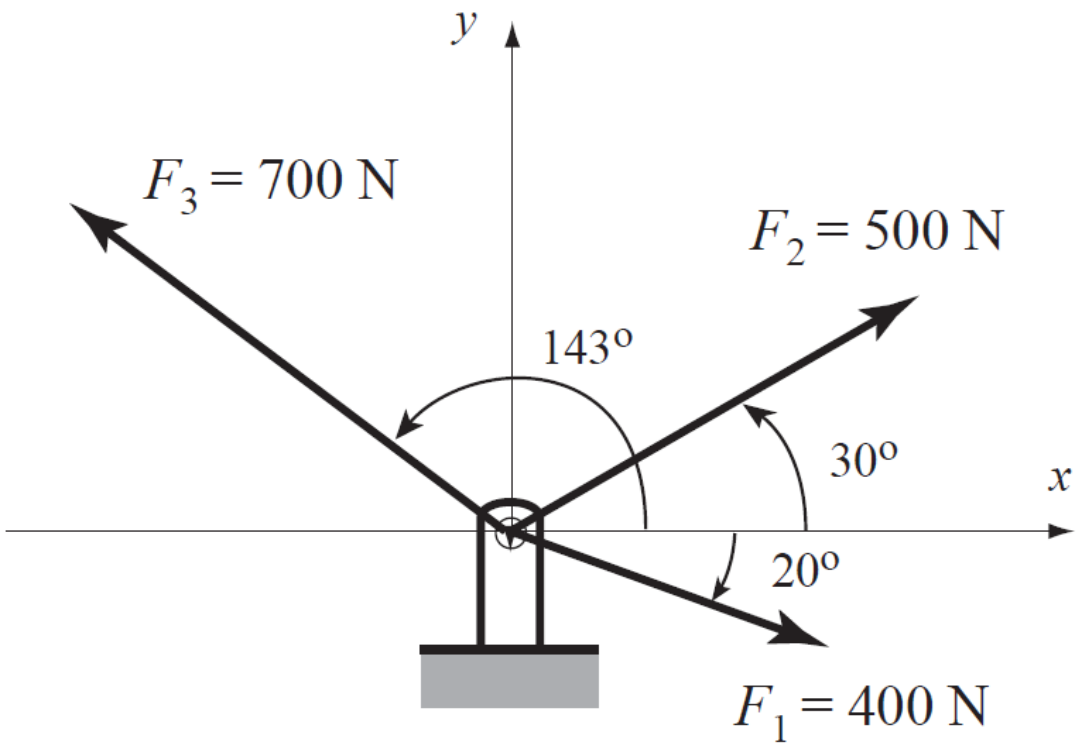
Engineering applications

Three forces are applied to a bracket as shown.
Determine the total (equivalent) force applied to the bracket.

$$\mathbf{F} = F_x\mathbf{i} + F_y\mathbf{j} = F\cos\theta\mathbf{i} + F\sin\theta\mathbf{j} = F(\cos\theta\mathbf{i} + \sin\theta\mathbf{j})$$

$$F = \sqrt{F_x^2 + F_y^2}$$

$$\tan\theta = \frac{F_y}{F_x}$$



Engineering applications

- Write each force as a vector with two elements, where the first element is the *x* component of the vector and the second element is the *y* component.
- Determine the vector form of the equivalent force by adding the vectors.
- Determine the magnitude and direction of the equivalent force.

Engineering applications

```
% Sample Problem 3-2 solution (script file)
clear
F1M=400; F2M=500; F3M=700;

Th1=-20; Th2=30; Th3=143;

F1=F1M*[cosd(Th1) sind(Th1)]
F2=F2M*[cosd(Th2) sind(Th2)]
F3=F3M*[cosd(Th3) sind(Th3)]
Ftot=F1+F2+F3
FtotM=sqrt(Ftot(1)^2+Ftot(2)^2)
Th=atand(Ftot(2)/Ftot(1))
```

Define variables with the magnitude of each vector.

Define variables with the angle of each vector.

Define the three vectors.

Calculate the total force vector.

Calculate the magnitude of the total force vector.

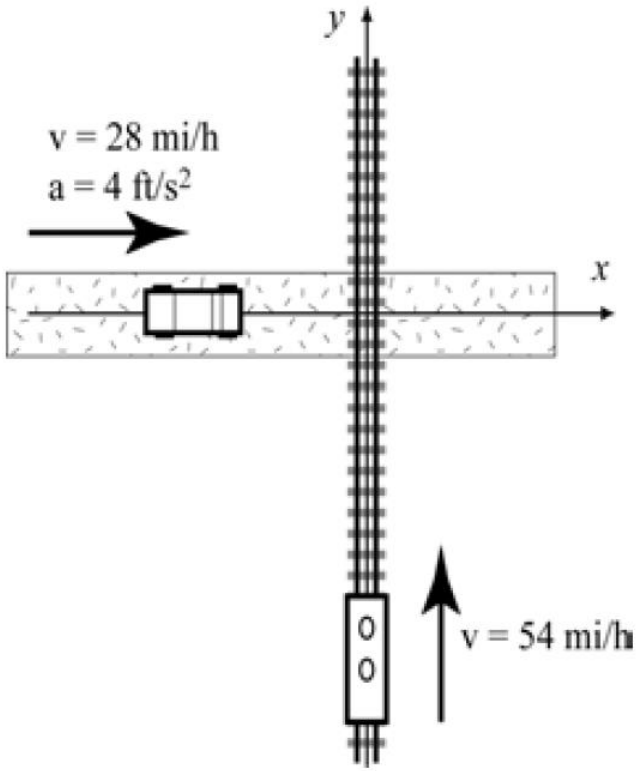
Calculate the angle of the total force vector.

Engineering applications

- A train and a car are approaching a road crossing. At time the train is 400ft south of the crossing traveling north at a constant speed of 54mi/h.
- At the same time the car is 200ft west of the crossing traveling east at a speed of 28mi/h and accelerating at $4m/s^2$.

1. Determine the positions of the train and the car;
2. the distance between them;
3. and the speed of the train relative to the car

every second for the next 10 seconds.



Engineering applications

Car

$$x = -200 + v_{ocar}t + \frac{1}{2}a_{car}t^2$$

$$\mathbf{v}_{car} = (v_{ocar} + a_{car}t)\mathbf{i}$$

Train

$$y = -400 + v_{otrain}t$$

$$\mathbf{v}_{train} = v_{otrain}\mathbf{j}$$

Distance between them

$$d = \sqrt{x^2 + y^2}$$

Relative speed

$$\mathbf{v}_{t/c} = \mathbf{v}_{train} - \mathbf{v}_{car} = -(v_{ocar} + a_{car}t)\mathbf{i} + v_{otrain}\mathbf{j}$$

Engineering applications

```
v0train=54*5280/3600; v0car=28*5280/3600; acar=4;
```

Create variables for the initial velocities (in ft/s) and the acceleration.

```
t=0:10;
```

Create the vector `t`.

```
y=-400+v0train*t;
x=-200+v0car*t+0.5*acar*t.^2;
```

Calculate the train and car positions.

```
d=sqrt(x.^2+y.^2);
```

Calculate the distance between the train and car.

```
vcar=v0car+acar*t;
```

Calculate the car's velocity.

```
speed_trainRcar=sqrt(vcar.^2+v0train^2);
```

Calculate the speed of the train relative to the car.

```
table=[t' y' x' d' vcar' speed_trainRcar']
```

Create a table (see note below).

End of Elementary Matrix Operations

Data type

Constant

Inf or inf: Infinity

>> 1/0

NaN: Not-a-Number

>> 0/0 >> Inf/Inf

pi:Ratio of circle's circumference to its diameter

>> pi

String

- A string is an array of characters. It is created by typing the characters within single quotes.
- Strings can include letters, digits, other symbols, and spaces.
- When a variable is defined as a string, the characters of the string are stored in an array just as numbers are.
- Each character, including a space, is an element in the array.
- The elements of the vectors are addressed by position.

```
>> name='kirin shi'
>> age='80';
>> degree='phd‘
>> info=[name ' is ' age ' years old,' 'and he has a ' degree ' degree'];
```

String

```
>> a='FRty 8'

a =
FRty 8

>> B='My name is John Smith'

B =
My name is John Smith
>>
```

```
>> B(4)

ans =
n

>> B(12)

ans =
```

```
>> B(12:15)='Bill'

B =
My name is Bill Smith
>>
```

Using a colon to assign new characters to elements 12 through 15 in the vector `B`.

String in a matrix

- Strings can also be placed in a matrix by typing a semicolon ; (or pressing the Enter key) at the end of each row.
- All rows must have the same number of elements.
- MATLAB has a built-in function named char that creates an array with
  rows having the same number of characters from an input of rows not all of the same length.
- MATLAB makes the length of all the rows equal to that of the longest row by adding spaces at the end of the short lines.

```
variable_name=char('string 1','string 2','string 3')
```

String in a matrix

```
variable_name = char('string 1','string 2','string 3')
```

```
>> Info=char('Student Name:','John Smith','Grade:','A+')

Info =
Student Name:
John Smith
Grade:
A+
>>
```

A variable named `Info` is assigned four rows of strings, each with different length.

The function `char` creates an array with four rows with the same length as the longest row by adding empty spaces to the shorter lines.

```
>> x=536

x =
   536

>> y='536'

y =
536
>>
```

Unsigned/Signed integer; float

- **uint8**

unsigned 8-bit integers (0-2^8-1,0~255)

```
>> uint8(1.5)
```

- **uint16**

unsigned 16-bit integers (0-2^16-1,0~65535)

```
>> uint16(2^32)
```

- **uint32**

unsigned 32-bit integers (0-2^32-1)

```
uint32 (2^32)
```

- **uint64**

unsigned 64-bit integers (0-2^64-1)

```
>> uint64(-6)
```

Unsigned/Signed integer; float

- **int8**

signed 8-bit integers (-128~127)

```
>> int8(1.5)
```

- **int16**

signed 16-bit integers (-32768 ~ 32767)

```
>> int16(2^32)
```

- **int32**

signed 32-bit integers (-2^32/2 ~ 2^32/2-1)

```
int32 (2^32)
```

- **int64**

signed 64-bit integers (-2^64/2 ~ 2^64/2-1)

>> uint64(-6)

intmin('int8')

intmax('int64')

Unsigned/Signed integer; float

double (default)

MATLAB constructs the double-precision (or double) data type according to IEEE Standard 754 for double precision. Any value stored as a double requires 64 bits,

single

MATLAB constructs the double-precision (or double) data type according to IEEE Standard 754 for double precision. Any value stored as a double requires 32 bits,

Use double-precision to store values greater than approximately 3.4×10^{38} or less than approximately -3.4×10^{38}.

For numbers that lie between these two limits, you can use either double- or single-precision, but single requires less memory.

Constant

eps: Floating-point relative accuracy

>> eps('single') >> eps('double')

realmin: Smallest positive normalized floating-point number

>> realmin('double') >> realmin('single')

realmin: Largest positive normalized floating-point number

>> realmax('double') >> realmax('single')

Data input and output

Input data from Keyboard

```
variable_name = input('string with a message that
                      is displayed in the Command Window')
```

Matlab command:

eval_varable = input('prompt')

str_varable = input('prompt', 's')

>> score = **input('**What is the score you want for your Matlab course?: **')**;

What is the score you want for your Matlab course?:

>> reply = **input('**Are you a good student? Y/N: **', 's')**;

Are you a good student? Y/N:

Input data from Keyboard

```
% This script file calculates the average of points scored in three games.
% The points from each game are assigned to the variables by
% using the input command.
game1=input('Enter the points scored in the first game ');
game2=input('Enter the points scored in the second game ');
game3=input('Enter the points scored in the third game ');
ave_points=(game1+game2+game3)/3
```

```
>> Chapter4Example4
Enter the points scored in the first game   67
Enter the points scored in the second game   91
Enter the points scored in the third game   70

ave_points =
     76
>>
```

The computer displays the message. Then the value of the score is typed by the user and the **Enter** key is pressed.

Output data to Screen

The **disp** command is used to display the elements of a variable without displaying the name of the variable, and to display text.

```
disp(name of a variable) or disp('text as string')
```

```
>> x=[1,2,3];
>> disp(x)

>> sjtu = [' Shanghai Southwest Some School '];
disp(sjtu)
```

Output data to Screen

```
>> abc = [5  9  1;  7  2  4];
>> disp(abc)
     5     9     1
     7     2     4

>> disp('The problem has no solution.')

The problem has no solution.
>>
```

A 2×3 array is assigned to variable `abc`.

The `disp` command is used to display the `abc` array.

The array is displayed without its name.

The `disp` command is used to display a message.

Output data to Screen

```
% This script file calculates the average points scored in three games.
% The points from each game are assigned to the variables by
% using the input command.
% The disp command is used to display the output.

game1=input('Enter the points scored in the first game    ');
game2=input('Enter the points scored in the second game   ');
game3=input('Enter the points scored in the third game    ');
ave_points=(game1+game2+game3)/3;
disp(' ')
disp('The average of points scored in a game is:')
disp(' ')
disp(ave_points)
```

Display empty line.

Display text.

Display empty line.

Display the value of the variable `ave_points`.

```
>> Chapter4Example5
Enter the points scored in the first game    89
Enter the points scored in the second game   60
Enter the points scored in the third game    82

The average of points scored in a game is:

   77
```

An empty line is displayed.

The text line is displayed.

An empty line is displayed.

The value of the variable `ave_points` is displayed.

Output data to Screen

```
yr=[1984 1986 1988 1990 1992 1994 1996];
pop=[127 130 136 145 158 178 211];
tableYP(:,1)=yr';
tableYP(:,2)=pop';
disp('          YEAR      POPULATION')
disp('                    (MILLIONS)')
disp(' ')
disp(tableYP)
```

The population data is entered in two row vectors.

`yr` is entered as the first column in the array `tableYP`.

`pop` is entered as the second column in the array `tableYP`.

Display heading (first line).

Display heading (second line).

Display an empty line.

Display the array `tableYP`.

```
>> PopTable
          YEAR      POPULATION
                    (MILLIONS)

          1984          127
          1986          130
```

Headings are displayed.

An empty line is displayed.

Output data to Screen

- The **fprintf** command can be used to display output (text and data) on the screen or to save it to a file.

- With this command (unlike with the disp command) the output can be formatted. E.g. text and numerical values of variables can be intermixed and displayed in the same line.

```
fprintf('text typed in as a string')
```

Output data to Screen

```
fprintf('The problem, as entered, has no solution. Please check the input data.')
```

```
The problem, as entered, has no solution. Please check the input data.
```

```
fprintf('The problem, as entered, has no solution.\nPlease check the input data.')
```

```
The problem, as entered, has no solution.
Please check the input data.
```

```
fprintf('The problem, as entered, has no solution. Please check the input data.')
x = 6; d = 19 + 5*x;
fprintf('Try to run the program later.')
y = d + x;
fprintf('Use different input values.')
```

```
The problem, as entered, has no solution. Please check the input data.Try to run the program later.Use different input values.
```

Output data to Screen

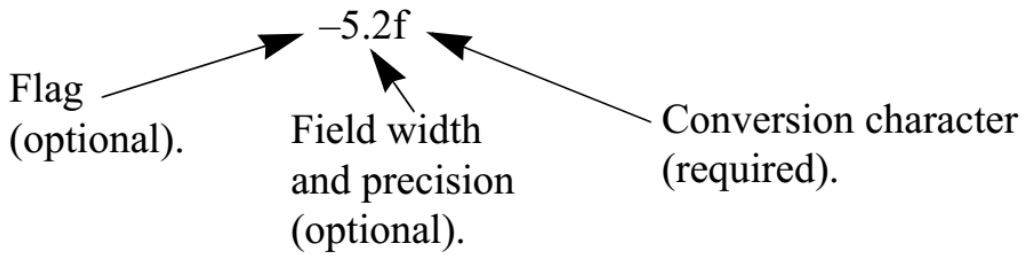
- Using the fprintf command to display a mix of text and numerical data:

```
fprintf('text as string %-5.2f additional text',
                                    variable_name)
```

The % sign marks the spot where the number is inserted within the text.

Formatting elements (define the format of the number).

The name of the variable whose value is displayed.



Character used for flag	Description
– (minus sign)	Left-justifies the number within the field.
+ (plus sign)	Prints a sign character (+ or –) in front of the number.
0 (zero)	Adds zeros if the number is shorter than the field.

Output data to Screen

- The field width and precision (5.2 in the previous example) are optional.
- The first number (5 in the example) is the field width, which specifies the minimum number of digits in the display. If the number to be displayed is shorter than the field width, spaces or zeros are added in front of the number.
- The precision is the second number (2 in the example). It specifies the number of digits to be displayed to the right of the decimal point.
- The last element in the formatting elements, which is required, is the conversion character, which specifies the notation in which the number is displayed.

e	Exponential notation using lower-case e (e.g., 1.709098e+001).
E	Exponential notation using upper-case E (e.g., 1.709098E+001).
f	Fixed-point notation (e.g., 17.090980).
g	The shorter of e or f notations.
G	The shorter of E or f notations.
i	Integer.

Output data to Screen

```
% This script file calculates the average points scored in three games.
% The values are assigned to the variables by using the input command.
% The fprintf command is used to display the output.
game(1) = input('Enter the points scored in the first game    ');
game(2) = input('Enter the points scored in the second game   ');
game(3) = input('Enter the points scored in the third game    ');
ave_points = mean(game);
```

```
fprintf('An average of %f points was scored in the three games.',ave_points)
```

- Text
- % marks the position of the number.
- Additional text.
- The name of the variable whose value is displayed.

```
>> Chapter4Example6
Enter the points scored in the first game    75
Enter the points scored in the second game   60
Enter the points scored in the third game    81
An average of 72.000000 points was scored in the three games.
>>
```

The display generated by the `fprintf` command combines text and a number (value of a variable).

Output data to Screen

- With the fprintf command it is possible to insert more than one number (value of a variable) within the text. This is done by typing %g (or % followed by any formatting elements) at the places in the text where the numbers are to be inserted.

```
fprintf('..text...%g...%g...%f...',variable1,variable2,variable3)
```

```
% This program calculates the distance a projectile flies,
% given its initial velocity and the angle at which it is shot.
% the fprintf command is used to display a mix of text and numbers.

v=1584;  % Initial velocity (km/h)
theta=30;  % Angle (degrees)
vms=v*1000/3600;
t=vms*sind(30)/9.81;
d=vms*cosd(30)*2*t/1000;
```

Changing velocity units to m/s.

Calculating the time to highest point.

Calculating max distance.

Output data to Screen

- With the fprintf command it is possible to insert more than one number (value of a variable) within the text. This is done by typing %g (or % followed by any formatting elements) at the places in the text where the numbers are to be inserted.

```
fprintf('..text...%g...%g...%f...',variable1,variable2,variable3)
```

```
fprintf('A projectile shot at %3.2f degrees with a velocity
of %4.2f km/h will travel a distance of %g km.\n',theta,v,d)
```

```
>> Chapter4Example7

A projectile shot at 30.00 degrees with a velocity of
1584.00 km/h will travel a distance of 17.091 km.
>>
```

Input data from a file

Matlab command:

eval_varable = load('filename', '-mat', 'variables')

eval_varable = load('filename', '-ascii')

eval_varable = dlmread('filename', delimiter, range)

>> load('C:\Matlab_course\ example2_0.mat','-mat','a')

>> load('C:\Matlab_course\example2_0.mat','-mat')

>> load('C:\Matlab_course\example2_1.dat', '-ascii')

>> dlmread('C:\Matlab_course\example2_2.dat', '\t', [4,0,10,3])

R=0, C=0 specifies the first value in the file

imread

Output data to a file

Matlab command:

eval_varable = save('filename', 'variables', 'format')

'-mat'	Binary MAT-file format (default).
'-ascii'	8-digit ASCII format.
'-ascii', '-tabs'	Tab-delimited 8-digit ASCII format.
'-ascii', '-double'	16-digit ASCII format.
'-ascii', '-double', '-tabs'	Tab-delimited 16-digit ASCII format.

Command Window

```
>> V=[3 16 -4 7.3];
>> A=[6 -2.1 15.5; -6.1 8 11];
>> save('DatSavAsci.dat','A','-ascii');
>>
```

Output data to a file

Matlab command:

fileID = fopen(filename, permission)

fprintf(fileID, format, A, ...)

'r'	Open file for reading (default).
'w'	Open or create new file for writing. Discard existing contents, if any.
'a'	Open or create new file for writing. Append data to the end of the file.
'r+'	Open file for reading and writing.
'w+'	Open or create new file for reading and writing. Discard existing contents, if any.
'a+'	Open or create new file for reading and writing. Append data to the end of the file.
'A'	Append without automatic flushing. (Used with tape drives.)
'W'	Write without automatic flushing. (Used with tape drives.)

To open files in text mode, attach the letter 't' to the permission argument

Output data to a file

Example:

```
data1=[1;2;3;4];   data2=[5;6;7;8]; data3=[9;10;11;12];
result=[data1,data2,data3];
file_head = ['This is an example for fprint fopen command'];
fid = fopen('C:\Matlab_course\example2_3.dat', 'wt+');
fprintf(fid, '%s \n', file_head, 'VARIABLES = "X""Y""Z"');
[row,col] = size(result);
for i= 1:row
    for j= 1:col-1
        fprintf(fid, '%f\t', result(i, j));
    end
     fprintf(fid, '%f\n', result(i, col));
end
fclose(fid);
```

xlsread, xlswrite

imwrite, fwrite, fscanf, fread, fgetl, fgets

Script &
Function Files

Programs: definition

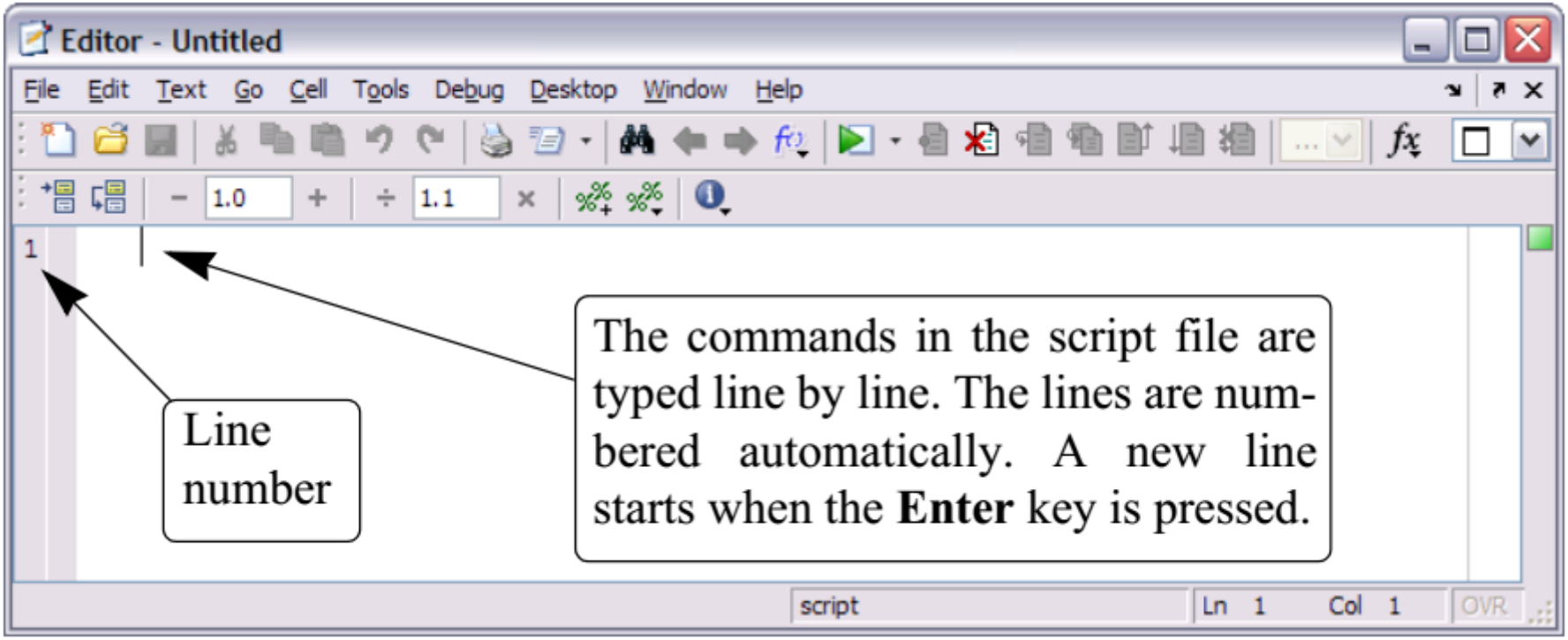
- **Program:** sequences of commands which are stored in MATLAB files.
  - These files must have the extension **.m** (e.g. **filename.m**) and that is why they are called ***M-files***.
- **An M-file** is executed by typing its name (***without*** *the extension* ***.m***) at the prompt.
  - This would be equivalent to typing all the commands in the file at the prompt one by one, in the order they have been written.

Creating M files

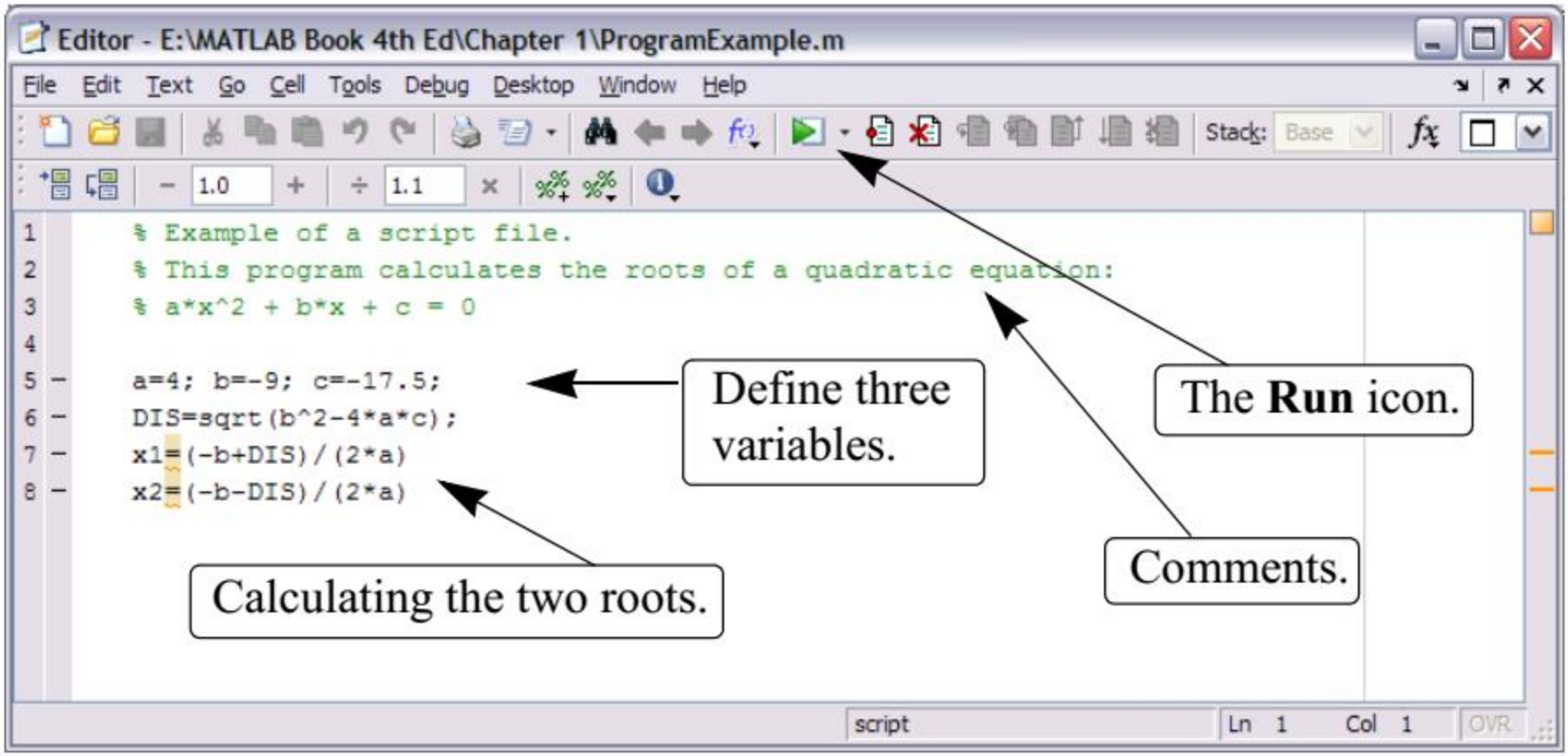
- An **Edit/Debug** window is used to create new **M-files**, or to modify existing ones.

- To create a new M-file
  - use the **'File/New/M-file'** selection from the **desktop menu,** or
  - by clicking the **'New M-File'** button on the desktop toolbar.

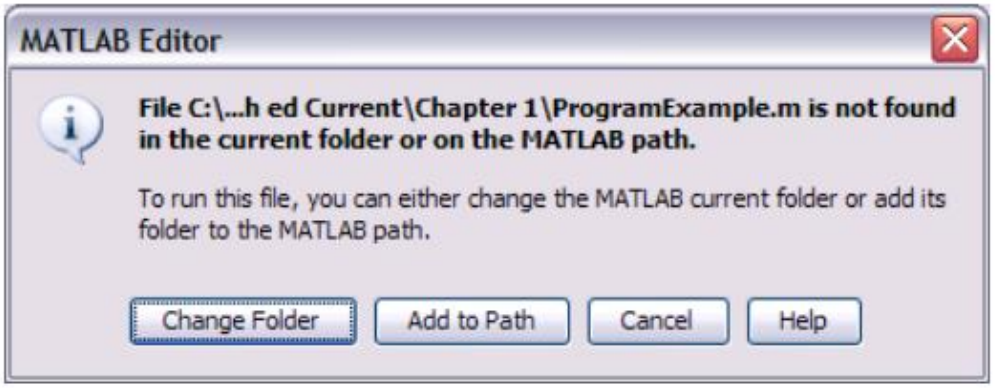
Creating M files

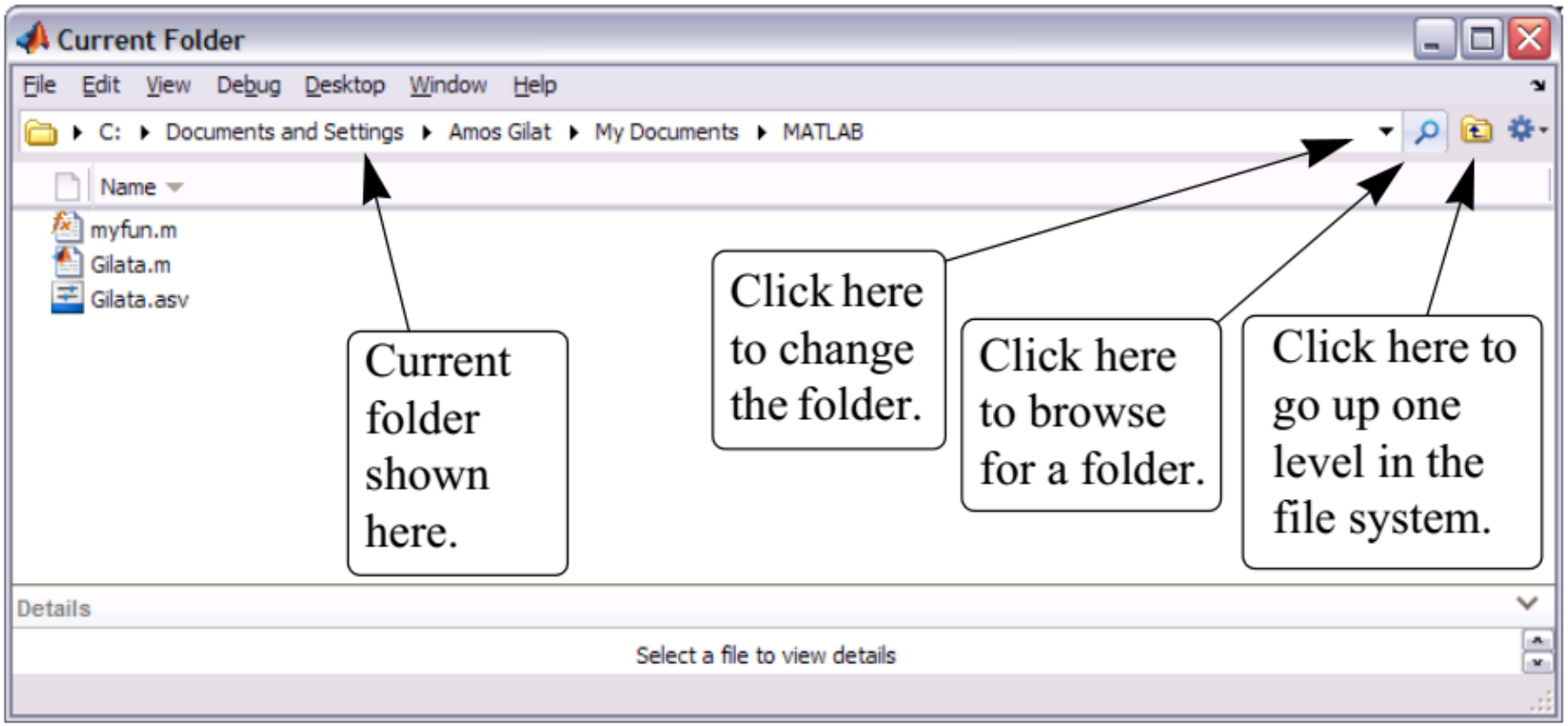


Creating M files

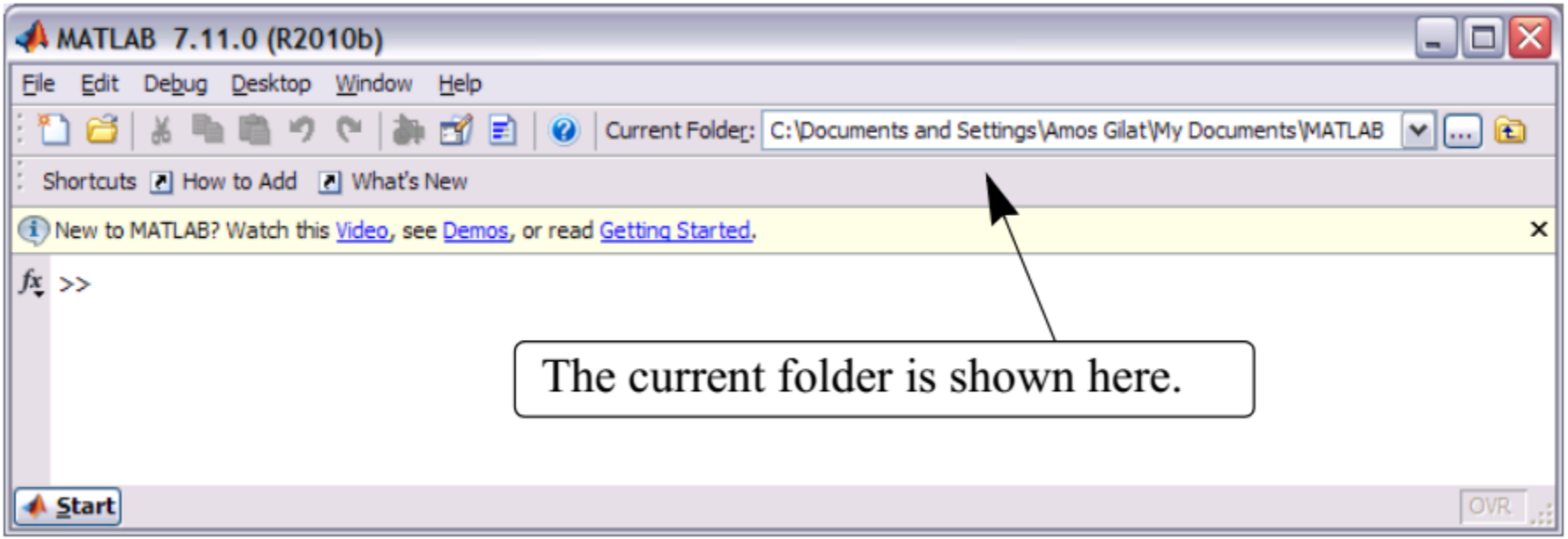


Creating M files





Editor Window




```
>> cd E:

>> ProgramExample
x1 =
    3.5000
x2 =
   -1.2500
```

The current directory is changed to drive E.

The script file is executed by typing the name of the file and pressing the **Enter** key.

The output generated by the script file (the roots `x1` and `x2`) is displayed in the Command Window.

Opening M files

- To **open** an existing M-file
  - use the **“File/Open”** selection from the desktop menu or
  - click on the **‘open file’** button on the desktop toolbar.
  - double click on the file name in the current directory.

Edit/Debug window

- The Edit/Debug window is essentially a text editor.
- MATLAB language features are highlighted in different colours.
  - comments appear **in green**,
  - variables and numbers appear in black,
  - character strings appear in **purple**,
  - language keywords appear in **blue**.

Different types of M-files

- M-files are of two types: *Script files* and *function files*.
  - ***Script files*** are just collections of MATLAB statements that are stored in a file.

    Script files do not have clearly defined input and output arguments.
  - **A *function file*** receives data through **input** arguments, and return results through **output** arguments.

Script Files

Script Files

- Quite often you want to reuse bits of codes over again.
- One way to do this is to use a script file.
- A **script file** is a collection of MATLAB commands which are saved in a file rather than being typed at the MATLAB prompt one by one.

Example: body mass index of an individual

```
>> height = 1.7;
>> weight = 70;
>> body_mass_index = weight/height^2

body_mass_index =

    24.2215
```

Example of a script file

- **Open an Edit/debug window and then type the following statements in it.**

```
%Program to calculate the body mass index of a person
%The person's height in meters
height = 1.70;
%The person's weight in Kilograms
weight = 70;
%calculate the body mass index
body_mass_index = weight/height^2
```

Running a script file

- To run your program, type its name at the command prompt without the extension **.m**.
- Make sure the name of the program is not the same as the name of a variable you have used or the name of a variable in your program.

```
>> bodymassindex

body_mass_index =

   24.2215
```

Changing data in your script file

- To change the weight and height of the individual,
  - **open the file containing your code** (**bodymassindex.m**) and
  - change the height and weight to, for example, **1.80m** and **86kg**.
- Then run the program again.

```
>> bodymassindex

body_mass_index =

   26.5432
```

File Names: Caution

- MATLAB checks for variable names first. So you should **never** use a variable with the same name as a M-file.
  - **If you do so, that function will become inaccessible.**
- Also never create an M-file with the same name **as a standard MATLAB function** (such as **sin**, **cos**, **max**, **eig**).

Function Files

Meaning of Function

- Consider the following simple function

$$Z = f(x,y) = x^2 + y + xy$$

- The function ***f*** has two **input** arguments (***x*** and ***y***) and one **output** argument (***Z***).
- function ***f*** defines how the output argument ***Z*** is calculated from the input arguments ***x*** and ***y***.
- For example, α **= *f* (2,3)** will be calculated as

Meaning of Function

$$Z = f(2,3) = 2^2 + 3 + 2*3 = 13$$

$$\alpha = Z$$

- In other words,
  - first, numbers 2 and 3 will be assigned to variables ***x*** and ***y*** respectively;
  - then, ***Z*** will be calculated according to the definition of the function ***f***.
  - Finally, the calculated value of ***Z*** will be assigned to variable α.

Function files in MATLAB

- In MATLAB, functions behave in the same way as the foregoing algebraic function.
- For example, we can write a function file to calculate the body mass index of an individual in the following way:

Function file: an example

```
function bmi = bodymassindex(height, weight)
%Program to calculate the body mass index
%Two input arguments:
   %the first one is height in meters
   %the second one is weight in kilograms
%the only output is the body mass index (bmi)
%Last revision: 2 October 2010
%Example: my_bmi = bodymassindex(1.81,91)


%calculate the body mass index
bmi = weight/height^2;
```

Running a function file

- Now type the following command and see the result

```
>> my_bmi = bodymassindex(1.81,91)

 my_bmi =

   27.7769
```

```
function bmi = bodymassindex(height, weight)

bmi = weight/height^2;
```

Function file: input arguments

- You can choose any values for the input arguments.
- So if you are 1.65m tall and your weight is 55kg, then your body mass index can be calculated by the following command

```
>> my_bmi = bodymassindex(1.65,55)

my_bmi =

    20.2020
```

Giving values to input arguments

- You can assign the height and weight values to variables and then use these variables as input arguments. For example,

```
>> my_height = 1.65; my_weight = 55;

>> my_index = bodymassindex(my_height,my_weight)

my_index =

   20.2020
```

```
function bmi = bodymassindex(height, weight)

bmi = weight/height^2;
```

The workspace of a function

- The workspace is an area in computer memory where variables are stored.
- A script file does not have an independent work space.
- A function file has its own independent (private) workspace.
  - In other words, the variables inside a function file are all **local variables**, whose values are stored in a separate place from MATLAB workspace.

The workspace of a function

```
>> my_bmi = bodymassindex(1.81,91)
my_bmi =
   27.7769
>> bmi
??? Undefined function or variable 'bmi'.
```

- MATLAB communicates with a function only through its input and output arguments.

```
function bmi = bodymassindex(height, weight)
bmi = weight/height^2;
```

Example

Write a function file *(name it chap7one)* for the function

$$f(x) = \frac{x^4\sqrt{3x+5}}{(x^2+1)^2}$$

The input to the function is **x** and the output is **f(x)**.Write the function such that **x** can be a vector. Use the function to calculate:

(a) f(x) for x = 6.

(b) f(x) for x =1,3,5,7,9 and 11.

Example

Write a user defined function *(name it Ftoc)* that converts temperature in degrees F to temperature in degrees C. Use the function to solve the following problem:

The change in the length of an object, ΔL, due to a change in temperature, ΔT, is given by :$\Delta L = \alpha \Delta T$, where α is the coefficient of the thermal expansion.

Determine the change in the area of a rectangular (4.5m and 2.25m) aluminium $\left(\alpha = 23 * 10^{-6}\ \frac{1}{^{\circ}\mathrm{C}}\right)$ plate if the temperature changes from $40°F$ to $92°F$.

COMPARISON BETWEEN SCRIPT FILES AND FUNCTION FILES

- Both script and function files are saved with the extension .
- The first executable line in a function file is (must be) the function definition line.
- The variables in a function file are local. The variables in a script file are recognized in the Command Window.
- Script files can use variables that have been defined in the workspace.
- Script files contain a sequence of MATLAB commands (statements).
- Function files can accept data through input arguments and can return data through output arguments.
- When a function file is saved, the name of the file should be the same as the name of the function.

ANONYMOUS AND INLINE FUNCTIONS

An **anonymous function** is a simple (one-line) user-defined function that is defined without creating a separate function file (M-file). An anonymous function is created by typing the following command:

```
name = @ (arglist) expr
```

The name of the anonymous function.

The @ symbol.

A list of input arguments (independent variables).

Mathematical expression.

- The expr consists of a single valid mathematical MATLAB expression.
- The mathematical expression can have one or several independent variables.
- The independent variable(s) is (are) entered in the (arglist). Multiple independent variables are separated with commas.
- The expression must be written according to the dimensions of the arguments (element-by-element or linear algebra calculations).

Examples of anonymous function

- Example of an anonymous function with **one independent variable**:

The function $f(x) = \frac{e^{x^2}}{\sqrt{x^2+5}}$ can be defined (in the Command Window) as an anonymous function for x as a scalar by:

```
>> FA = @ (x) exp(x^2)/sqrt(x^2+5)

FA =
    @(x)exp(x^2)/sqrt(x^2+5)
>> FA(2)
ans =
   18.1994
>> z = FA(3)
z =
  2.1656e+003
```

If **x** is expected to be an array, with the function calculated for each element, then the function must be modified for element-by-element calculations.

```
>> FA = @ (x) exp(x.^2)./sqrt(x.^2+5)

FA =
    @(x)exp(x.^2)./sqrt(x.^2+5)
```

Examples of anonymous function

- Example of an anonymous function with several independent variables:

The function $f(x,y) = 2x^2 - 4xy + y^2$ can be defined as an anonymous function by:

```
>> HA = @ (x,y) 2*x^2 - 4*x*y + y^2
HA =
    @(x,y)2*x^2-4*x*y+y^2
```

Then the anonymous function can be used for different values of x and y. For example, typing HA(2,3) gives:

```
>> HA(2,3)
ans =
    -7
```

ANONYMOUS AND INLINE FUNCTIONS

An inline function is a simple user-defined function that is defined without creating a separate function file (M-file). anonymous functions replace the inline functions used in earlier versions of MATLAB.

```
name = inline('math expression typed as a string')
```

- Any letter except i and j can be used for the independent variables in the expression.
- The mathematical expression can include any built-in or user-defined functions.
- The expression must be written according to the dimension of the argument (element-by-element or linear algebra calculations).
- The expression ***cannot*** include pre assigned variables.
- The inline function can be used as an argument in other functions.

Examples of inline function

The function $f(x) = \frac{e^{x^2}}{\sqrt{x^2+5}}$ can be defined as an inline function for **x** by:

```
>> FA=inline('exp(x.^2)./sqrt(x.^2+5)')
FA =
     Inline function:
     FA(x) = exp(x.^2)./sqrt(x.^2+5)
>> FA(2)
ans =
   18.1994
>> FA([1 0.5 2])
ans =
    1.1097    0.5604   18.1994
```

Expression written with element-by-element operations.

Using a scalar as the argument.

Using a vector as the argument.

FUNCTION FUNCTIONS

- A function that accepts another function is called in MATLAB a function function.

For example, MATLAB has a built-in function called ***fzero*** (Function A) that finds the zero of a math function (Function B), i.e., the value of ***x*** where $f(x) = 0$. When ***fzero*** is called, the specific function to be solved is passed into f***zero***, which finds the zero of the $f(x)$.

- There are two methods, function handle and function name, for listing the name of an imported function in the argument list of a function function.

Using Function Handles for Passing a Function into a Function Function

- **A function handle** is a MATLAB value that is associated with a function. It is a MATLAB data type and can be passed as an argument into another function.
- For built-in and user-defined functions, a function handle is created by typing the symbol @ in front of the function name. For example, ***@cos*** is the function handle of the built-in function ***cos***, and ***@FtoC*** is the function handle of the user-defined function ***FtoC***.
- The function handle can also be assigned to a variable name. For example,

cosHandle=@cos assigns the handle ***@cos*** to ***cosHandle***. Then the name ***cosHandle*** can be used for passing the handle.

- As anonymous functions, their name is already a function handle.

Writing a function function

- The function function and the imported function must have the same number and type of input and output arguments.

The following is an example of a user-defined function function, named ***funplot***, that makes a plot of a function (any function that is imported into it) between the points $x = a$ and $x = b$. The input arguments are **(Fun,a,b),** where **Fun** is a dummy name that represents the imported function, and a and b are the end points of the domain. The function funplot also has a numerical output xyout, which is a 3X2 matrix with the values of **x** and **f(x)** at the three points $x = a$, $x = (a + b)/2$ and $x = b$. Note that in the program, the dummy function Fun has one input argument **(x)** and one output argument y, which are both vectors.

Writing a function function

A name for the function that is passed in.

```
function xyout=funplot(Fun,a,b)
% funplot makes a plot of the function Fun which is passed in
% when funplot is called in the domain [a, b].
% Input arguments are:
% Fun:  Function handle of the function to be plotted.
% a:  The first point of the domain.
% b:  The last point of the domain.
% Output argument is:
% xyout: The values of x and y at x=a, x=(a+b)/2, and x=b
% listed in a 3 by 2 matrix.

x=linspace(a,b,100);
y=Fun(x);
xyout(1,1)=a; xyout(2,1)=(a+b)/2; xyout(3,1)=b;
xyout(1,2)=y(1);
xyout(2,2)=Fun((a+b)/2);
xyout(3,2)=y(100);
plot(x,y)
xlabel('x'), ylabel('y')
```

Using the imported function to calculate $f(x)$ at 100 points.

Using the imported function to calculate $f(x)$ at the midpoint.

Passing a user-defined function into a function function

- To pass a user-defined function into a function function, firstly, a user-defined function is written for $f(x)$.

```
function y=Fdemo(x)
y=exp(-0.17*x).*x.^3-2*x.^2+0.8*x-3;
```

- Next, the function Fdemo is passed into the user-defined function function funplot, which is called in the Command Window.

```
>> ydemo=funplot(@Fdemo,0.5,4)
ydemo =
    0.5000   -2.9852
    2.2500   -3.5548
    4.0000    0.6235
```

Enter a handle of the user-defined function Fdemo.

Using Function Name for Passing a Function into a Function Function

- A second method for passing a function into a function function is by typing the **function name** of the function that is being imported as a string. When a user-defined function is imported by using its name, the value of the imported function inside the function function has to be calculated with the **feval command.**

The **feval** (short for “function evaluate”) command evaluates the value of a function for a given value (or values) of the function’s argument (or arguments). The format of the command is:

```
variable = feval('function name', argument value)
```

Examples using the feval command with built-in functions follow

- The function name is typed as string.
- The function can be a built-in or a user-defined function.
- If there is more than one input argument, the arguments are separated with commas.
- If there is more than one output argument, the variables on the left-hand side of the assignment operator are typed inside brackets and separated with commas.

```
>> feval('sqrt',64)
ans =
     8
>> x=feval('sin',pi/6)
```

```
x =
    0.5000
```

Examples using the feval command with built-in functions follow

- The following shows the use of the feval command with the user-defined function **loan** which was created earlier. This function has three input arguments and two output arguments.

```
>> [M,T]=feval('loan',50000,3.9,10)
M =
         502.22
T =
       60266.47
```

A $50,000 loan, 3.9% interest, 10 years.

Monthly payment.

Total payment.

A name for the function that is passed in.

```
function xyout=funplotS(Fun,a,b)
% funplotS makes a plot of the function Fun which is passed in
% when funplotS is called in the domain [a, b].
% Input arguments are:
% Fun: The function to be plotted. Its name is entered as
string expression.
% a:  The first point of the domain.
% b:  The last point of the domain.
% Output argument is:
% xyout: The values of x and y at x=a, x=(a+b)/2, and x=b
% listed in a 3 by 2 matrix.

x=linspace(a,b,100);
y=feval(Fun,x);
xyout(1,1)=a; xyout(2,1)=(a+b)/2; xyout(3,1)=b;
xyout(1,2)=y(1);
xyout(2,2)=feval(Fun,(a+b)/2);
xyout(3,2)=y(100);
plot(x,y)
xlabel('x'), ylabel('y')
```

Using the imported function to calculate $f(x)$ at 100 points.

Using the imported function to calculate $f(x)$ at the midpoint.

Passing a user-defined function into another function by using a string expression

- The following demonstrates how to pass a user-defined function into a function function by typing the name of the imported function as a **string** in the input argument. The function $f(x) = e^{-0.17x}x^3 0\text{-}2x^2 + 0.8x - 3$, created as a user-defined function named **Fdemo**, is passed into the user-defined function **funplotS**.

```
>> ydemoS=funplotS('Fdemo',0.5,4)
ydemoS =
    0.5000   -2.9852
    2.2500   -3.5548
    4.0000    0.6235
```

The name of the imported function is typed as a string.

Subfunctions

- A function file can contain more than one user-defined function. The functions are typed one after the other. Each function begins with a function definition line. The first function is called the primary function and the rest of the functions are called subfunctions. The subfunctions can be typed in any order.
- The name of the function file that is saved should correspond to the name of the primary function.
- Each of the functions in the file can call any of the other functions in the file. Outside functions, or programs (script files), can call only the primary function.
- The program in the primary function can be divided into smaller tasks, each of which is carried out in a subfunction.

Sample Problem : Average and standard deviation

- Write a user-defined function that calculates the average and the standard deviation of a list of numbers. Use the function to calculate the average and the standard deviation of the following list of grades:

  80 75 91 60 79 89 65 80 95 50 81

The average and standard deviation of a given set of n numbers is given by:

$$x_{ave} = (x_1 + x_2 + \ldots + x_n)/n \qquad \sigma = \sqrt{\frac{\sum_{i=1}^{i=n} (x_i - x_{ave})^2}{n-1}}$$

- A user-defined function, named *stat*, is written for solving the problem. The function file includes *stat* as a primary function, and two subfunctions called ***AVG*** and ***StandDiv***. The function AVG calculates the average, and the function ***StandDiv*** calculates standard deviation.

Sample Problem : Average and standard deviation

Function file:

```
function [me SD] = stat(v)
n=length(v);
me=AVG(v,n);
SD=StandDiv(v,me,n);

function av=AVG(x,num)
av=sum(x)/num;

function Sdiv=StandDiv(x,xAve,num)
xdif=x-xAve;
xdif2=xdif.^2;
Sdiv= sqrt(sum(xdif2)/(num-1));
```

The primary function.

Subfunction.

Subfunction.

Command Window:

```
>> Grades=[80 75 91 60 79 89 65 80 95 50 81];
>> [AveGrade StanDeviation] = stat(Grades)
AveGrade =
   76.8182
StanDeviation =
   13.6661
```

Sample Problem : Average and standard deviation

Function file:

```
function [me SD] = stat(v)
n=length(v);
me=AVG(v,n);
SD=StandDiv(v,me,n);

function av=AVG(x,num)
av=sum(x)/num;

function Sdiv=StandDiv(x,xAve,num)
xdif=x-xAve;
xdif2=xdif.^2;
Sdiv= sqrt(sum(xdif2)/(num-1));
```

The primary function.

Subfunction.

Subfunction.

Command Window:

```
>> Grades=[80 75 91 60 79 89 65 80 95 50 81];
>> [AveGrade StanDeviation] = stat(Grades)
AveGrade =
   76.8182
StanDeviation =
   13.6661
```

Nested Function

- A nested function is a user-defined function that is written inside another user-defined function. The portion of the code that corresponds to the nested function starts with a function definition line and ends with an end statement. An ***end*** statement must also be entered at the end of the function that contains the nested function.

- One nested function:

The format of a user-defined function A (called the primary function) that contains
one nested function B is:

```
function y=A(a1,a2)
.......
function z=B(b1,b2)
.......
end
.......
end
```

One nested function

- The format of a user-defined function A (called the primary function) that contains one nested function B is:

```
function y=A(a1,a2)
 .......
      function z=B(b1,b2)
      .......
      end
 .......
end
```

- Note the end statements at the ends of functions B and A.
- A variable defined in the primary function A can be read and redefined in nested function B and vice versa.
- Function A can call function B, and function B can call function A.

Two (or more) nested functions at the same level

- The format of a user-defined function A (called the primary function) that contains two nested functions B and C at the same level is:

```
function y=A(a1,a2)
  .......
      f unction z=B(b1,b2)
      .......
      end
  .......
      function w=C(c1,c2)
      .......
      end
  .......
  end
```

- The three functions can access the workspace of each other.
- The three functions can call each other.

Example

Function file:

```
function [me SD]=statNest(v)
n=length(v);
me=AVG(v);

    function av=AVG(x)
    av=sum(x)/n;
    end

    function Sdiv=StandDiv(x)
    xdif=x-me;
    xdif2=xdif.^2;
    Sdiv= sqrt(sum(xdif2)/(n-1));
    end

SD=StandDiv(v);
end
```

The primary function.

Nested function.

Nested function.

Command Window:

```
>> Grades=[80 75 91 60 79 89 65 80 95 50 81];
>> [AveGrade StanDeviation] = statNest(Grades)
AveGrade =
   76.8182
StanDeviation =
   13.6661
```

Sample Problem : Exponential growth and decay

- A model for exponential growth or decay of a quantity is given by

$$A(t) = A_0 e^{kt}$$

where $A(t)$ and A_0 are the quantity at time *t* and time 0, respectively, and *k* is a constant. Write a user-defined function that uses this model to predict the quantity at time *t* from knowledge of and at some other time .

Function name and arguments: At = expGD (A0,At1,t1,t) .

Output argument At corresponds to $A(t)$, input arguments use A0, At1, t1, t, corresponding to A_0, $A(t1)$, t_1, and t, respectively.

- Use the function file in the Command Window for the following two cases:

(a) The population of Mexico was 67 million in the year 1980 and 79 million in 1986. Estimate the population in 2000.

(b) The half-life of a radioactive material is 5.8 years. How much of a 7-gram sample will be left after 30 years?

Sample Problem : Exponential growth and decay

Solution

The value of the constant *k* has to be determined first by solving for *k* in terms of $A(t)$ and A_0.

$$k = \frac{1}{t_1} \ln \frac{A(t_1)}{A_0}$$

Once *k* is known, the model can be used to estimate the population at any time. The user-defined function that solves the problem is:

```
function At=expGD(A0,At1,t1,t)
% expGD calculates exponential growth and decay
% Input arguments are:
% A0: Quantity at time zero.
% At1: Quantity at time t1.
% t1: The time t1.
% t: time t.
% Output argument is:
% At: Quantity at time t.
k=log(At1/A0)/t1;
At=A0*exp(k*t);
```

Function definition line.

Determination of *k*.

Determination of $A(t)$.
(Assignment of value to output variable.)

Sample Problem : Exponential growth and decay

For case *a*) A_0=67 , $A(t_1)$ =79, t_1=6 and t =20 :

```
>> expGD(67,79,6,20)
ans =
        116.03
```

Estimation of the population in the year 2000.

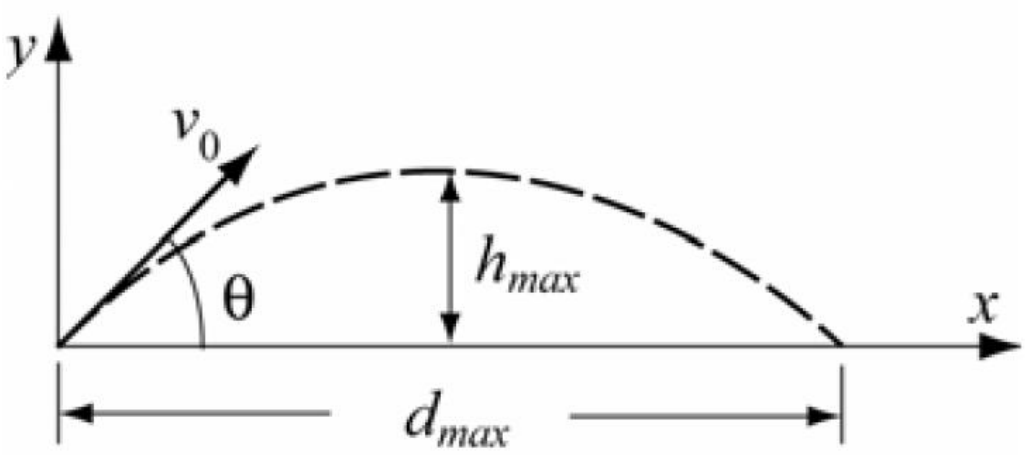
For case *b*) A_0=7 , $A(t_1)$ =3.5, (since t_1 corresponds to the half-life, which is the time required for the material to decay to half of its initial quantity), t_1=5.8 and t =30 :

```
>> expGD(7,3.5,5.8,30)
ans =
          0.19
```

The amount of material after 30 years.

Sample Problem : Motion of a projectile

- Create a function file that calculates the trajectory of a projectile. The inputs to the function are the initial velocity and the angle at which the projectile is fired. The outputs from the function are the maximum height and distance. In addition, the function generates a plot of the trajectory. Use the function to calculate the trajectory of a projectile that is fired at a velocity of 230 m/s at an angle of 39° .



Sample Problem : Motion of a projectile

Solution

- The motion of a projectile can be analyzed by considering the horizontal and vertical components. The initial velocity can be resolved into horizontal and vertical components

$$v_{0x} = v_0 \cos(\theta) \quad v_{0y} = v_0 \sin(\theta)$$

$$v_y = v_{0y} - gt \quad y = v_{0y}t - \frac{1}{2}gt^2$$

The time it takes the projectile to reach the highest point , $v_y = 0$ and the corresponding height are given by:

$$t_{hmax} = \frac{v_{0y}}{g} \text{ and } h_{max} = \frac{v_{0y}^2}{2g}$$

Sample Problem : Motion of a projectile

- IMATLAB function [hmax,dmax] = trajectory(v0,theta). The function file is:

```
function [hmax,dmax]=trajectory(v0,theta)
% trajectory calculates the max height and distance of a
projectile, and makes a plot of the trajectory.
% Input arguments are:
% v0: initial velocity in (m/s).
% theta: angle in degrees.
% Output arguments are:
% hmax: maximum height in (m).
% dmax: maximum distance in (m).
% The function creates also a plot of the trajectory.
g=9.81;
v0x=v0*cos(theta*pi/180);
v0y=v0*sin(theta*pi/180);
thmax=v0y/g;
hmax=v0y^2/(2*g);
ttot=2*thmax;
dmax=v0x*ttot;
% Creating a trajectory plot
tplot=linspace(0,ttot,200);
x=v0x*tplot;
y=v0y*tplot-0.5*g*tplot.^2;
plot(x,y)
xlabel('DISTANCE (m)')
ylabel('HEIGHT (m)')
title('PROJECTILE''S TRAJECTORY')
```

Function definition line.

Creating a time vector with 200 elements.

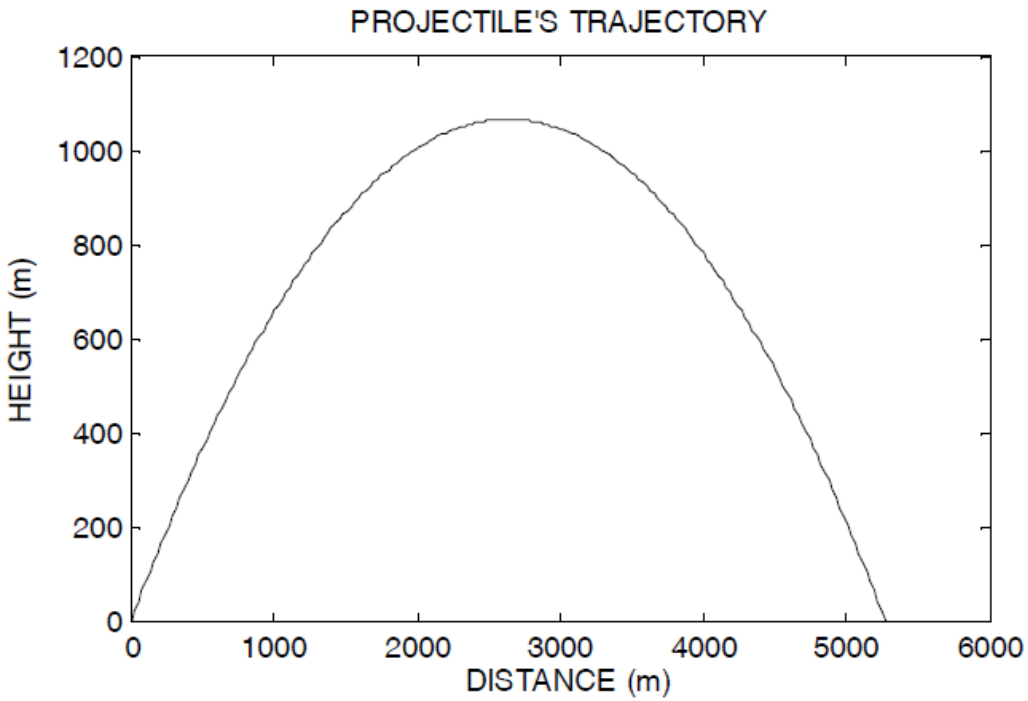
Calculating the x and y coordinates of the projectile at each time.

Note the element-by-element multiplication.

Sample Problem : Motion of a projectile

- For a projectile that is fired at a velocity of 230 m/s and an angle of 39° .

```
>> [h d]=trajectory(230,39)
h =
   1.0678e+003
d =
   5.2746e+003
```



Coursework 1: MATLAB Basics

TOTAL MARK: 5 out of 20

Date set: 03/03/2017

Required date of submission: **10/03/2017**

Save the following in a single word file and email it to TA

1. Original question
2. Your codes/commands
3. Matlab final outputs