University *of* Alexandria

Faculty of Engineering

Computer and Systems Engineering Department

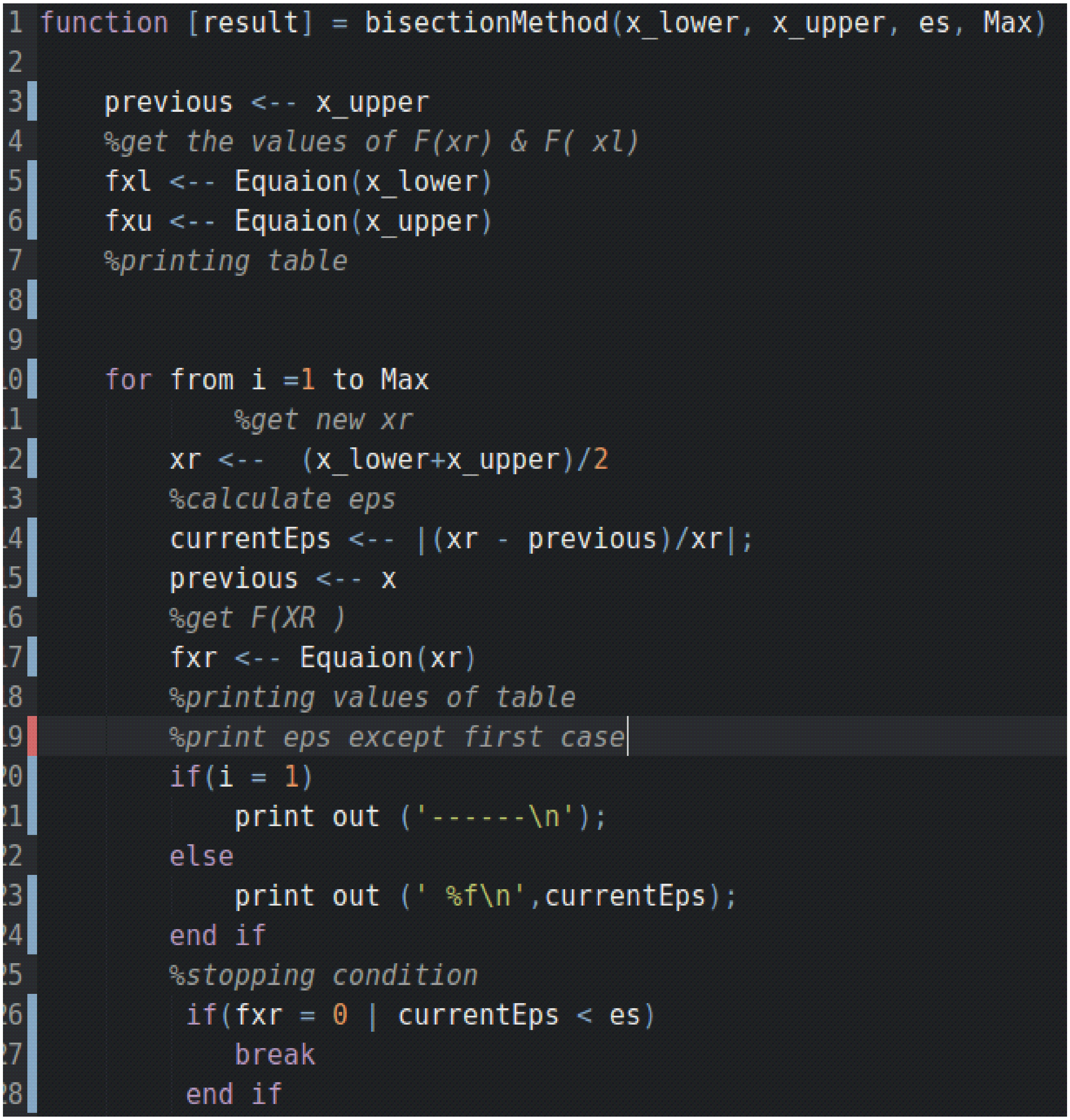
Numerical Analysis

Ahmed Helmi	06
Ahmed Tarek	08
Ahmed Eid	09
Shehab Kamal	32
Mark Philip	48

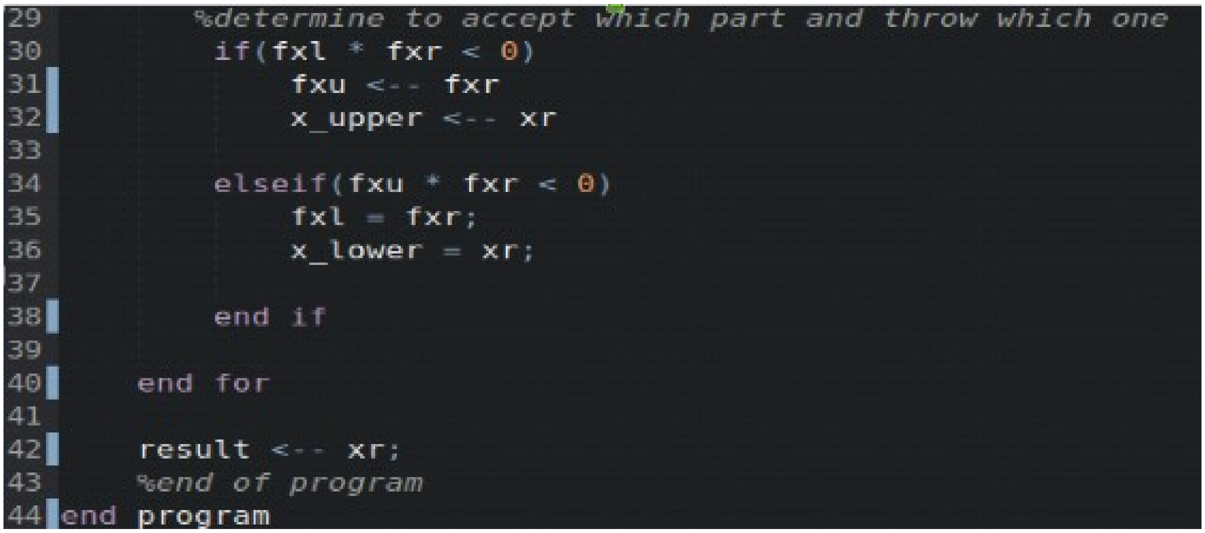
PART1

Bisection Method

pseudo-code :



```
function [result] = bisectionMethod(x_lower, x_upper, es, Max)

    previous <-- x_upper
    %get the values of F(xr) & F( xl)
    fxl <-- Equaion(x_lower)
    fxu <-- Equaion(x_upper)
    %printing table


    for from i =1 to Max
            %get new xr
        xr <--  (x_lower+x_upper)/2
        %calculate eps
        currentEps <-- |(xr - previous)/xr|;
        previous <-- x
        %get F(XR )
        fxr <-- Equaion(xr)
        %printing values of table
        %print eps except first case
        if(i = 1)
            print out ('------\n');
        else
            print out (' %f\n',currentEps);
        end if
        %stopping condition
         if(fxr = 0 | currentEps < es)
            break
         end if
```



```
       %determine to accept which part and throw which one
        if(fxl * fxr < 0)
            fxu <-- fxr
            x_upper <-- xr

        elseif(fxu * fxr < 0)
            fxl = fxr;
            x_lower = xr;

        end if

    end for

    result <-- xr;
    %end of program
end program
```

general algorithm:

1- Choose lower xl and upper xu guesses for the root such that the function changes sign over the interval . This can be checked by ensuring that f(xl)f(xu)< 0 .

2-estimate the root xr is determined by :

$$x_r = \frac{x_l + x_u}{2}$$

3-Make the following evaluations to determine in which subinterval the root lies :

a) if f(xl)f(xr) <0 , the root lies in the lowersubinterval . Therefore , set xu =xr and return to step 2 .

b) if f(xl)f(xr) >0 , the root lies in the upper subinterval . Therefore ,set xl =xr and return to step 2 .

c)if f(xl)f(xr)=0 , the root equals xr, terminate the computation .

4- calculate relative error estimate :

$$\text{Relative error estimate:} \quad \varepsilon = \frac{\left|x_r^{new} - x_r^{old}\right|}{\left|x_r^{new}\right|} 100\%$$

5- Termination criteria: epsilon < epsilon tol OR Max number of Iteration is reached .

The reason behind your decisions :

Pros	Cons
Easy	Slow
Always finds a root	Need to find initial guesses for x l and x u
Number of iterations required to attain an absolute error can be computed a priori.	No account is taken of the fact that if f(x l) is closer to zero, it is likely that root is closer to x l .

<u>Data structure used :</u>

- No special data structure used to calculate the root .

<u>Analysis:</u>

- The bisection method in mathematics is a root finding method that repeatedly bisects an interval and then selects a sub interval in which a root must lie for further processing . It is very simple method but it is also relatively slow . Because of this , it is often used to obtain a rough approximation to a solution which is then used as a starting point for more rapidly converging methods . The method is also called binary search method .

- The method is guaranteed to converge to a root of f if f is continuous function on interval [a,b] .

- The absolute error is halved at each step so the method converges linearly which considered slow .

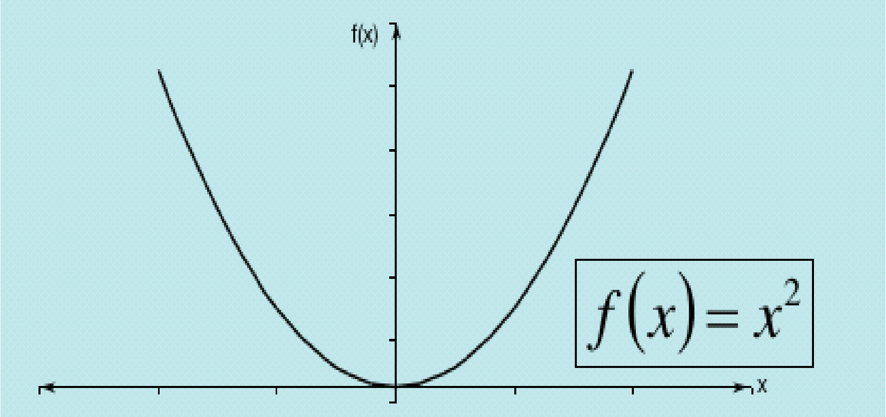
- there is at least one zero crossing within the interval .

-If initial guesses is correct the method will converge And number of iterations

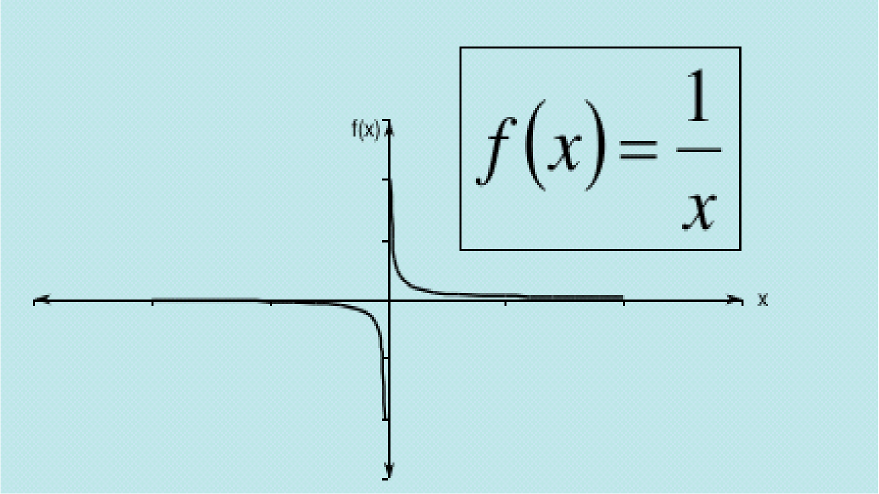
$$k \geq \log_2\left(\left|\frac{L_0}{x_l * \varepsilon_{es}}\right|\right)$$

Drawbacks :

- 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.



- Function changes sign but root does not exist .

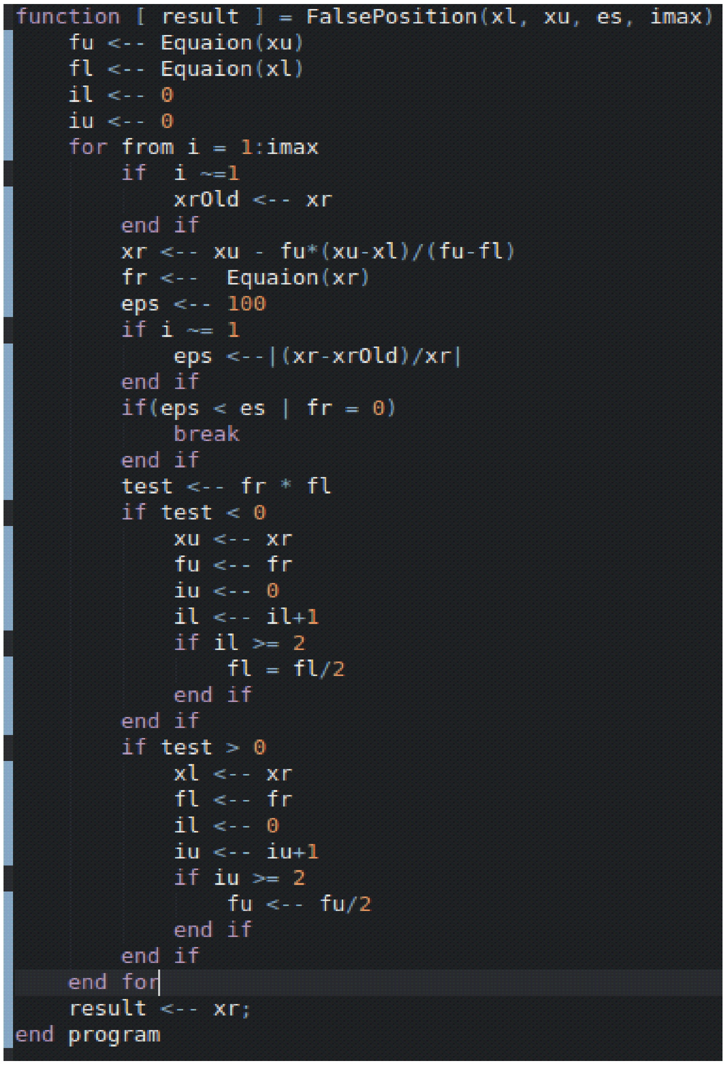


Error Analysis:

- an initial bound on the problem [*a*, b], then the maximum error of using either *a* or *b* as our approximation is *h* = *b* − *a*. Because we halve the width of the interval with each iteration, the error is reduced by a factor of 2, and thus, the error after *n* iterations will be *h*/2*n*.

The False-Position Method (Regula-Falsi)

pseudo-code :



```
function [ result ] = FalsePosition(xl, xu, es, imax)
    fu <-- Equaion(xu)
    fl <-- Equaion(xl)
    il <-- 0
    iu <-- 0
    for from i = 1:imax
        if  i ~=1
            xrOld <-- xr
        end if
        xr <-- xu - fu*(xu-xl)/(fu-fl)
        fr <--  Equaion(xr)
        eps <-- 100
        if i ~= 1
            eps <--|(xr-xrOld)/xr|
        end if
        if(eps < es | fr = 0)
            break
        end if
        test <-- fr * fl
        if test < 0
            xu <-- xr
            fu <-- fr
            iu <-- 0
            il <-- il+1
            if il >= 2
                fl = fl/2
            end if
        end if
        if test > 0
            xl <-- xr
            fl <-- fr
            il <-- 0
            iu <-- iu+1
            if iu >= 2
                fu <-- fu/2
            end if
        end if
    end for
    result <-- xr;
end program
```

general algorithm:

1- Fina a pair of values of x ,xl and xu such that f1 =f(x1)<0 and fu= f(xu)> 0 .

2-Estimate the value of the root from the following formula :

$$x_r = \frac{x_l f_u - x_u f_l}{f_u - f_l}$$

3- use the new point to replace one of the original points ,keeping the two points on opposite side of the X axis .

a) if f(xr)<0 then xl=xr.

b) if f(xr)>0 then xu=xr .

- If f(x r)=0 then you have found the root and need go no further!

4 -see if the new xl and xu are close enough for convergence to be declared . If they are not go back to step 2 .

The reason behind your decisions :

-Faster than bisection method .

-Always converges for a single root.

Data structure used :

- No special data structure used to calculate the root .

Analysis:

- One can try for a better convergence-rate, at the risk of a worse one, or none at all.

- Most numerical equation-solving methods usually converge faster than Bisection. The price for that is that some of them (e.g. Newton’s method and Secant) can fail to converge at all, and all of them can sometimes converge very much more slowly than Bisection—sometimes prohibitively slowly. None can guarantee Bisection’s reliable and steady guaranteed convergence rate. Regula Falsi, like Bisection, always converges, usually considerably faster than Bisection--but sometimes very much more slowly than Bisection.

- The Effect of Non-linear Functions :

If we cannot assume that a function may be interpolated by a linear function, then applying the false-position method can result in worse results than the bisection method.

Error Analysis

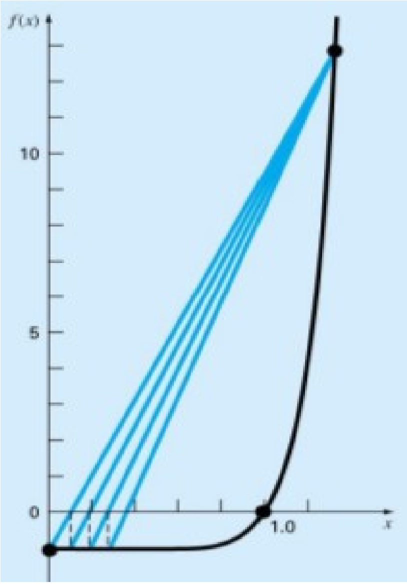
The error analysis for the false-position method is not as easy as it is for the bisection method, however, if one of the end points becomes fixed, it can be shown that it is still an O(h) operation, that is, it is the same rate as the bisection method, usually faster, but possibly slower. For differentiable functions, the closer the fixed end point is to the actual root, the faster the convergence.

Drawbacks :

- One way to mitigate the “one-sided” nature of the false position

Solution :

have the algorithm detect when one of the bounds is stuck. And when that happen the original formula $x_r = (x_l + x_u)/2$ can be used .

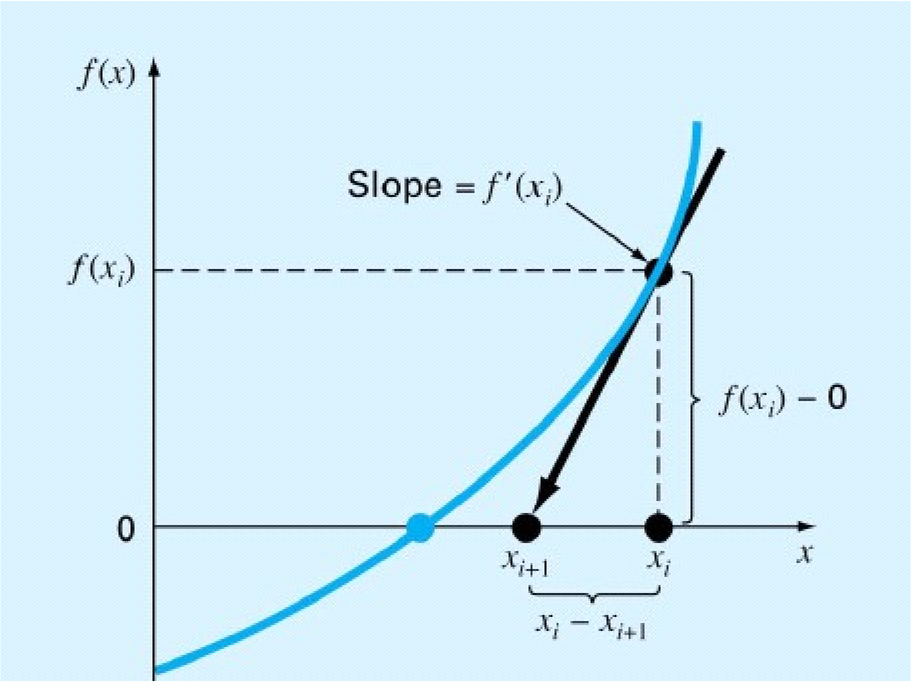


NEWTON-RAPHSON Method

-pseudoCode:

```
Newton( equation , max , es , x0 )
x(1)<-- x0

dfX <-- diff(equation)

ea <-- 0

i <-- 2

while i is less than (max)+1

    if(dfX(x(i-1)) is equal 0)

        set divByZero equal 1

        endWhile

    end if

x(i) <-- x(i-1)-(equation(x(i-1))/dfX(x(i-1)))

    ea <-- abs((x(i-1)-x(i)))

    errors(i) <-- ea

    if(ea < es)

        break;
    end if

    increment I

    numberOfItrations <-- i – 1

end while
```

2-general algorithm:



steps of algorithm :

1-evaluate F ' (x) symbolically.
2- Use an initial guess of the root x_i to estimate the new value of the root X_{i+1} .

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

3 - check the error by using this equation :

$$|\in_a| = \left|\frac{x_{i+1} - x_i}{x_{i+1}}\right| \times 100$$

4-terminates when $e_a <= e_s$ or max iteration is reached

The reason behind your decisions :

this method converges quadratically when it converges.so it's faster than many other methods.

Data structure used :

diff() method : to get F ' (x) as we use it to calculate F(x_{i+1})

vectors : as the X $_i$ is saved in a vector to be printed after each iteration also error is saved in a vector.

- why we use it ?

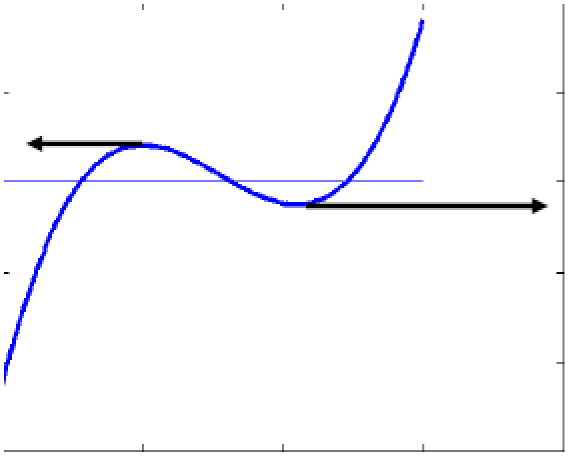
- Random access – constant *O(1)*
- Insertion or removal of elements at the end - amortized constant *O(1).*
- Insertion or removal of elements - linear in distance to the end of the vector *O(n).*

Analysis:

this is a very fast method to find roots of any equation but we can't use it in for all functions because it have some restrictions the most important one is that it doesn't converge in all cases .

Drawbacks :

1-Division by zero

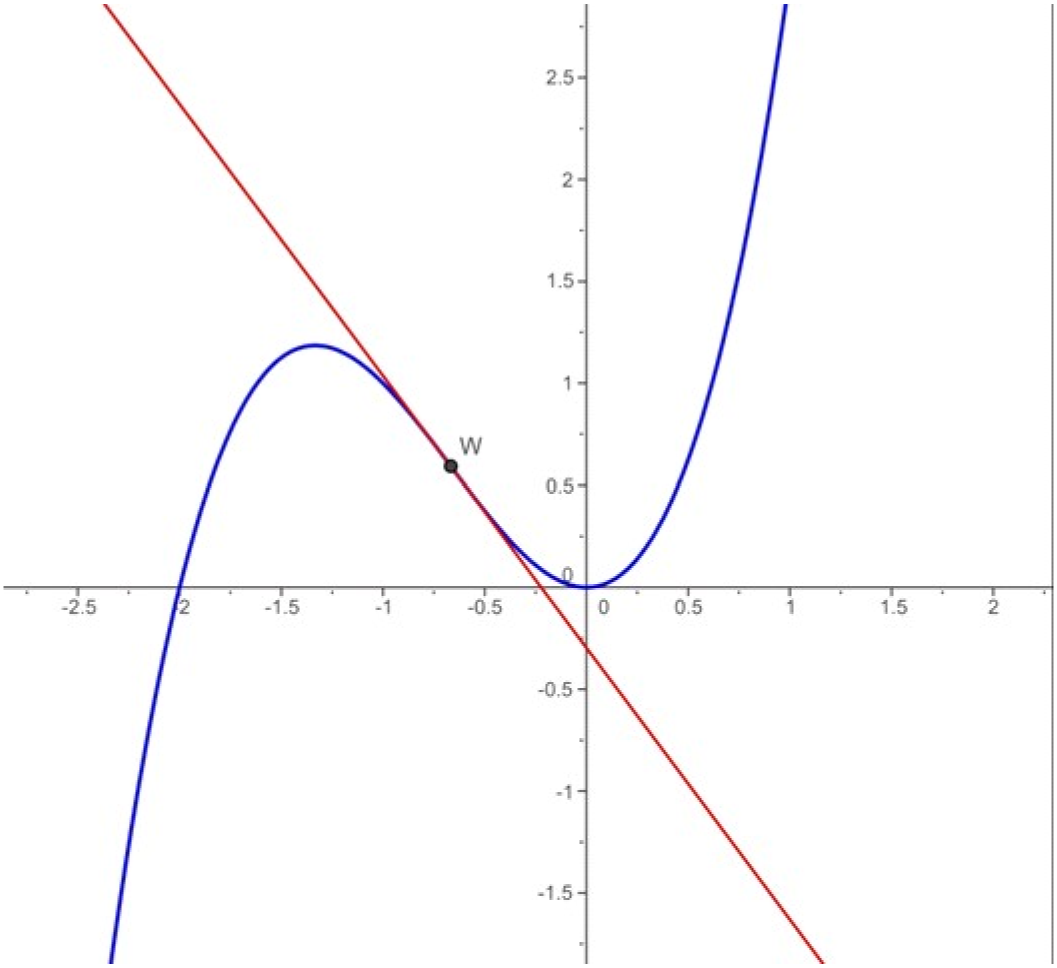


$$x_{i+1} = x_i - \frac{f(x_i)}{f'(x_i)}$$

if F ' (x_i) is equal to zero this leads to division by zero the result equal infinity which will result in strongly wrong answer

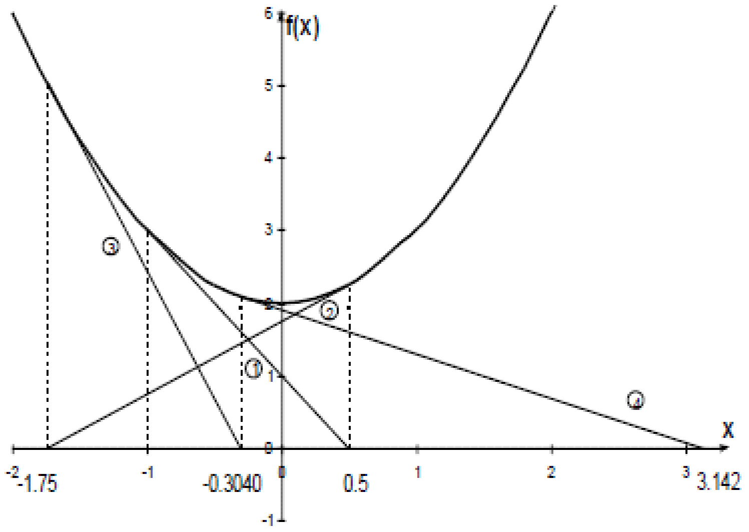
2- Divergence at inflection points

Selection of the initial guess or an iteration value of the root that is close to the inflection point of the function f(x) may start diverging away from the root in the Newton-Raphson method.



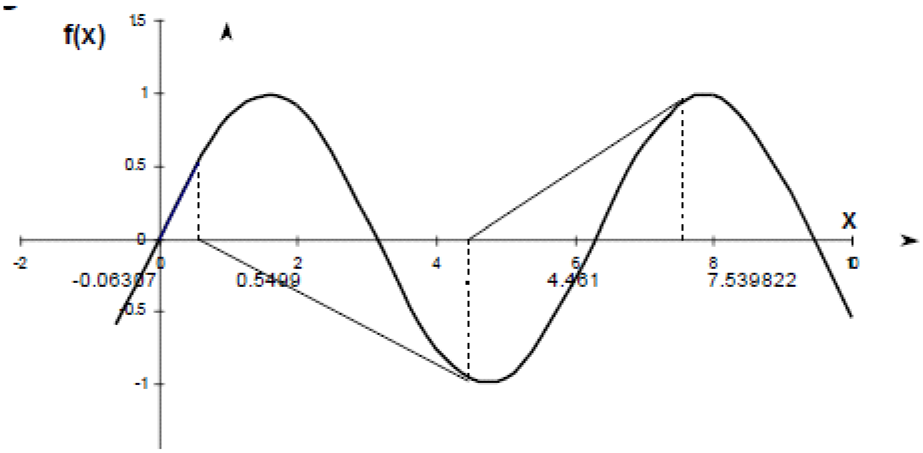
3 Oscillations near local maximum and minimum

choosing a point near a local minimum or a local maximum the computer can consider it a root this lead to try to converge towards a virtual root.



4. **Root Jumping :**

using a point near some root may makes you to jump to anther root away from the initial guess.



Error Analysis:

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

$$\Rightarrow f(x_i) = f'(x_i)(x_i - x_{i+1})$$

$$\Rightarrow f(x_i) + f'(x_i)(-x_i) = f'(x_i)(-x_{i+1})$$

$$\Rightarrow f(x_i) + f'(x_i)(\alpha - x_i) = f'(x_i)(\alpha - x_{i+1})$$

Suppose α is the true value (i.e., $f(\alpha) = 0$).
Using Taylor's series

$$f(\alpha) = f(x_i) + f'(x_i)(\alpha - x_i) + \frac{f''(c)}{2}(\alpha - x_i)^2$$

$$\Rightarrow 0 = f(x_i) + f'(x_i)(\alpha - x_i) + \frac{f''(c)}{2}(\alpha - x_i)^2$$

$$\Rightarrow 0 = f'(x_i)(\alpha - x_{i+1}) + \frac{f''(c)}{2}(\alpha - x_i)^2 \qquad \text{(from(3))}$$

$$\Rightarrow 0 = f'(x_i)(\delta_{i+1}) + \frac{f''(c)}{2}(\delta_i)^2 \qquad \text{(from(1) and (2))}$$

$$\Rightarrow \delta_{i+1} = \frac{-f''(c)}{2f'(x_i)}\delta_i^2 \cong \frac{-f''(\alpha)}{2f'(\alpha)}\delta_i^2$$

When x_i and α are very close to each other, c is between x_i and α.

The iterative process is said to be of second order.

this means that each error is nearly equal to square of the previous error

$$(E_i) = O(E_{i-1}^2).$$

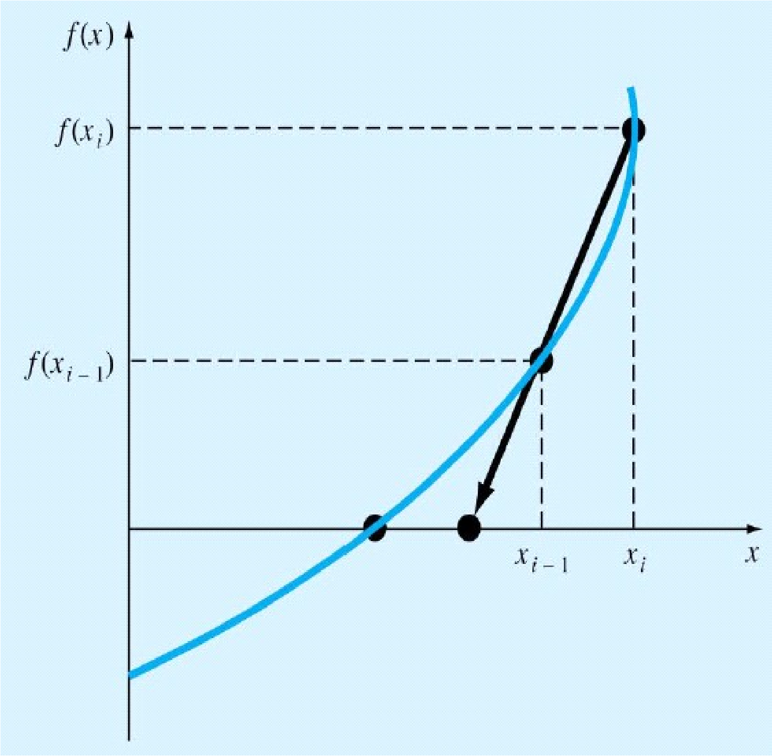
SECANT METHOD

1-pseudoCode:

```
Secant( equation , max , es , x0 , x1 )

x(1) <-- x0
x(2) <-- x1
i <-- 3
while i is less than (max)+1
    if equation(x(i-2))-equation(x(i-1))) is equal zero
        diffIsNearZero <-- 1
        end While
    end if

    x(i) <-- x(i-1)-(equation(x(i-1))*(x(i-2)-x(i-1)))/(equation(x(i-2))-equation(x(i-1)))
    ea <-- abs((x(i-1)-x(i)))
    errors(i)<-- ea

    if(ea < es)
        break
    end if

    i <-- i+1
    numberofItrations = i - 2
end while
```

General algorithm:

this algorithm is simillar to false position method but the only difference is that in this method we just evaluate the next point using the previous 2 points without make check that one of them is lower x-axis and the other is upper x-axis.



1-choose two intial proper guesses X_{-1} & X_0

2- substitute in this equation to get X_{i+1}

$$x_{i+1} = x_i - \frac{f(x_i)(x_i - x_{i-1})}{f(x_i) - f(x_{i-1})}$$

3-calculate the absolute relative error from this equation.

$$|\in_a| = \left|\frac{x_1 - x_0}{x_1}\right| \times 100$$

4- check if E_a <= E_s or the Max iteration is reached then terminates

The reason behind your decisions :

-this is a fast method for finding the roots which needs no many calculations.
-it converges quadrically when it converges.
-Requires two guesses that do not need to bracket the root.

Data structure used :

vectors : as the X $_i$ is saved in a vector to be printed after each iteration also error is saved in a vector.

- why we use it ?

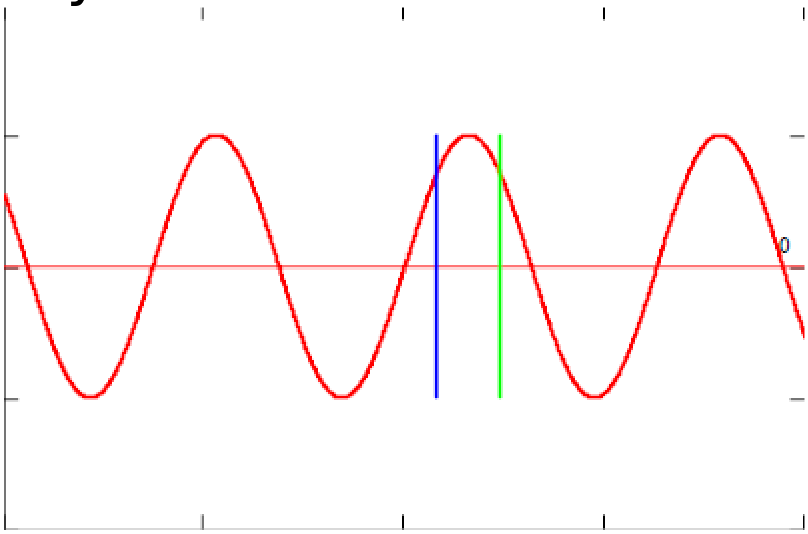
- Random access – constant *O(1)*
- Insertion or removal of elements at the end - amortized constant *O(1).*
- Insertion or removal of elements - linear in distance to the end of the vector *O(n).*

Analysis:

it converges quadrically when it converges and it doesn't need to make check for lower and upper bounds . this decreases the calculations used which decreases the time used . but in case of multiple roots in converges linearly.

Drawbacks :

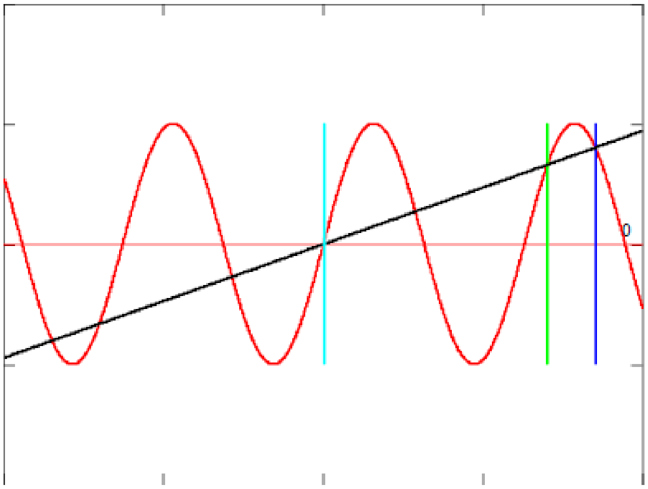
1-Division by zero



$$x_{i+1} = x_i - \frac{f(x_i)(x_i - x_{i-1})}{f(x_i) - f(x_{i-1})}$$

- if f (X_i) - f(X_{i-1}) is near to zero this leads to division by zero . which result in a stronglywrong answer.

2. **Root Jumping :**



some times we take an intial guess to get a root near this guess but sometimes what happens is that this function converges to another root.

Fixed Point

Pseudo Code:

```
function [x,ea,runningTime] = fixed_point( fn , x , es, iter_max)
fn ß string_concatenate(fn,' + x')
x ß eval(fn)
iter ß 0
x_old ß 0
x_old ß x
x ß evaluate(fn)
if (x ~= 0)
    ea ß absolute_value((x - x_old) / x) * 100;
end
iter ß iter+1

while (ea > es && iter < iter_max)
    x_old ß x
    x ß evaluate(fn)
    if (x != 0)
        ea ß absolute_value((x - x_old) / x) * 100
    end
    iter ß iter+1;
end
end
```

Data-Structures Used:

- 2-D Array for (x, f(x))

General Idea about Fixed-Point:

- We reformulate f(x) = 0, to get g(x) (x on one side of the equation)
- We then successively calculate $X_{i+1} = g(x_i)$ until X_{i+1} converges to x

General Algorithm:

- The fn takes the entered function by the user as an input, the value of x, the value of e_s and maximum number of iterations.
- To prepare G(x) we concatenate ‘x’ to the string function
- After that we calculate the first value for ‘x’
- Then, we enter the loop to apply the successive substitution theme of fixed-point method.

Analysis of behavior:

As known before, the function convergence or divergence depends on the prepared G(x)

Suppose we have the following f(x) = x^2 – 2x - 3 = 0, we can hold G(x) in many ways like this:

Case a:

$$\begin{aligned}&x^2 - 2x - 3 = 0\\ &\Rightarrow x^2 = 2x + 3\\ &\Rightarrow x = \sqrt{2x+3}\\ &\Rightarrow g(x) = \sqrt{2x+3}\end{aligned}$$

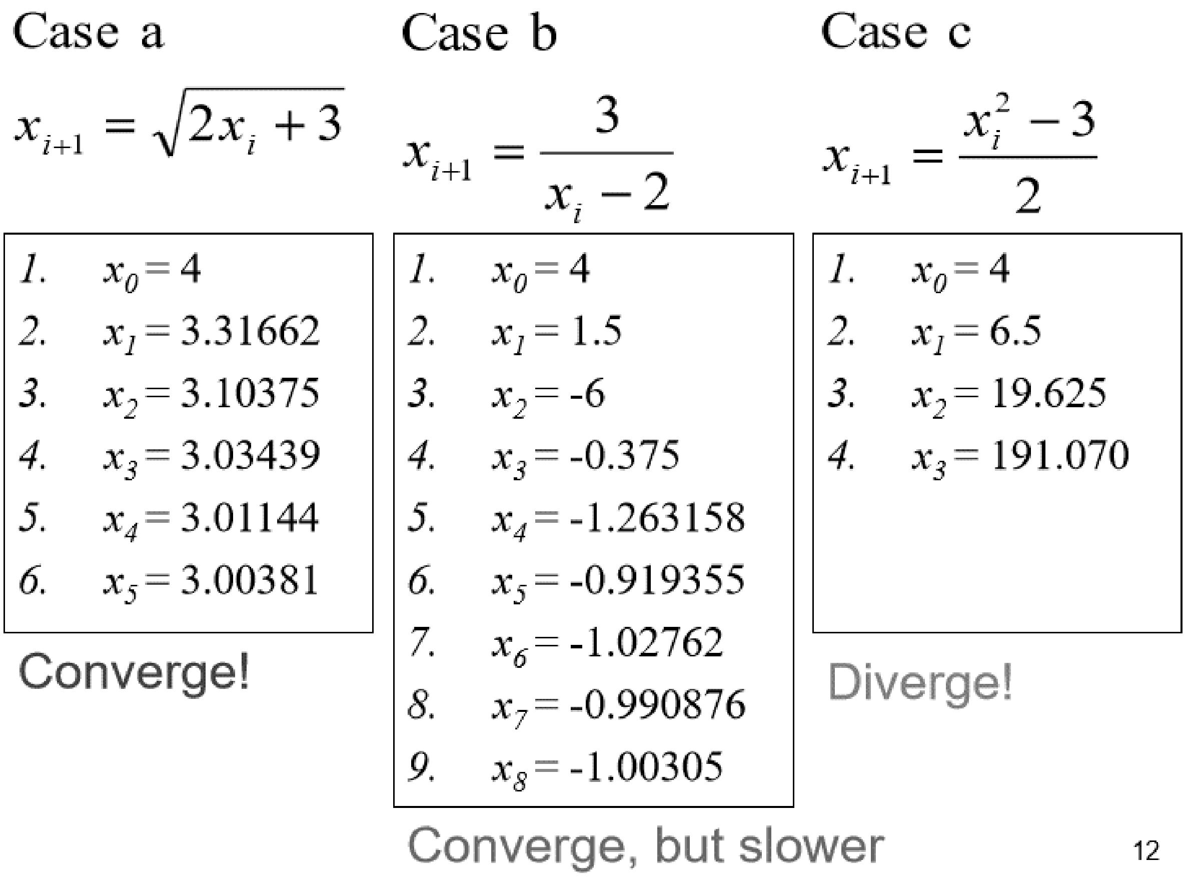
Case b:

$$\begin{aligned}&x^2 - 2x - 3 = 0\\ &\Rightarrow x(x-2) - 3 = 0\\ &\Rightarrow x = \frac{3}{x-2}\\ &\Rightarrow g(x) = \frac{3}{x-2}\end{aligned}$$

Case c:

$$\begin{aligned}&x^2 - 2x - 3 = 0\\ &\Rightarrow 2x = x^2 - 3\\ &\Rightarrow x = \frac{x^2-3}{2}\\ &\Rightarrow g(x) = \frac{x^2-3}{2}\end{aligned}$$

The results are as following:



Case a

$$x_{i+1} = \sqrt{2x_i + 3}$$

1. $x_0 = 4$
2. $x_1 = 3.31662$
3. $x_2 = 3.10375$
4. $x_3 = 3.03439$
5. $x_4 = 3.01144$
6. $x_5 = 3.00381$

Converge!

Case b

$$x_{i+1} = \frac{3}{x_i - 2}$$

1. $x_0 = 4$
2. $x_1 = 1.5$
3. $x_2 = -6$
4. $x_3 = -0.375$
5. $x_4 = -1.263158$
6. $x_5 = -0.919355$
7. $x_6 = -1.02762$
8. $x_7 = -0.990876$
9. $x_8 = -1.00305$

Converge, but slower

Case c

$$x_{i+1} = \frac{x_i^2 - 3}{2}$$

1. $x_0 = 4$
2. $x_1 = 6.5$
3. $x_2 = 19.625$
4. $x_3 = 191.070$

Diverge!

12

In the first case: the prepared G(x) is proper enough for x to converge.

In the second G(x), fixed-point converged, but it was slower than the first one.

In the third case: the method diverged originally, cause G(x) was not proper for it.

Convergence Rate Analysis :

- If |g’(x)| < 1, the error decreases and the function converges.
- If |g’(x)| < 1, the error decreases and the function converges.
- If derivatives is Positive, the iteration solution will be monotonic.
- If the derivative is negative, the errors will oscillate.

Solutions:

- The fixed-point iteration method diverges when
- |g’(x)| <1,
-

so to handle it, we must choose the appropriate G(x) that make the function converge (|g’(x)| >

Birge vieta

pseudo-code :

```
function[result] = birage_veta(x0, es, numOfItrations, polynomial)

Coff = sym2poly(sym(polynomial));  // Make “coff” matrix as the coefficients of
the polynomial.
matrix = zeros(length(coff), 3); // Make “matrix” matrix of zeros with number of
rows of “coff” length and 3 columns.
matrix(:, 1) =  coff;   // Equal first column of matrix with the coefficients.

e = 100;
matrix(1,2) = matrix(1,1);
matrix(1,3) = matrix(1,1);
numOfItr  =  0;
while abs(e) >= es
      for I = 2 ▭ length(coff)
              matrix(i,2) = matrix(i,1) + x0 * matrix(i-1,2);
              matrix(i,3) = matrix(i,2) + x0 * matrix(i-1,3);
```

```
          end
         element = matrix(length(coff),2) / matrix(length(coff) - 1,3);
          x1 = x0 - element;
          e =  (x1 - x0) / x1;
          x0 =  x1;
         numOfItr =  numOfItr + 1;
         if numOfItr = numOfItrations
                 break;
         end
      end
      result = x1;
      end
```

General algorithm:

1- Take the polynomial, the initial ‘x’, the required ‘epsilon’ and the max number of iterations from the user.

2- Form a vector from the coefficients of the polynomial and insert it in the
First column of the matrix.

3- Move the first coefficient of the polynomial –the coefficient of the highest degree” to the first row of the whole matrix.

$$Coff_1 = B_1 = C_1$$

4- iterate over the matrix to form the second and the third column from the first column and the initial ‘x’ from the equation

$$B_i = coff_i + x\_estimate * B_{i-1}$$
$$C_i = B_i + x0 * C_{i-1}$$

5- calculate the next estimate from the equation

$$x\_estimate = B_n / C_{n-1}$$

where ‘n’ is the number of coefficients or the degree of the polynomial + 1.

6- calculate the estimate error from the equation

Epsilon = (x_newEstimate – x_oldEstimate) / x_newEstimate

7- repeat the previous three steps until it reaches the required epsilon or the max number of iterations

Data structure used :

- 2-D matrix for the calculation.
- A vector for the coefficients of the polynomial.

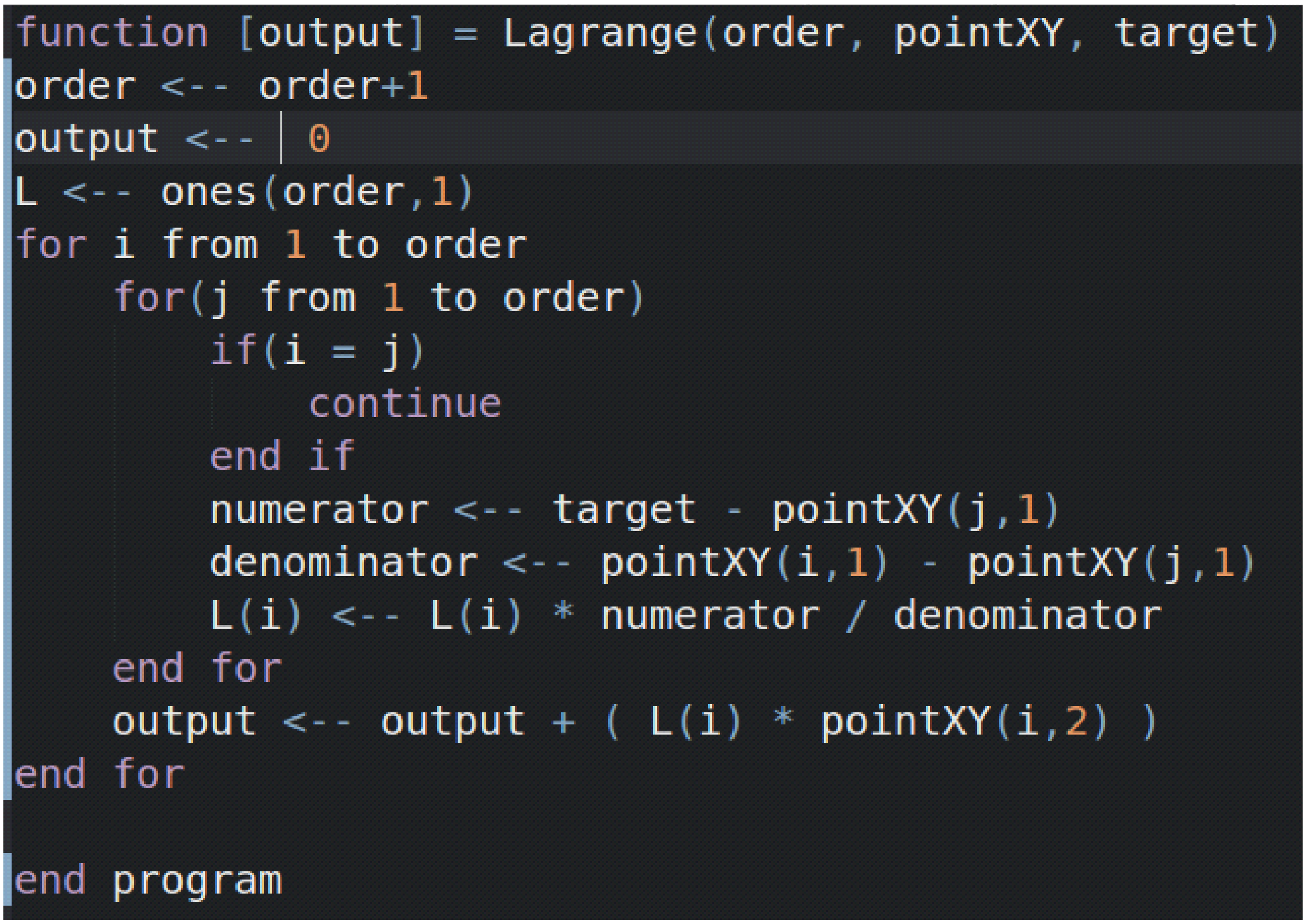
Drawbacks :

- It can works only on polynomials. It won’t work on any other type of equations like sinusoidal or exponential.

PART 2

Lagrange interpolation

pseudo-code :



```
function [output] = Lagrange(order, pointXY, target)
order <-- order+1
output <-- 0
L <-- ones(order,1)
for i from 1 to order
    for(j from 1 to order)
        if(i = j)
            continue
        end if
        numerator <-- target - pointXY(j,1)
        denominator <-- pointXY(i,1) - pointXY(j,1)
        L(i) <-- L(i) * numerator / denominator
    end for
    output <-- output + ( L(i) * pointXY(i,2) )
end for

end program
```

General algorithm:

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

Given a set of k + 1 data points .
(X_0,Y_0),............,(X_j,Y_j)...................(X_k,Y_k) .

- where no two X_j are the same, the interpolation polynomial in the Lagrange form is a Linear combination .

$$L(x) := \sum_{j=0}^{k} y_j \ell_j(x)$$

Where :

$$\ell_j(x) := \prod_{\substack{0\le m\le k \\ m\ne j}} \frac{x-x_m}{x_j-x_m} = \frac{(x-x_0)}{(x_j-x_0)} \cdots \frac{(x-x_{j-1})}{(x_j-x_{j-1})} \frac{(x-x_{j+1})}{(x_j-x_{j+1})} \cdots \frac{(x-x_k)}{(x_j-x_k)},$$

Data structure used :

- vector:

  - vector is a sequence container that encapsulates dynamic size arrays.

  - why we use it ?

- Random access – constant *O(1)*
- Insertion or removal of elements at the end - amortized constant *O(1).*
- Insertion or removal of elements - linear in distance to the end of the vector *O(n).*

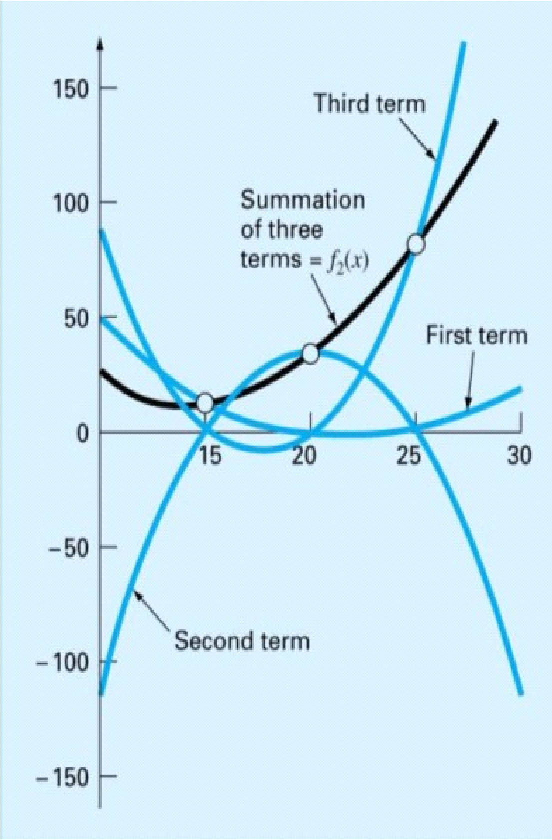
- we use vector of points ,every point consist of X and f(x).

Analysis:

we will talk about second order case :

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

The figure (1) shows :



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 .

- Same for other orders .

Error Analysis :

The Lagrange form of the interpolation polynomial shows the linear character of polynomial interpolation and the uniqueness of the interpolation polynomial . Therefore , we can consider that the error in both Lagrange and newtons is very close to each other .

NEWTON_N_DEVISION INTERPOLATION

1-pesudoCode

```
Newton_N_Devision( xy )
for i from 2 to  length of x
    for j from 2 to i
        mabX(i,j) <-- (mabX(i,j-1)-mabX(i-1,j-1))/(x(i)-x(i - j + 1))

    end

end

for i from 1 to length of x
    element <-- mab(i,i)
    for j from 1 to i - 1
        element <-- element +( x - x(j) )
    end for j

end for I

end while
```

general algorithm:

objective :

having some discrete points and we want to find a curve passing them .according to the number of points we can get a general formula equation that passes throw all of the given points.

Data structure used :

- vector:

- vector is a sequence container that encapsulates dynamic size arrays.

- why we use it ?

- Random access – constant *O(1)*
- Insertion or removal of elements at the end - amortized constant *O(1).*
- Insertion or removal of elements - linear in distance to the end of the vector *O(n).*

- we use 2D vectors to evaluate the coeffients of $F_n(x)$.

Analysis:

in case of having only two points the equation can be formulated easily from the eqaution of the slope .

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

we can reformulate this equation to be like this :

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

but newton made a general formula that fits for n^{th} order eqaution.

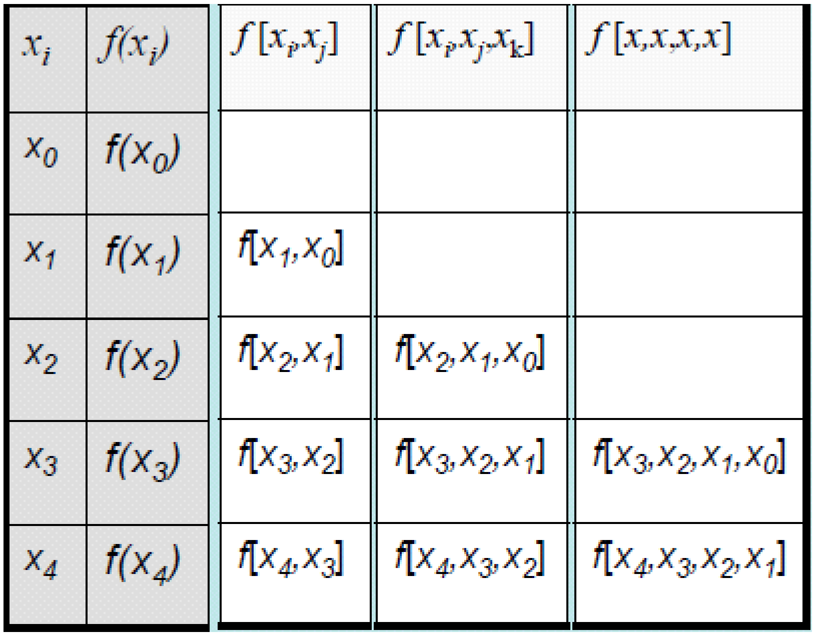
$$f_n(x) = b_0 + b_1(x - x_0) + b_2(x - x_0)(x - x_1) + \cdots + b_n(x - x_0)(x - x_1)\cdots(x - x_{n-1})$$

$$b_0 = f(x_0) \quad b_1 = f[x_1, x_0] \quad b_2 = f[x_2, x_1, x_0] \ \ldots \ b_n = f[x_n, x_{n-1}, \cdots, x_1, x_0]$$

where

$$f[x_i, x_j] = \frac{f(x_i) - f(x_j)}{x_i - x_j}$$

$$f[x_n, x_{n-1}, \ldots, x_1, x_0] = \frac{f[x_n, x_{n-1}, \ldots, x_1] - f[x_{n-1}, x_{n-2}, \ldots, x_0]}{x_n - x_0}$$



x_i	$f(x_i)$	$f[x_i,x_j]$	$f[x_i,x_j,x_k]$	$f[x,x,x,x]$
x_0	$f(x_0)$			
x_1	$f(x_1)$	$f[x_1,x_0]$		
x_2	$f(x_2)$	$f[x_2,x_1]$	$f[x_2,x_1,x_0]$	
x_3	$f(x_3)$	$f[x_3,x_2]$	$f[x_3,x_2,x_1]$	$f[x_3,x_2,x_1,x_0]$
x_4	$f(x_4)$	$f[x_4,x_3]$	$f[x_4,x_3,x_2]$	$f[x_4,x_3,x_2,x_1]$

Error of Interpolations :

For an *nth*-order interpolating polynomial, an analogous relationship for the error is:

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

For non differentiable functions, if an additional point *f(xn+1)*is available, an alternative formula can be used that does not require prior knowledge of the function:

$$R_n \cong f[x_{n+1}, x_n, x_{n-1}, \ldots, x_0](x-x_0)(x-x_1)\cdots(x-x_n)$$

The GUI

Part one

How to use root Finder :

1- run the gui.m the gui will start .

2- now you have two options
first --> take the input "equation" from the file
second --> write the input "equation" in text field in gui

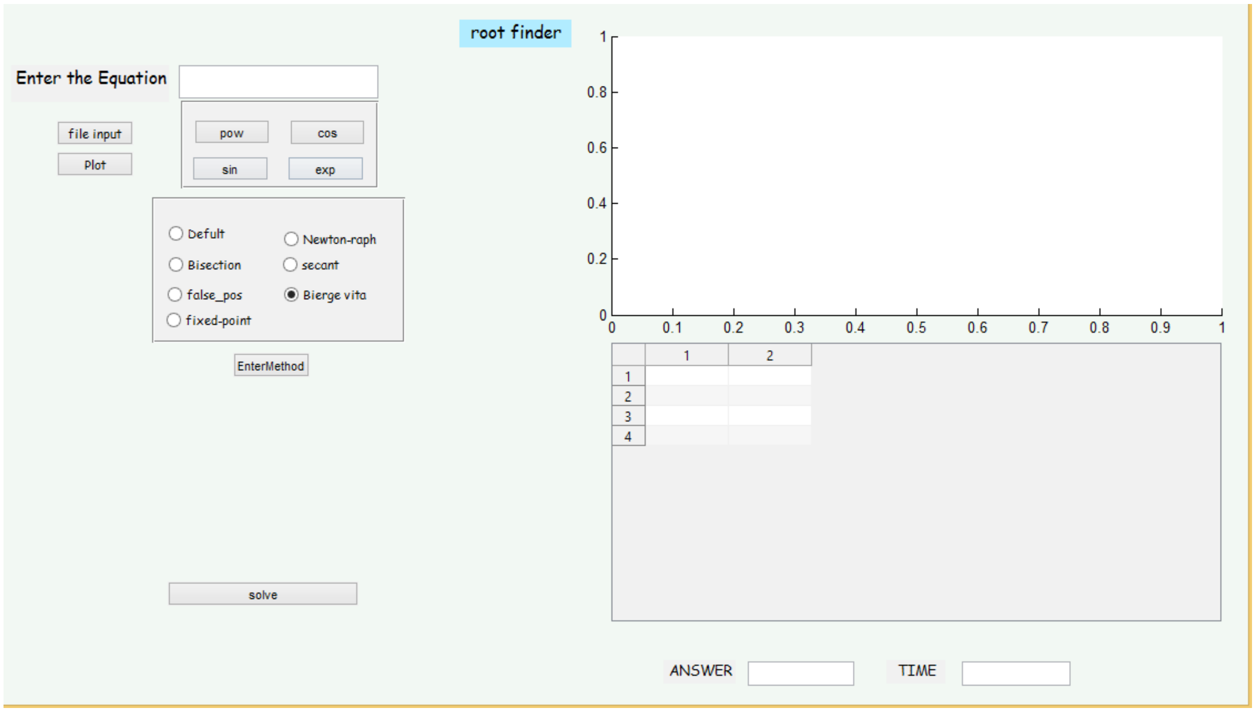
3-then choose the method you have 6 options :
a- Bisection Method
b- False Position Method
c- Fixed Point Method
d- Newton Method
e- Secant Method
f- Bierge Vita Method
g- Default Method this run if the user don't choose any thing

4-the panel will appear according to the method

5-click on solve to solve the equation

6-the answer will appear in answer field and the time and the plot of function and signals

This is the program when you run it :



○ Defult
○ Newton-raph
○ Bisection
○ secant
○ false_pos
● Bierge vita
○ fixed-point

	1	2
1		
2		
3		
4		

IMPORTANT POINTS IN PART ONE :

- Part one contain 7 Methods to get the root of equations : (Bisection ,False Position ,Fixed Point ,Newton ,Secant ,Bierge Vita and Default)

- Default Max Iterations = 50, Default Epsilon = 0.00001;

- The Default method use bisection method but after get a good initial guess and it loop for -1000 to 1000 and in every loop try to find the upper and the lower of the function where the f(xl) * f(xu) < 0 when the condition is true the two points will be the upper and the lower initial guess.

- You can zoom on in the plot of the function to see signals and to focus on the paint

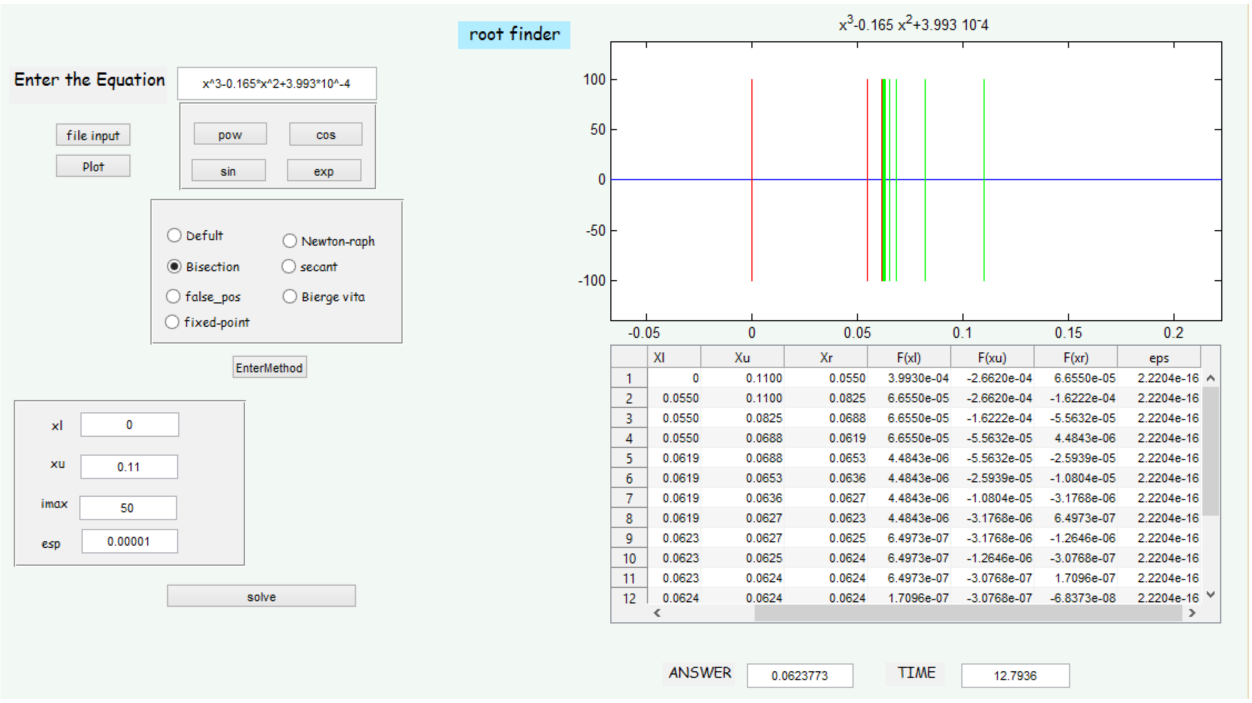
- The gui include the answer of the equation and the time of using the specific method and the table contains every iteration and some information like : number of iterations, execution time, approximate root , error, and precision.

- You can get the input from file or enter the input in gui .

- The part one include the **bonus** : Single step mode simulation showing the iterations on the drawn function for one method of choice

-

1- <u>Bisection Method :</u>



Enter the Equation: x^3-0.165*x^2+3.993*10^-4

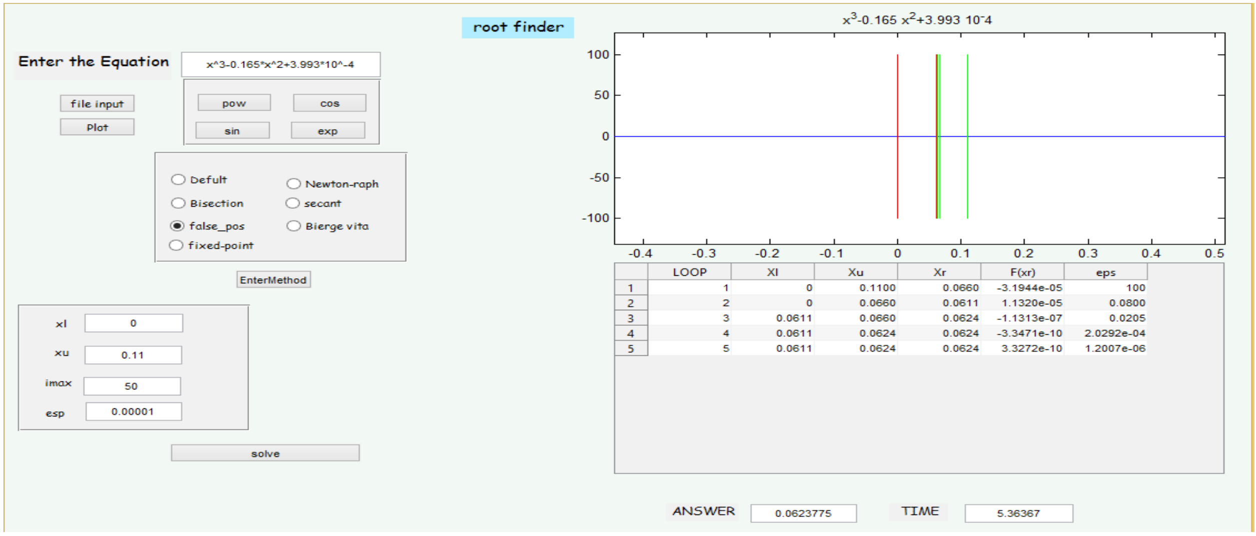
○ Defult ○ Newton-raph
● Bisection ○ secant
○ false_pos ○ Bierge vita
○ fixed-point

xl: 0
xu: 0.11
imax: 50
esp: 0.00001

	Xl	Xu	Xr	F(xl)	F(xu)	F(xr)	eps
1	0	0.1100	0.0550	3.9930e-04	-2.6620e-04	6.6550e-05	2.2204e-16
2	0.0550	0.1100	0.0825	6.6550e-05	-2.6620e-04	-1.6222e-04	2.2204e-16
3	0.0550	0.0825	0.0688	6.6550e-05	-1.6222e-04	-5.5632e-05	2.2204e-16
4	0.0550	0.0688	0.0619	6.6550e-05	-5.5632e-05	4.4843e-06	2.2204e-16
5	0.0619	0.0688	0.0653	4.4843e-06	-5.5632e-05	-2.5939e-05	2.2204e-16
6	0.0619	0.0653	0.0636	4.4843e-06	-2.5939e-05	-1.0804e-05	2.2204e-16
7	0.0619	0.0636	0.0627	4.4843e-06	-1.0804e-05	-3.1768e-06	2.2204e-16
8	0.0619	0.0627	0.0623	4.4843e-06	-3.1768e-06	6.4973e-07	2.2204e-16
9	0.0623	0.0627	0.0625	6.4973e-07	-3.1768e-06	-1.2646e-06	2.2204e-16
10	0.0623	0.0625	0.0624	6.4973e-07	-1.2646e-06	-3.0768e-07	2.2204e-16
11	0.0623	0.0624	0.0624	6.4973e-07	-3.0768e-07	1.7096e-07	2.2204e-16
12	0.0624	0.0624	0.0624	1.7096e-07	-3.0768e-07	-6.8373e-08	2.2204e-16

ANSWER 0.0623773 TIME 12.7936

False Position Method :



Enter the Equation x^3-0.165*x^2+3.993*10^-4

○ Defult
○ Bisection
● false_pos
○ fixed-point
○ Newton-raph
○ secant
○ Bierge vita

xl 0

xu 0.11

imax 50

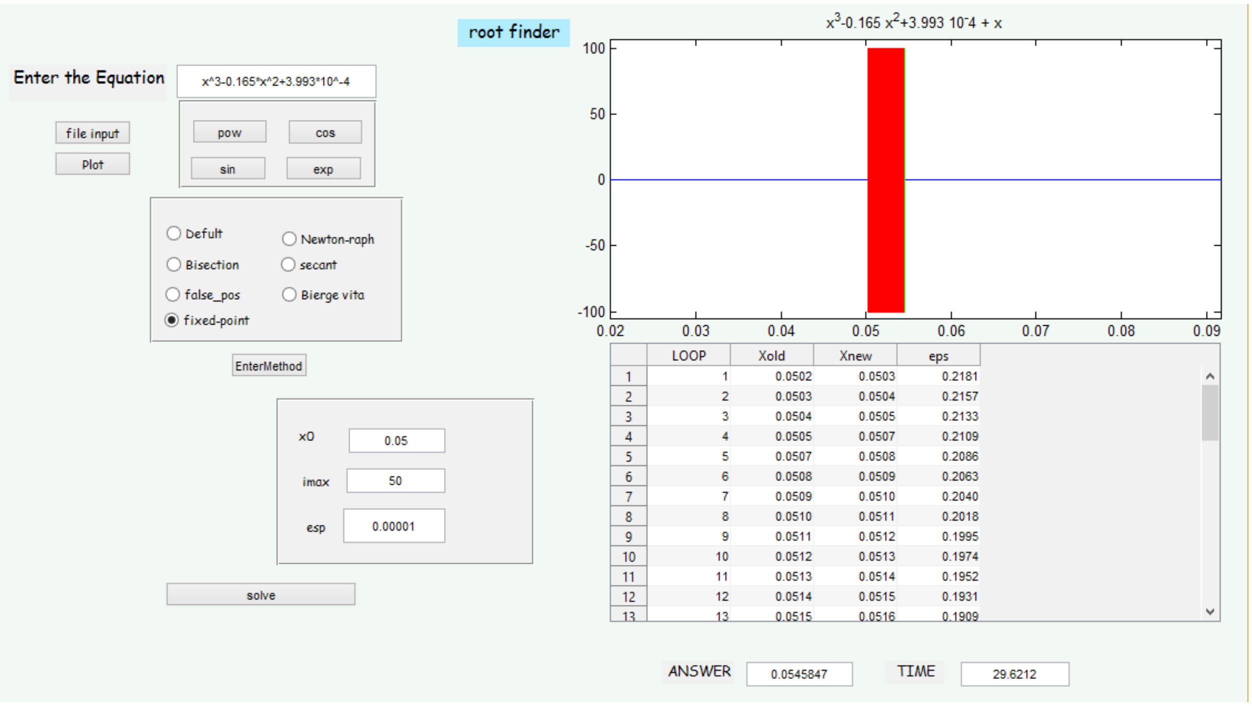
esp 0.00001

	LOOP	Xl	Xu	Xr	F(xr)	eps
1	1	0	0.1100	0.0660	-3.1944e-05	100
2	2	0	0.0660	0.0611	1.1320e-05	0.0800
3	3	0.0611	0.0660	0.0624	-1.1313e-07	0.0205
4	4	0.0611	0.0624	0.0624	-3.3471e-10	2.0292e-04
5	5	0.0611	0.0624	0.0624	3.3272e-10	1.2007e-06

ANSWER 0.0623775

TIME 5.36367

2- Fixed Point Method :



Enter the Equation x^3-0.165*x^2+3.993*10^-4

○ Defult
○ Bisection
○ false_pos
● fixed-point
○ Newton-raph
○ secant
○ Bierge vita

x0 0.05

imax 50

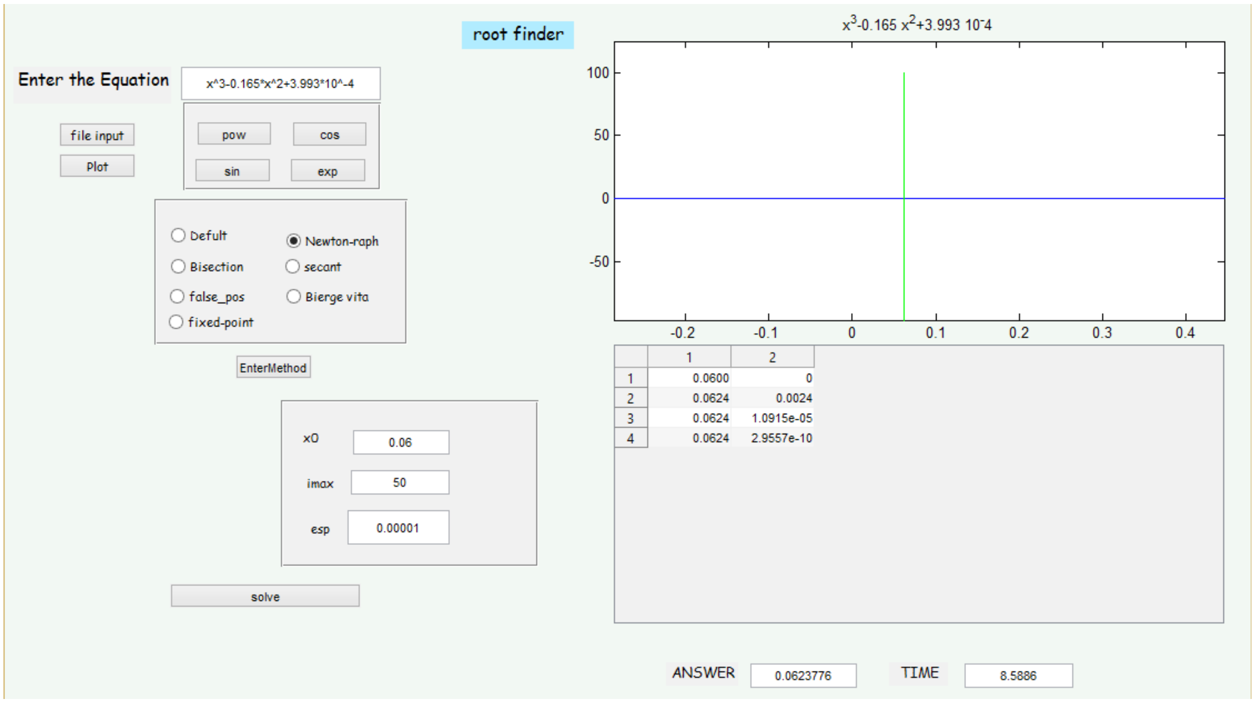
esp 0.00001

	LOOP	Xold	Xnew	eps
1	1	0.0502	0.0503	0.2181
2	2	0.0503	0.0504	0.2157
3	3	0.0504	0.0505	0.2133
4	4	0.0505	0.0507	0.2109
5	5	0.0507	0.0508	0.2086
6	6	0.0508	0.0509	0.2063
7	7	0.0509	0.0510	0.2040
8	8	0.0510	0.0511	0.2018
9	9	0.0511	0.0512	0.1995
10	10	0.0512	0.0513	0.1974
11	11	0.0513	0.0514	0.1952
12	12	0.0514	0.0515	0.1931
13	13	0.0515	0.0516	0.1909

ANSWER 0.0545847

TIME 29.6212

Newton Method :



Enter the Equation: x^3-0.165*x^2+3.993*10^-4

○ Defult ● Newton-raph
○ Bisection ○ secant
○ false_pos ○ Bierge vita
○ fixed-point

x0: 0.06

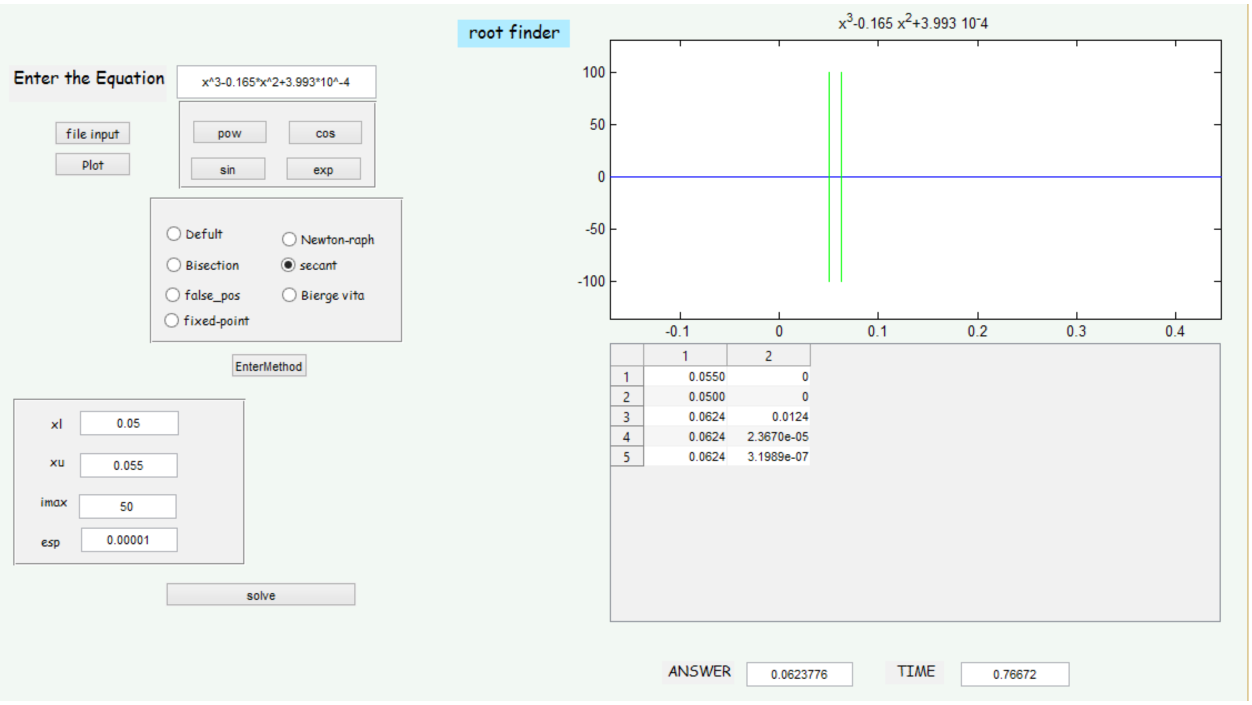
imax: 50

esp: 0.00001

	1	2
1	0.0600	0
2	0.0624	0.0024
3	0.0624	1.0915e-05
4	0.0624	2.9557e-10

ANSWER: 0.0623776 TIME: 8.5886

3- Secant Method :



Enter the Equation: x^3-0.165*x^2+3.993*10^-4

○ Defult ○ Newton-raph
○ Bisection ● secant
○ false_pos ○ Bierge vita
○ fixed-point

xl: 0.05

xu: 0.055

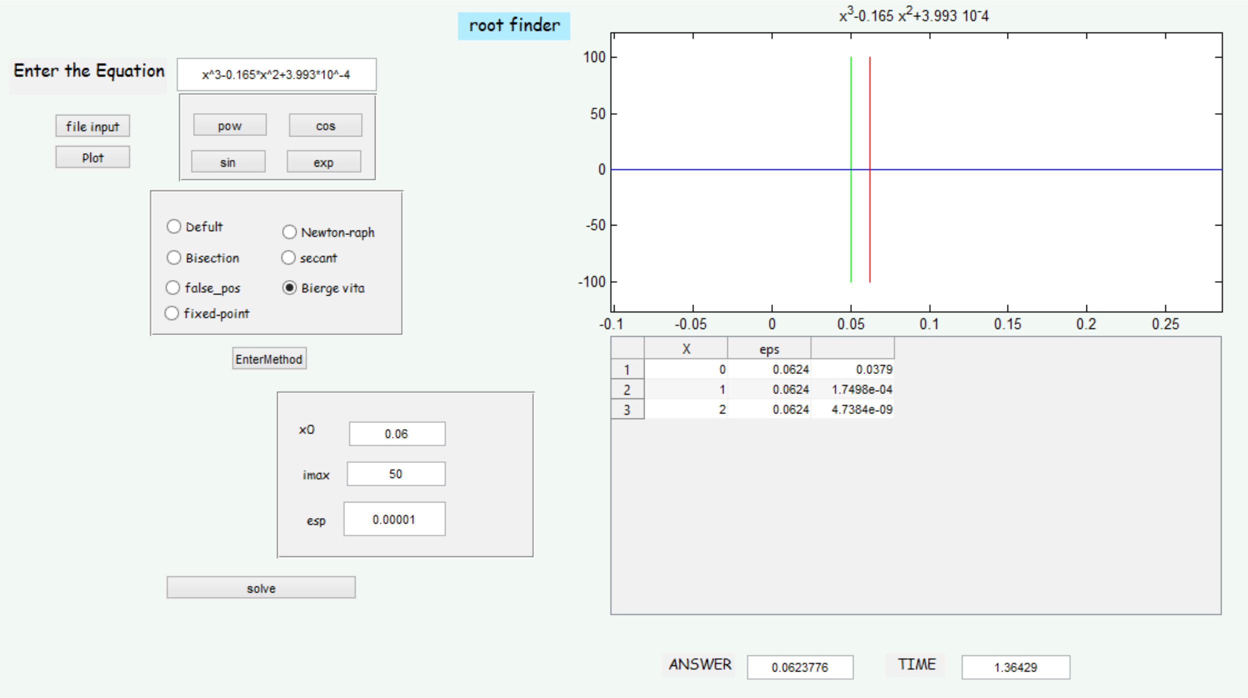
imax: 50

esp: 0.00001

	1	2
1	0.0550	0
2	0.0500	0
3	0.0624	0.0124
4	0.0624	2.3670e-05
5	0.0624	3.1989e-07

ANSWER: 0.0623776 TIME: 0.76672

4- Bierge Vita Method :



Enter the Equation: x^3-0.165*x^2+3.993*10^-4

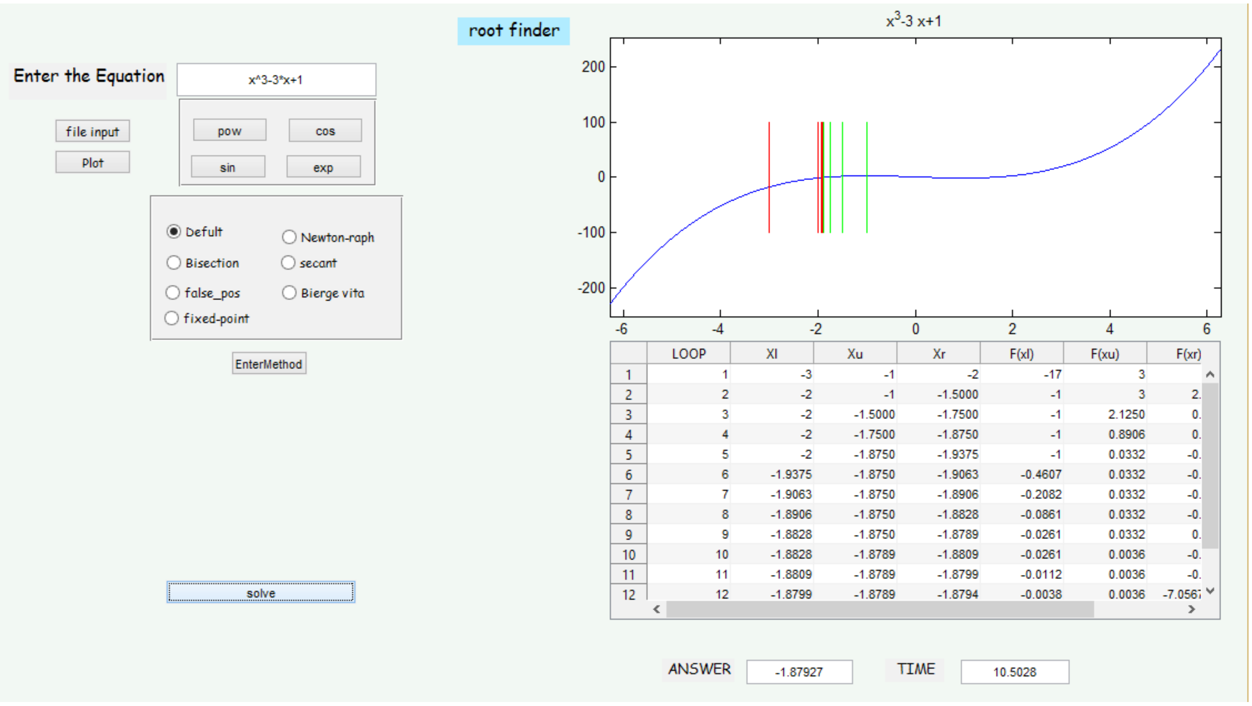
○ Defult ○ Newton-raph
○ Bisection ○ secant
○ false_pos ● Bierge vita
○ fixed-point

x0: 0.06
imax: 50
esp: 0.00001

	X	eps	
1	0	0.0624	0.0379
2	1	0.0624	1.7498e-04
3	2	0.0624	4.7384e-09

ANSWER: 0.0623776 TIME: 1.36429

5- Default Method :



Enter the Equation: x^3-3*x+1

● Defult ○ Newton-raph
○ Bisection ○ secant
○ false_pos ○ Bierge vita
○ fixed-point

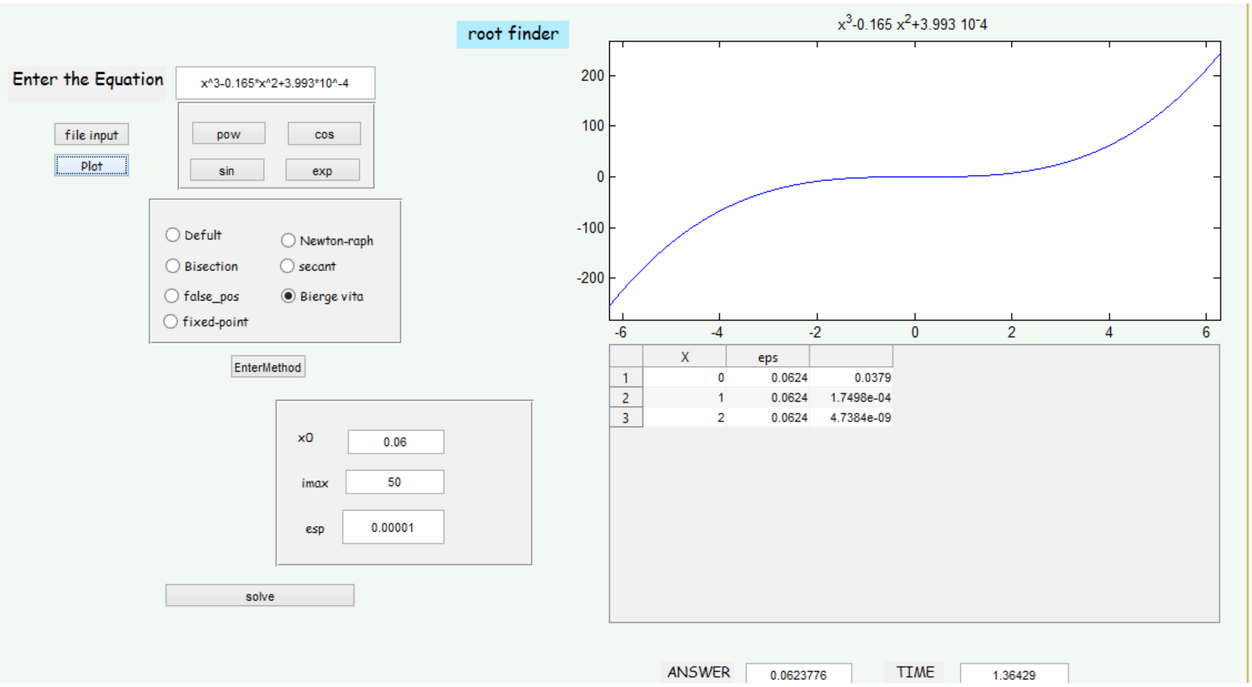
	LOOP	Xl	Xu	Xr	F(xl)	F(xu)	F(xr)
1	1	-3	-1	-2	-17	3	
2	2	-2	-1	-1.5000	-1	3	2.
3	3	-2	-1.5000	-1.7500	-1	2.1250	0.
4	4	-2	-1.7500	-1.8750	-1	0.8906	0.
5	5	-2	-1.8750	-1.9375	-1	0.0332	-0.
6	6	-1.9375	-1.8750	-1.9063	-0.4607	0.0332	-0.
7	7	-1.9063	-1.8750	-1.8906	-0.2082	0.0332	-0.
8	8	-1.8906	-1.8750	-1.8828	-0.0861	0.0332	-0.
9	9	-1.8828	-1.8750	-1.8789	-0.0261	0.0332	0.
10	10	-1.8828	-1.8789	-1.8809	-0.0261	0.0036	-0.
11	11	-1.8809	-1.8789	-1.8799	-0.0112	0.0036	-0.
12	12	-1.8799	-1.8789	-1.8794	-0.0038	0.0036	-7.0567

ANSWER: -1.87927 TIME: 10.5028

Input from file :

```
x^3-0.165*x^2+3.993*10^-4
```

Plot of Function From Plot Button :



Enter the Equation: x^3-0.165*x^2+3.993*10^-4

○ Defult
○ Newton-raph
○ Bisection
○ secant
○ false_pos
● Bierge vita
○ fixed-point

x0: 0.06

imax: 50

esp: 0.00001

	X	eps	
1	0	0.0624	0.0379
2	1	0.0624	1.7498e-04
3	2	0.0624	4.7384e-09

ANSWER: 0.0623776

TIME: 1.36429

Part Two

How to use interpolation :

1- run tne gui2.m the gui will start

2- now you have two options
first --> take the input from the file
second --> write the inputin gui

3- first enter the number of point that you want to use to get the equation and then enter the points one by one after every one enter the button.

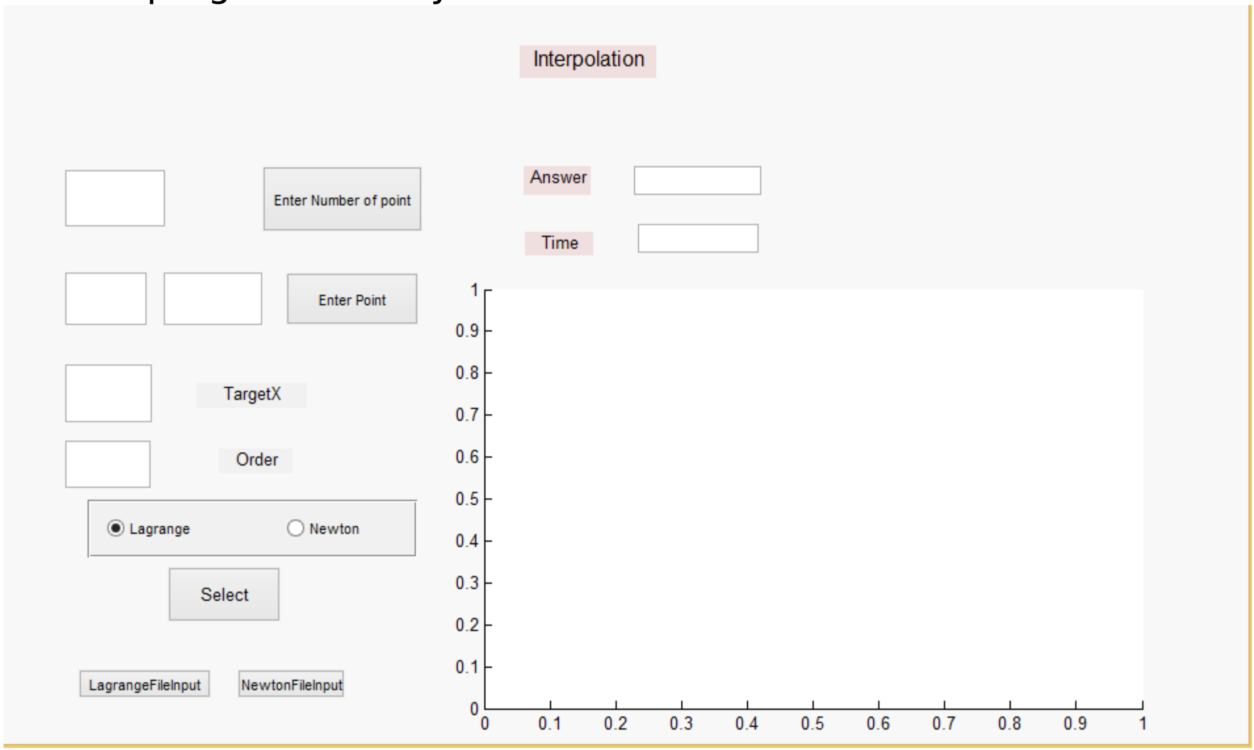
4- the choose one of two methods

a-Newton
b-Lagrange

5- click on solve to get the equation

6- the answer will appear in answer field and the time and the plot of function

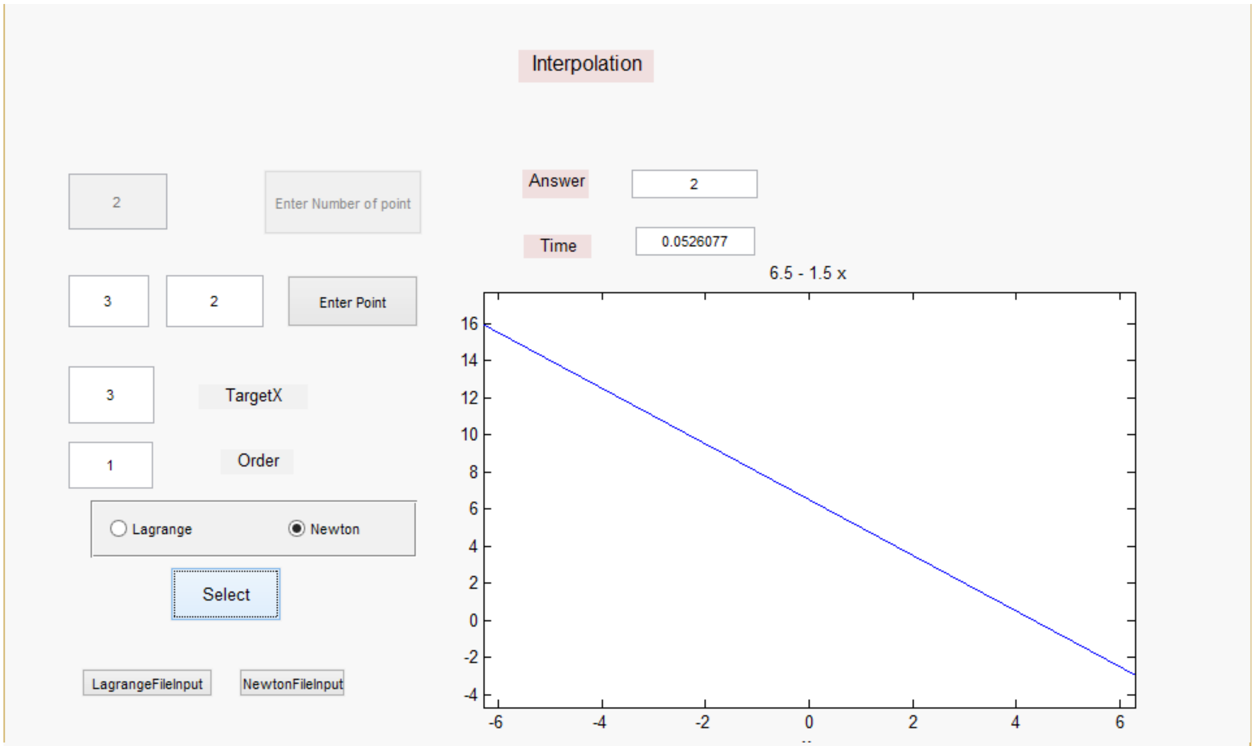
This is the program when you run it :



IMPORTANT POINTS IN PART TWO:

- The input can be from file and can be from the gui
- The gui will contain the equation and the answer from the target point and the time that every method take and the plot of the equation
- It contains two method to get the equation newton and lagrange

Newton :



Interpolation

2 | Enter Number of point

3 | 2 | Enter Point

TargetX: 3

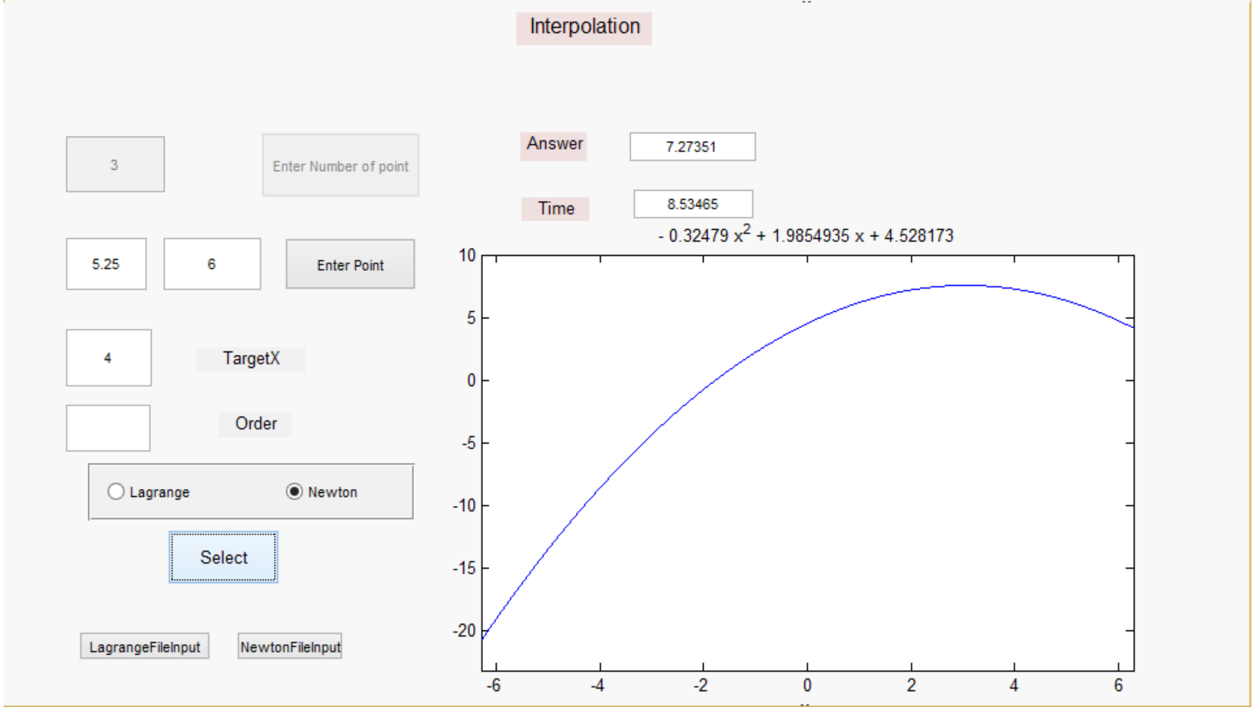
Order: 1

○ Lagrange ● Newton

Answer: 2

Time: 0.0526077

6.5 - 1.5 x



Interpolation

3 | Enter Number of point

5.25 | 6 | Enter Point

TargetX: 4

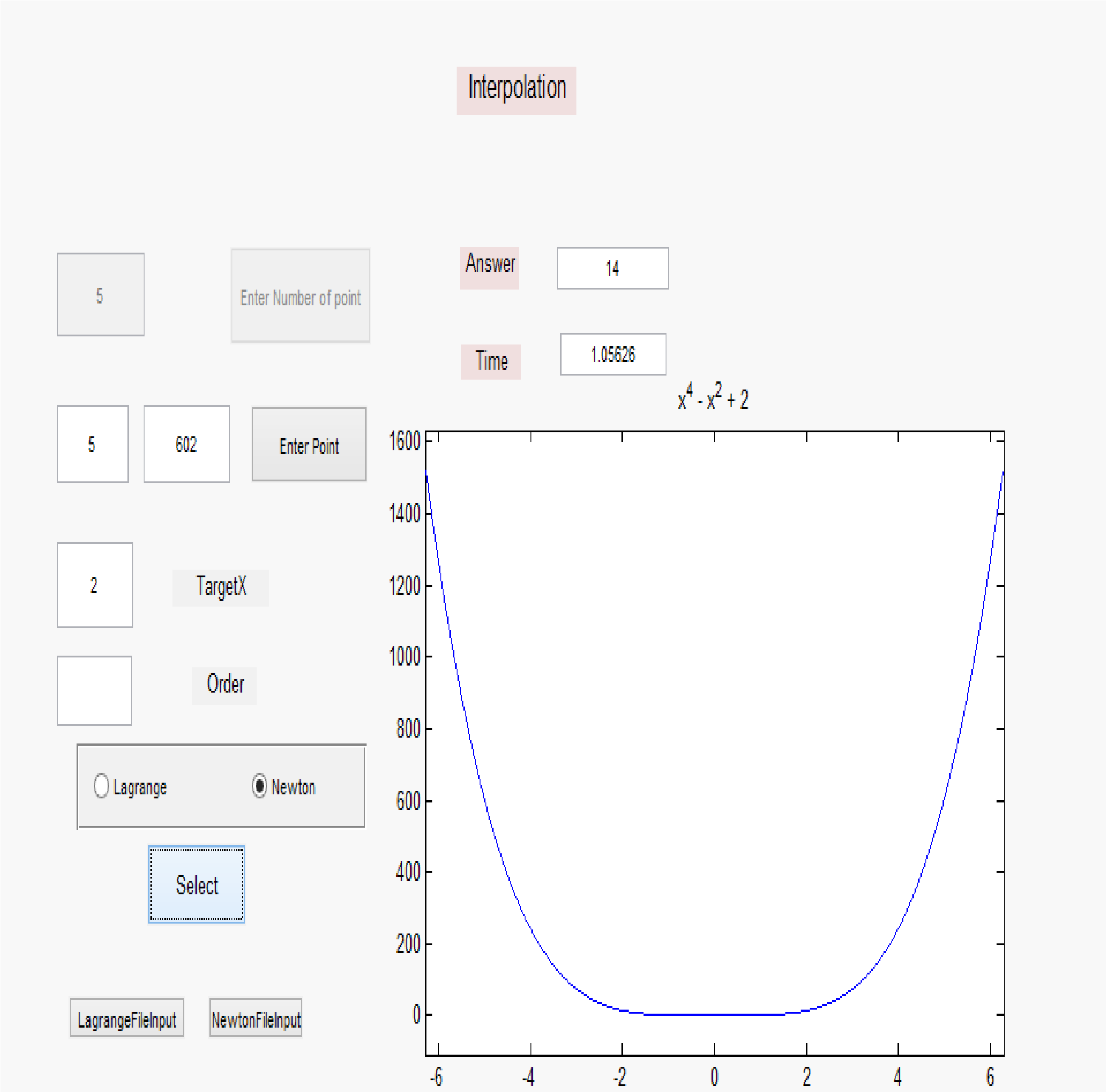
Order:

○ Lagrange ● Newton

Answer: 7.27351

Time: 8.53465

$- 0.32479 x^2 + 1.9854935 x + 4.528173$



Interpolation

5 Enter Number of point

5 602 Enter Point

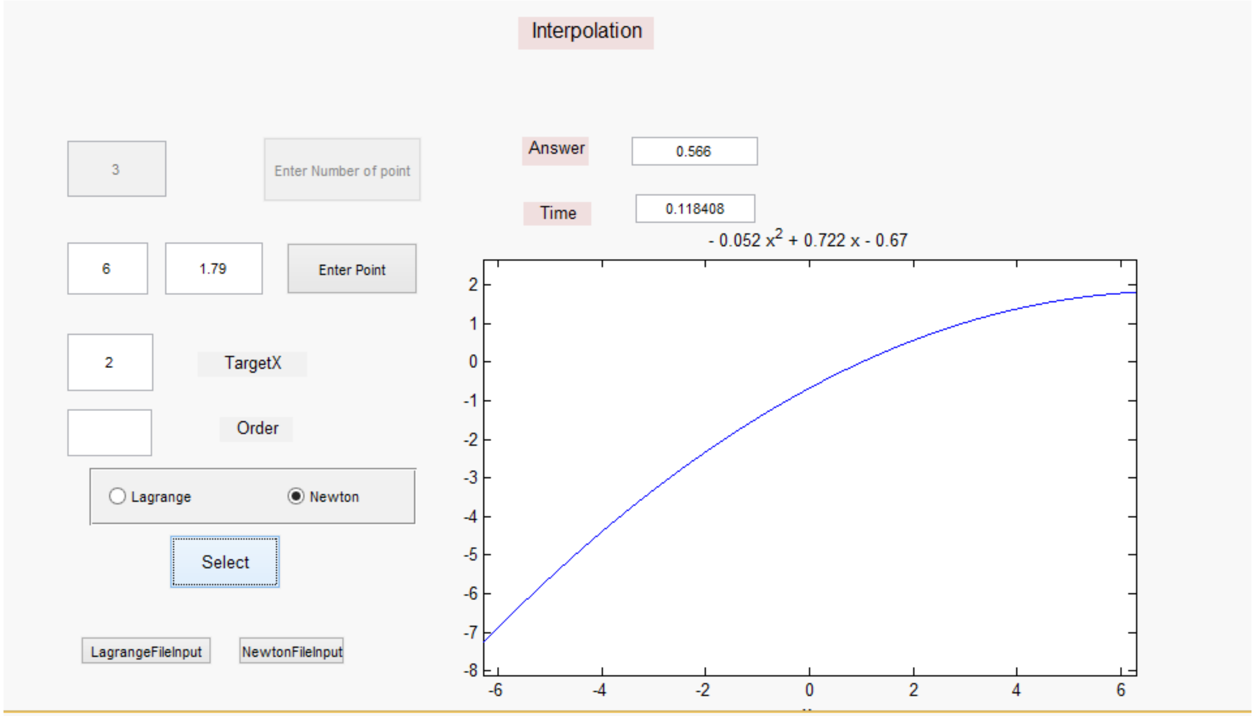
2 TargetX

Order

○ Lagrange ● Newton

Answer 14

Time 1.05626



Interpolation

3 — Enter Number of point

6 — 1.79 — Enter Point

2 — TargetX

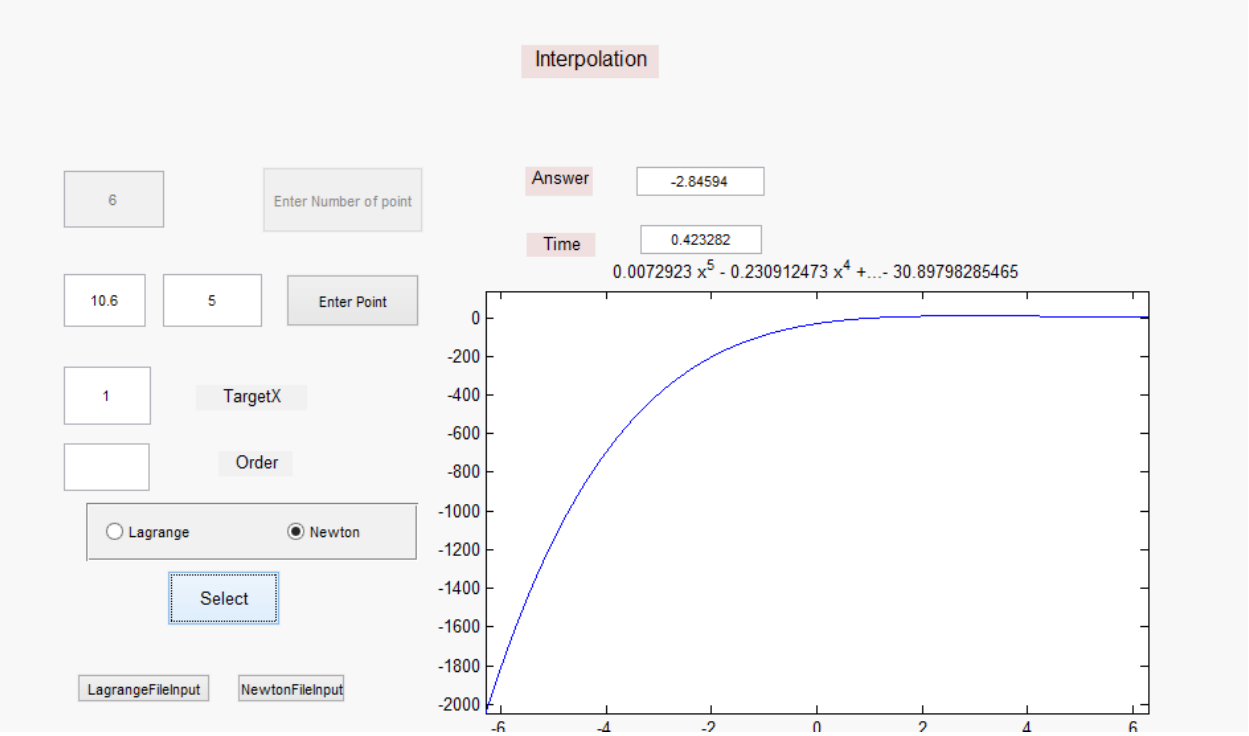
Order:

○ Lagrange ● Newton

Answer: 0.566

Time: 0.118408

$- 0.052 x^2 + 0.722 x - 0.67$



Interpolation

6 — Enter Number of point

10.6 — 5 — Enter Point

1 — TargetX

Order:

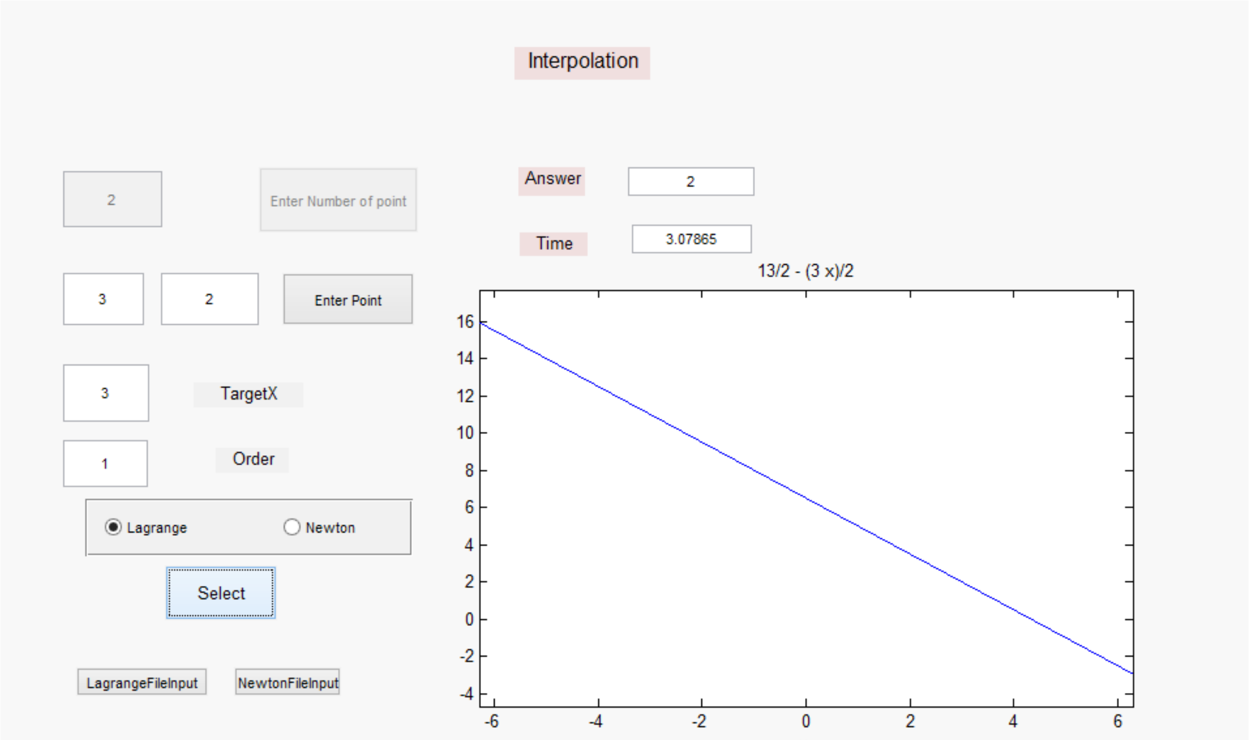
○ Lagrange ● Newton

Answer: -2.84594

Time: 0.423282

$0.0072923 x^5 - 0.230912473 x^4 +...- 30.89798285465$

Lagrange :



Interpolation

2 Enter Number of point

3 2 Enter Point

3 TargetX

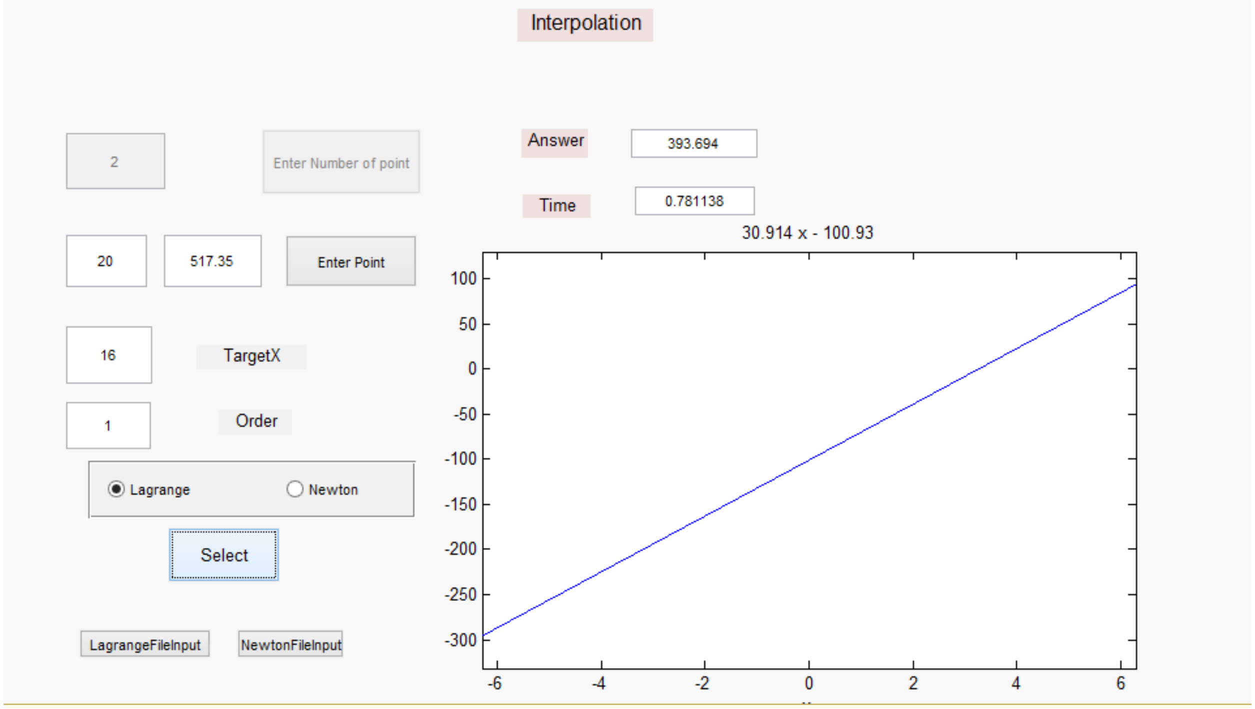
1 Order

● Lagrange ○ Newton

Answer 2

Time 3.07865

13/2 - (3 x)/2



Interpolation

2 Enter Number of point

20 517.35 Enter Point

16 TargetX

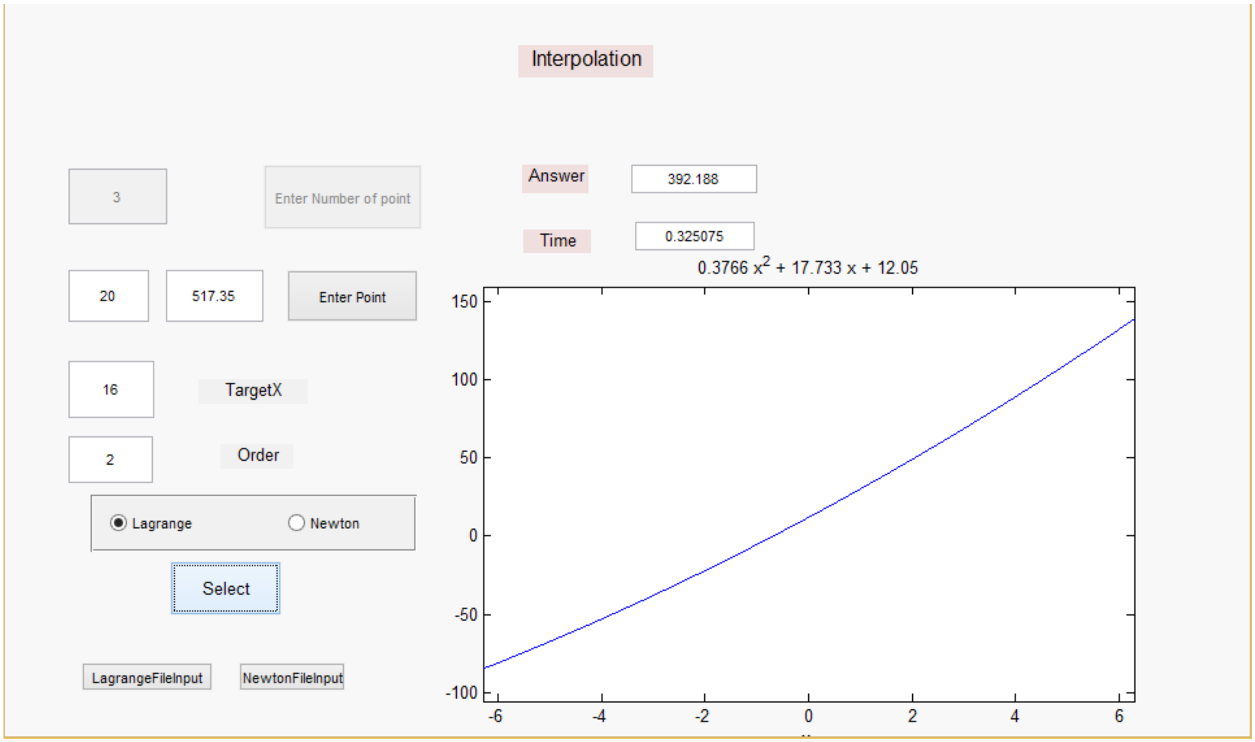
1 Order

● Lagrange ○ Newton

Answer 393.694

Time 0.781138

30.914 x - 100.93



Interpolation

3 Enter Number of point

Answer 392.188

Time 0.325075

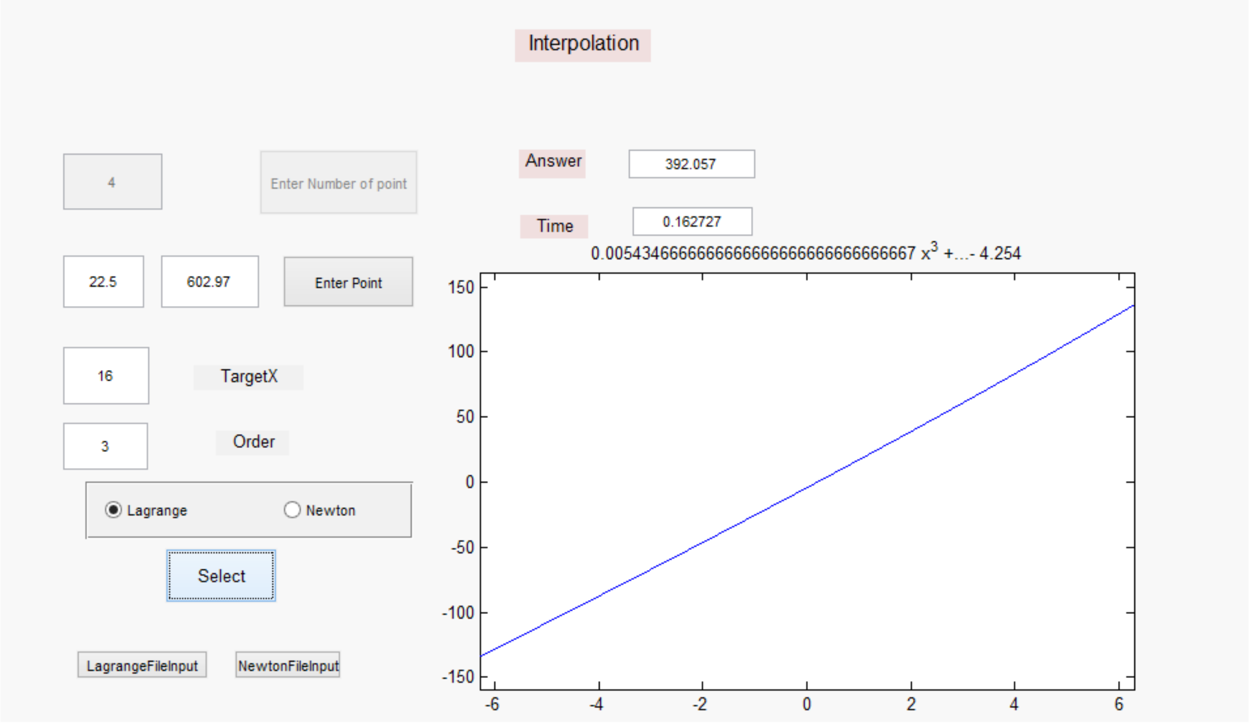
20 517.35 Enter Point

16 TargetX

2 Order

● Lagrange ○ Newton

$0.3766 x^2 + 17.733 x + 12.05$



Interpolation

4 Enter Number of point

Answer 392.057

Time 0.162727

22.5 602.97 Enter Point

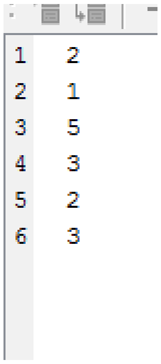
16 TargetX

3 Order

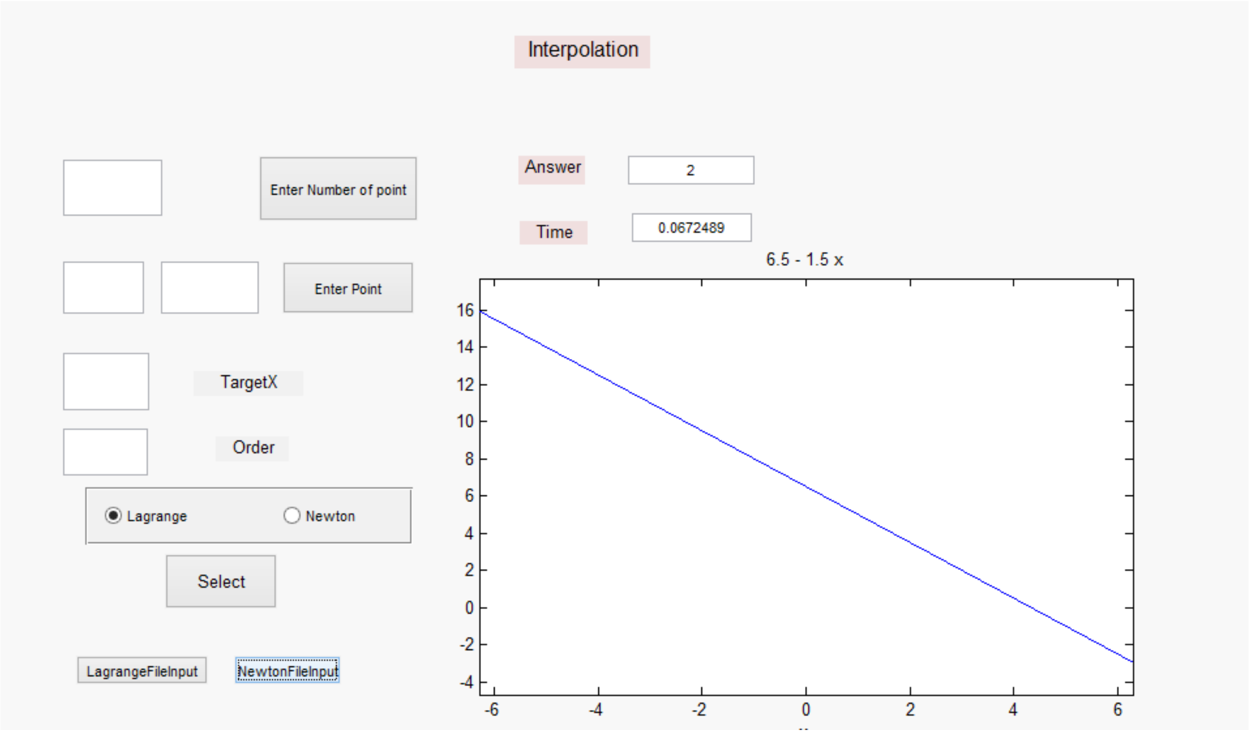
● Lagrange ○ Newton

$0.00543466666666666666666666666667 x^3 +...- 4.254$

Get input from file Newton :



```
2
1
5
3
2
3
```



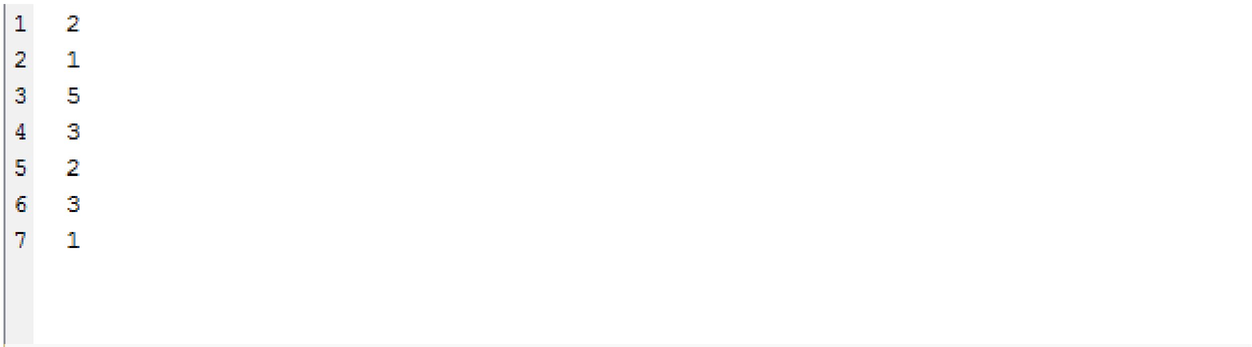
Interpolation

Answer: 2

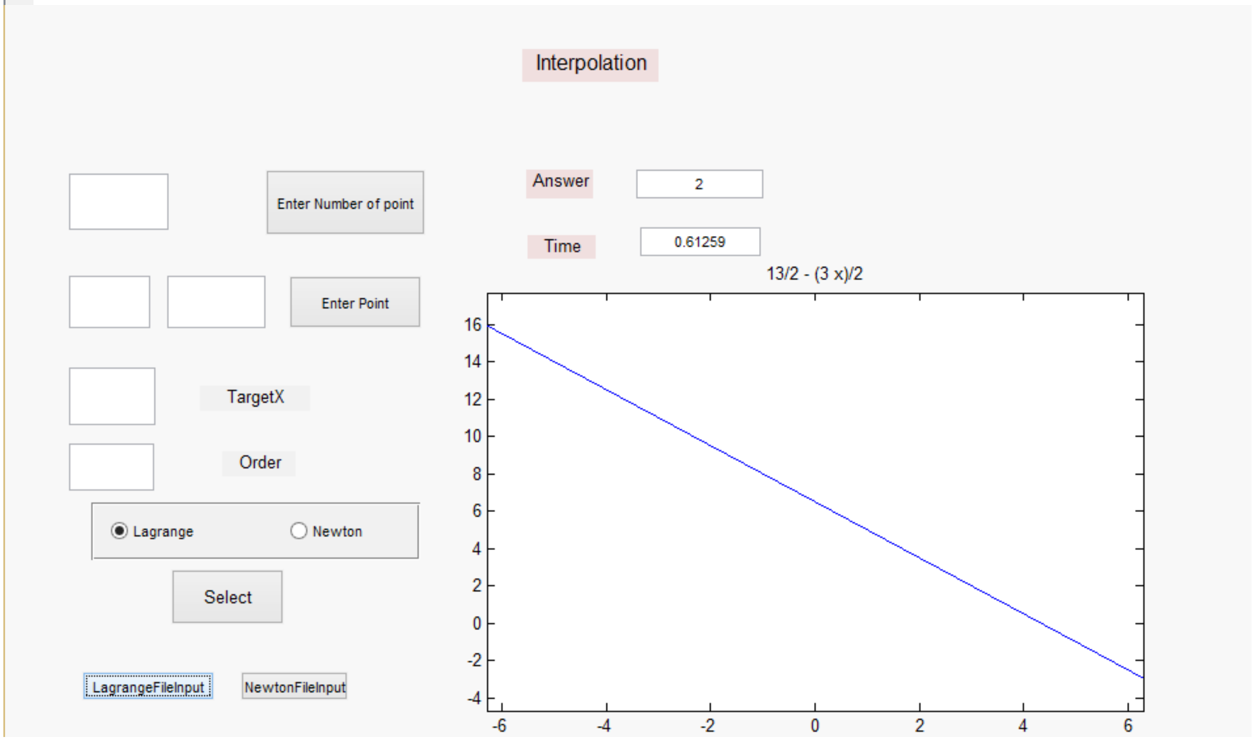
Time: 0.0672489

● Lagrange ○ Newton

Get input from file Lagrange :



```
2
1
5
3
2
3
1
```



Interpolation

Answer: 2

Time: 0.61259

● Lagrange ○ Newton

13/2 - (3 x)/2

References :

- Applied Numerical Methods with MATLAB for Engineers and Scientists - Chapra, Second Edition

-Numerical Methods using MATLAB - Mathews & Fink, Third Edition

-Numerical Methods For Engineers sixth Edition .