Numerical Methods with Scilab
Delhi University

1 To solve Transcendental and Polynomial equations

1.1 Solving a Polynomial Equation using the Bisection Method

```
// Solving polynomial equations
// Bisection Method

// Consider the following polynomial equation:
// x^2 + x = 1

function y = f(x)
    y = x^2 + x - 1
endfunction

// We first define an interval over which we know a root exists.
a = 0
b = 5

// We verify that the above interval end-points have different signs:
f(a)
f(b)

for i = 1:20 // 20 iterations
    printf("Iteration number %d\n", i)
    c = (a+b)/2
    if sign(f(c)) == 0 then
        the_root = 0,
        break
    elseif sign(f(c)) == sign(f(a)) then
        a = c
    else
        b = c
    end
end

the_root = c
```

1.2 Solving a Transendental Equation using the Bisection Method

```
// Solving transcendental equations
// Bisection Method

// Consider the following transcendental equation:
```

```
// sin(x) = 0

function y = g(x)
    y = sin(x)
endfunction

// We first define an interval over which we know a root exists.
a = -%pi/4
b =  %pi/2

// We verify that the above interval end-points have different signs:
g(a)
g(b)

for i = 1:20 // 20 iterations
    printf("Iteration number %d\n", i)
    c = (a+b)/2
    if sign(g(c)) == 0 then
        the_root = 0,
        break
    elseif sign(g(c)) == sign(g(a)) then
        a = c
    else
        b = c
    end
end

the_root = c
```

1.3 Solving a Polynomial Equation using the Secant Method

```
// Solving polynomial equations
// Secant method:

// Consider the following polynomial equation:
// x^2 + x = 1

function y = f(x)
    y = x^2 + x - 1
endfunction

// We define our first two guesses
x(1) = 0
x(2) = 5

for i = 3:9 // 7 iterations
    x(i) = x(i-1) - f(x(i-1))*(x(i-1) - x(i-2))/(f(x(i-1)) - f(x(i-2)));
    disp(x(i))
end

the_root = x(i)
```

1.4 Solving a Transcendental Equation using the Secant Method

```
// Solving transcendental equations
// Secant method:

// Consider the following transcendental equation:
// sin(x) = 0

function y = g(x)
    y = sin(x)
endfunction

// We define our first two guesses
```

```
x(1) = -%pi/4
x(2) =  %pi/2

for i = 3:9 // 7 iterations
    x(i) = x(i-1) - g(x(i-1))*(x(i-1) - x(i-2))/(g(x(i-1)) - g(x(i-2)));
    disp(x(i))
end

the_root = x(i)
```

1.5 Solving a Polynomial Equation using the Regula Falsi Method

```
// Solving polynomial equations
// Regula Falsi Method

// Consider the following polynomial equation:
// x^2 + x = 1

function y = f(x)
    y = x^2 + x - 1
endfunction

// We first define an interval over which we know a root exists.
a = 0
b = 5

for i = 1:25 // 10 iterations
    printf("Iteration number %d\n", i)
    c = (f(b)*a - f(a)*b)/(f(b) - f(a))
    if sign(f(c)) == 0 then
        the_root = 0,
        break
    elseif sign(f(c)) == sign(f(a)) then
        a = c
    else
        b = c
    end
end

the_root = c
```

1.6 Solving a Transcendental Equation using the Regula Falsi Method

```
// Solving transcendental equations
// Regula Falsi Method

// Consider the following transcendental equation:
// sin(x) = 0

function y = g(x)
    y = sin(x)
endfunction

// We first define an interval over which we know a root exists.
a = -%pi/4
b =  %pi/2

for i = 1:20 // 20 iterations
    printf("Iteration number %d\n", i)
    c = (g(b)*a - g(a)*b)/(g(b) - g(a))
    if sign(g(c)) == 0 then
        the_root = 0,
        break
```

```
    elseif sign(g(c)) == sign(g(a)) then
        a = c
    else
        b = c
    end
end

the_root = c
```

1.7 Solving a Polynomial Equation using the Newton Raphson Method

```
// Solving polynomial equations
// Newton Raphson method:

// Consider the following polynomial equation:
// x^2 + x = 1

function y = f(x)
    y = x^2 + x - 1
endfunction

// Our initial guess:

x(1) = 0

// The derivative:
function y = df(x)
    y = 2*x + 1
endfunction

for i = 2:5 // 4 iterations
    x(i) = x(i-1) - f(x(i-1))/df(x(i-1));
    disp(x(i))
end

the_root = x(i)
```

1.8 Solving a Transcendental Equation using the Newton Raphson Method

```
// Solving transcendental equations
// Newton Raphson method:

// Consider the following transcendental equation:
// sin(x) = 0

function y = g(x)
    y = sin(x)
endfunction

// Our initial guess:

x(1) = -%pi/4

// The derivative:
function y = dg(x)
    y = cos(x)
endfunction

for i = 2:5 // 4 iterations
    x(i) = x(i-1) - g(x(i-1))/dg(x(i-1));
    disp(x(i))
end
```

```
the_root = x(i)
```

2 To find the Complex Roots of equations

```
// Solving a quadratic equation:

function [x1,x2] = qroots(a,b,c);
    i=%i;
    D=(b^2)-(4*a*c);

    if (D>0)

        x1=(-b+D^(1/2))/(2*a);
        x2=(-b-D^(1/2))/(2*a);
    elseif (D==0)
        x1=(-b/(2*a));
        x2=(-b/(2*a));

    else
        xr=(-b)/(2*a);
        xi=((-D)^(1/2))/(2*a);
        x1=xr+(i*xi);
        x2=xr-(i*xi);
    end
endfunction

// Consider the equation:
//     x^2 + 2x + 5
// The roots are
//     -1 + 2i and -1 - 2i

// We find them using the above function:
[x1, x2] = qroots(1, 2, 5)

// We compare them using Scilab's functions:
roots(poly([5 2 1], 'x', 'c'))
```

3 Interpolation and Polynomial Approximations

3.1 Linear Interpolation using the Nearest Neighbour Method

```
// Linear interpolation (nearest neighbour):

// Our data-set:
x = [0:1:5]'; y = exp(x); // exponent
plot(x, y, 'ro')

xp = [0:0.01:5]';
yp = []

for i = xp'
    x_nearest = floor(i+0.5);
    yp_i = y(find(x == x_nearest));
    yp = [yp; yp_i];
end

plot(xp, yp, 'b')

// We compare with Scilab's internal interpolation function:
// Linear interpolation (nearest neighbour) using Scilab's function:

yp = interp1(x, y, xp, 'nearest');
plot(xp, yp, 'g')
```

3.2 Linear Interpolation

```
// Linear interpolation:

// Our data-set:
x = [0:1:5]'; y = exp(x); // exponent
plot(x, y, 'ro')

xp = [0:0.01:5]';
yp = []

for i = xp'
    x0 = floor(i);
    x1 = ceil(i);
    if x0 == x1 then
        yp_i = y(find(x == i));
    else
        y0 = y(find(x == x0));
        y1 = y(find(x == x1));
        yp_i = y0 + (i - x0)*(y1 - y0)/(x1 - x0);
    end
    yp = [yp; yp_i];
end

plot(xp, yp, 'b')

// We compare with Scilab's internal interpolation function:
// Linear interpolation using Scilab's Interpolation function:

yp = interp1(x, y, xp);
plot(xp, yp, 'g')
```

3.3 Polynomial Interpolation

```
// Polynomial interpolation

// Our data-set:
x = [0:1:5]'; y = exp(x); // exponent
plot(x, y, 'ro')

xp = [0:0.01:5]';

// We first form the Vandermonde matrix:

V = []
n=length(x);

for i=0:n-1
    V = [V, x.^i];
end

// The coefficients of the polynomial equation are given by:
c = V\y

// The interpolating polynomial is given by:
p  = poly(c, 'a', 'coeff')

// We interpolate the points using the above polynomial
yp = horner(p, xp);

plot(xp, yp, 'b')
```

4 Curve Fitting

4.1 Linear Curve Fitting

```
//This code is written by Rupak Rokade, IIT Bombay
//The code fits a noisy data with a linear curve

x = [0:1:5]'; y = exp(x);

function [slope,intercept,lserror]=lincurvefit(data)
      n=length(data);
      x=0:n-1;
      [slope,intercept,lserror]=reglin(x,data);
    y_fit=slope*x+intercept;//fitted line equation
    plot2d(x,y_fit,2)
    plot2d(x,data,1)
    xlabel('x');
    ylabel('y');
    legend('fitted_line','actual_data');
endfunction

[s, i, lse] = lincurvefit(y')
```

4.2 Second Order Curve Fitting

```
//This code is written by Rupak Rokade, IIT Bombay
//The code fits a noisy data with a second order curve fit

x = [0:1:5]'; y = exp(x);

function [slope,intercept,sig]=second_order_fit(data)
      n=length(data);
      x=0:n-1;
      X=[x;x.^2]; //the two regressors
      [slopes,intercept,sig]=reglin(X,data);
      yfitted=intercept+slopes(1)*x+slopes(2)*x.^2;
    plot2d(x,yfitted,2) // the fitted solution
    plot2d(x,data,1)
    xlabel('x');
    ylabel('y');
    legend('fitted_line','actual_data');
endfunction

[s, i, lse] = second_order_fit(y')
```

5 Numerical Integration

5.1 Simple Trapezoidal Rule

```
mode(-1);
// Numerical Integration
//    Written by
//    Rakhi.R
//    IIT Bombay

function f = func(x)
   f = sqrt(x); // define a function in x as f(x) = squareroot of x
  //f = 1/(1+x^2); //define a function in x as f(x) = 1/(1+x^2)
endfunction


mode(-1);
// Numerical Integration
//    Written by
```

```
//    Rakhi.R
//    IIT Bombay

// Simple Trapezoidal Rule - S T rule, ie n = 1
// This computes I = integral from a to b f(x) dx using S T rule
// fa , fb are the values of the function f(x) at a and b respectively
//I is given by I = (fa + fb)*h/2

function I = trap(a,b,fa,fb)
    h = (b-a); // length of the interval
    I = (fa + fb)*h/2; //value of the integral
endfunction


mode(-1);
// Numerical Integration
//    Written by
//    Rakhi.R
//    IIT Bombay

// Example of simple trapezoidal rule  for f(x) = sqroot(x)

exec('func.sci'); //execute the function that defines f(x)
exec('trap.sci'); // execute the function for simple trapezoidal

a = input("Enter a = "); // for inputting a and b
b = input("Enter b = ");

fa = func(a);// Calculate fa and fb
fb = func(b);
I = trap(a,b,fa,fb);
//display result
disp("The value of the integral is  =");
disp(I);
```

5.2 Composite Trapezoidal Rule

```
mode(-1);
// Numerical Integration
//    Written by
//    Rakhi.R
//    IIT Bombay

function f = func(x)
   f = sqrt(x); // define a function in x as f(x) = squareroot of x
   //f = 1/(1+x^2); //define a function in x as f(x) = 1/(1+x^2)
endfunction


mode(-1);
// Numerical Integration
//    Written by
//    Rakhi.R
//    IIT Bombay

// Simple Trapezoidal Rule - S T rule, ie n = 1
// This computes I = integral from a to b f(x) dx using S T rule
// fa , fb are the values of the function f(x) at a and b respectively
//I is given by I = (fa + fb)*h/2

function I = trap(a,b,fa,fb)
    h = (b-a); // length of the interval
    I = (fa + fb)*h/2; //value of the integral
endfunction


mode(-1);
// Numerical Integration
//    Written by
```

```
//    Rakhi.R
//    IIT Bombay


// Composite Trapezoidal Rule - C T rule
// This computes I = integral from a to b f(x) dx using C T rule
// fa , fb are the values of the function f(x) at a and b respectively
// The interval (a,b) is divided into n panels, each of width h
// so h = (b-a)/n
// I is given by I = (fa + 2fx2 + 2fx3 + .....+2fxn + fb)*h/2

exec('func.sci'); // execute the function which defines f(x)
temp_sum = 0;

function I = trapc(a,b,n,fa,fb)
    h = (b-a)/n; // interval
    x(1) = a ;
    for i = 1:n-1
        x(i+1) = x(i) + h;  // finds abscissas each separated by h units
        temp_sum = temp_sum + 2*func(x(i+1)); // finds 2fx1 + .....+2fxn
            -1
        // temp_sum is also a vector whose last element gives
        // 2fx1 + 2fx2 + .....+2fxn-1
    end
    I = (fa + temp_sum($) + fb)*h/2; // finds I as per the formula

endfunction


mode(-1);
// Numerical Integration
//    Written by
//    Rakhi.R
//    IIT Bombay

// Example of composite trapezoidal rule  for f(x) = sqroot(x)

exec('func.sci'); //execute the function that defines f(x)
exec('trapc.sci'); // execute the function for simple trapezoidal
exec('trap.sci');

a = input("Enter a = "); // for inputting x0 and xn
b = input("Enter b = ");
n = input("Enter n = ");
fa = func(a);// Calculate fx0 and fxn
fb = func(b);
if n > 1
    I = trapc(a,b,n,fa,fb);
else
    I = trap(a,b,fa,fb);
end

//display result
disp("The value of the integral is  =");
disp(I);
```

5.3 Simpson's One Third Rule

```
mode(-1);
// Numerical Integration
//    Written by
//    Rakhi.R
//    IIT Bombay

function f = func(x)
    f = sqrt(x); // define a function in x as f(x) = squareroot of x
  //f = 1/(1+x^2); //define a function in x as f(x) = 1/(1+x^2)
endfunction
```

```
mode(-1);
// Numerical Integration
//    Written by
//    Rakhi.R
//    IIT Bombay

//Composite Simpson's 1/3rd rule
// This computes I = integral from a to b f(x) dx using this rule
// fa , fb are the values of the function f(x) at a and b respectively
// The interval (a,b) is divided into n panels, each of width h
//n has to be even!
// so h = (b-a)/n
// I is given by
// I = (fa + 2(fx2 + fx4 +....+ fxn-2) + 4(fx1 + fx3 +.....+fxn-1) + fb)
    *h/3
//If we rename a as x(1), b as x(n+1),(since arrays in sclab start with
    index of 1)
//I can be written as
//I = (fx1 + 4(fx2 + fx4 +....+ fxn-1) + 2(fx3 +.....+fxn) + fxn+1)*h/3


exec('func.sci'); // execute the function which defines f(x)

temp_sum = 0;
function I = simp3(a,b,n,fa,fb)
    h = (b-a)/n; // interval
    x(1) = a ;
    for i = 1:n-1
        x(i+1) = x(i) + h;  // finds abscissas each separated by h units
        if ~modulo(i+1,2)// if i+1 is divisible by 2
            temp_sum = temp_sum + 4*func(x(i+1)); // for odd terms
        else
            temp_sum = temp_sum + 2*func(x(i+1)); // for even terms
        // temp_sum is also a vector whose last element gives
        // 4(fx2 + fx4 +....+ fxn-1) + 2(fx3 +.....+fxn)
        end
    end
    I = (fa + temp_sum($) + fb)*h/3; // finds I as per the formula

endfunction

mode(-1);
// Numerical Integration
//    Written by
//    Rakhi.R
//    IIT Bombay

// Simple Simpson's 1/3rd Rule - S T rule, ie n = 2
// This computes I = integral from a to b f(x) dx using S T rule
// fa , fb are the values of the function f(x) at a and b respectively
//I is given by I = (fa + 4*f((a+b)/2) + fb)*h/3
exec('func.sci'); //execute the function that defines f(x)

function I = sim_simp3(a,b,fa,fb)
    h = (b-a)/2; // length of the interval
    I = (fa + 4*func((a+b)/2) + fb)*h/3; //value of the integral
endfunction

mode(-1);
// Numerical Integration
//    Written by
//    Rakhi.R
//    IIT Bombay

//Example of composite simpson's 1/3rd rule  for f(x) = sqroot(x)
//n is the number of panels into which the interval (a,b) is to be
    divided
//n has to be even!
```

```
exec('func.sci'); //execute the function that defines f(x)
exec('simp3.sci'); // execute the function for simpson's 1/3rd rule
exec('sim_simp3.sci');

a = input("Enter a = "); // for inputting a and b
b = input("Enter b = ");
n = input("Enter n = ");
if modulo(n,2)//remainder on dividing by 2 != 0
    disp("n has to be even!!");
    abort;
end
fa = func(a);// Calculate fa and fb
fb = func(b);
if n > 2
    I = simp3(a,b,n,fa,fb);
end
if n == 2 // if n = 2 use simple simpson's 1/3rd rule
   I = sim_simp3(a,b,fa,fb);
end

//display result
disp("The value of the integral is  =");
disp(I);
```

5.4 Simpson's Three Eights Rule

```
mode(-1);
// Numerical Integration
//    Written by
//    Rakhi.R
//    IIT Bombay

function f = func(x)
   //f = sqrt(x); // define a function in x as f(x) = squareroot of x
  f = 1/(1+x^2); //define a function in x as f(x) = 1/(1+x^2)
endfunction
```

```
mode(-1);
// Numerical Integration
//    Written by
//    Rakhi.R
//    IIT Bombay

//Composite Simpson's 3/8th rule
// This computes I = integral from a to b f(x) dx using this rule
// fa , fb are the values of the function f(x) at a and b respectively
// The interval (a,b) is divided into n panels, each of width h
// so h = (b-a)/n
//n has to be a multiple of 3!
// I is given by
// I = (fa  + 3(fx1 + fx2 + fx4.....+ fxn-1) + 2(fx3 + fx6 +....+ fxn-3)
     + fb)*3h/8
//If we rename a as x(1), b as x(n+1), (since arrays in scilab start
    with index of 1)
//I can be written as
//I = (fx1 + 3(fx2 + fx3 +....+ fxn) + 2(fx4 + fx7 +.....+ fxn-2) + fxn
    +1)*3h/8


exec('func.sci'); // execute the function which defines f(x)

temp_sum = 0;
mult3 = 4; // for separating out values at intervals of 3
function I = simp8(a,b,n,fa,fb)
    h = (b-a)/n; // interval
    x(1) = a ;
```

```
    for i = 1:n-1
       x(i+1) = x(i) + h;  // finds abscissas, each separated by h units
       if i+1 == mult3
          temp_sum = temp_sum + 2*func(x(i+1));
          mult3 = mult3 + 3;
       else
          temp_sum = temp_sum + 3*func(x(i+1));
       end
        // last element of temp_sum gives the value of
        // 3(fx1 + fx2 + fx4.....+ fxn-1) + 2(fx3 + fx6 +....+ fxn-3)
    end
    I = (fa + temp_sum($) + fb)*3*h/8; // finds I as per the formula

endfunction


mode(-1);
// Numerical Integration
//    Written by
//    Rakhi.R
//    IIT Bombay

//Example of composite simpson's 3/8th rule  for f(x) = sqroot(x)
//n is the number of panels into which the interval (a,b) is to be
    divided
//n has to be a multiple of 3!

exec('func.sci'); //execute the function that defines f(x)
exec('simp8.sci'); // execute the function for simpson's 1/3rd rule

a = input("Enter a = "); // for inputting a and b
b = input("Enter b = ");
n = input("Enter n = ");
if modulo(n,3)//remainder on dividing by 3 != 0
    disp("n has to be a multiple of 3!!");
    abort;
end
fa = func(a);// Calculate fa and fb
fb = func(b);
I = simp8(a,b,n,fa,fb);
//display result
disp("The value of the integral is  =");
disp(I);
```

6 Numerical Differentiation

6.1 Forward Difference Method

```
mode(-1);
// Numerical Differentiation
//    Written by
//    Rakhi.R
//    IIT Bombay

//Defining the delta function
//delta(f(x)) = f(x+h) - f(x)
// here y is the set of values of f(x)
//corresponding to each x
i = 1;

function delta_n = delta(y,N)
  for i = 1:N-1
    delta_n(i) = y(i+1) - y(i);
  end
endfunction
```

```
//f(x+3h)-2f(x+2h)+2f(x+h)- f(x)
//f(x+4h)-f(x+3h)


mode(-1);
// Numerical differentiation
//   Written by
//   Rakhi.R
//   IIT Bombay

// function to find index of a value in a given array
// x is the array, a is the value whose index is to
// be found out. n is the index found out

function n = index(x,a)
  for i = 1:length(x)
    if x(i) == a //compares all values in x
      n = i;// and gets the index
    end
  end
endfunction


mode(-1);
// Numerical differentiation
//   Written by
//   Rakhi.R
//   IIT Bombay

// function to find higher order delta values if they exist
global delta_2 delta_3 delta_4 delta_5

function [delta_2,delta_3,delta_4,delta_5] = get_deltas(delta_1,N)
if N >= 3
    delta_2 = delta(delta_1,N-1);
    delta_2 = [delta_2 ;(zeros(1,N-length(delta_2)))'];// pad with zeros
  if N >= 4
    delta_3 = delta(delta_2,N-2);
    delta_3 = [delta_3 ;(zeros(1,N-length(delta_3)))'];
    if N >= 5
      delta_4 = delta(delta_3,N-3);
      delta_4 = [delta_4 ;(zeros(1,N-length(delta_4)))'];
      if N >= 6
        delta_5 = delta(delta_4,N-4);
        delta_5 = [delta_5; (zeros(1,N-length(delta_5)))'];
      else
        delta_5 = (zeros(1,N))';//if these do not exist
      end
    else
      delta_4 = (zeros(1,N))';
    end
  else
    delta_3 = (zeros(1,N))';
  end
else
  delta_2 = (zeros(1,N))';
end
endfunction


mode(-1);
// Numerical differentiation
//   Written by
//   Rakhi.R
//   IIT Bombay
```

```
//Example of first order derivative using
//forward difference formula.
//This uses a set of values of x and the corresponding
//values of y = f(x). x values are equally spaced.
//a is also an element of the x array
//dy/dx at x = a is given by
//[delta(f(a)) - (1/2)*deltasquared(f(a)) +
//(1/3)*deltacubed(f(a)) - (1/4)*delta^4(f(a)) +
// (1/5)*delta^5(f(a)) - ....]*(1/h)
// the formula is approximated here till the 5th order delta
// h is the gap between the x values
exec('delta.sci');//function to calculate delta
exec('index.sci');//function to get index
exec('get_deltas.sci');//function to get higher order deltas if exists

x = input("Enter the values of x = ");//x values
y = input("Enter the values of y = ");//y values
a = input("Enter the value of x at which derivative is to be found out =
    ");
n = index(x,a);//get index of a

N = length(x);// find length of x (or y)
if N <= 1 // if x has only 1 value
  disp("N has to be greater than 1!");
  abort;
end
h = x(2) - x(1); // length of h

delta_1 = delta(y,N);// first oder delta
[delta_2,delta_3,delta_4,delta_5] = get_deltas(delta_1,N);//higher order
    deltas

f_dash_x = (1/h)*(delta_1(n) - (1/2)*delta_2(n) + (1/3)*delta_3(n) -
    (1/4)*delta_4(n) + (1/5)*delta_5(n))
//evaluate dy/dx as per the formula
disp("The value of the derivative at a is = ");
disp(f_dash_x);
```

7 Solution of Differentiation Equation

7.1 Solving an Ordinary Differential Equation using Euler's Method

```
// Euler Method for solving an ODE

// Suppose we wish to solve the following ODE:
//     y' = y
// with the initial point,
//     y0 = 1
// The analytical solution for this is:
//     y = y0*e^(x - x0)

clc
clear

// This is our ODE:
function dy = f(y)
    dy = y
endfunction

// Over the following points:
x = 0:0.5:10;
// We see the difference is:
h = 0.5
// therefore,
```

```
y(1) = 1 // since indexing begins at 1.

// We implement the Euler method:
// The Euler method is:
//      y{n+1} = yn + h*f(xn, yn)

for i = 1:length(x),
    y(i+1) = y(i) + h*f(y(i));
end

// The output, y is:
// disp(y)

// We compare the answer we have obtained using the analytical solution:
plot(exp(x'), 'green')
plot(y, 'red')
```

7.2 Solving an Ordinary Differential Equation using the Midpoint Method

```
// Midpoint method for solving an ODE

// Suppose we wish to solve the following ODE:
//      y' = y
// with the initial point,
//      y0 = 1
// The analytical solution for this is:
//      y = y0*e^(x - x0)

clc
clear

// This is our ODE:
function dy = f(y)
    dy = y
endfunction

// Over the following points:
x = 0:0.5:10;
// We see the difference is:
h = 0.5
// therefore,
y(1) = 1 // since indexing begins at 1.

// We implement the Midpoint method:
// The Midpoint method is
//      y{n+1} = yn + h*f(xn + h/2, yn + (h/2)*f(xn, yn))

for i = 1:length(x),
    y(i+1) = y(i) + h*f(y(i) + (h/2)*f(y(i)));
end

// We compare the answer we have obtained using the analytical solution:
plot(exp(x'), 'green')
plot(y, 'blue')
```

7.3 Solving an Ordinary Differential Equation using Runge Kutta Method

```
// Runge Kutta method of the fourth order for solving an ODE

// Suppose we wish to solve the following ODE:
//      y' = y
// with the initial point,
//      y0 = 1
```

```
// The analytical solution for this is:
//      y = y0*e^(x - x0)

clc
clear

// This is our ODE:
function dy = f(y)
    dy = y
endfunction

// Over the following points:
x = 0:0.5:10;
// We see the difference is:
h = 0.5
// therefore,
y(1) = 1 // since indexing begins at 1.

// We implement the Runge Kutta method:
// The Runge Kutta method (fourth order) is
//      y{n+1} = yn + (1/6)*h(k1 + 2*k2 + 2*k3 + k4)
// where,
//      k1 = f(xn          , yn)
//      k2 = f(xn + (1/2)*h, yn + (1/2)*h*k1)
//      k3 = f(xn + (1/2)*h, yn + (1/2)*h*k2)
//      k4 = f(xn + h      , yn + h*k3)

for i = 1:length(x),
    k1 = f(y(i));
    k2 = f(y(i) + 0.5*h*k1);
    k3 = f(y(i) + 0.5*h*k2);
    k4 = f(y(i) + h*k3);
    y(i+1) = y(i) + (1/6)*h*(k1 + 2*k2 + 2*k3 + k4);;
end

// We compare the answer we have obtained using the analytical solution:
plot(exp(x'), 'green')
plot(y, 'black')
```

7.4 Solving an Ordinary Differential Equation using Scilab's Internal ODE solver

```
// Scilab's internal ODE solver

// Suppose we wish to solve the following ODE:
//      y' = y
// with the initial point,
//      y0 = 1
// The analytical solution for this is:
//      y = y0*e^(x - x0)

clc
clear

// This is our ODE:
function dy = f(t, y)
    dy = y
endfunction

// Over the following points:
x = 0:0.5:10;
// Since we will be using Scilab's internal ODE solver, we do not
// need to specify the step size; Scilab will decide the appropriate
// step size

// therefore,
y0 = 1 // here, we are not initializing a vector
```

```
x0 = 0

y = ode(y0, x0, x, f);

// We compare the answer we have obtained using the analytical solution:
plot(exp(x'), 'green')
plot(y, 'magenta')
```

8 To find the Roots of Linear Equations

8.1 Finding the Solution of Linear Equations using Gaussian Elimination

```
// Solving a linear equation using Gaussian Elimination:

// Consider the following system of equations:
//      2x + y -  z =  8
//     -3x - y + 2z = -11
//     -2x + y + 2z = -3

// We can represent the above system in matrix form as follows:
//      Ax = b
// where,
A = [ 2  1 -1
     -3 -1  2
     -2  1  2]
// and
b = [ 8
     -11
     -3]

// We find the augmented matrix:
Aug = [A, b]

// We have to perform the following elimination steps to bring
// the entries below the first pivot to zero:
// R2 --> R2 + (3/2)*R1
// R3 --> R3 + R1

Aug(2, :) = Aug(2, :) + (3/2)*Aug(1, :)
Aug(3, :) = Aug(3, :) + Aug(1, :)

// We now perform the following elimination steps to bring
// the entry below the second pivot to zero:
// R3 --> R3 - 4*R2

Aug(3, :) = Aug(3, :) - 4*Aug(2, :)

// Therefore, we have the matrices:
U = Aug(:, 1:3),
c = Aug(:, 4)

// There, we find x, y and z in the reverse order by back substitution:
z = c(3)/U(3, 3)
y = (c(2) - U(2, 3)*z)/U(2, 2)
x = (c(1) - U(1, 3)*z - U(1, 2)*y)/U(1, 1)

// We compare this with the answer that Scilab's internal solver gives:
A\b
```

8.2 Finding the Solution of Linear Equations using Cramer's Method

```
// Solving a linear equation using Cramer's rule

// Consider the following system of equations:
//      2x + y -  z =  8
//     -3x - y + 2z = -11
//     -2x + y + 2z = -3

// We can represent the above system in matrix form as follows:
//      Ax = b
// where,
A = [ 2  1 -1
     -3 -1  2
     -2  1  2]
// and
b = [ 8
     -11
     -3]

// The values of x, y and z are given, by Cramer's rule, as:
x = det([b, A(:, [2 3])])/det(A)
y = det([A(:, 1) b A(:, 3)])/det(A)
z = det([A(:, [1 2]), b])/det(A)

// We compare this with the answer that Scilab's internal solver gives:
A\b
```