NUMERICAL

ASSIGNMENT 1
ROOT FINDER & INTERPOLATION

Team Members

Alaa Samir (1)

Bassant Ahmed (18)

Sara Eldafrawy (29)

Sohayla Mohammed (32)

Root finder

Introduction

We can easily find the roots or an approximation of them for a nonlinear equations by the use of iterative methods :

- Graphical method
    - By plotting the function and finding its intersections with the x-axis.
- Bracketing methods
    - Bisection method.
    - False Position.
- Open methods
    - Fixed Point.
    - Newton-Raphson.
    - Secant.

And many others.

The aim of this project is to implement different numerical methods and to compare and analyse their behaviour.

Bisection

Algorithm and Implementation

Code

```
function [root, xls, xus, xrs, itr, error, errorMsg, time] = bisection(f,xl, xu, es, imax)
tic;
f = inline(f,'x');
errorMsg = 'VALID';
ea = 100000;
itr = 1;
time = 0;
xls = nan(1,imax);
xus = nan(1,imax);
xrs = nan(1,imax);
error = nan(1,imax);
if  feval(f,xl) * feval(f, xu)>0 % if guesses do not bracket, exit
    root = Nan;
    errorMsg = 'the function of the two points have the same sign';
    return
end
for i=1:1:imax
    xls(i) = xl;
    xus(i) = xu;
    xr = (xu+xl)/2;
    xrs(i)= xr;% compute the midpoint xr
    if (i > 1)
        ea = abs((xrs(i)-xrs(i - 1))); % approx. relative error
        error(i)= ea;
    end
    test = feval(f,xl) * feval(f, xr); % compute f(xl)*f(xr)
    if (test < 0)
        xu = xr;
    else
        xl = xr;
    end
    if (test == 0)
        ea=0;
    end
    if (ea < es)
        break;
    end
    itr = i;
end
root = xr;
time = toc;
```

Procedure

- Starting with two initial values (xl , xu) , we find values of f(xl), f(xu).

- Looping over the maximum number of iterations given by the user (50 by default), and for i_{th} iteration we find the value of approximated xr(i) by using the formula :
    - $xr(i) = \frac{xl + xu}{2}$
    - f(xr)*f(xl) to decide which interval will be taken next for next iteration.
- We get the absolute value of approximated error err(i) where : $|Xr_i - Xr_{i-1}|$

Data-structure and Built in Functions

- Array of zeros :
    - Stores the values of (xls, xus, xrs, error) resulted in each iteration.
- Built-in functions :
    - feval()
        - To evaluate f(x).
    - abs()
        - To compute absolute of $|Xr_i - Xr_{i-1}|$.

Analysis and Pitfalls

Introduction

Bisection is applicable for numerically solving the equation f(x) = 0 for the real variable x, where f is a continuous function defined on an interval [a, b] and where f(a) and f(b) have opposite signs. In this case a and b are said to bracket a root since, by the intermediate value theorem, the continuous function f must have at least one root in the interval (a, b).

Derivation of the formula

- Every new x is derived as mid point in the interval [a , b].
    - $x_n = \frac{a + b}{2}$

Convergence

- The method is guaranteed to converge to a root of *f* if *f* is a continuous function on the interval [*a*, *b*] and *f*(*a*) and *f*(*b*) have opposite signs.

- The absolute error is halved at each step so the method converges linearly, which is comparatively slow.
- $\varepsilon_{n+1} = c \times \varepsilon_n$

Notes

- This formula can be used to determine in advance the number of iterations that the bisection method would need to converge to a root to within a certain tolerance.
    - $n = \frac{log(\varepsilon_0) - log(\varepsilon)}{log(2)}$; where $\varepsilon = b - a$
- No account is taken of the fact that if f(xl) is closer to zero, it is likely that root is closer to xl.
- If a function f(x) is such that it just touches the x-axis it will be unable to find the lower and upper guesses.
    - $f(x) = x^2$
- Function changes sign but root does not exist
    - $f(x) = 1/x$
- Can't solve if there's an even number of roots in the interval.

False Position

Algorithm and Implementation

Code

```
function [root, xls, xus, xrs, error, errorMsg, time, itr] = false_position(f, xl, xu, es, maxit)
tic;
time = 0;
xls = nan(1,maxit);
xus = nan(1,maxit);
xrs = nan(1,maxit);
fxl = nan(1,maxit);
fxu = nan(1,maxit);
fxr = nan(1,maxit);
error = nan(1,maxit);
xls(1) = xl;
xus(1) = xu;
fxl(1) = feval (f, xls(1));
fxu(1) = feval (f, xus(1));
errorMsg = 'VALID';
root = Nan;

if fxl(1) * fxu(1) > 0.0
    errorMsg = 'the function of the two points have the same sign';
end
itr = 1;
for i = 1:maxit
    xrs(i) = xus(i) - fxu(i)*(xus(i) - xls(i)) / (fxu(i) - fxl(i));
    fxr(i) = feval (f, x(i));
    root = xrs(i);
    if fxr(i) == 0
        break;
    elseif fxr(i) * fxl(i) > 0
        xus(i+1) = xus(i);
        fxu(i+1) = fxu(i);
        xls(i+1) = xrs(i);
        fxl(i+1) = fxr(i);
    else
        xls(i+1) = xls(i);
        fxl(i+1) = fxl(i);
        xus(i+1) = xrs(i);
        fxu(i+1) = fxr(i);
    end
         if ( i > 1 )
             error(i) = abs (x(i)-x(i-1));
             if (error(i) < es)
                break;
             end
         end
         itr = i;
end
time = toc;
if itr > maxit
    errorMsg = 'zero not found to desired tolerance';
end
```

Procedure

- Starting with two initial values (xl , xu) , we find values of f(xl), f(xu).
- Looping over the maximum number of iterations given by the user (50 by default), and for i_{th} iteration we find the value of approximated xr(i) by using the formula :
    - $x_r = \frac{xlfu - xufl}{fu - fl}$
    - f(xr)*f(xl) to decide which interval will be taken next for next iteration.
- We get the absolute value of approximated error err(i) where : $| Xr_i - Xr_{i-1} |$

Data-structure and Built in Functions

- Array of zeros :
    - Stores the values of (xls, xus, xrs, fxl, fxu, fxr, error) resulted in each iteration.
- Built-in functions :
    - feval()
        - To evaluate f(x).
    - abs()
        - To compute absolute of $| x_i - x_{i-1} |$.

Analysis and Pitfalls

Introduction

Since the speed of convergence is important, then it could be preferable to first try a usually-faster method, going to Bisection only if the faster method fails to converge, or fails to converge at a useful rate.

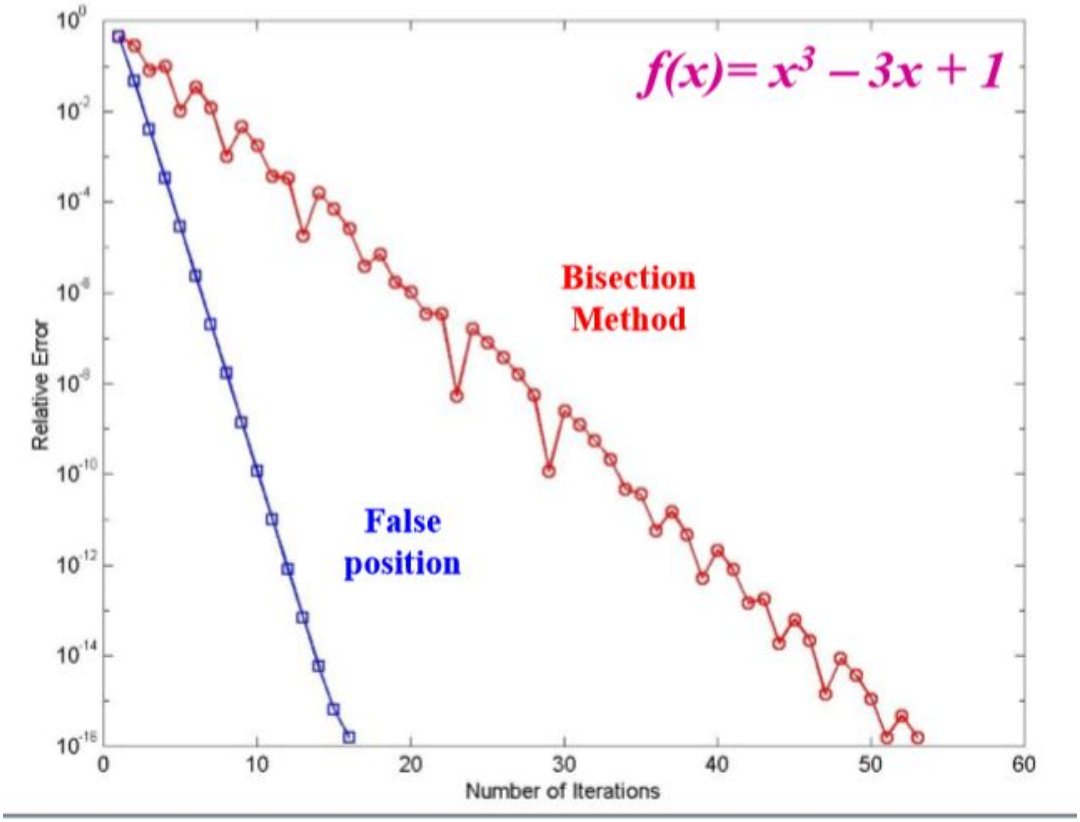
Regula Falsi always converges, and has versions that do well at avoiding slowdowns, which makes it a good choice when speed is needed, and when Newton's method doesn't converge, or when the evaluation of the derivative is too time-consuming for Newton's to be useful.

Derivation of the formula

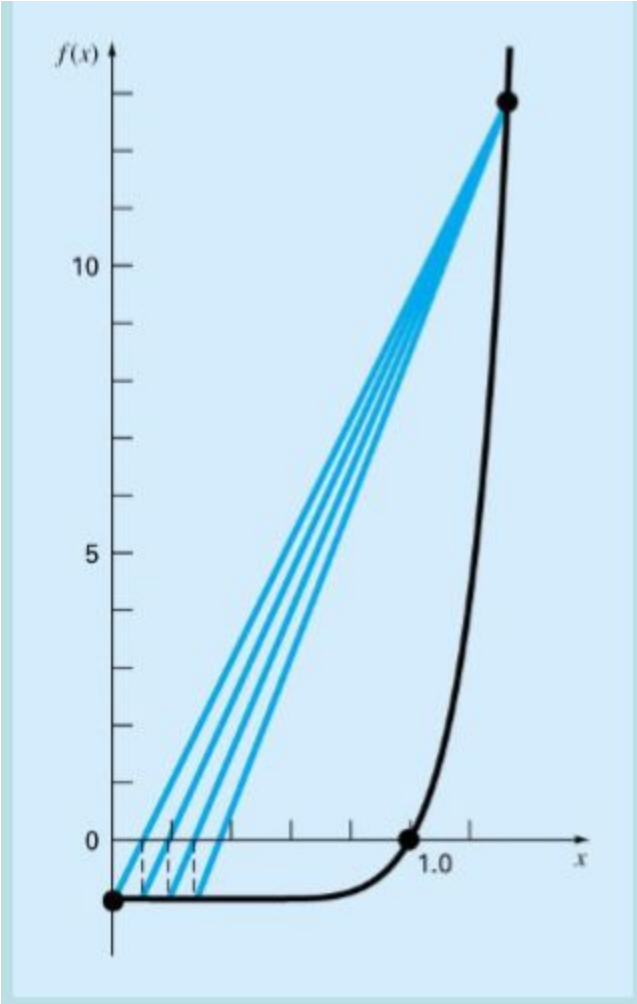
- Regula Falsi assumes that f(x) is linear—even though these methods are needed only when f(x) is *not* linear, and usually work well anyway.
- The ratio of the change in x, to the resulting change in y is:
  - $\frac{x_2 - x_1}{y_2 - y_1}$
- Because y, most recently, is y_2, and we want y to be 0, then the change that we want in y is:
  - $0 - y_2 = -y_2$
- So, given that desired change in y, and given the expected ratio of change in x to change in y, then the best (linearly-gotten) estimate for the right x-value—the best estimate for the solution—is:
  - $x_{i+1} = x_i - f(x_i)\frac{x_i - x_{i-1}}{f(x_i) - f(x_{i-1})}$
- Which equals :-
  - $x_{i+1} = \frac{x_{-1i} f(x_i) - x_i f(x_{i-1})}{f(x_i) - f(x_{i-1})}$

Convergence

- False position has linear convergence.
- It converges faster than bisection but not in all cases.



- Case where it converges slower than bisection


f(x)
10
5
0
1.0
x

Secant

Algorithm and Implementation

Code

```
function [root, err, xs0,xsl, xsn, it, time, errorM] = secant(f, xi_1, xi, ess, maxI)
    tic;
    f = char(f);
    errorM = 'VALID';
    xs0 = zeros(1, (maxI+1));
    xsl = zeros(1, (maxI+1));
    xsn = zeros(1, (maxI+1));
    fxi = zeros (1, maxI+1);
    fxi_1 = zeros (1, maxI+1);
    err = zeros(1, maxI+1);
    xs0(1) = xi_1;
    xsl(1) = xi;
    it = 1;
    syms x;
    breakError = false;
    try
    fxi_1(it) = subs(f,x, xs0(it));
    fxi(it) = subs(f,x, xsl(it));
    catch
        errorM = "Invalid equation";
        return;
    end
try
    while(it <= maxI)
           if(fxi_1(it) - fxi(it) == 0)
                errorM = "division by zero";
                return;
           end
         xsn(it) = xsl(it) - (fxi(it) * (xs0(it) - xsl(it))) / (fxi_1(it) - fxi(it));

         err(it) = abs(xsn(it) - xsl(it));
```

```
            if(err(it) < abs(ess))
                breakError = true;
               break;
            end
           it = it+1;
           fxi_1(it) = fxi(it-1);
           xs0(it) = xs1(it-1);
           xs1(it) = xsn(it-1);
           try
           fxi(it) = subs(f,x,xs1(it));
           catch
               errorM = "Invalid equation";
               return;
           end
           if(fxi(it) == 0)
               break;
           end
       end
catch
       errorM = "Un-identified error";
       return;
end
        time = toc;
       if(breakError)
           root = xsn(it);
       else
           root = xs1(it);
       end
end
```

Procedure

- Starting with two initial values (xi_1 , xi) , we find values of f(xi_1), f(xi).
- Looping over the maximum number of iterations given by the user (50 by default), and for i_{th} iteration we find the value of approximated xsn(i) by using the formula :
    - $xsn(i) = xi - f(xi) \frac{xi\_1 - xi}{f(xi\_1) - f(xi)}$
- Before using the formula, We check if $f(xi\_1) - f(xi)$ is equal zero or not as to avoid division by zero.
- We get the absolute value of approximated error err(i) where :
    - $err(i) = |xsn(i) - xi|$

- Check if this value exceeds the tolerance entered by the user; if it does then we break from the loop and return the approximated root.
- We find the values of the new xi_1 , xi , f(xi_1) and f(xi) where :
    - $xi\_1_{new} = xi_{old}$
    - $xi_{new} = xsn_{old}$
    - $f(xi\_1)_{new} = f(xi)_{old}$
    - $f(xi)_{new} = f(xsn_{old})$
- If $f(xi)_{new}$ is equal zero, break from the loop and return true value of root.

Data-structure and Built in Functions

- Array of zeros :
    - Stores the values of (xi_1, xi, xsn, f(xi), f(xi_1) and err) resulted in each iteration.
- Built-in functions :
    - Syms -> to make 'x' a symbolic variable.
    - Subs -> to substitute a value of 'x' in equation f(x).

Analysis and Pitfalls

Introduction

In Newton-Raphson method, a potential problem may occur while evaluating the derivative of a function whose derivative is difficult or inconvenient to evaluate. For this cases we use Secant method instead. Although, Newton-Raphson converges faster (order of 2).

Derivation of formula

From Newton

- The derivative of a function at point x, can be approximated by a backward finite divided difference:
    - $\bar{f}(x) \approx \frac{f(x_0) - f(x)}{x_0 - x}$

- Substitute $\bar{f}(x)$ in Newton's formula yields :
  - $x_{i+1} = x_i - f(x_i) \frac{x_{i-1} - x_i}{f(x_{i-1}) - f(x_i)}$

Analytically

- If we had two initial points x_0 , x_1 , we construct a line through the points (x_0 , f(x_0)) and (x_1 , f(x_1)).
- In slope-intercept form, the line has the equation :
  - $Y = \frac{f(x_1) - f(x_0)}{x_1 - x_0}(x - x_1) + f(x_1)$
- Solving this equation for y equal zero yield :
  - $x = x_1 - f(x_1) \frac{x_0 - x_1}{f(x_0) - f(x_1)}$

Convergence

- The secant method converges to a root of equation f(x), if the initial x_0, x_1 values are sufficiently close to the root.
  - There is no general definition of "close enough", but the criterion has to do with how "wiggly" the function is on the interval $[x_0, x_1]$. For example, if f is differentiable on that interval and there is a point where $\bar{f}$ is equal zero on the interval, then the algorithm may not converge.
  - There's no certainty on the convergence or the divergence of the secant method but the above conditions give us a sense that it MAY diverge.
- Secant method is sublinear with order of convergence approximately equal to the golden ratio $\alpha \approx 1.168$.

Notes

- Secant method can be modified so that we only need one initial guess instead of two.
- To do so, replace $x_0 - x_1$ with δ approximation of $\bar{f}$ will be :
  - $\bar{f}(x) \approx \frac{f(x_i + \delta x_i) - f(x_i)}{\delta x_i}$

- Modified secant method will be :
  - $x_{i+1} = x_i - f(x_i) \frac{\delta x_i f(x_i)}{f(x_i + \delta x_i) - f(x_i)}$
- If δ is too small, the method can be swamped by roundoff error caused by subtractive cancellation in the denominator.
- If δ is too big, this technique can become inefficient and even divergent.
- If δ is selected properly, this method provides a good alternative for cases when developing two initial guess is inconvenient.

Newton-Raphson

Algorithm and Implementation

Code

```
function [fxi, dfxi, oldX, newX, root, itr, error, errorMsg, executionTime] = newtonRaphson (f, xi, es, maxItr)
  tic;
  errorMsg = 'VALID';
  syms x
  df = diff(f,x);
  newX = zeros(1, maxItr);
  oldX = zeros(1, maxItr);
  fxi = zeros(1, maxItr);
  dfxi = zeros(1, maxItr);
  error = 100.0;
  itr = 1;
  root = 0.0;
  executionTime = 0.0;
  oldX(itr) = xi;
  %check if method won't converge
  d2f = diff(df,x);
  try
    d2fValue = feval(d2f,oldX(itr));
  catch
      d2fValue = feval(d2f);
  end
  dfValue = feval(df,oldX(itr));
  if (abs(d2fValue / (2 * dfValue)) > 1)
    errorMsg = 'Method will diverge';
    return;
  end
  --------
```

```
while (itr <= maxItr)
  fxi(itr) = feval(f, oldX(itr));
  dfxi(itr) = df(oldX(itr));

  if (dfxi == 0)
    errorMsg = 'Newton method failed to converge (Division by Zero)';
    break;
  end
  newX(itr) = oldX(itr) - (fxi(itr) / dfxi(itr));
  root = newX(itr);

  if (itr ~= 1)
    error = (abs(newX(itr) - newX(itr - 1)) / newX(itr)) * 100;
  end
  if (error < es || feval(f, root) == 0)
    break;
  end
  itr = itr + 1;
  oldX(itr) = newX(itr - 1);

end
executionTime = toc;
```

Procedure

- Starting with one initial value (xi) , we find values of f(xi) and f`(xi).
- Looping over the maximum number of iterations given by the user (50 by default), and for i_{th} iteration we find the value of approximated x_{i+1} by using the formula :
  - $x_{i+1} = xi - \frac{f(xi)}{\dot{f}(xi)}$
- Before using the formula, We check if $\dot{f}(xi)$ is equal zero or not as to avoid division by zero.
- We get the relative value of approximated error err(i) where :
  - $err(i)\% = \frac{|x_{i+1} - xi|}{x_{i+1}} \times 100\%$
- Check if this value is less than the tolerance entered by the user; if it does then we break from the loop and return the approximated root.

- We find the values of the new xi , f(xi) and f`(xi) where :
    - $x_i = x_{i+1}$

Data-structure and Built in Functions

- Array of zeros to store:
    - $x_{i+1},\ x_i,\ f(x_i),\ \bar{f}(x_i)$
- Built in functions :
    - diff()
        - To differentiate f(x).
    - feval()
        - To evaluate f(x).

Analysis and Pitfalls

Introduction

The idea of the method is :

Starting with an initial guess which is reasonably close to the true root, then the function is approximated by its tangent line, and one computes the x-intercept of this tangent line. This x-intercept will typically be a better approximation to the function's root than the original guess, and the method can be iterated until a reasonable value of the root is reached .

Derivation of formula

- Suppose we have initial point xi.
- The equation of the tangent line of the curve $y = f\ (x)$ at the point $x = x_i$ is :
    - $Y = \bar{f}(x_i)(x - x_i) + f(x_i)$
- Solving the tangent equation at y equals zero yields :
    - $x = xi - \frac{f(xi)}{\bar{f}(xi)}$

Convergence

- The Newton-Raphson method diverges from a root of equation f(x), if :
  - $\left|\frac{f'(x_o)}{2f(x_o)}\right| > 1$
- Newton-Raphson method is quadratic with order of convergence equal to 2, except when the root is a multiple roots .
- When the initial guess is close to the root, Newton-Raphson method usually converges.

Notes

- An inflection point (f''(x)=0) at the vicinity of a root causes divergence.
- A local maximum or minimum causes oscillations.
- It may jump from one location close to one root to a location that is several roots away.
- A zero slope causes division by zero.
- It can be slow (slower than a bracketing method) in some cases.
- Multiple roots :
  - A multiple root corresponds to a point where a function is tangent to the x axis.
  - For multiple roots, Newton-Raphson and Secant methods converge linearly, rather than quadratic convergence.
- Modification of Newton-Raphson for multiple roots. Where 'm' is the multiplicity of the root
  - If 'm' is known :
    - $\widehat{f} = f^{\frac{1}{m}}$, $\widehat{f}(\alpha) = 0$ where α is a single root.
    - $x_{i+1} = x_i - m\frac{f(x)}{f'(x_i)}$
  - If 'm' is unknown :
    - $\widehat{f} = \frac{f(x)}{f'(x_i)}$
    - $x_{i+1} = x_i - m\frac{f(x_i)f'(x_i)}{[f'(x_i)]^2 - f(x_i)f''(x_i)}$

Fixed Point

Algorithm and Implementation

Code

```
function[g,arrX,arrGofX,root,i,arrError,errorMsg,executionTime] = fixedPoint(f, xi, ess, maxI)
    tic;
    errorMsg = 'VALID';
    arrX = zeros(1, maxI);
    arrGofX = zeros(1, maxI);
    arrError = zeros(1, maxI);
%     syms ftemp(x);
%     ftemp(x) = f;
    syms x;
    g = f + x
    g_dash = diff(g,x)
    i = 1;
    if (abs(feval(g_dash,xi)) > 1)
        disp("can't converge")
        errorMsg = "Fixed Point method failed to converge because |g'(x)| > 1";
    end
    %first iteraion
    err = 100;
    arrX(i) = xi;
    arrGofX(i) = g(xi)
    arrError(i) = err
    xi = arrGofX(i)
    i = i+1
    while (i <= maxI)
                xi = g(xi)
                arrX(i) = arrGofX(i-1)
                arrGofX(i) = xi
        err = abs(arrGofX(i) - arrX(i))
                arrError(i) = err
        if(err < ess)
            break;
        end
                i= i+1;
    end
        root = arrX(i);
    disp(root);
        executionTime = toc;
```

Procedure

- Starting with one initial value (xi) , we find values of g`(xi).
- If |g'(xi)| > 1 then it won't converge.
- Looping over the maximum number of iterations given by the user (50 by default), and for i_{th} iteration we find the value of approximated x_{i+1} by computing g(xi) :
    - $x_{i+1} = g(x_i)$

- If $x_{i+1} = x_i$ then we found the root
- We get the value of approximated error err(i) where :
    - $err(i) = |x_{i+1} - x_i|$

Data-structure and Built in Functions

- Array of zeros to store:
    - $x_i,\ g(x_i)$
- Built in functions :
    - diff()
        - To differentiate f(x).
    - feval()
        - To evaluate f(x).
    - abs()
        - To compute absolute of g'(xi)

Analysis and Pitfalls

Introduction

To find the root for f(x) = 0 , we construct a magic formulae $x_{i+1} = g(x_i)$ to predict the root iteratively until x converge to a root.

To find the root for f(x) = 0, we reformulate f(x) = 0 so that there is an x on one side of the equation.

- f(x) = 0 ⇔ g(x) = x

If we can solve g(x) = x, we solve f(x) = 0

X is known as the fixed point of g(x)

We solve g(x) =x by computing

Xi+1 = g(xi) with xo given

until xi+1 converges to x.

Convergence

By definition

$$\delta_i = \alpha - x_i \quad (1)$$

$$\delta_{i+1} = \alpha - x_{i+1} \quad (2)$$

Fixed point iteration

$$\alpha = g(\alpha) \quad (3)$$

$$x_{i+1} = g(x_i) \quad (4)$$

(3) - (4) => $\alpha - x_{i+1} = g(\alpha) - g(x_i)$ (5)

Sub (2) in (5) => $\delta_{i+1} = g(\alpha) - g(x_i)$ (6)

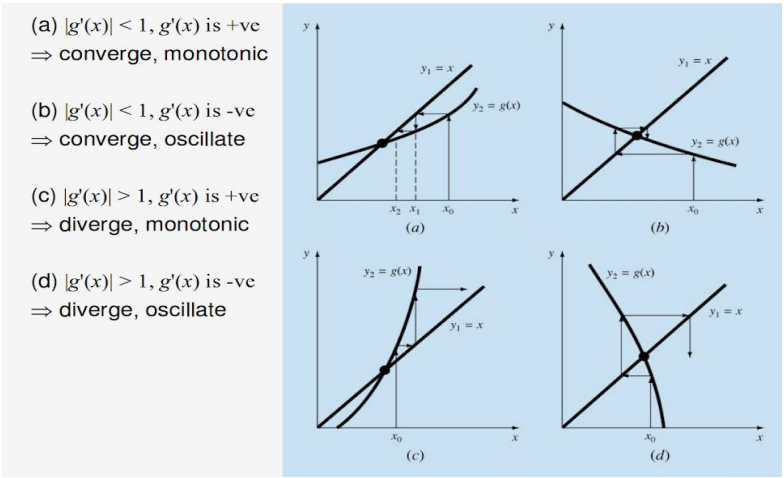
According to the derivative mean-value theorem, if g(x) and g′(x) are continuous over an interval xi x α, there exists a value x = c within the interval such that

- $g'(c) = \frac{g(\alpha) - g(xi)}{\alpha - xi}$ (7)

From (1) and (6), we have $\delta_i = \alpha - x_i$ and $\delta_{i+1} = g(\alpha) - g(x_i)$

Thus (7) => $g'(c) = \frac{\delta_{i+1}}{\delta_i}$ => $\delta_{i+1} = g'(c)\delta_i$

- Therefore, if |g′(c)| < 1, the error decreases with each iteration. If |g′(c)| > 1, the error increases.
- If the derivative is positive, the iterative solution will be monotonic.
- If the derivative is negative, the errors will oscillate.



Bierge Vieta

Algorithm and Implementation

Code

```
function [aArray, bArray, cArray, root, error, errorMsg, itr, time] = BirgeVieta(f,xo,es,maxit);
tic;
syms x
coefficients = coeffs(f(x), 'All');
orderOfPoly = size(coefficients,2);
initialVal = xo;
aArray = zeros(1,orderOfPoly);
for j = 1: orderOfPoly
    aArray(j) = coefficients(j);
end
bArray(:,:,maxit) = zeros(1,orderOfPoly);
cArray(:,:,maxit) = zeros(1,orderOfPoly);
error = nan(1,maxit);
tempbArray = zeros(1,orderOfPoly);
tempcArray = zeros(1,orderOfPoly);
tempbArray(1) = coefficients(1);
tempcArray(1) = coefficients(1);
itr = 1;
errorMsg = "VALID";
while (itr <= maxit)
for i=2:orderOfPoly
    tempbArray(i) =  tempbArray(i-1)*initialVal + coefficients(i);
    if (i < orderOfPoly)
    tempcArray(i) =  tempcArray(i-1)*initialVal + tempbArray(i);
    end
end
root = initialVal - (tempbArray(orderOfPoly) / tempcArray(orderOfPoly - 1));
bArray(:,:,itr) = tempbArray;
cArray(:,:,itr) = tempcArray;
```

```
error(itr) = (abs (root-initialVal));
        if (error(itr) < es )
            break;
        end
initialVal = root;
itr = itr + 1;
end
errorMsg = "Birge Vieta method has diverged";
time = toc;
```

Procedure

- Starting with xi as an initial value and using the coefficients of the given equation as values of ai we calculate the values of all bi and ci.

- We calculate xi+1 by using b0 and c1 as values for f(xi) and f`(xi) using the formula :

$$x_{i+1} = xi - \frac{b1}{c0}$$

- We get the relative value of approximated error err(i) where :
  - $err(i)\% = \frac{|x_{i+1} - xi|}{x_{i+1}} \times 100\%$
- Check if this value is less than the tolerance entered by the user; if it does then we break from the loop and return the approximated root.
- We find the values of the new xi , f(xi) and f`(xi) where :
  - $x_i = x_{i+1}$

Data-structure and Built in Functions

- Array of arrays to store the values of bi and ci at each iteration.
- 1D array to store the coefficients of the equation.

Analysis and Pitfalls

Introduction

NR method with f(x) and f′(x) evaluated using Horner's method. Once a root is found, reduce order of polynomial

Derivation of the formula

f(x) $= a_0 + a_1 x + a_2 x^2 + \ldots + a_m x^m$

$= (x - r)(b_1 + b_2 x + \ldots + b_m x^{m-1}) + b_0 = (x - r)h(x) + b_0$

$b_m = a_m \qquad b_j = a_j + r b_{j+1}$ j=m-1,m-2,...,1,0

$f'(x) = h(x) + (x - r)h'(x)$

f′(r) = h(r)

$$h(x) = (b_1 + b_2 x + \ldots + b_m x^{m-1})$$

$$= (x - r)(c_1 + c_2 x + \ldots + c_m x^{m-2}) + c_1$$

$c_m = b_m$ $\quad c_j = b_j + r c_{j+1}$ $\quad$ j=m-1,m-2,...,1

f(r) = b_0 $\quad f'(r) = h(r) = c_1$

$$x_{i+1} = g(x_i) = x_i - \frac{f(x)}{f'(x_i)} = x_i - \frac{b_0}{c_1}$$

Convergence

Bierge Vieta is somehow applied in the same way as newton Raphson and therefore we can consider its divergence to be similar to newton which is as follows:
The Newton-Raphson method diverges from a root of equation f(x), if :

- $\left| \frac{f'(x_o)}{2f(x_o)} \right| > 1$

General Algorithm

Algorithm and Implementation

Code

```
function [roots, itr, berrorMsg] = generalAlgorithm(f, startInterval, endInterval, es)
equ = inline(f,'x');
syms x
c = coeffs(equ(x), 'All');
number = size(c,2);
roots = nan(1,number - 1);
berrorMsg = 'VALID';
itr = 1;
isPoly = checkIfPoly(f);
if (isPoly)
    points = linspace(startInterval, endInterval,20);
else
    points = linspace(startInterval, endInterval, 3);
end
numOfPoints = size(points,2);
if (isPoly)
for i = 1:numOfPoints - 1
    if (itr <= number - 1)
    [root, errorMsg] = bisection(equ,points(i),points(i + 1),es,50);
    berrorMsg = errorMsg;
    bRoot = root;
    if (~isnan(bRoot))
        if (~checkIfExist(roots, bRoot, es))
            roots(itr) = bRoot;
            itr = itr + 1;
        end
    end
    else
        break;
    end
end
```

```
else
for i = 1:numOfPoints - 1
    if (itr <= number - 1)
        [root, errorMsg] = secant(f,points(i),points(i + 1),es,3);
        berrorMsg = errorMsg;
        sRoot = root;
        if (~isnan(sRoot))
            if (sRoot < endInterval && sRoot > startInterval && ~checkIfExist(roots, sRoot, es))
              roots(itr) = sRoot;
              itr = itr + 1;
            end
        end
    else
        break;
    end
end
```

Procedure

- The input function is first checked to determine whether it is a polynomial or a non polynomial function.
- If Function is found to be polynomial the following procedure is considered:
    - The Given interval is then divided into a number of intervals so that the step between each 2 values is lower than 1 yet greater than 0.1 in order to avoid errors.
    - Each interval is given to the Bisection method to find the roots within.
    - The output roots is then stored in an array for plotting.
- If the Function is found to be a non polynomial, whether sin, cos, ln, etc. the following procedures are considered:
    - The interval is divided into 3 intervals as when more intervals are given the same root might with a high probability appear more than once.
    - Each is given to the Secant method to find the roots.
    - The output roots is plotted.

Data-structure and Built in Functions

- Array of nan to store:
    - Roots
- Built in functions :
    - inline
        - To convert string to symbolic function
    - size
        - To determine size of an array

Analysis and Pitfalls

Introduction

General Algorithm method doesn't have a specific implementation, as it could be implemented with different ways and with either one of the already implemented methods or with a group of them.

Our implementation is based upon the nature of the function, as polynomial functions were chosen to be plotted with bisection due to its definite convergence, it could as well be plotted using false position as both bracketing methods do converge. And the chosen interval is small so that either 1 or no root could be found in the interval, however taking into consideration the round off error the interval is not that small.

As for the non-polynomial function, the secant method was chosen to solve then as it doesn't require the root to be bracketed by the 2 points.

Pitfalls:

- In secant, the multiplicity will not be calculated and multiple roots won't be found
- Due to the small interval taken in bisection, if the intervals in which the roots are to be found will be more than 15, each will go under bisection method with about 0.0001 error in root which is slow.
- Not all roots will be found in secant, however in the default trials they were found.

Interpolation

Newton Interpolation

Algorithm and Implementation

Code

```
function funcOutput = DividedDifference (Xs,Ys)
  [~, col] = size(Xs);
  coefficients = zeros(1,col);
  newXs = zeros(1,col);
  tempArray = zeros(1,col);

  for i = [1:1:col]
    newXs(i) = i + 1;
    tempArray(i) = Ys(i);
  end

  coefficients(1) = Ys(1);
  loops = col - 1;

  while(loops >= 1)
   for i = [1:1:loops]
     tempArray(i) = (tempArray(i+1) - tempArray(i)) / (Xs(newXs(i)) - Xs(i));
     newXs(i) = newXs(i) + 1;
   end
   loops = loops - 1;
   coefficients(col - loops) = tempArray(1);
  end
```

```
func = num2str(coefficients(1));
loops = 2;
while loops <= col
  start = true;
  for i =[1:1:loops]
    if start
      s1 = '+';
      s2 = num2str(coefficients(loops));
      func = strcat(func, s1,s2);
      start = false;
  else
      s3 = '*(x-';
      s4 = num2str(Xs(i-1));
      s5 = ')';
      func = strcat(func, s3,s4,s5);
    end
  end
  loops = loops + 1;
end

syms f(x)
f(x) = func;
funcOutput = simplify(f);
```

Procedure

- Input the polynomial order required
- Add points and their corresponding values, where the number of points are greater than the polynomial order by one.
- Using the table method we loop on the array of values in order to calculate the next column in the table, where the stopping condition is when the table contains only 1 row.
- The output result is then concatenated to form a non simplified equation.
- Using simplify the equation is set in the simplest form.

Data-structure and Built in Functions

- Array of zeros to store:
    - Every iteration Coefficients.
    - Corresponding Xs to new column.
- Built in functions :
    - num2str()
        - To convert numbers to string for concatenation.
    - strcat()
        - To concatenate strings to find final equation.
    - simplify()
        - To simplify the output equation.

Analysis and Pitfalls

Introduction

The Interpolation concepts is based upon the fact that consider a data set of n+1 points where yi=f(xi) at n+1 through all x points x0,x1…xi,…xn: xj> xj-1, there is a unique polynomial gn(x) of order n: that passes through all n+1 points.

Error

Structure of interpolating polynomials is similar to the Taylor series expansion in the sense that finite divided differences are added sequentially to capture the higher order derivatives. For an nth-order interpolating polynomial, an analogous relationship for the error is:

$$R_n = \frac{f^{(n+1)}(\zeta)}{(n+1)!}(x - x_0)(x - x_1)....(x - x_n)$$

Notes

- Linear Interpolation formula derivation:

- $\frac{f_1(x)-f_0(x)}{x-x_0} = \frac{f(x_1)-f(x_0)}{x_1-x_0}$
- $f_1(x) = f(x_0) + \frac{f(x_1)-f(x_0)}{x_1-x_0}(x-x_0)$

- General Form of Newton Interpolation Formula:
  - $f_n(x) = b_0 + b_1(x-x_0) + b_2(x-x_0)(x-x_1) + ... + b_n(x-x_0)(x-x_1).....(x-x_{n-1})$
  - $f[x_i, x_j] = \frac{f(x_i)-f(x_j)}{x_i-x_j}$

Lagrange Interpolation

Algorithm and Implementation

Code

```
function[func executionTime] = LaGrange(arrOfX, arrOfY)
    tic;
    syms func(x);
        syms temp1(x) temp2(x);
        temp2(x) = 1;
        func(x) = 0;
        i = 1;
    executionTime = 0.0;
        arrOfPts = [arrOfX;arrOfY]
        while (i <= size(arrOfPts,1)+1)
                j = 1;
                while (j <= size(arrOfPts,1)+1)
                        disp("i,j " + i +" " + j );
                        if (i ~= j)
                        temp1(x) = (x-arrOfPts(1,j)) / (arrOfPts(1,i) - arrOfPts(1,j));
                        temp2(x) = temp2(x) * temp1(x);
                        end
                        j = j + 1;
                end
                func(x) = func(x) + (temp2(x) * arrOfPts(2,i));
                i = i + 1;
                temp2(x) = 1;
        end
        func(x) = simplify(func(x));

    executionTime = toc;
```

Procedure

- Input the polynomial order required

- Add points and their corresponding values, where the number of points are greater than the polynomial order by one.
- Then there's two nested loops:
    - The first one to compute Li
    - The second one summation of Li*f(xi)
- The output result is then concatenated to form a non simplified equation.
- Using simplify the equation is set in the simplest form.

Data-structure and Built in Functions

- simplify()
    - To simplify the output equation.

Analysis and Pitfalls

Introduction

The Lagrange interpolating polynomial is simply a reformulation of the Newton's polynomial that avoids the computation of divided differences:

- $f_n(x) = \Sigma L_i(x) f(x_i)$
- $L_i(x) = \Pi \frac{x - x_j}{x_i - x_j}$

Above formula can be easily verified by plugging in x0, x1, … in the equation one at a time and checking if the equality is satisfied.

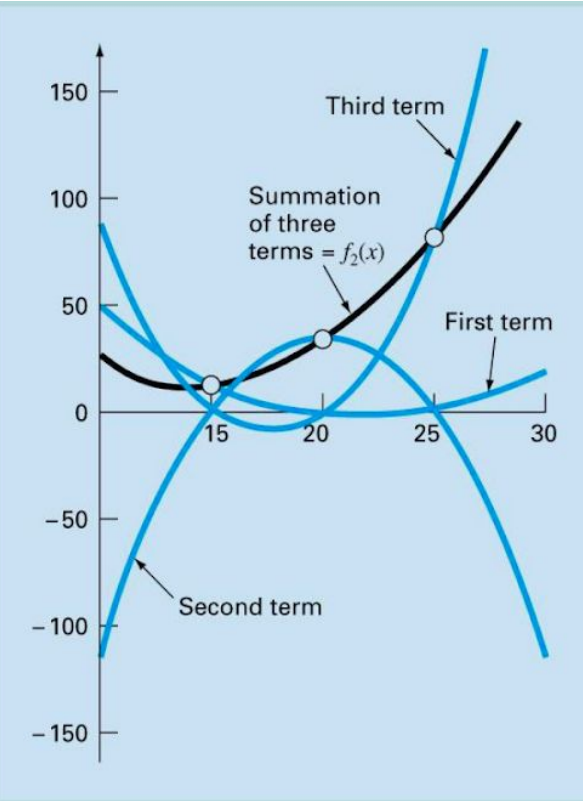
Notes

A visual depiction of the rationale behind the Lagrange polynomial. The figure shows a second order case:

- $f_2(x) = \frac{(x - x_1)(x - x_2)}{(x_0 - x_1)(x_0 - x_2)} f(x_0) + \frac{(x - x_0)(x - x_2)}{(x_1 - x_0)(x_1 - x_2)} f(x_1)$

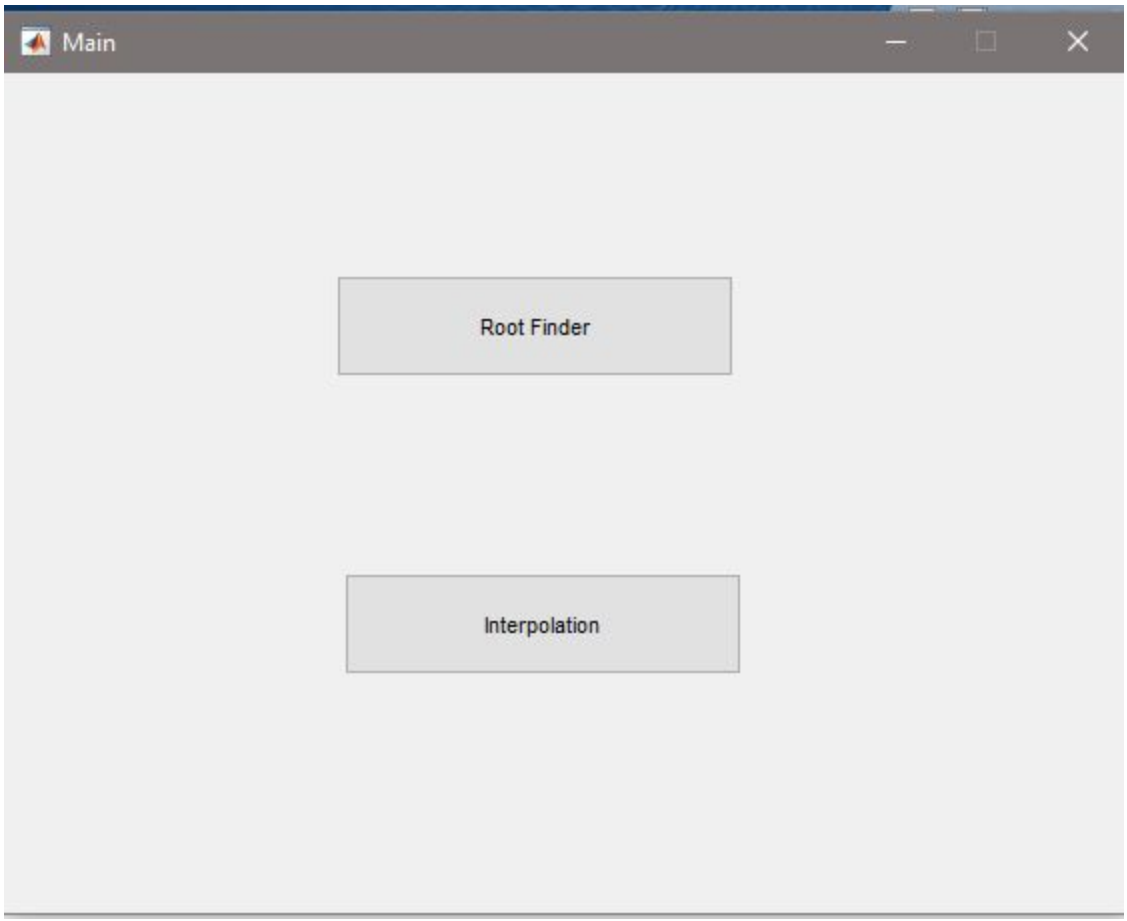
  $+ \frac{(x - x_0)(x - x_1)}{(x_2 - x_0)(x_2 - x_1)} f(x_2)$

Each of the three terms passes through one of the data points and zero at the other two. The summation of the three terms must, therefore, be unique second order polynomial $f_2(x)$ that passes exactly through three points.



GUI

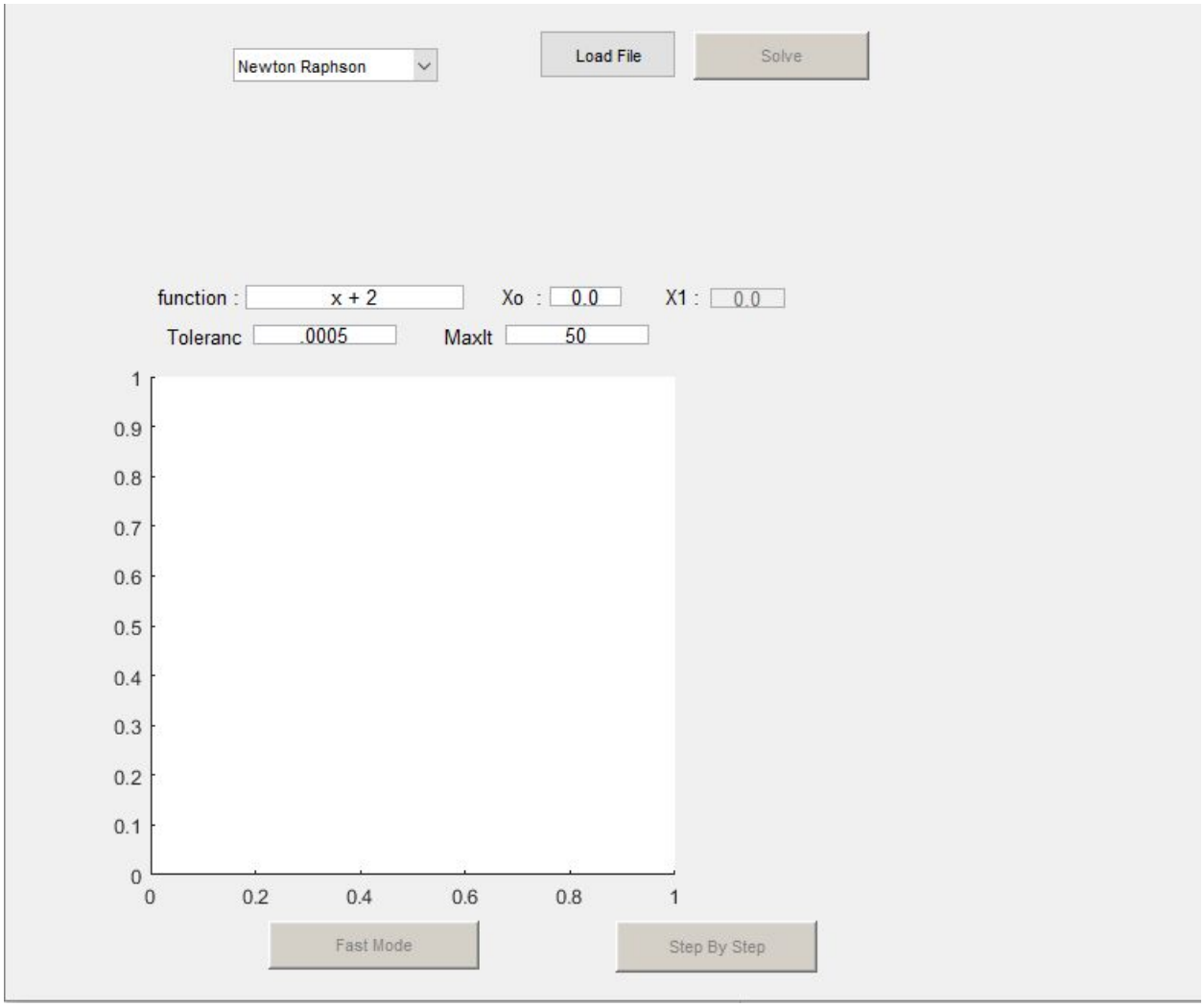
Main



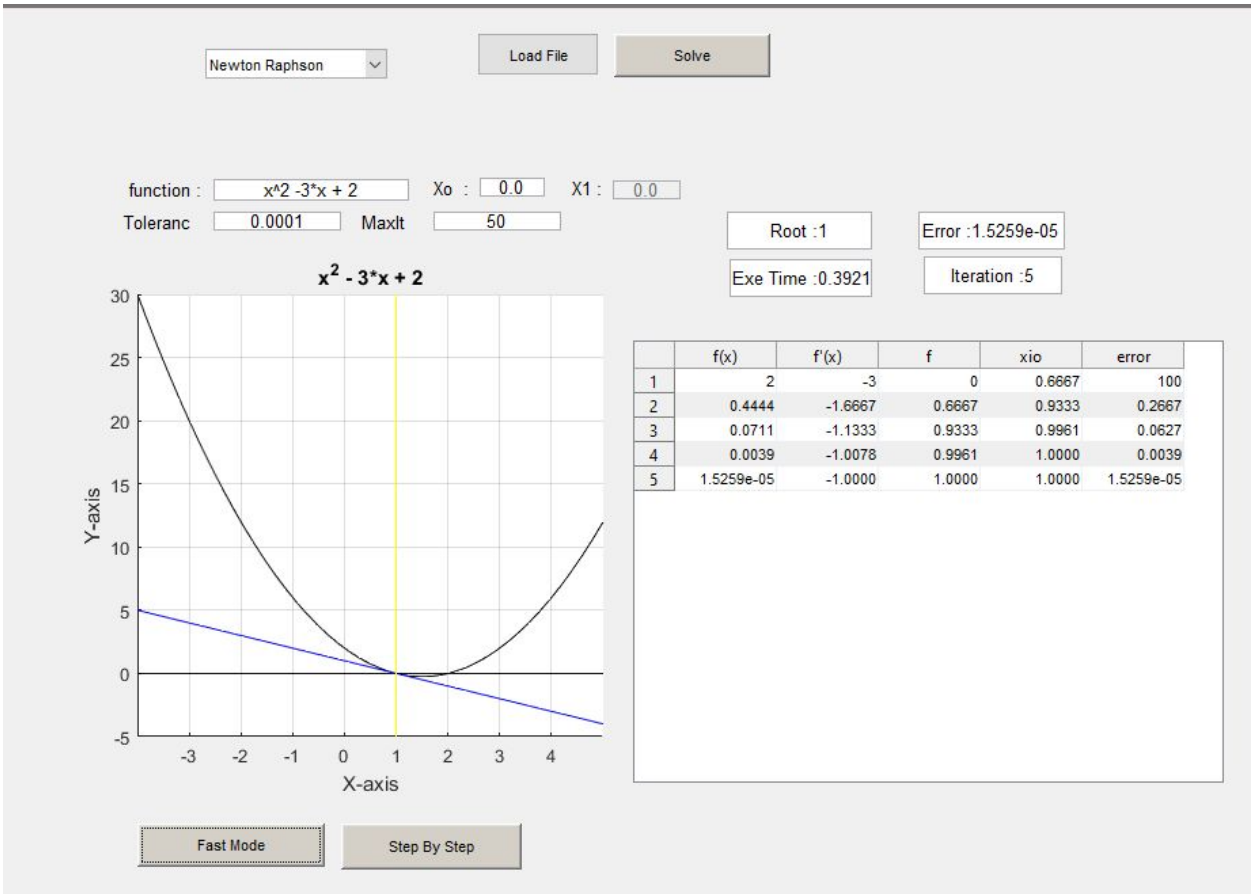
User Guide

- Buttons :
    - Root Finder
        - To open the Root Finder window.
    - Interpolation
        - To open the Interpolation window.

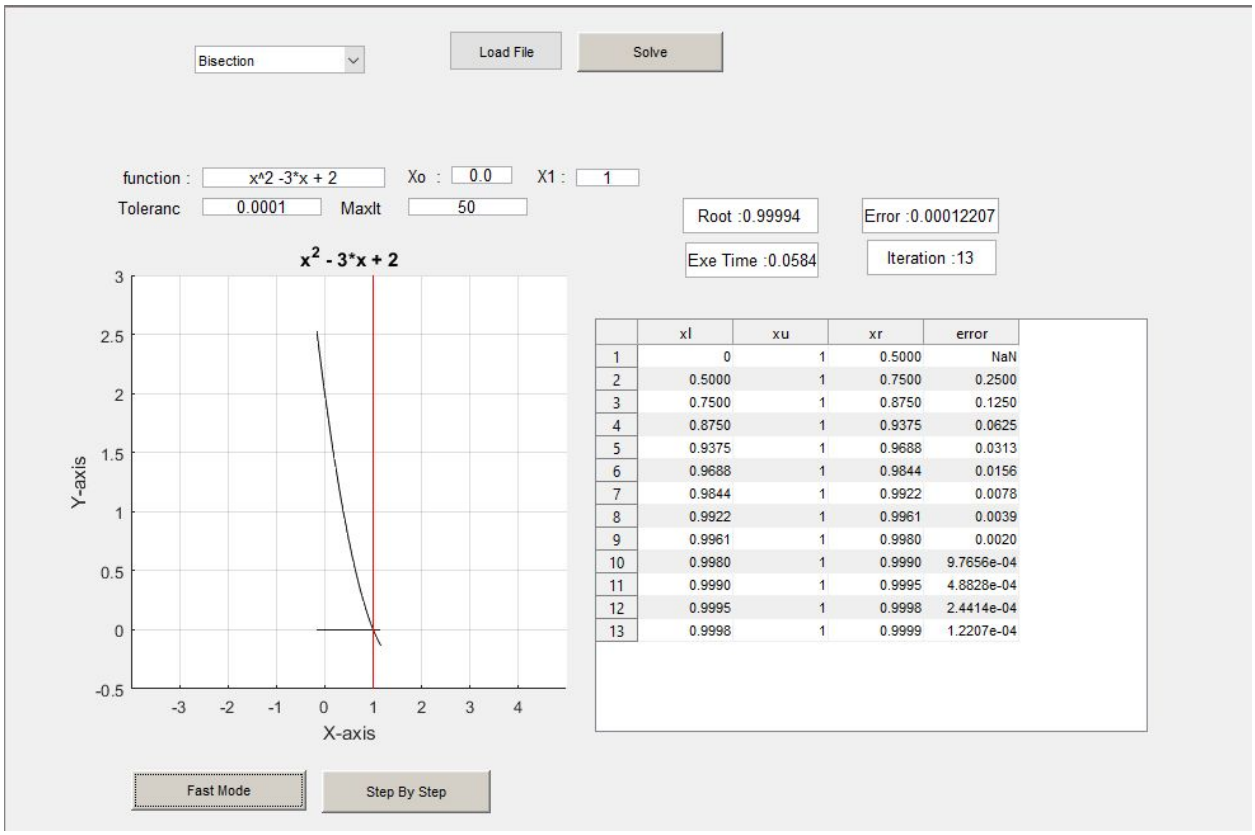
Root Finder


Newton Raphson
Load File
Solve
function :
x + 2
Xo :
0.0
X1 :
0.0
Toleranc
.0005
MaxIt
50
1
0.9
0.8
0.7
0.6
0.5
0.4
0.3
0.2
0.1
0
0
0.2
0.4
0.6
0.8
1
Fast Mode
Step By Step

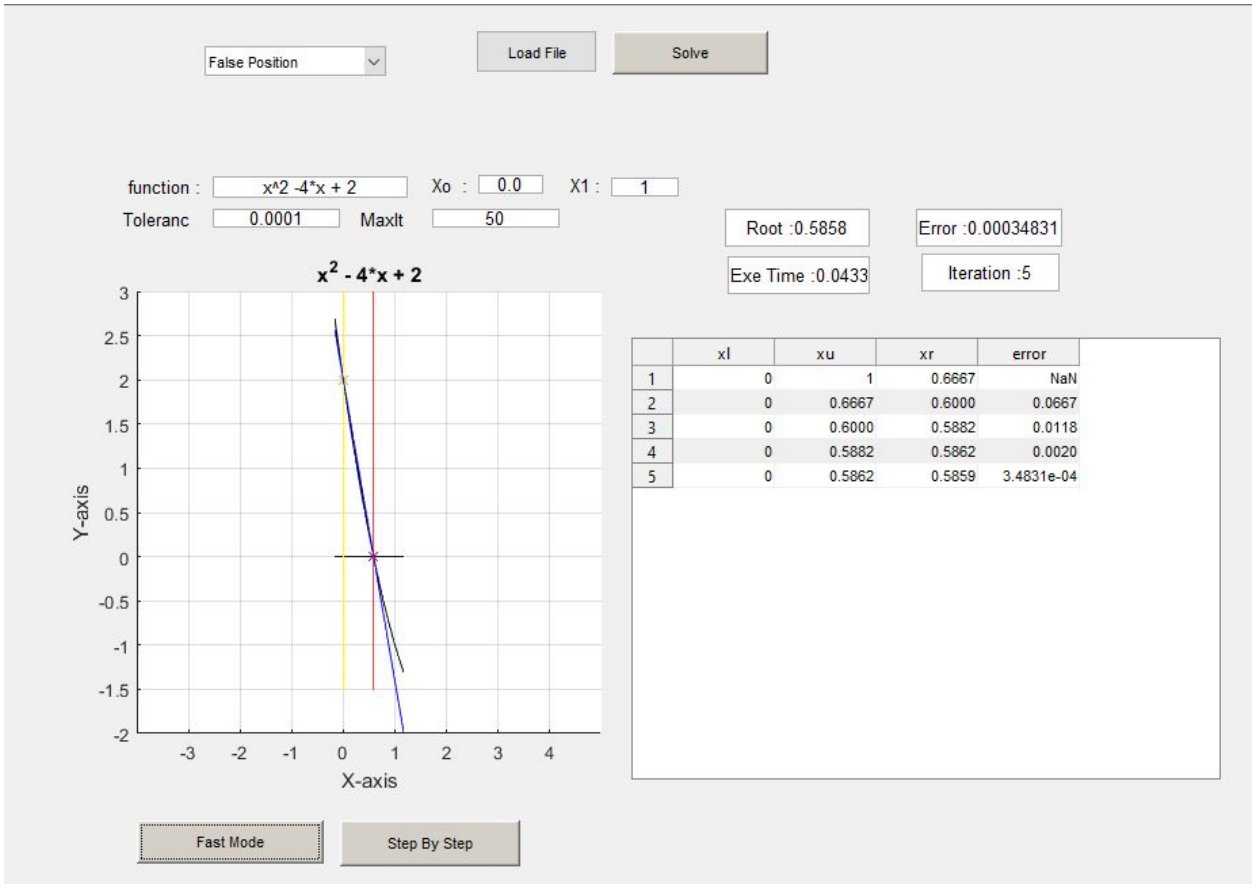
Sample runs



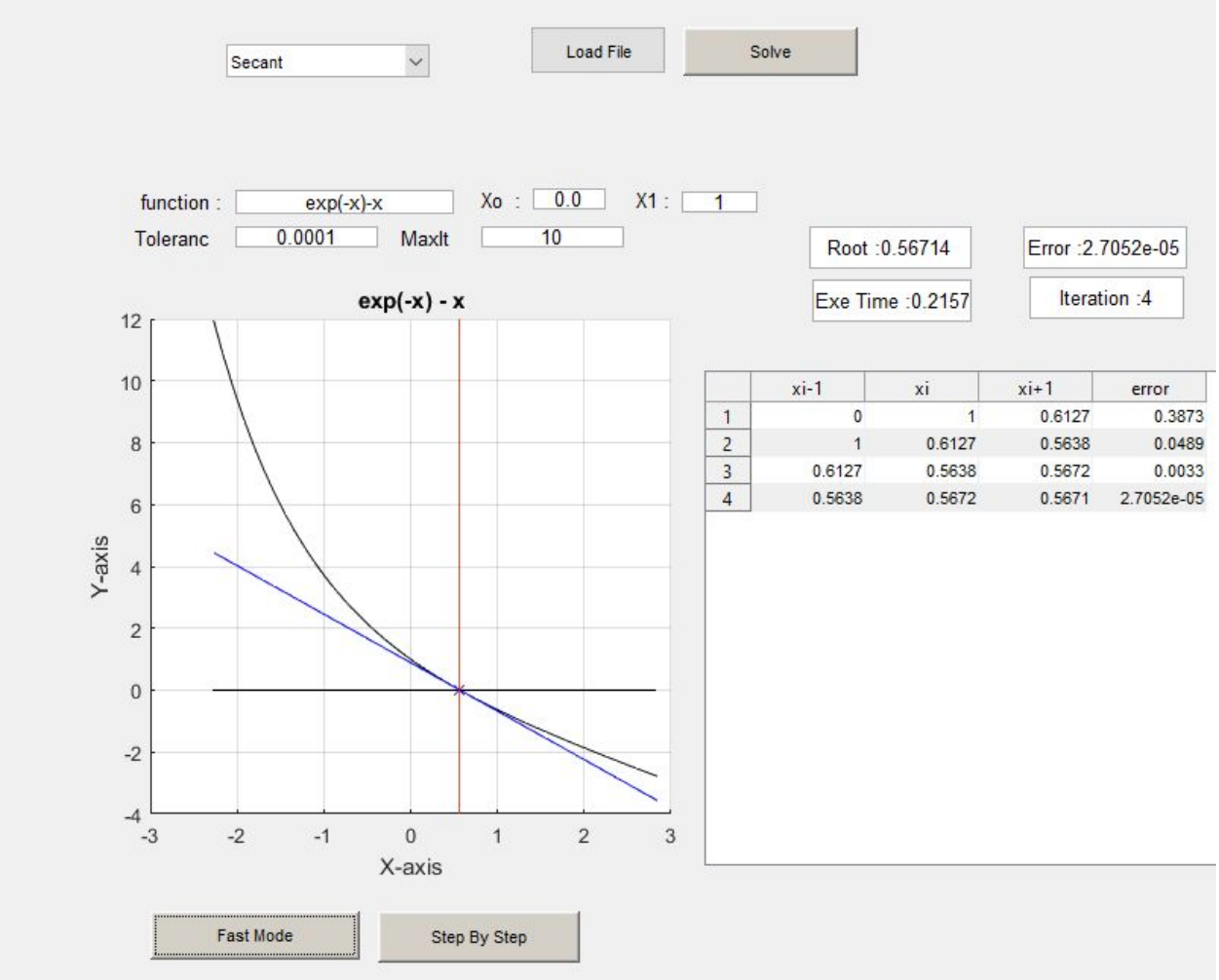
	f(x)	f'(x)	f	xio	error
1	2	-3	0	0.6667	100
2	0.4444	-1.6667	0.6667	0.9333	0.2667
3	0.0711	-1.1333	0.9333	0.9961	0.0627
4	0.0039	-1.0078	0.9961	1.0000	0.0039
5	1.5259e-05	-1.0000	1.0000	1.0000	1.5259e-05



	xl	xu	xr	error
1	0	1	0.5000	NaN
2	0.5000	1	0.7500	0.2500
3	0.7500	1	0.8750	0.1250
4	0.8750	1	0.9375	0.0625
5	0.9375	1	0.9688	0.0313
6	0.9688	1	0.9844	0.0156
7	0.9844	1	0.9922	0.0078
8	0.9922	1	0.9961	0.0039
9	0.9961	1	0.9980	0.0020
10	0.9980	1	0.9990	9.7656e-04
11	0.9990	1	0.9995	4.8828e-04
12	0.9995	1	0.9998	2.4414e-04
13	0.9998	1	0.9999	1.2207e-04


False Position
Load File
Solve
function :
x^2 -4*x + 2
Xo :
0.0
X1 :
1
Toleranc
0.0001
MaxIt
50
Root :0.5858
Error :0.00034831
Exe Time :0.0433
Iteration :5
x2 - 4*x + 2
Y-axis
X-axis
xl xu xr error
1 0 1 0.6667 NaN
2 0 0.6667 0.6000 0.0667
3 0 0.6000 0.5882 0.0118
4 0 0.5882 0.5862 0.0020
5 0 0.5862 0.5859 3.4831e-04
Fast Mode
Step By Step

	g(x)	Xr	error
1	0	1	NaN
2	1	0.3679	0.6321
3	0.3679	0.6922	0.3243
4	0.6922	0.5005	0.1917
5	0.5005	0.6062	0.1058
6	0.6062	0.5454	0.0608
7	0.5454	0.5796	0.0342
8	0.5796	0.5601	0.0195
9	0.5601	0.5711	0.0110
10	0.5711	0.5649	0.0063

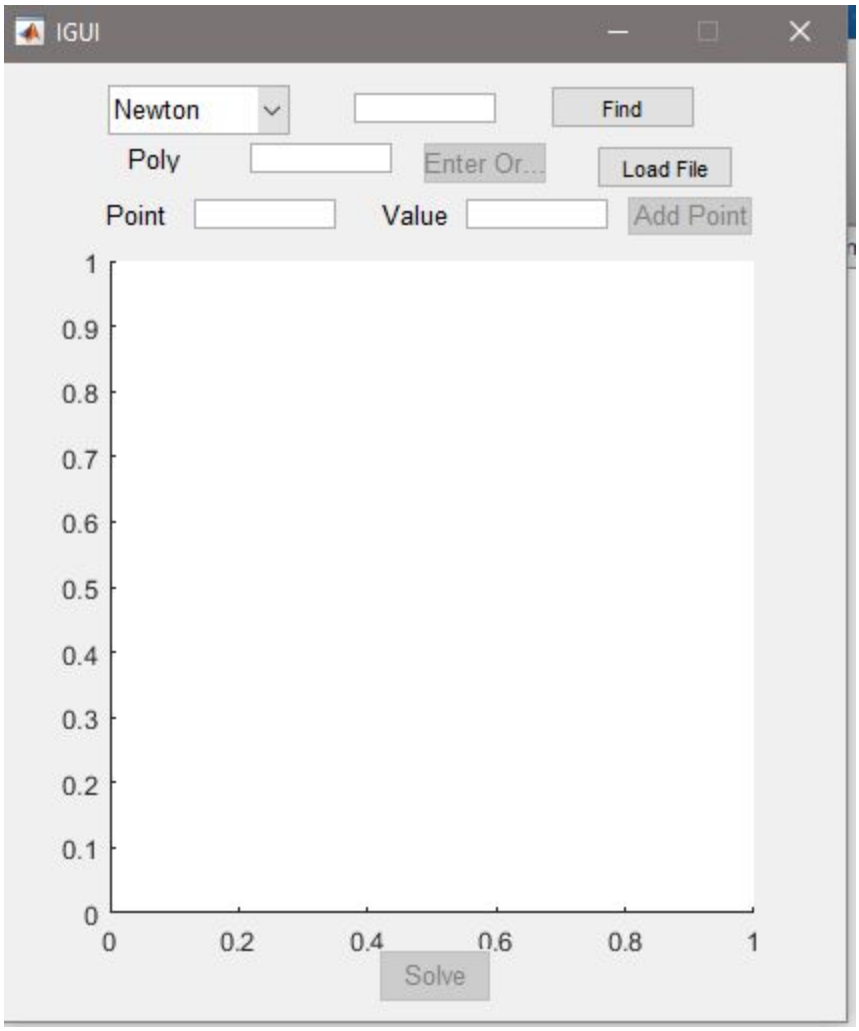


User Guide

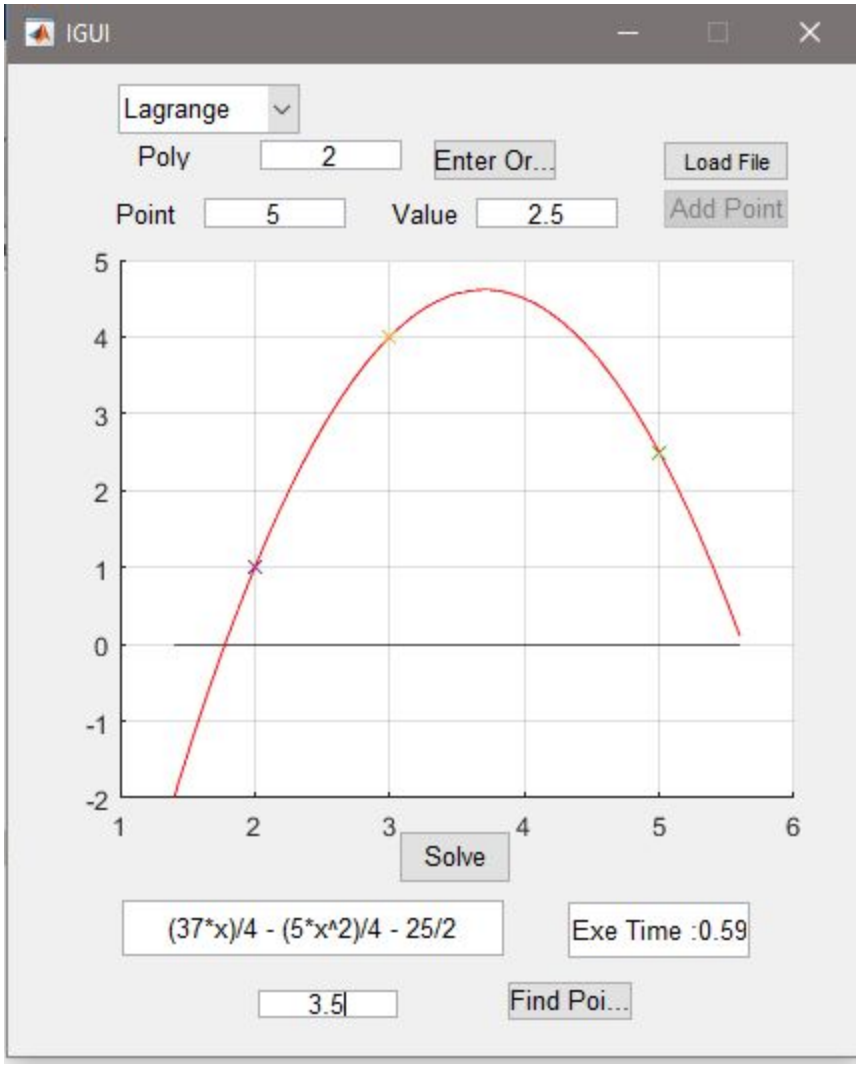
- Buttons :
  - Fast Mode & Step By Step :
    - To choose how do you want to display the results. Either all iterations at once or each iteration by choice.
  - Load File :
    - To load input data from existing (*.txt) file.
  - Solve :
    - Solve the given equation with the chosen method.
- Text areas:
  - Function :

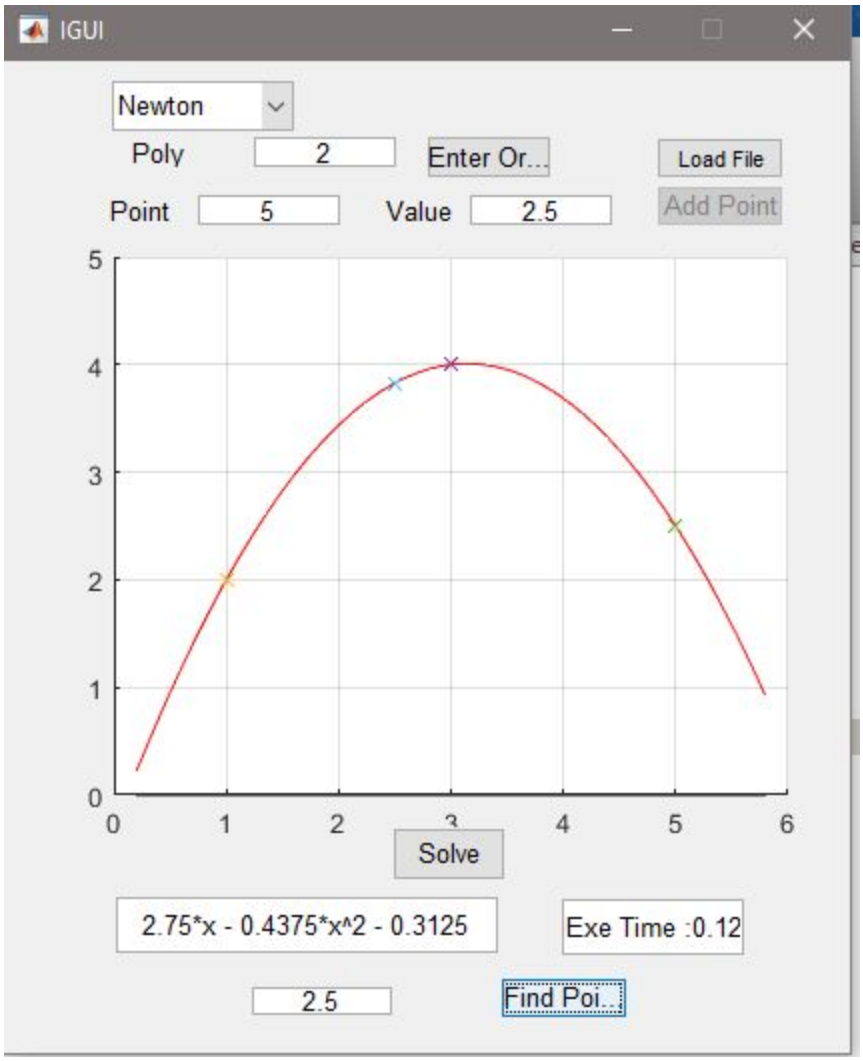
        - User inputs the desired function. The function must be compatible with matlab standards. By default “x+2”
    - X0 :
        - The first initial value.
    - X1 :
        - The second initial value. Only enabled in (Bisection, False Position, Secant, General Algorithm).
    - Tolerance :
        - To enter the tolerance of error.
        - We’re using absolute value for the error.
        - By default (.0001).
    - MaxIt :
        - To enter the maximum iterations.
        - By default (50).

Interpolation



Sample runs


IGUI
Lagrange
Poly
2
Enter Or...
Load File
Point
5
Value
2.5
Add Point
5
4
3
2
1
0
-1
-2
1
2
3
4
5
6
Solve
(37*x)/4 - (5*x^2)/4 - 25/2
Exe Time :0.59
3.5
Find Poi...



User Guide

- Buttons :
  - Solve :
    - To get the equation and plot it.
  - EnterOrder :
    - Enter the order of polynomial.
  - LoadFile :
    - Load input data from (*.txt) file.
  - Add Point :
    - Add Points needed to form the equation.

        - Number of equations must be equal to order+ 1.
- Text areas :
    - Poly :
        - Enter the order of polynomial.
    - Point :
        - Enter the x coordinate of a point.
    - Value :
        - Enter the Y coordinate of given X point in ‘Point’.
    - Un-named :
        - Enter X coordinate of a query point.