COMPUTATIONAL PHYSICS LAB

submitted by
S.ASHWIN BALAJI
B170632EP

DEPARTMENT OF PHYSICS

NATIONAL INSTITUTE OF TECHNOLOGY CALICUT

Bisection Method

Aim

To implement the Bisection method to numerically compute the roots of non-trivial equations using MATLAB.

Algorithm

- Step 1: Read the range to work with, say x_1 and x_2
- Step 2: Read the tolerance limit, say e
- Step 3: Compute the no of iterations using the formula $\frac{\log(|x_2 - x_1|/e)}{\log(2)}$
- Step 4: If $f(x_1) \times f(x_2)$ is greater than 0 prompt the user about the invalid range and terminate the program
- Step 5: If $f(x_1) \times f(x_2)$ is less than 0 compute $x = \frac{(x_1+x_2)}{2}$
- Step 6: If $|((x_1 - x_2)/x)|$ is greater than or equal to e proceed to step 7 else goto step 10
- Step 7: Calculate the percentage error value using $\left|\frac{x_i - x_{i-1}}{x_i}\right| \times 100$
- Step 8: If $f(x_1) \times f(x)$ is less than 0 update x_2 as x and goto step 5 else goto step 9
- Step 9: Update x_1 as x and goto step 5
- Step 10: Plot percentage error vs no of iterations graph
- Step 11: Display the root of the equation in given interval, no of iterations computed earlier and actual no of iterations

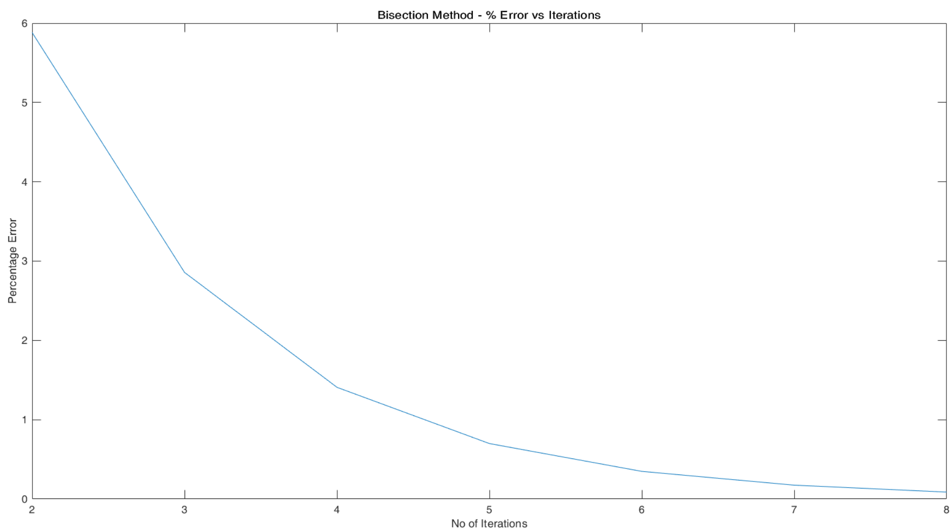
Code

```
function outputnum = func(x)
outputnum = x.^2 - 4;
end

function root = bisection()

er = zeros(100);
co = 0;

x1 = input('Enter value of x1: ');
x2 = input('Enter value of x2: ');
e = input('Enter value tolerance limit: ');

nTh = log( abs(x2 - x1) / e ) / log(2);

if func(x1) * func(x2) > 0
    disp('Wrong Choice of Range, please check')
    return;
else
    x = (x1 + x2) / 2;
    while ( abs((x1 - x2) / x ) >= e)
        co = co + 1;
        er(co) = x;
```

```
        x = (x1 + x2) / 2;
        er(co) = 100 * abs(( x - er(co) ) / x);
        if ((func(x) * func(x1)) < 0)
            x2 = x;
        else
            x1 = x;
        end
    end
end

r = 1:co;
plot(r(2:co),er(2:co));
xlabel('No of Iterations');
ylabel('Percentage Error');
title('Bisection Method - % Error vs Iterations');

fprintf('\nTheoretical value of No of Iterations: %i', floor(nTh));
fprintf('\nTotal no. of Iterations:  %i', co);
fprintf('\nRoot of the given function: %f', x);

root = x;
end
```

Output

```
Command Window
>> bisection()
Enter value of x1: 2.0
Enter value of x2: 2.5
Enter value tolerance limit: 0.001

Theoretical value of No of Iterations: 8
Total no. of Iterations:  8
Root of the given function: 2.236328
ans =

    2.2363

fx >>
```



Newton Raphson Method

Aim

To implement the Newton Raphson method to numerically improve the accuracy of the roots of non-trivial equations using MATLAB.

Algorithm

- Step 1: Read the approximate root value to the given function to work with, say x
- Step 2: Read the tolerance limit, say e
- Step 3: Compute a better approximation using Newton Raphson formula
  $x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}$
- Step 4: $f'(x_n)$ could be computed by directly providing the derivative manually to MATLAB or by a simple approximation
  $f'(x_n) = \frac{f(x_n+0.001)-f(x_n-0.001)}{0.002}$
- Step 5: Calculate the percentage error value using
  $\left|\frac{x_i - x_{i-1}}{x_i}\right| \times 100$
- Step 6: If $|(x_{n+1} - x_n)/x_n|$ is greater than or equal to e proceed to step 7 else goto step 3
- Step 7: Plot percentage error vs no of iterations bar graph
- Step 8: Display the root value computed earlier

Code

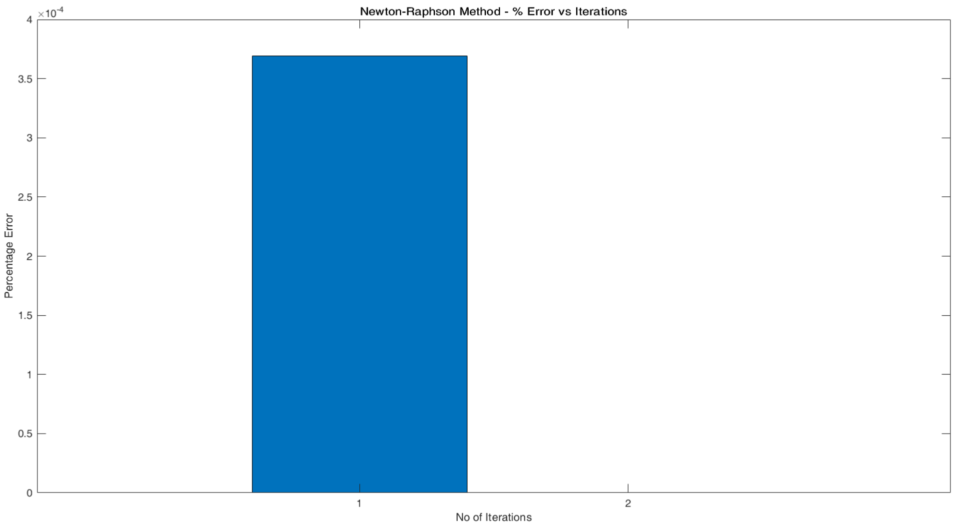
```
function outputnum = func(x)
outputnum = x.^2 - 5;
end

function root = newtonRaphson(x, e)

tmp = 0.0;
er = zeros(100);
count = 1;

while (abs((x - tmp) / x ) >= e)
    tmp = x;
    x = x - 2 * 0.001 * func(x) / (func(x + 0.001) - func(x - 0.001));
    er(count) = abs( (x - tmp) / x ) * 100;
    count = count + 1;
end

n = count;
r = 1:n;
bar(r(1:n-1),er(2:n));
xlabel('No of Iterations');
ylabel('Percentage Error');
title('Newton-Raphson Method - % Error vs Iterations');

root = vpa(x);
```

```
end
```

Output

Command Window

```
>> newtonRaphson(2.23, 0.0001)

ans =

2.2360679775150300585551121912431

>>
```



Secant Method

Aim

To implement the Secant method to numerically improve the accuracy of the roots of non-trivial equations using MATLAB.

Algorithm

- Step 1: Read the approximate root value to the given function to work with, say x
- Step 2: Read the tolerance limit, say e
- Step 3: Compute a better approximation using Secant method formula $x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}$
- Step 4: Calculate the percentage error value using $\left|\frac{x_i - x_{i-1}}{x_i}\right| \times 100$
- Step 5: If $|(x_{n+1} - x_n)/x_n|$ is greater than or equal to e proceed to step 6 else goto step 3
- Step 6: Plot percentage error vs no of iterations bar graph
- Step 7: Display the root value computed earlier
- Step 8: Compare percentage error vs no of iterations graph with Newton-Raphson's results

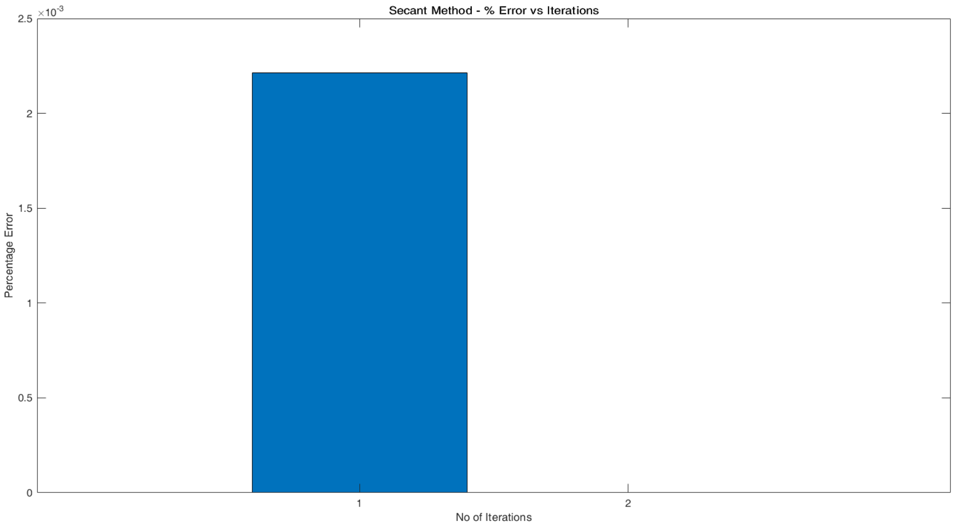
Code

```
function outputnum = func(x)
outputnum = x.^2 - 5;
end

function root = secant(x1, x2, e)

x = 0.0;
er = zeros(100);
count = 1;

while (abs((x1 - x2) / x2 ) >= e)
    x = x2;
    x2 = ( x1 * func(x2) - x2 * func(x1) ) / ( func(x2) - func(x1) );
    x1 = x;
    er(count) = abs( (x1 - x2) / x2 ) * 100;
    count = count + 1;
end

n = count;
r = 1:n;
bar(r(1:n-1),er(2:n));
xlabel('No of Iterations');
ylabel('Percentage Error');
title('Secant Method - % Error vs Iterations');

root = vpa(x2);

end
```

Output

Command Window

```
>> secant(2.2, 2.23, 0.001)

ans =

2.2360679103760947583623419632204

>>
```



Gauss Elimination Method

Aim

To implement Gauss Elimination method to solve a system of linear equations exactly in MATLAB.

Algorithm

- Step 1: Check for proper dimensions of Matrix A and Vector b
- Step 2: Compute the formula for iterations of
  $k = 1 \ldots n-1, i = k+1 \ldots n \text{ and } j = k+1 \ldots n$
  $a_{ij} = a_{ij} - \frac{a_{ik} \times a_{kj}}{a_{kk}}$
- Step 3: Check a_{ii} is not equal to zero. If zero, swap rows $i \text{ and } i-1$. This is called partial pivoting.
- Step 4: Now for the back substitution, use
  $x_n = b_n / a_{nn}$
- Step 5: Compute remaining solutions by iterating $k = n-1 \ldots 1$ the below relation
  $x_k = \frac{a_{kn+1} - \sum_{j=k+1}^{n} a_{kj} \times x_j}{a_{kk}}$
- Step 6: Return/Print the vector x to display the solutions for given system of linear equations represented by A and b.

Code

```
function [ x ] = gaussElimination( A, b)

sz = size(A);
if sz(1) ~= sz(2)
    fprintf('A is not n by n\n');
    clear x;
    return;
end

n = sz(1);

if n ~= sz(1)
    fprintf('b is not 1 by n.\n');
    return
end

x = zeros(n,1);
aug = [A b];
tempmatrix = aug;

for i= 2:sz(1)
    tempmatrix(1,:) = tempmatrix(1,:) / max(tempmatrix(1,:));
    temp = find(abs(tempmatrix) - max(abs(tempmatrix(:,1))));
    if length(temp) > 2
        for j = 1:length(temp)-1
            if j ~= temp(j)
                maxi = j;
```

```
                break;
            end
        end
    else
        maxi = 1;
    end
    if maxi ~= 1
        temp = tempmatrix(maxi,:);
        tempmatrix(maxi,:) = tempmatrix(1,:);
        tempmatrix(1,:) = temp;
    end
    for j = 2:length(tempmatrix)-1
        tempmatrix(j,:) = tempmatrix(j,:) - tempmatrix(j,1) ...
            / tempmatrix(1,1) * tempmatrix(1,:);
        if tempmatrix(j,j) == 0 || ...
                isnan(tempmatrix(j,j)) || abs(tempmatrix(j,j)) == Inf
            fprintf('Error: Matrix is singular.\n');
            clear x;
            return
        end
    end
    aug(i-1:end,i-1:end) = tempmatrix;
    tempmatrix = tempmatrix(2:end,2:end);
end

x(end) = aug(end,end) / aug(end,end-1);

for i = n-1:-1:1
    x(i) = (aug(i,end) - dot(aug(i,1:end-1),x)) / aug(i,i);
end

end
```

Output

Command Window

```
>> A = [-3,2,-6;5,7,-7;1,4,-2]

A =

    -3     2    -6
     5     7    -7
     1     4    -2

>> b = [6;6;8]

b =

     6
     6
     8

>> gaussElimination( A, b)

ans =

   -1.6491
    2.7895
    0.7544
```

Gauss Jordan Method

Aim

To implement Gauss Jordan method to solve a system of linear equations exactly in MATLAB.

Algorithm

- Start from $k = 0$ and $l = 0$.
- Increment k by one unit.
- Increment l by one unit.
- Stop the algorithm if $l > L$. Else proceed to the next step.
- If $A_{il} = 0$ for $i = k, \cdots, K$, return to step 3. Else proceed to the next step.
- Interchange the k-th equation with any equation i (with $i > k$) such that $A_{il} \neq 0$ (if $i = k$ there is no need to perform an interchange).
- Divide the k-th equation by A_{kl}.
- For $i = 1, \cdots, k-1$ and $i = k+1, \cdots, K$, subtract the k-th equation multiplied by A_{il} from the i-th equation.
- If $k < K$, return to step 2. Else stop the algorithm.

Code

```
function solution = gauss_jordan(A, b, n)
     M = [A, b];
    for i = 2:n
        for j = 1:i-1
            M(i,:) = M(i,:) - M(j,:)*(M(i,j)/M(j,j));
        end
    end

   for i = n-1:-1:1
        for j = n:-1:i+1
            M(i,:) = M(i,:) - M(j,:)*(M(i,j)/M(j,j));
        end
   end

   A = M(1:n,1:n);
   b = M(:,n+1);

   disp(A)
    for i = 1:n
        solution(i,1) = b(i)/A(i,i);
    end
end
```

Output

Command Window

```
>> n = 3;

rnum = randi(20,n,n);
rnum2 = randi(20,n,1);

disp(rnum);
disp(rnum2);

sol = gauss_jordan(rnum, rnum2, n);
disp(sol);
    17    19     6
    19    13    11
     3     2    20

    20
     4
    20

   17.0000         0         0
         0   -8.2353         0
         0         0   18.2357

   -2.3141
    2.7857
    1.0685
```

Trapezoidal Rule

Aim

To implement Trapezoidal Rule in MATLAB and compute integrals numerically.

Algorithm

- Step 1: Generate a vector x such that it contains domain values in desired integration range, say $[x_0, x_n]$ for given function $f(x)$
- Step 2: Generate a vector y for any desired function $f(x)$ such that $y = f(x)$
- Step 3: Determine step size h and number of points n using x
- Step 4:Compute the integral $\int_{x_0}^{x_n} ydx$ using the \textbf{Trapezoidal Rule} formula $\int_{x_0}^{x_n} ydx = \frac{h}{2}[y_0 + 2(y_1 + y_2 + \cdots + y_{n-1}) + y_n]$
- Step 5: Display the numerical value of the given definite integral
- Step 6: Compare the numerical accuracy of Simpson's 1/3 Rule with Trapezoidal Rule for different functions

Code

```
function numInt = trapezoidal(x, y)

h = x(2) - x(1);
[m, n] = size(y);
sum = 0.000;

for count = 1:n
    sum = sum + y(count);
end

sum = 2.0 * sum - (y(1) + y(n));
sum = h * sum / 2.0;

numInt = vpa(sum);
end
```

Output

Command Window

```
>> x = 1:0.01:10;
y = x.^4 + exp(-x);
>> trapezoidal(x, y)

ans =

20000.20113710354053182527422905

>>
```

Simpson's 1/3 Rule

Aim

To implement Simpson's 1/3 Rule in MATLAB and compute integrals numerically.

Algorithm

- Step 1: Generate a vector x such that it contains domain values in desired integration range, say $[x_0, x_n]$ for given function $f(x)$
- Step 2: Generate a vector y for any desired function $f(x)$ such that $y = f(x)$
- Step 3: Determine step size h and number of points n using x
- Step 4:Compute the integral $\int_{x_0}^{x_n} ydx$ using the **Simpson's 1/3 Rule** formula $\int_{x_0}^{x_n} ydx = \frac{h}{3}[y_0 + 4(y_1 + y_3 + \cdots + y_{n-1}) + 2(y_2 + y_4 + \cdots + y_{n-2}) + y_n]$
- Step 5: Display the numerical value of the given definite integral

Code

```
function numInt = simpson13(x, y)

h = x(2) - x(1);
[m, n] = size(y);
sum = 0.000;

for count = 1:n
    if ( count >= 2 ) & ( count <= n-1 ) & ( mod(count, 2) ==  0)
        sum = sum + 4 * y(count);
    end
    if ( count >= 3 ) & ( count <= n-2 ) & ( mod(count, 2) ==  1)
        sum = sum + 2 * y(count);
    end
end

sum = sum + y(1) + y(n);
sum = h * sum / 3.0;

numInt = vpa(sum);
end
```

Output

Command Window

```
>> x = 1:0.01:10;
>> y = x.^4 + exp(-x);
>> simpson13(x, y)

ans =

20000.16783405328897060826420784

>>
```

Romberg integration

Aim

To implement romberg integration recursively in MATLAB to compute definite integrals numerically with varying accuracy.

Algorithm

To compute
$\int_a^b f(x)dx$
using **Romberg's Integration** Method, define a **recursive function** using the below definitions

$h_n = \frac{b-a}{2^n}$

$R(0,0) = \frac{f_b - f_a}{h_1}$

$R(n,0) = \frac{R(n-1,0)}{2} + h_n \sum_{k=1}^{2^{n-1}} f\left(a + \frac{2k-1}{h_n}\right)$

$R(n,m) = R(n,m-1) + \frac{R(n,m-1) - R(n-1,m-1)}{4^m - 1}$

where $n \geq m$ and $m \geq 1$. Here,

- The zeroth extrapolation, R(n, 0), is equivalent to the trapezoidal rule with $2^n + 1$ points
- The first extrapolation, R(n, 1), is equivalent to Simpson's rule with $2^n + 1$ points
- The second extrapolation, R(n, 2), is equivalent to Boole's rule with $2^n + 1$ points

Code

```
function [ Result ] = romberg( fun, x1, x2, n, m )

func = inline(fun);
sum = 0.0;
h = @(t) ((x2 - x1) / (2^t));

if( (n == 0) & (m == 0) )
    Result = h(1) * (func(x1) + func(x2));
elseif ( m == 0)
    for k = 1:(2^(n-1))
        sum = sum + func(x1 + (2*k - 1)*h(n));
    end
    Result = ( 0.5 * romberg(fun, x1, x2, n-1, 0) + sum*h(n));
else
    Result = ((4^m)*romberg(fun, x1, x2, n, m-1) - romberg(fun, x1, x2, n-1, m-
1)) / (4^m - 1);
end

end
```

Output

Command Window

```
>> romberg('x.^2', 0, 5, 1, 0)

ans =

   46.8750

>> romberg('x.^2', 0, 5, 1, 1)

ans =

   41.6667

>> romberg('x.^2', 0, 5, 3, 1)

ans =

   41.6667

>>
```

Euler Method

Aim

To implement Euler Method in MATLAB to solve the differential equation $f'(x) = f(x, y)$ for it's particular solution numerically.

Algorithm

- Define function $f(x)$
- Get the values of x_0, y_0, h and x_n. Here x_0 and y_0 are the initial conditions, h is the interval, x_n is the required value
- Define $n = \frac{(x_n - x_0)}{h} + 1$
- Start loop from $i = 1$ to n
- $y = y_0 + h \times f(x_0, y_0)$
- $x = x + h$
- Print values of y_0 and x_0
- Check if $x \leq x_n$
- If yes, assign $x_0 = x$ and $y_0 = y$ and continue loop
- If no, End loop i

Code

```
function euler(fun, x0, y0, h, fname)

func = inline(fun);

x = zeros(1001);
y = x;

x(1) = x0;
y(1) = y0;

fid = fopen(fname,'w');

i = 0;
n = 1/h;

for i = 2:n
    x(i) = x(i-1) + h;
    y(i) = y(i-1) + h * func(x(i-1),  y(i-1));

    fprintf(fid,'%8.4f %8.4f \n', x(i), y(i));
end

plot(x,y);

end
```

Output

Command Window

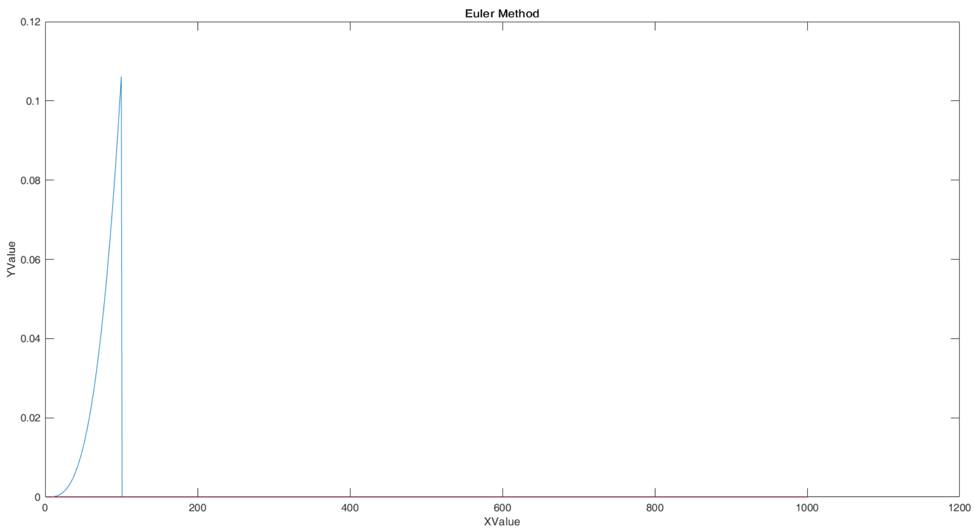
```
>> euler( '(x^2)/3 + 0*y', 0, 0, 0.01, 'eulerxyza')
>>
```

eulerxyza - Notepad

File Edit Format View Help

```
0.7500   0.0459
0.7600   0.0478
0.7700   0.0497
0.7800   0.0517
0.7900   0.0537
0.8000   0.0558
0.8100   0.0580
0.8200   0.0601
0.8300   0.0624
0.8400   0.0647
0.8500   0.0670
0.8600   0.0694
0.8700   0.0719
0.8800   0.0744
```

Unix (LF) Ln 1, Col 1 100%



Runge Kutta Method

Aim

To implement 2nd and 4th order Runge Kutta Methods to solve a Differential Equation for it's particular solution numerically using MATLAB for the derivative function $f(x, y)$ in the domain $[a, b]$

Algorithm

- Step 1: Define the function $f(x, y)$
- Step 2: Determine step size h for desired accuracy
- Step 3: Compute y_1 from (x_0, y_0) and $x_1 = x_0 + h$ using the following formulas

For 2nd Order:
$y_{i+1} = y_i + \frac{k_1+k_2}{2}h$

where

$k_1 = f(x_i, y_i) \text{ and } k_2 = f(x_i + h, k_1 h)$
and for 4th Order:

$y_{i+1} = y_i + \frac{k_1+2k_2+2k_3+k_4}{6}$

where
$k1 = hf(x_i, y_i),\ k_2 = hf(x_i + \frac{h}{2}, y_i + \frac{k_1}{2})$

$k_3 = hf(x_i + \frac{h}{2}, y_i + \frac{k_2}{2}),\ k_4 = hf(x_i + \frac{h}{2}, y_i + \frac{k_3}{2})$

- Step 5: Check if x_n is equal to b, if yes then, plot the result else continue.
- Step 6: Save the vector x and y to a file for further reference.

Code

2nd Order Implementation

```
function rkutta2(x0, y0, h, fname)

func = @(x,y) 4 * x ^ 3 + 0 * y;

fid = fopen(fname,'w');

i = 0;
n = 1/h;

x = zeros(n);
y = x;

x(1) = x0;
y(1) = y0;
```

```
k2 = @(x,y) func((x + h), (y + (func(x, y) * h)));

for i = 2:n
    x(i) = x(i-1) + h;
    y(i) = y(i-1) + 0.5 * (func(x(i-1), y(i-1)) ...
        + k2(x(i-1), y(i-1))) * h;

    fprintf(fid,'%8.4f %8.4f \n', x(i), y(i));
end

plot(x,y);

end
```

4th Order Implementation

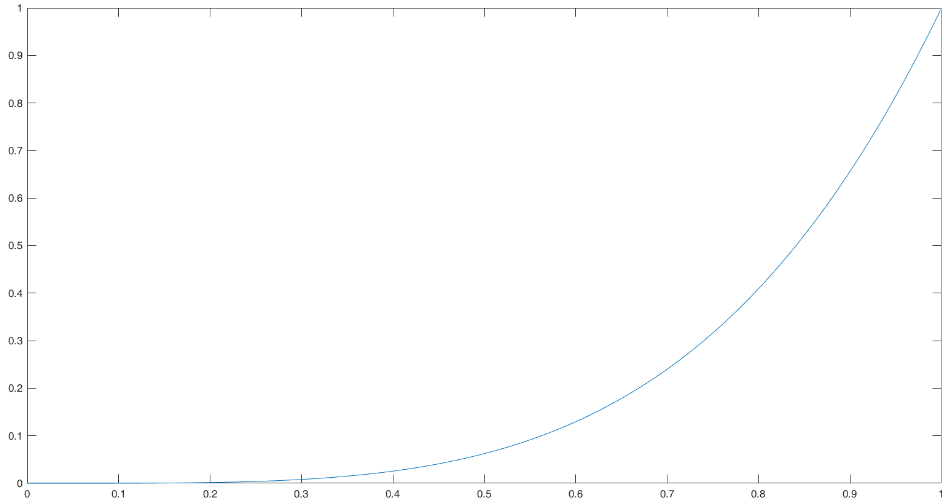
```
function rkutta4(x0, y0, h, fname)

func = @(a, b) 4 * a ^ 3 + 0 * b;

x = zeros(10001);
y = x;

x(1) = x0;
y(1) = y0;

fid = fopen(fname,'w');

i = 0;
n = 1/h;

k1 = @(a,b) func(a, b);
k2 = @(a,b) func((a + h / 2.0), (b + (k1(a,b) / 2.0)));
k3 = @(a,b) func((a + h / 2.0), (b + (k2(a,b) / 2.0)));
k4 = @(a,b) func((a + h), (b + k3(a,b)));

for i = 2:n
    x(i) = x(i-1) + h;
    y(i) = y(i-1) + h * (k1(x(i-1),y(i-1)) + ...
        2 * k2(x(i-1),y(i-1)) + ...
        2 * k3(x(i-1),y(i-1)) + k4(x(i-1),y(i-1))) / 6.0;

    fprintf(fid,'%8.4f %8.4f \n', x(i), y(i));
end

plot(x,y);

end
```

Output

Runge Kutta 2nd Order:

Command Window

```
>> rkutta2(0, 0, 0.001, 'rkutta2.txt')
>> rkutta2(0, 0, 0.001, 'rkutta2.txt')
fx >>
```



rkutta2.txt - Notepad

File Edit Format View Help

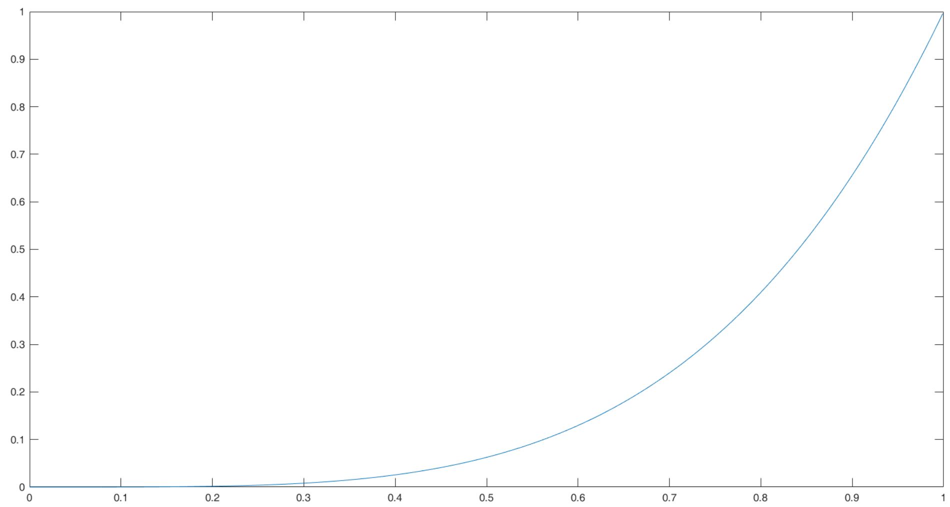
```
0.5170   0.0714
0.5180   0.0720
0.5190   0.0726
0.5200   0.0731
0.5210   0.0737
0.5220   0.0742
0.5230   0.0748
0.5240   0.0754
0.5250   0.0760
0.5260   0.0765
0.5270   0.0771
0.5280   0.0777
0.5290   0.0783
0.5300   0.0789
```

Unix (LF) Ln 1, Col 1 100%

Runge Kutta 4th Order:

Command Window

```
>> rkutta4(0, 0, 0.001, 'rkutta4.txt')
fx >>
```



rkutta4.txt - Notepad

File Edit Format View Help

```
  0.0010   0.0000
  0.0020   0.0000
  0.0030   0.0000
  0.0040   0.0000
  0.0050   0.0000
  0.0060   0.0000
  0.0070   0.0000
  0.0080   0.0000
  0.0090   0.0000
  0.0100   0.0000
  0.0110   0.0000
  0.0120   0.0000
  0.0130   0.0000
  0.0140   0.0000
```

Unix (LF) Ln 1, Col 1 100%

Least Square Method

Aim

To implement Least Square Method in MATLAB to find the least error fit for the given data $(x_1, y_1) \to (x_n, y_n)$.

Algorithm

- Step 1: Compute $\sum_{i=1}^{n} x_i$, $\sum_{i=1}^{n} x_i^2$, $\sum_{i=1}^{n} x_i y_i$ and $\sum_{i=1}^{n} y_i$
- Step 2: Define a matrix $\mathcal{A}$

$$\mathcal{A} = \begin{pmatrix} n & \sum_{i=1}^{n} x_i \\ \sum_{i=1}^{n} x_i & \sum_{i=1}^{n} x_i^2 \end{pmatrix}$$

and a vector v

$$v = \begin{pmatrix} \sum_{i=1}^{n} y_i \\ \sum_{i=1}^{n} x_i y_i \end{pmatrix}$$

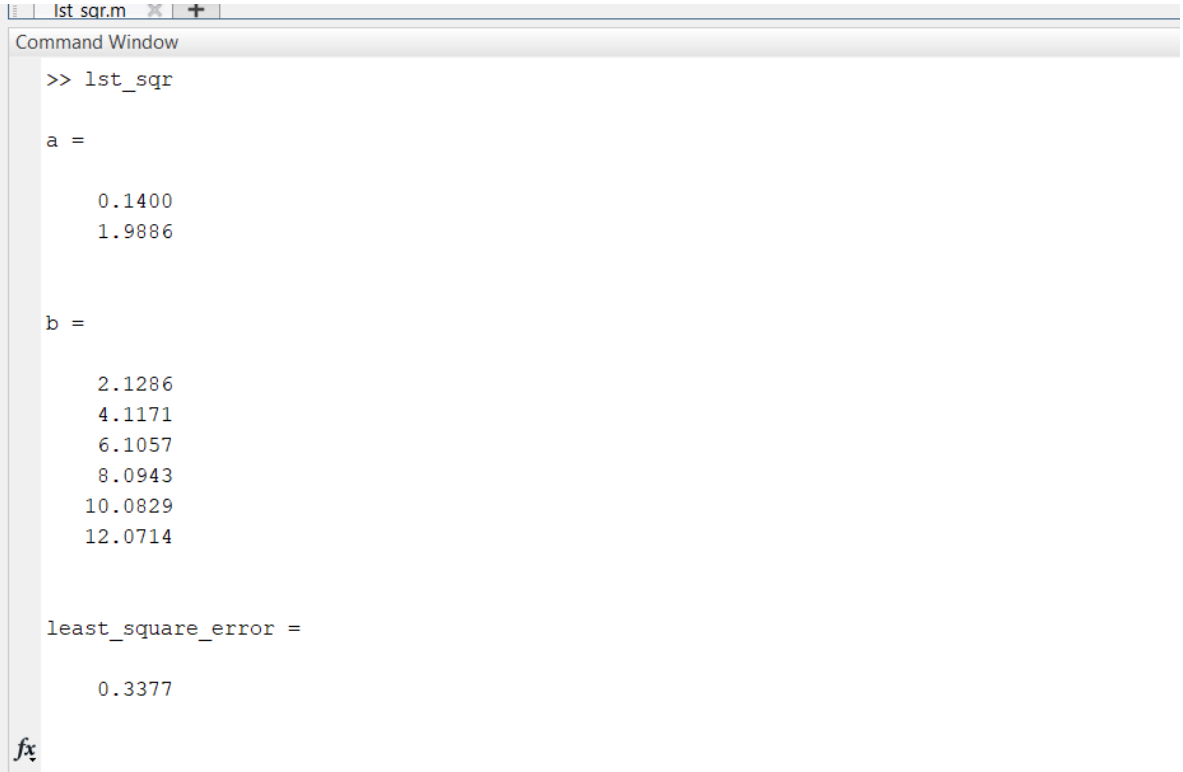
- Use Gauss Elimination to solve $\mathcal{A}\Phi = v$
- The components of Φ gives us the slope and intercept of the best fit line.

Code

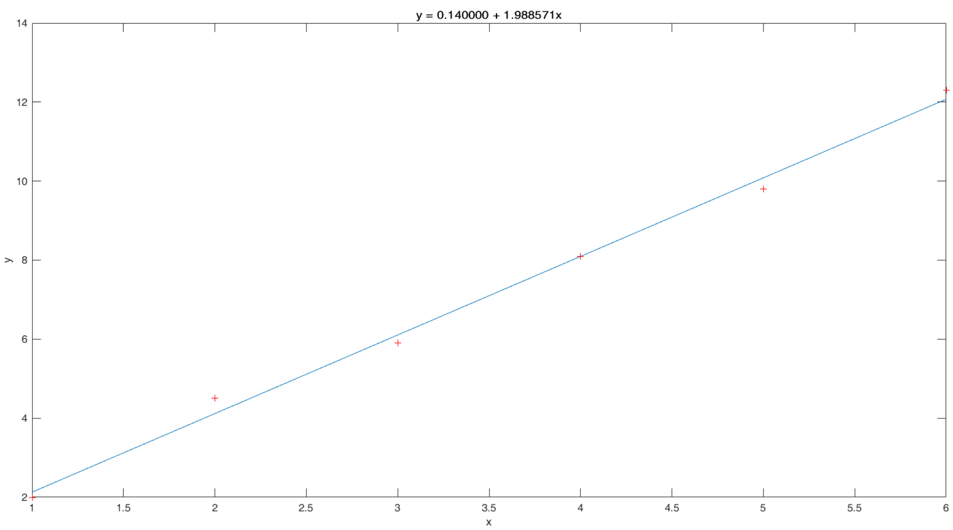
```
% input in the form of matrix, each row is a (x, y).
input = [...
  1, 2;...
  2, 4.5;...
  3, 5.9;...
  4, 8.1;...
  5, 9.8;...
  6, 12.3];

m = size(input, 1);
n = size(input, 2);
x = input(:,1:n-1);
y = input(:,n);

% The first column of matrix X is populated with ones,
% and the rest columns are the x columns of the input.
X = ones(m, n);
X(:,2:n) = input(:,1:n-1);

% Try to find the a that minimizes the least square error Xa - y.
% Project y onto the C(X) will give us b which is Xa.

% The relationship is X'Xa = X'b

% Use left division \ to solve the equation, which is equivalent
% to a = inverse(X'*X)*X'*y, but computationally cheaper.
a = (X' * X) \ (X' * y)
b = X*a
```

```
least_square_error = sum((b - y) .^ 2)

% Plot the best fit line.
plot(x, b);
title(sprintf('y = %f + %fx', a(1), a(2)));
xlabel('x');
ylabel('y');

hold on;
% Plot the input data.
plot(x, y, '+r');
hold off;
pause;
```

Output



```
lst_sqr.m
Command Window
>> lst_sqr

a =

    0.1400
    1.9886


b =

    2.1286
    4.1171
    6.1057
    8.0943
   10.0829
   12.0714


least_square_error =

    0.3377

fx
```



Hermite Cubic Interpolation

Aim

To implement Cubic Hermite Interpolator spline to fit a cubic polynomial to given data points numerically using MATLAB.

Algorithm

- Form a linear system of equations for input values of y and a dy sampled at points x (for y) and dx (for dy)
- Form a Matrix A for all value for x and dx. Form a (column) vector b for corresponding values of y and dy
- Use Gauss Elimination or \ operator to solve the system

Code

```
function [a, b, c, d] = hermite_cubic_interpolate(x,y,dx,dy)

% hermite_cubic_interpolate: Function to compute a Cubic Hermit Polynomial
%
% Assume a cubic polynomial of the form
%
%  y = a*x*x*x + b*x*x + c*x + d
%
%  with derivative:
%
%  dy = 3A*x*X + 2*b*x + c
%
% Solution is quite simple: Form a linear system of equations for input
% values of y and a dy sampled at points x (for y) and dx (for dy)
%
% Form a MATRIX A for all value for x and dx. Form a (column) vector b for
% corresaponding values of y and dy
%
% Use MATLAB \ operator to solve the system
%
% Inputs: x --- values of x where y samples are given
%         y --- values of y at given x values
%         dx --- values of dx where dy samples are given
%         dy --- values of dy at given dx values
%
% All inputs assumed row vectors.
%
% Outputs: a,b,c,d --- parameters of Hermite Cubic


% Get number of x data points
[m n] = size(x);

% Form matrix A for X values
```

```
A = [x.*x.*x; x.*x; x; ones(1,n)]';


% Get number of x data points
[m n] = size(dx);

% Form matrix dA for X values

dA = [3*dx.*dx; 2*dx; ones(1,n); zeros(1,n)]';

% Concatentate A and da  for final A matrix
A = [A ; dA];

% Concatentate y and dy  for b COLUMN vector ---
%   so columnise y and dy (use ')
b = [y' ; dy'];

% solve linear system of equations

p = A\b;

% output a,b,c,d

a = p(1);
b = p(2);
c = p(3);
d = p(4);
```

Output

Command Window

```
>> x = 0:0.01:1;
>> y = x.^3 - 4*x;
>> dx = 0.01 * ones(101,1);
>> dx = dx';
>> dy = 3*x.^2 -4;
>> hermite_cubic_interpolate(x,y,dx,dy)

ans =

    2.4630

>> hermite_cubic_interpolate(x,y,dx,dy)

a =

    2.4630


b =

   -2.3600


c =

   -2.9495


d =

   -0.1040

fx
```