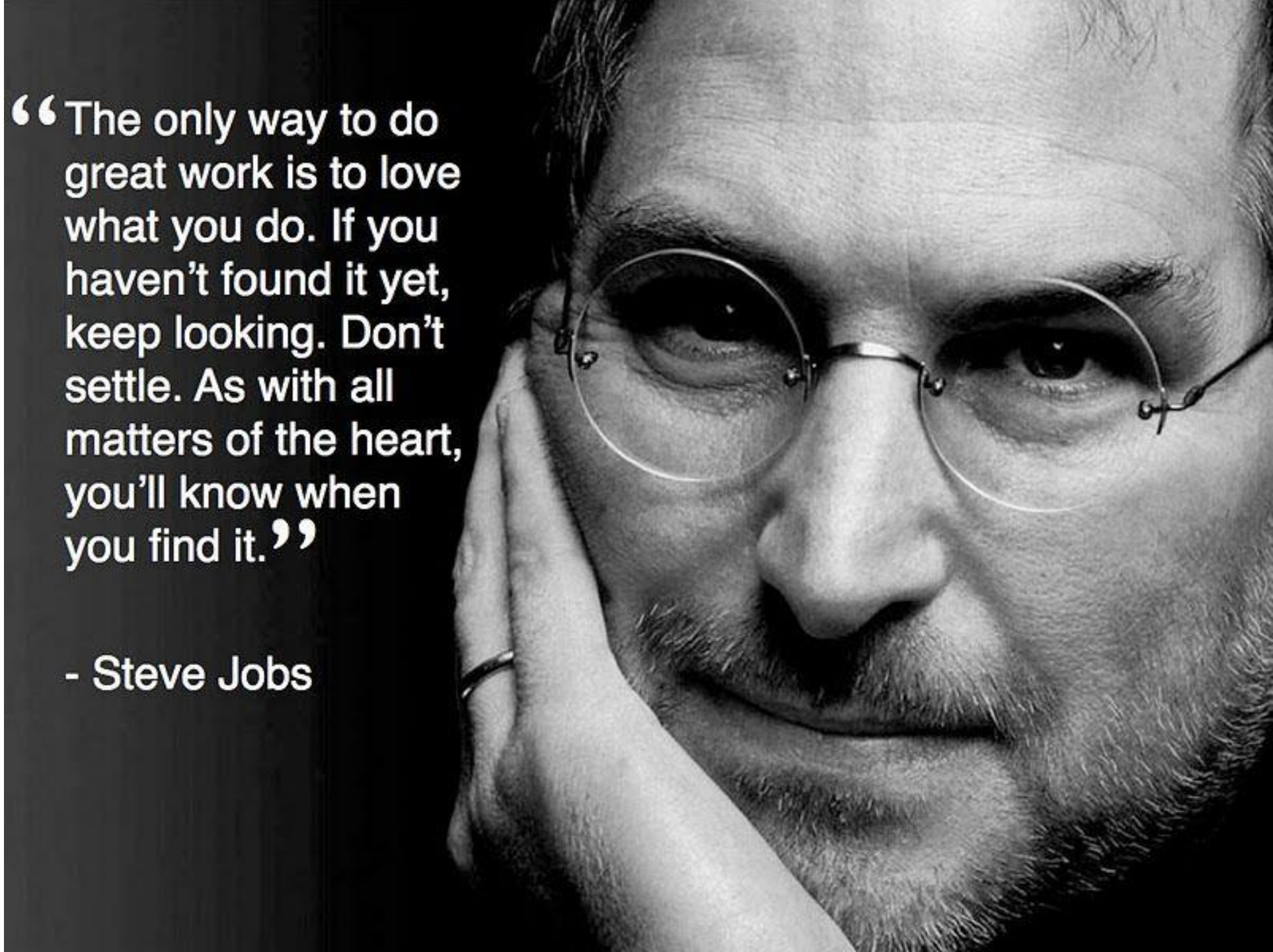
ME153 – Engineering Computation Laboratory
Class 2 – Curve Fitting & ODE Solvers

G.R.K.Gupta, MED, NITW
Mobile: 9392231833
E-mail: gupta@nitw.ac.in
12-04-2022, 1100 AM


HELLO
EVERYBODY


Take every chance
you get in life,
because some things
only happen once.
Good Morning

Today's Inspiring Quote



Peter Safar (12-04-1924)

He is a physician whose pioneering "Kiss of Life" procedure of mouth-to-mouth resuscitations is credited with saving countless lives. In the 1960s the technique was combined with new chest compressions, producing what's known today as CPR, or cardio-pulmonary resuscitation.

He also helped create the organization that, in 1976, became the World Association for disaster and Emergency Medicine. Although there are ancient references to the apparent use of mouth-to-mouth resuscitation in the Bible, the technique fell out of practice until rediscovered by Safar in the 1950s.

Also credited with playing a key role was his colleague, Dr James Elam. Safar survived a Nazi labor camp before emigrating to the U.S. after WW II

Revision of Class 1

1. Matlab Desktop – Command window,
2. Matlab Variables
3. Matlab arithmetic operators
4. Matlab Logical Operators
5. Output formats
6. Arrays - creation and Operations
7. Matrices - creation and Operations
8. Built-in Functions
9. Plotting 2-D Graphs
10. Creating, saving and executing a script file

Revision of Class 1 – Contd…

11. Creating and Executing a Function File
12. Elementary Math functions
13. Solving a Linear System of Equations

Interactive Computation

Utility Matrices

`eye(m,n)`	returns an m by n matrix with ones on the main diagonal,
`zeros(m,n)`	returns an m by n matrix of zeros,
`ones(m,n)`	returns an m by n matrix of ones,
`rand(m,n)`	returns an m by n matrix of random numbers,
`randn(m,n)`	returns an m by n matrix of normally distributed numbers,
`diag(v)`	generates a diagonal matrix with vector v on the diagonal,
`diag(A)`	extracts the diagonal of matrix A as a vector, and
`diag(A,1)`	extracts the first upper off-diagonal vector of matrix A.

```
>> A = rand(4,3)

A =
    0.8147    0.6324    0.9575
    0.9058    0.0975    0.9649
    0.1270    0.2785    0.1576
    0.9134    0.5469    0.9706
```

Create a 4 x 3 random matrix *A*.

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

ans =
    0.2785    0.1576
    0.5469    0.9706
```

Get those elements of *A* that are located in rows 3 to 4 and columns 2 to 3.

```
>> A(:,4) = A(:,1)

A =
    0.8147    0.6324    0.9575    0.8147
    0.9058    0.0975    0.9649    0.9058
    0.1270    0.2785    0.1576    0.1270
    0.9134    0.5469    0.9706    0.9134
```

Add a fourth column to *A* and set it equal to the first column of *A*.

```
>> A(2:4,2:4) = eye(3)
```

Replace the last 3 x 3 submatrix of A (rows 2 to 4, columns 2 to 4) by a 3 x 3 identity matrix.

```
A =
    0.8147    0.6324    0.9575    0.8147
    0.9058    1.0000         0         0
    0.1270         0    1.0000         0
    0.9134         0         0    1.0000
```

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

Delete the first and third rows of A.

```
A =
    0.9058    1.0000         0         0
    0.9134         0         0    1.0000
```

```
>> A = round(A)
```

Round off all entries of A.

```
A =
     1     1     0     0
     1     0     0     1
```

```
>> A(:)'
```

String out all elements of A in a row (note the transpose at the end).

```
ans =
     5     1     1     0     0     1
```

```
>> eye(3)

ans =

     1     0     0
     0     1     0
     0     0     1
```

`eye(n)` creates an $n \times n$ identity matrix. The commands `zeros`, `ones`, and `rand` work in a similar way.

```
>> B = [ones(3) zeros(3,2); zeros(2,3) 4*eye(2)]

B =

     1     1     1     0     0
     1     1     1     0     0
     1     1     1     0     0
     0     0     0     4     0
     0     0     0     0     4
```

Create a matrix *B* using submatrices made up of elementary matrices: ones, zeros, and the identity matrix of the specified sizes.

```
>> diag(B)'

ans =

     1     1     1     4     4
```

This command pulls out the diagonal of B in a row vector. Without the transpose, the result would obviously be a column vector.

```
>> diag(B,1)'

ans =

     1     1     0     0
```

The second argument of the command specifies the off-diagonal vector to be pulled out. Here we get the first upper off-diagonal vector. A negative value of the argument specifies the lower off-diagonal vectors.

```
>> d = [2 4 6 8];
>> d1 = [-3 -3 -3];
>> d2 = [-1 -1];
```

Create vectors d, $d1$, and $d2$ of length 4, 3, and 2, respectively.

```
>> D = diag(d) + diag(d1,1) + diag(d2,-2)

D =

     2    -3     0     0
     0     4    -3     0
    -1     0     6    -3
     0    -1     0     8
```

Create a matrix D by putting d on the main diagonal, $d1$ on the first upper diagonal, and $d2$ on the second lower diagonal.

`rot90`	rotates a matrix by 90^o,
`fliplr`	flips a matrix from left to right,
`flipud`	flips a matrix from up to down,
`tril`	extracts the lower triangular part of a matrix,
`triu`	extracts the upper triangular part of a matrix, and
`reshape`	changes the shape of a matrix.

Relational Operations

<	less than
<=	less than or equal
>	greater than
>=	greater than or equal
==	equal
~=	not equal

Examples: If $x = [1 \;\; 5 \;\; 3 \;\; 7]$ and $y = [0 \;\; 2 \;\; 8 \;\; 7]$, then

`k = x < y`	results in $k = [0\ 0\ 1\ 0]$	because $x_i < y_i$ for $i = 3$,
`k = x <= y`	results in $k = [0\ 0\ 1\ 1]$	because $x_i \leq y_i$ for $i = 3$ and 4,
`k = x > y`	results in $k = [1\ 1\ 0\ 0]$	because $x_i > y_i$ for $i = 1$ and 2,
`k = x >= y`	results in $k = [1\ 1\ 0\ 1]$	because $x_i \geq y_i$ for $i = 1, 2$, and 4
`k = x == y`	results in $k = [0\ 0\ 0\ 1]$	because $x_i = y_i$ for $i = 4$, and
`k = x ~= y`	results in $k = [1\ 1\ 1\ 0]$	because $x_i \neq y_i$ for $i = 1, 2$, and 3

Logical Operations

`&`	logical AND
`|`	logical OR
`~`	logical complement (NOT)
`xor`	exclusive OR

Examples: For two vectors $x = [0\ \ 5\ \ 3\ \ 7]$ and $y = [0\ \ 2\ \ 8\ \ 7]$,

`m = (x>y)&(x>4)` results in $m = [0\ 1\ 0\ 0]$, because the condition is true only for x_2,

`n = x|y` results in $n = [0\ 1\ 1\ 1]$, because either x_i or y_i is nonzero for $i = [2\ 3\ 4]$,

`m = ~(x|y)` results in $m = [1\ 0\ 0\ 0]$, which is the logical complement of `x|y`, and

`p = xor(x,y)` results in $p = [0\ 0\ 0\ 0]$, because there is no such index i for which x_i or y_i, but not both, is nonzero.

Elementary Math Functions

Trigonometric functions

`sin`, `sind`	sine,	`sinh`	hyperbolic sine,
`asin`, `asind`	inverse sine,	`asinh`	inverse hyperbolic sine,
`cos`, `cosd`	cosine,	`cosh`	cyperbolic cosine,
`acos`, `acosd`	inverse cosine,	`acosh`	inverse hyperbolic cosine,
`tan`, `tand`	tangent,	`tanh`	hyperbolic tangent,
`atan`, `atand`	inverse tangent,	`atanh`	inverse hyperbolic tangent,
`atan2`	four-quadrant $\tan^{-1}$,		
`sec`, `secd`	secant,	`sech`	hyperbolic secant,
`asec`, `asecd`	inverse secant,	`asech`	inverse hyperbolic secant,
`csc`, `cscd`	cosecant,	`csch`	hyperbolic cosecant,
`acsc`, `acscd`	inverse cosecant,	`acsch`	inverse hyperbolic cosecant,
`cot`, `cotd`	cotangent,	`coth`	hyperbolic cotangent,
`acot`, `acotd`	inverse cotangent, and	`acoth`	inverse hyperbolic cotangent.

The angles given to these functions as arguments must be in *radians* for **sin**, **cos**, etc., and in *degrees* for **sind**, **cosd**, etc. Thus, **sin(pi/2)** and **sind(90)** produce the same result. All of these functions, except **atan2**, take a single scalar, vector, or matrix as input argument. The function **atan2** takes two input arguments, **atan2(y,x)**, and produces the four-quadrant inverse tangent such that $-\pi \leq \tan^{-1} \frac{y}{x} \leq \pi$. This gives the angle a rectangular to polar conversion.

Examples: If **q=[0 pi/2 pi]**, **x=[1 -1 -1 1]**, and **y=[1 1 -1 -1]**, then

sin(q)	gives **[0 1 0]**,
sinh(q)	gives **[0 2.3013 11.5487]**,
atan(y./x)	gives **[0.7854 -0.7854 0.7854 -0.7854]**, and
atan2(y,x)	gives **[0.7854 2.3562 -2.3562 -0.7854]**.

Exponential Functions

`exp`	exponential, *Example:* `exp(A)` produces a matrix with elements $e^{(A_{ij})}$. So how do you compute e^A? See the next section.
`log`	natural logarithm, *Example:* `log(A)` produces a matrix with elements $\ln(A_{ij})$.
`log10`	base 10 logarithm, *Example:* `log10(A)` produces a matrix with elements $\log_{10}(A_{ij})$.
`sqrt`	square root, *Example:* `sqrt(A)` produces a matrix with elements $\sqrt{A_{ij}}$. But what about $\sqrt{A}$? See the next section.
`nthroot`	real nth root of real numbers, *Example:* `nthroot(A,3)` produces a matrix with elements $\sqrt[3]{A_{ij}}$.

In addition, `log2`, `pow2`, `nextpow2`, `realpow`, `reallog`, `realsqrt`, `log1p` (for $\log(1+x)$), and `expm1` (for $e^x - 1$) functions exist in MATLAB. Clearly, these are array operations. You can, however, also compute matrix exponential e^A, matrix square root $\sqrt{A}$, etc. See Section 3.2.5.

Round-off Functions

`fix`	round toward 0, *Example:* `fix([-2.33 2.66])` $= [-2 \quad 2]$.
`floor`	round toward $-\infty$, *Example:* `floor([-2.33 2.66])` $= [-3 \quad 2]$.
`ceil`	round toward $+\infty$, *Example:* `ceil([-2.33 2.66])` $= [-2 \quad 3]$.
`mod`	modulus after division; `mod(a,b)` is the same as `a-floor(a./b)*b`, *Example:* `mod(26,5)` $= 1$ and `mod(-26,5)` $= 4$.
`round`	round toward the nearest integer, *Example:* `round([-2.33 2.66])` $= [-2 \quad 3]$.
`rem`	remainder after division, `rem(a,b)` is the same as `a-fix(a./b)*b`, *Example:* If `a=[-1.5 7], b=[2 3]`, then `rem(a,b)` $= [-1.5 \; 1]$.
`sign`	signum function, *Example:* `sign([-2.33 2.66])` $= [-1 \quad 1]$.

Matrix Functions

`expm(A)`	finds the exponential of matrix A, e^A,
`logm(A)`	finds $\log(A)$ such that $A = e^{\log(A)}$, and
`sqrtm(A)`	finds $\sqrt{A}$.

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

A =

     1     2
     3     4

>> asqrt = sqrt(A)

asqrt =

    1.0000    1.4142
    1.7321    2.0000

>> Asqrt = sqrtm(A)

Asqrt =

   0.5537 + 0.4644i   0.8070 - 0.2124i
   1.2104 - 0.3186i   1.7641 + 0.1458i
```

`sqrt` is an array operation. It gives the square root of each element of A as is evident from the output here.

`sqrtm`, on the other hand, is a true matrix function, i.e., it computes $\sqrt{A}$. Thus `[Asqrt]*[Asqrt]` $= [A]$.

```
>> exp_aij = exp(A)

exp_aij =

    2.7183    7.3891
   20.0855   54.5982

>> exp_A = expm(A)

exp_A =

   51.9690   74.7366
  112.1048  164.0738
```

Similarly, `exp` gives an element-by-element exponential of the matrix, whereas `expm` finds the true matrix exponential e^A. For information on other matrix functions, type `help matfun`.

Linear Algebra

Solving a Linear System

$$
\begin{aligned}
5x - 3y + 2z &= 10 \\
-3x + 8y + 4z &= 20 \\
2x + 4y - 9z &= 9
\end{aligned}
$$

```
>> A = [5 -3 2; -3 8 4; 2 4 -9];   % Enter matrix A
>> b = [10; 20; 9];                % Enter column vector b
>> x = A\b                         % Solve for x
x =
    3.4442
    3.1982
    1.1868

>> c = A*x                         % check the solution
c =
   10.0000
   20.0000
    9.0000
```

The backslash (\) or the left division is used to solve a linear system of equations $[\mathbf{A}]\{\mathbf{x}\} = \{\mathbf{b}\}$. For more information, type: `help slash`.

Eigen Values and Eigen Vectors

```
>> A = [5 -3  2; -3  8  4; 4  2  -9];
>> [V,D] = eig(A)

V =

    0.1725    0.8706   -0.5375
    0.2382    0.3774    0.8429
   -0.9558    0.3156   -0.0247


D =

  -10.2206         0         0
         0    4.4246         0
         0         0    9.7960
```

Here **V** is a matrix containing the eigenvectors of **A** as its columns. For example, the first column of **V** is the first eigenvector of **A**.

D is a matrix that contains the eigenvalues of **A** on its diagonal.

Matrix factorizations

1. **LU factorization:** The name of the built-in function is `lu`. To get the LU factorization of a square matrix **A**, type the command

```
[L,U] = lu(A);
```

MATLAB returns a lower triangular matrix **L** and an upper triangular matrix **U** such that `L*U=A`. It is also possible to get the permutation matrix as an output (see the on-line help on `lu`).

2. **QR factorization:** The name of the built-in function is `qr`. Typing the command

```
[Q,R] = qr(A);
```

returns an orthogonal matrix **Q** and an upper triangular matrix **R** such that `Q*R=A`. For more information, see the on-line help.

3. **Cholesky factorization:** If you have a positive definite matrix **A**, you can factorize the matrix with the built-in function `chol`. The command

```
R = chol(A);
```

produces an upper triangular matrix **R** such that `R'*R=A`.

4. **Singular value decomposition (svd):** The name of the built-in function is **svd**. If you type

```
[U,D,V] = svd(A);
```

MATLAB returns two orthogonal matrices, **U** and **V**, and a diagonal matrix **D**, with the *singular values* of **A** as the diagonal entries, such that `U*D*V=A`.

if, elseif, else - Syntax

```
if expression
        statements
elseif expression
        statements
else
        statements
end
```

Example for if, elseif, else (save in a file and execute)

```
x = input(‘give a value for x’)
minVal = 20;
maxVal = 60;
if (x >= minVal) && (x <= maxVal)
        disp('Value within specified range.’)
elseif (x > maxVal)
        disp('Value exceeds maximum value.’)
else
        disp('Value is below minimum value.’)
end
% Execute for different values of x
```

```
%Create a matrix of 1s.
nrows = 4;
ncols = 6;
A = ones(nrows,ncols);
 for c = 1:ncols
        for r = 1:nrows
                if r == c
                        A(r,c) = 2;
                elseif abs(r-c) == 1
                        A(r,c) = -1;
                else A(r,c) = 0;
                end
        end
end
A
```

Result

$A = 4 \times 6$

2	-1	0	0	0	0
-1	2	-1	0	0	0
0	-1	2	-1	0	0
0	0	-1	2	-1	0

for - Syntax

for *index* = *values*
    statements
end

index

1. *initVal*:*endVal* — Increment the *index* variable from *initVal* to *endVal* by 1, and repeat execution of *statements* until *index* is greater than *endVal*.
2. *initVal*:*step*:*endVal* — Increment *index* by the value *step* on each iteration, or decrements *index* when *step* is negative.
3. *valArray* — Create a column vector, *index*, from subsequent columns of array *valArray* on each iteration. For example, on the first iteration, *index* = *valArray*(:,1). The loop executes a maximum of *n* times, where *n* is the number of columns of *valArray*, given by numel(*valArray*(1,:)). The input *valArray* can be of any MATLAB® data type, including a character vector, cell array, or struct.

Example - for

```
for v = 1.0:-0.2:0.0
        disp(v)
end
```

Result

1
0.8000
0.6000
0.4000
0.2000
0

Example 2 - for

```
for v = [1 5 8 17]
        disp(v)
end
```

```
1
5
8
17
```

while - Syntax

```
while expression
      statements
end
```

while *expression*, *statements*, end evaluates an expression, and repeats the execution of a group of statements in a loop while the expression is true. An expression is true when its result is nonempty and contains only nonzero elements (logical or real numeric). Otherwise, the expression is false.

Example: calculate factorial(n)

```
n = input(‘give a value to get factorial=’)
f = n;
while n > 1
        n = n-1;
        f = f*n;
end
disp(['n! = ' num2str(f)])
```

Switch - Syntax

```
switch_expression
        case case_expression
                statements
        case case_expression
                statements
        ...
        otherwise
                statements
end
```

switch *switch_expression*, case *case_expression*, end evaluates an expression and chooses to execute one of several groups of statements. Each choice is a case.

The switch block tests each case until one of the case expressions is true. A case is true when:

1. For numbers, *case_expression* == *switch_expression*.
2. For character vectors, strcmp(*case_expression*,*switch_expression*) == 1.
3. For a cell array *case_expression*, at least one of the elements of the cell array matches *switch_expression*, as defined above for numbers, character vectors, and objects.

1. When a case expression is true, MATLAB® executes the corresponding statements and exits the switch block.
2. An evaluated *switch_expression* must be a scalar or character vector. An evaluated *case_expression* must be a scalar, a character vector, or a cell array of scalars or character vectors.
3. The otherwise block is optional. MATLAB executes the statements only when no case is true.

Example - switch

```
n = input('Enter a number: ');
switch n
    case -1
        disp('negative one')
    case 0
        disp('zero')
    case 1
        disp('positive one')
    otherwise
        disp('other value')
end
```

```
x = [12 64 24];
plottype = input('give plot type as bar or pie in single quotes=')
switch plottype
    case 'bar'
        bar(x)
        title('Bar Graph')
    case {'pie'}
        pie(x)
        title('Pie Chart')
    otherwise
        warning('Unexpected plot type. No plot created.')
end
```

Ordinary Differential Equation Solver

[t,y] = ode45(odefun,tspan,y0)

where tspan = [t0 tf], integrates the system of differential equations y′=f(t,y) from t0 to tf with initial conditions y0. Each row in the solution array y corresponds to a value returned in column vector t.

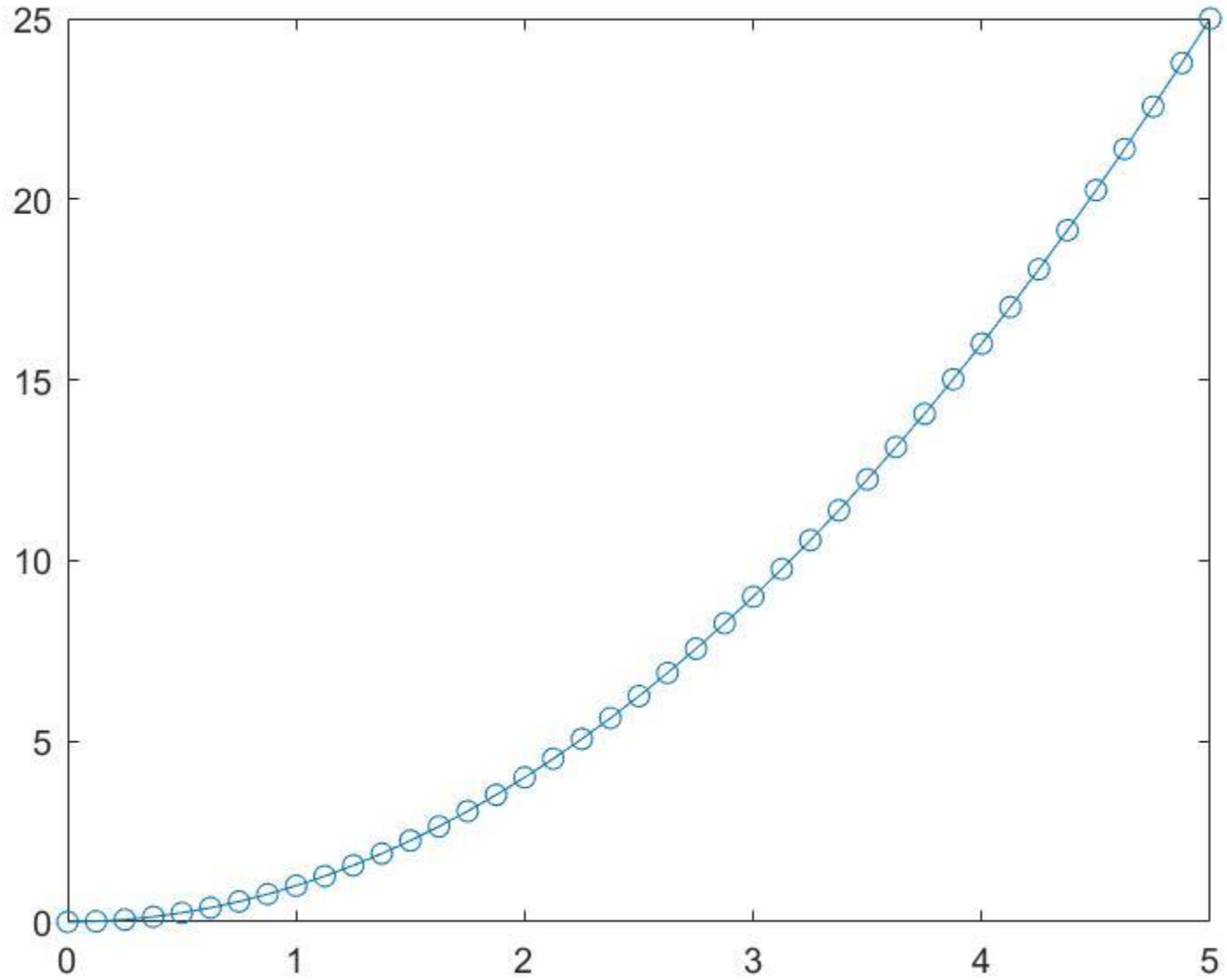
First order ODE

Solve the ODE

$y'=2t$.

Use a time interval of [0,5] and the initial condition y0 = 0.

```
tspan = [0 5];
y0 = 0;
[t,y] = ode45(@(t,y) 2*t, tspan, y0);
plot(t,y,'-o')
```


25
20
15
10
5
0
0
1
2
3
4
5

Solution of The van der Pol equation
Second Order ODE

The van der Pol equation is a second order ODE

$$y''_1 - \mu(1 - y_1^2) y'_1 + y_1 = 0$$

where $\mu > 0$ is a scalar parameter. Rewrite this equation as a system of first-order ODEs by making the substitution $y'_1 = y_2$.
The resulting system of first-order ODEs is

$$y'_1 = y_2$$
$$y'_2 = \mu(1 - y_1^2) y_2 - y_1$$

he function file vdp1.m represents the van der Pol equation using $\mu = 1$. The variables y_1 and y_2 are the entries y(1) and y(2) of a two-element vector, dydt.

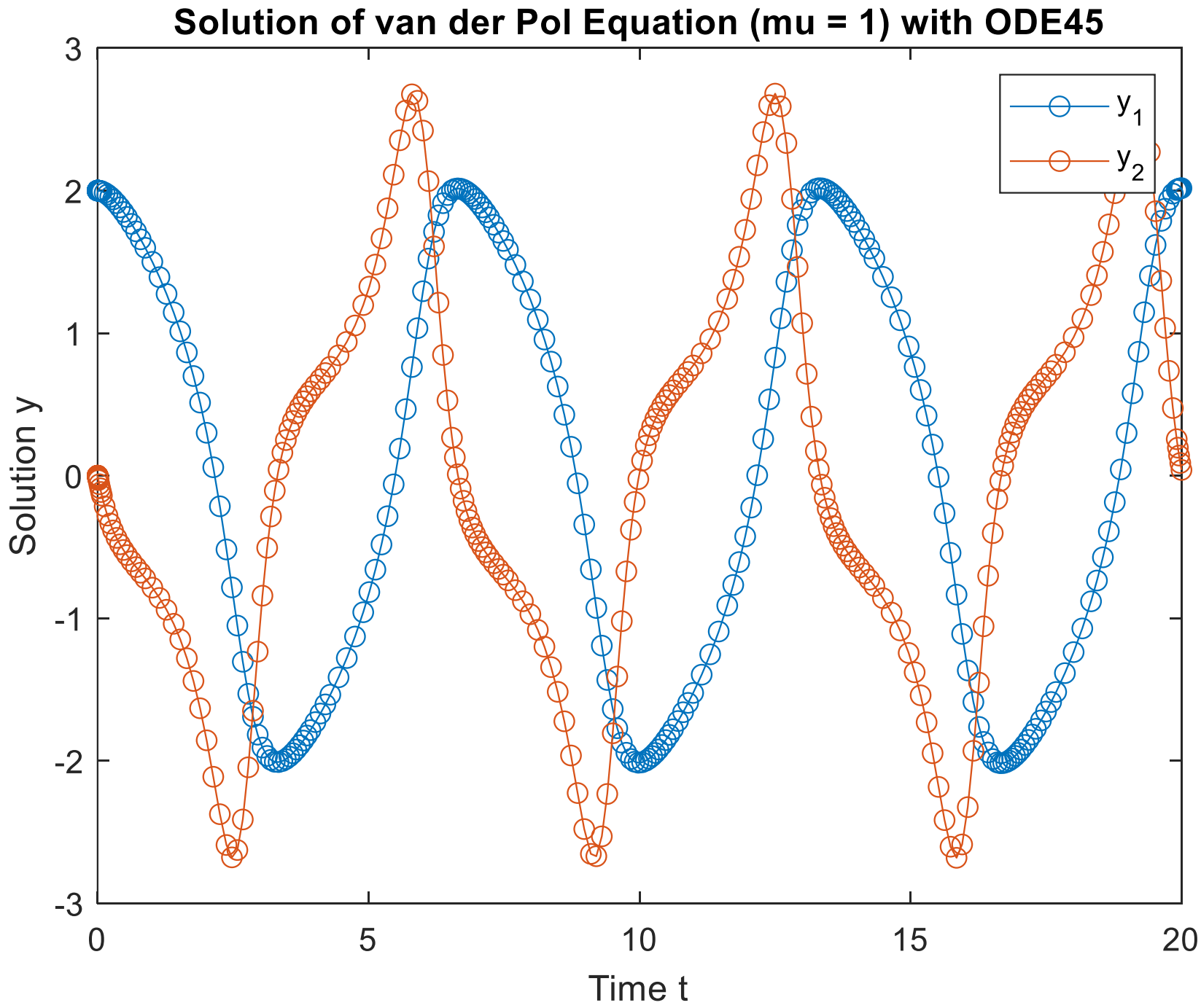
Function

```
function dydt = vdp1(t,y)
%VDP1  Evaluate the van der Pol ODEs for mu = 1
dydt = [y(2); (1-y(1)^2)*y(2)-y(1)];
```

Solve the ODE using the ode45 function on the time interval [0 20] with initial values [2 0]. The resulting output is a column vector of time points t and a solution array y. Each row in y corresponds to a time returned in the corresponding row of t. The first column of y corresponds to y_1, and the second column to y_2.

Main Program

```
[t,y] = ode45(@vdp1,[0 20],[2; 0]);
plot(t,y(:,1),'-o',t,y(:,2),'-o')
title('Solution of van der Pol Equation (\mu = 1) with ODE45');
xlabel('Time t');
ylabel('Solution y');
legend('y_1','y_2')
```


Solution of van der Pol Equation (mu = 1) with ODE45
Solution y
Time t
y1
y2
3
2
1
0
-1
-2
-3
0
5
10
15
20

Curve Fitting and Interpolation

Polynomial curve fitting on the fly

Curve fitting is a technique of finding an algebraic relationship that "best" (in a least squares sense) fits a given set of data. Unfortunately, there is no magical function (in MATLAB or otherwise) that can give you this relationship if you simply supply the data. You have to have an idea of what kind of relationship might exist between the input data (x_i) and the output data (y_i)· However, if you do not have a firm idea but you have data that you trust, MATLAB can help you explore the best possible fit. MATLAB includes Basic Fitting in it’s figure window's Tools menu that lets you fit a polynomial curve (up to the tenth order) to your data on the fly.

It also gives you options of displaying the residual at the data points and computing the norm of the residuals. This can help in comparing different fits and then selecting the one that makes you happy. Let us take an example and go through the steps.

Example 1: Straight-line (linear) fit

Let us say that we have the following data for x and y and we want to get the best linear (straight-line) fit through this data.

x	5	10	20	50	100
y	15	33	53	140	301

Here is all it takes to get the best linear fit, along with the equation of the fitted line.

Step 1: Plot raw data: Enter the data and plot it as a scatter plot using some marker, say, circles.

```
x = [5 10 20 50 100];          % x-data
y = [15 33 53 140 301];        % y-data
plot(x,y,'o')                  % plot x vs y using circles
```

Step 2: Use built-in Basic Fitting to do a linear fit: Go to your figure window, click on Tools, and select Basic Fitting from the pull-down menu (see Fig. 5.2). A separate window appears with Basic Fitting options.

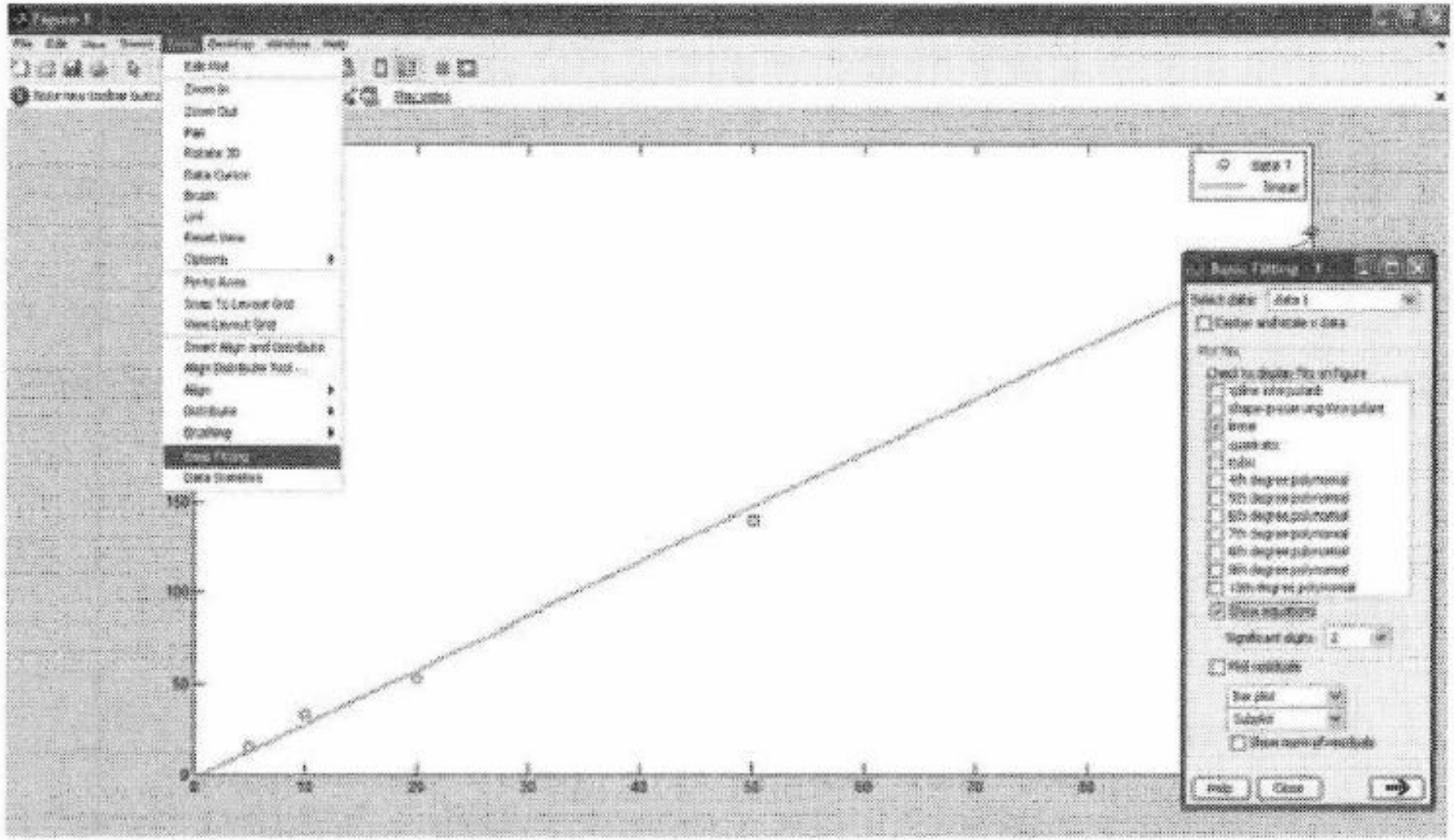


Figure 5.2: First plot the raw data. Then select Basic Fitting from the Tools menu of the figure window. A separate window appears with several Basic Fitting options for polynomial curve fits. Check appropriate boxes in this window to get the desired curve fit and other displays, such as the fitted equation.

Step 3: Fit a linear curve and display the equation: Check the boxes for linear and Show equations from the Basic Fitting window options. The best fitted line as well as its equation appear in the figure window. You are now done. The result is shown in Fig. 5.3.

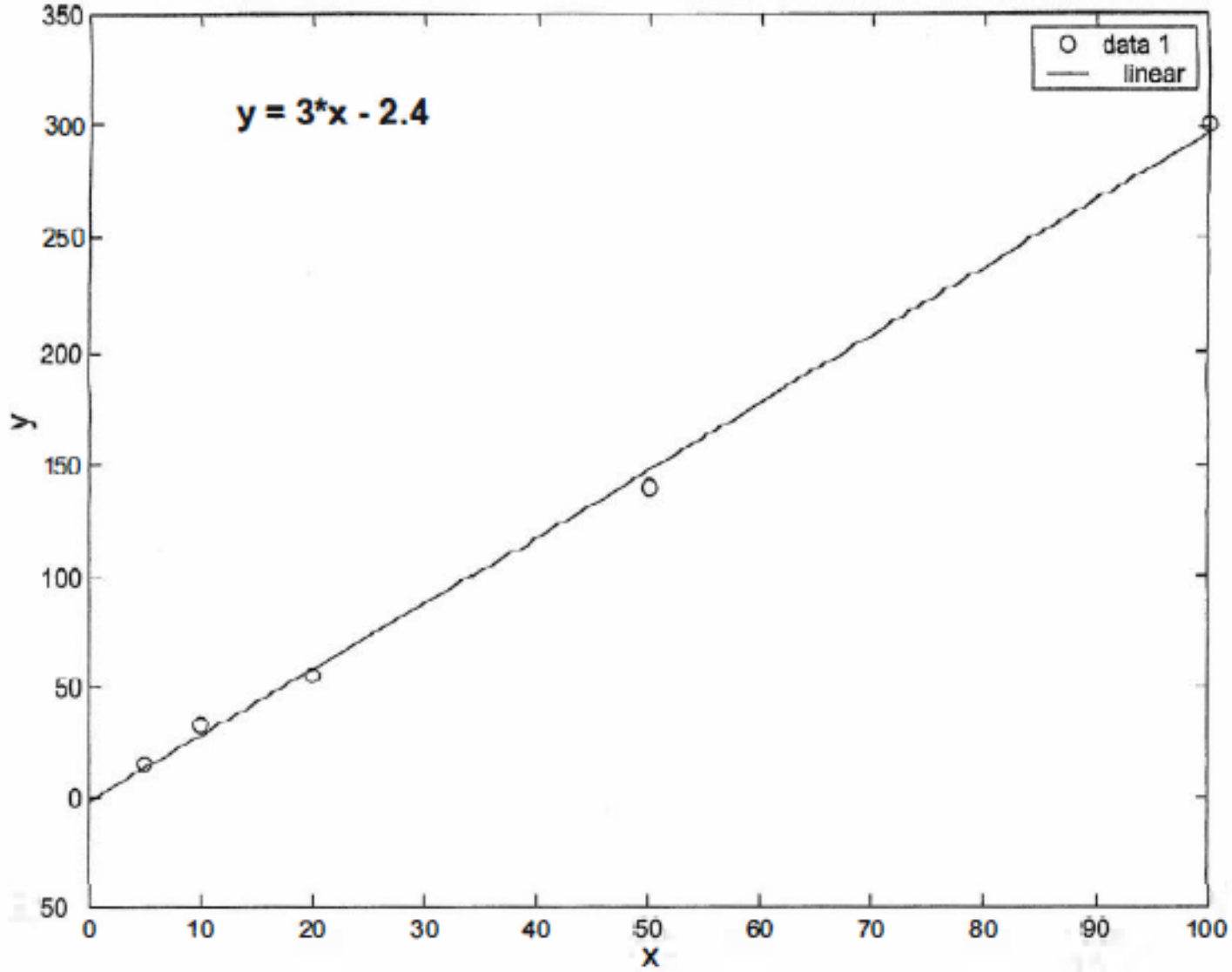


Figure 5.3: Linear curve fit through the data using Basic Fitting from the Tools menu of the figure window.

Functions polyfit and polyval

polyfit: Given two vectors x and y, the command **a = polyfit (x,y,n)** fits a polynomial of order n through the data points (x_i,y_i) and returns (n+ 1) coefficients of the powers of x in the row vector a. The coefficients are arranged in the decreasing order of the powers of x, i.e., a = [a_n a_{n-1}… a_1 a_o].

polyval: Given a data vector x and the coefficients of a polynomial in a row vector a, the command **y = polyval (a, x)** evaluates the polynomial at the data points x_i and generates the values y_i such that

$$y_i = a(1)x_i^n + a(2)x_i^{n-1} + \cdots + a(n)x + a(n+1).$$

Example for polyfit and polyval

The following data is obtained from an experiment aimed at measuring the spring constant of a given spring. Different masses m are hung from the spring and the corresponding deflections δ of the spring from its unstretched configuration are measured. From physics, we know that F = k δ and here F = mg. Thus, we can find k from the relationship k = mg/δ. Here, however, we are going to find k by plotting the experimental data, fitting the best straight line (we know that the relationship between δ and F is linear) through the data, and then measuring the slope of the best-fit line.

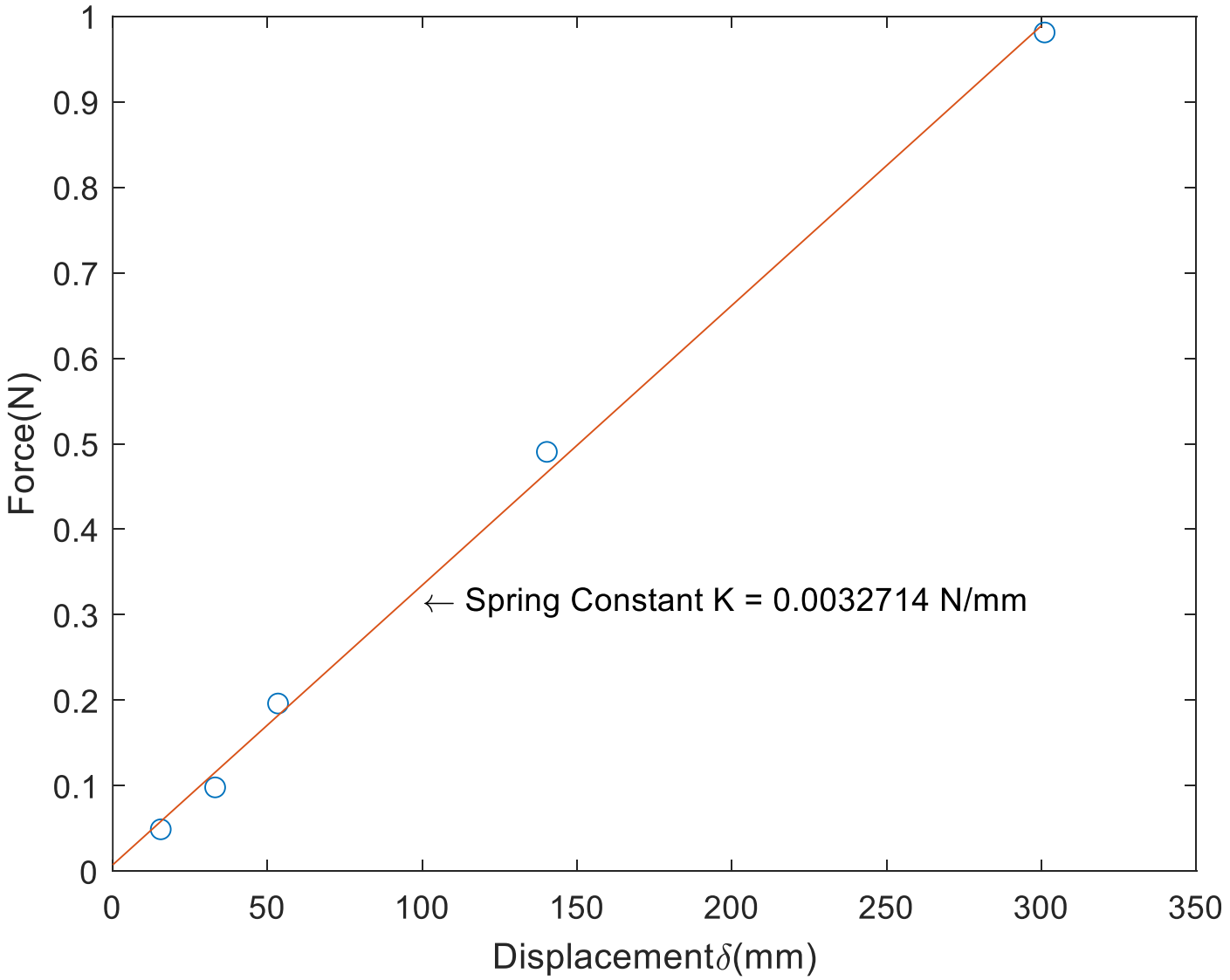
m(g)	5.00	10.00	20.00	50.00	100.00
δ(mm)	15.5	33.07	53.39	140.24	301.03

Fit a polynomial through theses points and evaluate at some values

Code

```
m=[5 10 20 50 100];                          % mass data (g)
d=[15.5 33.07 53.39 140.24 301.03]; % displacement data (mm)
g=9.81;                                      % g = 9.81 m/s^2
F=m/1000*g;                                  % compute spring force (N)
a=polyfit(d,F,1);                            % fit a line (1st order polynomial)
d_fit=0:10:300;                              % make a finer grid on d
F_fit=polyval(a,d_fit);                      % evaluate the polynomial at new points
plot(d,F,'o',d_fit,F_fit)                    % plot data and the fitted curve
xlabel('Displacement \delta (mm)'),ylabel('Force (N)')
k=a(1);                                      % Find the spring constant
text(100,.32,['\leftarrow Spring Constant K = ',num2str(k),' N/mm']);
```

Solution


Force(N)
Displacementδ(mm)
← Spring Constant K = 0.0032714 N/mm
0
0.1
0.2
0.3
0.4
0.5
0.6
0.7
0.8
0.9
1
0
50
100
150
200
250
300
350

Interpolation – interp1

One-dimensional data interpolation, i.e. , given y_i at x_i , finds y_j at desired x_j from $y_j = f(x_j)$. Here f is a continuous function that is found from interpolation. It is called one-dimensional interpolation because y depends on a single variable x. The calling syntax is

```
ynew = interp1(x,y,xnew,method)
```

You have an option of specifying a **method** of interpolation. The choices for method are **nearest, linear, cubic, or spline**. The choice of the method dictates the smoothness of the interpolated data. The default method is linear. To specify cubic interpolation instead of linear, for example, in interp1, use the syntax

```
ynew = interp1(x,y,xnew,'cubic')
```

Example

There are two simple steps involved in interpolation-providing a list (a vector) of points at which you wish to get interpolated data (this list may include points at which data is already available), and executing the appropriate function (e.g. , interp1) with the desired choice for the method of interpolation. We illustrate these steps through an example on the x and y data given in the following table.

x	0	0.785	1.570	2.356	3.141	3.927	4.712	5.497	6.283
y	0	0.707	1.000	0.707	0.000	−0.707	−1.000	−0.707	−0.000

Step 1: Generate a vector `xi` containing desired points for interpolation.

```
% take equally spaced fifty points.
xi = linspace(0,2*pi,50);
```

Step 2: Generate data *yi* at *xi*.

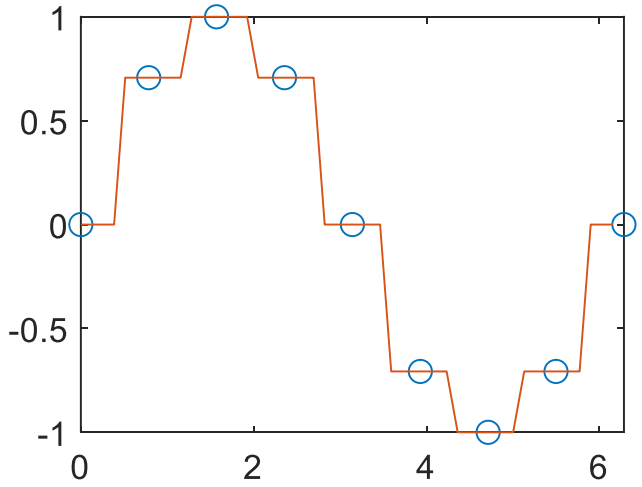
```
% generate yi at xi with cubic interpolation.
yi = interp1(x,y,xi,'cubic');
```

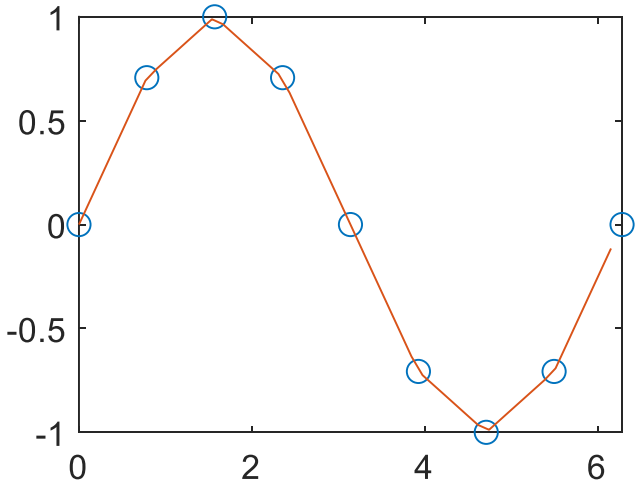
Here, `'cubic'` is the choice for interpolation scheme. The other schemes we could use are `nearest`, `linear`, and `spline`. The data generated by each scheme is shown in Fig. 5.7, along with the original data. The corresponding curves show the smoothness obtained.

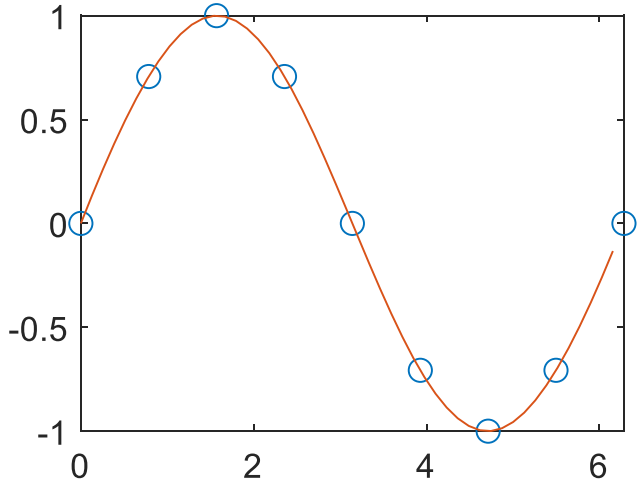
Code

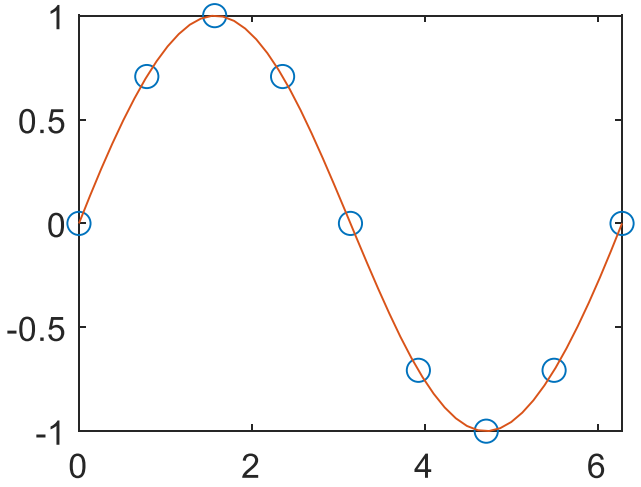
```
x=[0 0.785 1.570 2.356 3.141 3.927 4.712 5.497 6.283]
y=[0 0.707 1.000 0.707 0.000 -0.707 -1.000 -0.707 -0.000]
xi=linspace(0,2*pi,50);
yi=interp1(x,y,xi,'nearest')
subplot(2,2,1)
plot(x,y,'o',xi,yi)
yi=interp1(x,y,xi,'linear')
subplot(2,2,2)
plot(x,y,'o',xi,yi)
yi=interp1(x,y,xi,'cubic')
subplot(2,2,3)
plot(x,y,'o',xi,yi)
yi=interp1(x,y,xi,'spline')
subplot(2,2,4)
plot(x,y,'o',xi,yi)
```

Solution









Today’s Inspiring Story

General Motor’s Problem - Story

Never underestimate your clients' complaints. No matter how funny they might seem !

This is a real story that happened between a customer of General Motors and its Customer-Care Executive. Pls read on

A complaint was received by the Pontiac Division of General Motors:

'This is the second time I have written to you. I don't blame you for not answering me, because I sounded crazy, but it is a fact that we have a tradition in our family of Ice-Cream for dessert after dinner each night. But the kind of ice cream varies so, every night, after we've eaten, the whole family votes on which flavor of ice cream we should have and I drive down to the store to get it.

It's also a fact that I recently purchased a new Pontiac and since then my trips to the store have created a problem.

You see, every time I buy a vanilla ice-cream, when I start back from the store my car won't start. If I get any other flavour of ice cream, the car starts just fine. I want you to know I'm serious about this question, no matter how silly it sounds, "What is there about a Pontiac that makes it not start when I get vanilla ice cream, and easy to start whenever I get any other kind?"

The Pontiac President was understandably skeptical about the letter, but sent an Engineer to check it out anyway.

The latter was surprised to be greeted by an obviously well educated man in a fine neighborhood.

He had arranged to meet the man just after dinner time, so the two hopped into the car and drove to the ice cream store. It was vanilla ice cream that night and, sure enough, after they came back to the car, it wouldn't start.

The Engineer returned for three more nights. The first night, they got chocolate. The car started.

The second night, he got strawberry. The car started. The third night he ordered vanilla. The car failed to start.

Now the engineer, being a logical man, refused to believe that this man's car was allergic to vanilla ice cream. He arranged, therefore, to continue his visits for as long as it took to solve the problem. And toward this end he began to take notes: He jotted down all sorts of data: time of day, type of gas used, time to drive back and forth etc.

In a short time, he had a clue: the man took less time to buy vanilla than any other flavor. Why? The answer was in the layout of the store.

Vanilla, being the most popular flavor, was in a separate case at the front of the store for quick pickup. All the other flavors were kept in the back of the store at a different counter where it took considerably longer to check out the flavor.

Now, the question for the Engineer was why the car wouldn't start when it took less time.

Eureka - Time was now the problem - not the vanilla ice cream! The engineer quickly came up with the answer: "Vapor lock".

It was happening every night; but the extra time taken to get the other flavors allowed the engine to cool down sufficiently to start.

When the man got vanilla, the engine was still hot for the vapor lock to dissipate.

Even crazy looking problems are sometimes real and all problems seem to be simple only when we find the solution, with cool thinking.

What really matters is your attitude and your perception. So never be shy to ask out your questions.

Quite interesting!

That’s how insights occur.

It is called Gestalt in psychology - filling the gap in cognition.


THANK
YOU!