



Alexandria University
Faculty of Engineering
Computer and Systems Engineering Dept

Programming Assignment

NUMERICAL

Team (10)

Names

Salma Abd el Aziz Abd el Hamid 25

Salma Mohamed Mohamed 26

SHadwa Abd el Mobdy Aly 28

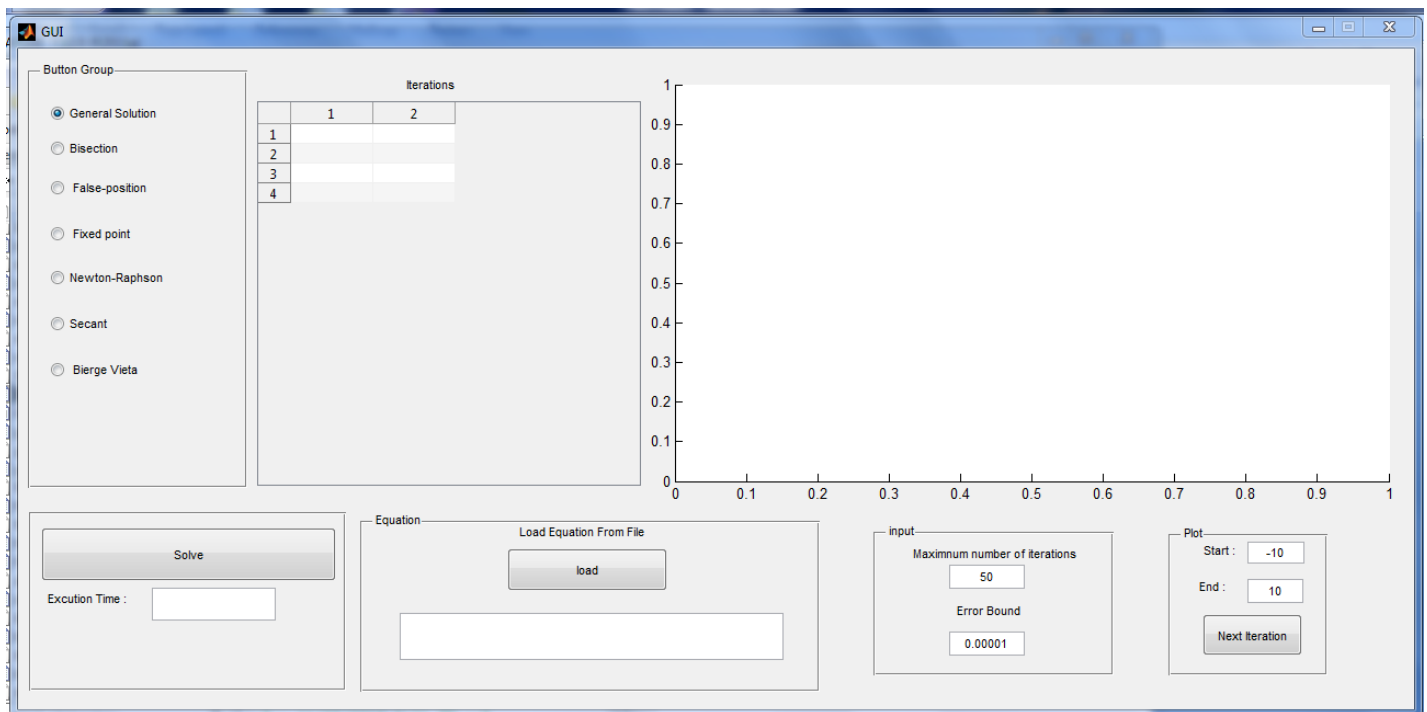
Amr Mahmoud Bekhiet 45

Mohamed Youssef Gewilli 63

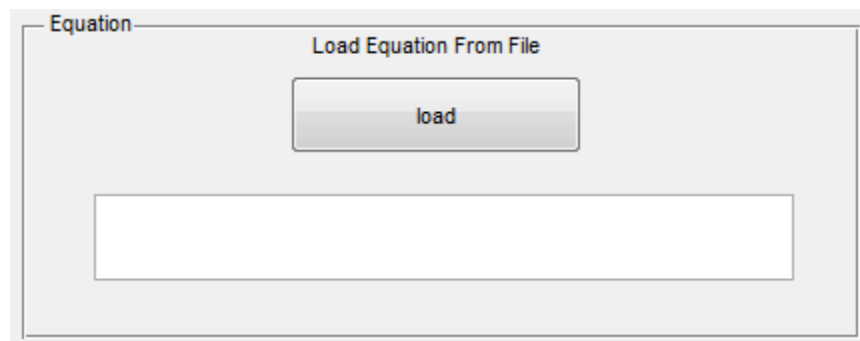
Part 1



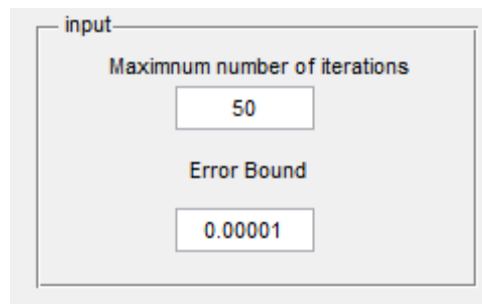
User Manual :



1. Enter the equation to be solved.



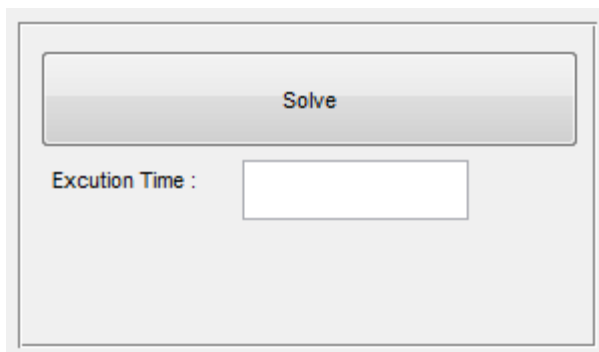
2. Enter maximum number of iterations and error bound .



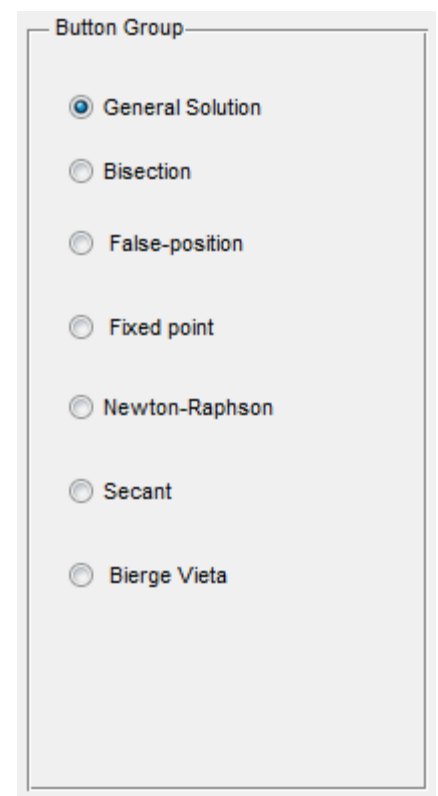
A dialog box titled "input" with a light gray background. It contains two text input fields. The first field is labeled "Maximum number of iterations" and contains the value "50". The second field is labeled "Error Bound" and contains the value "0.00001".

3. Choose the method of evaluation .

4. click solve.

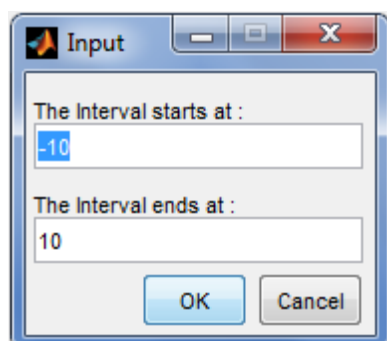


A dialog box with a light gray background. It features a large, rectangular button labeled "Solve" with a gradient. Below the button, the text "Excution Time :" is followed by a text input field.



A vertical panel titled "Button Group" with a light gray background. It contains seven radio buttons, each followed by a method name: "General Solution" (selected), "Bisection", "False-position", "Fixed point", "Newton-Raphson", "Secant", and "Bierge Vieta".

5. Enter the initial the guesses.



A standard Windows-style dialog box titled "Input" with a blue title bar and standard window controls. It contains two text input fields. The first field is labeled "The Interval starts at :" and contains the value "-10". The second field is labeled "The Interval ends at :" and contains the value "10". At the bottom are "OK" and "Cancel" buttons.

6. Iterations appear in this box .

Iterations		
	1	2
1		
2		
3		
4		

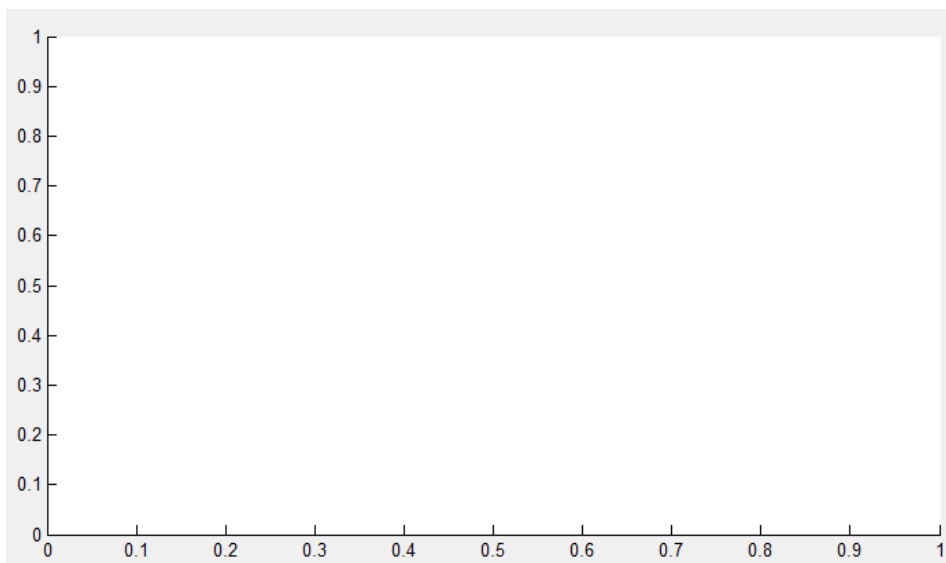
7. Then click here until we reach to the root .

8. the plot appears here.

Plot

Start :

End :



- In case loading the equation from a file click on this button.

The image shows a software window titled "Equation". Inside the window, there is a button labeled "Load Equation From File" and a button labeled "load". Below these buttons is a large, empty rectangular text input field.



Bisection :

• Algorithm:

1. choose lower " x_l " and upper " x_u " guesses for the root such that the function changes sign over the interval .

(check that : $f(x_l)f(x_u) < 0$)

2.Determine the estimate of the root " $x_r = \frac{x_u + x_l}{2}$ "

3.Determine the subinterval at which the root belongs to , by forming these evaluations :

$f(x_l)f(x_r) < 0$: The root lies at the lower subinterval,

$$\text{set : } x_u = x_r$$

$f(x_l)f(x_r) > 0$: The root lies at the lower subinterval,

$$\text{set : } x_l = x_r$$

$f(x_l)f(x_r) = 0$: The root equals x_r , then we stop .

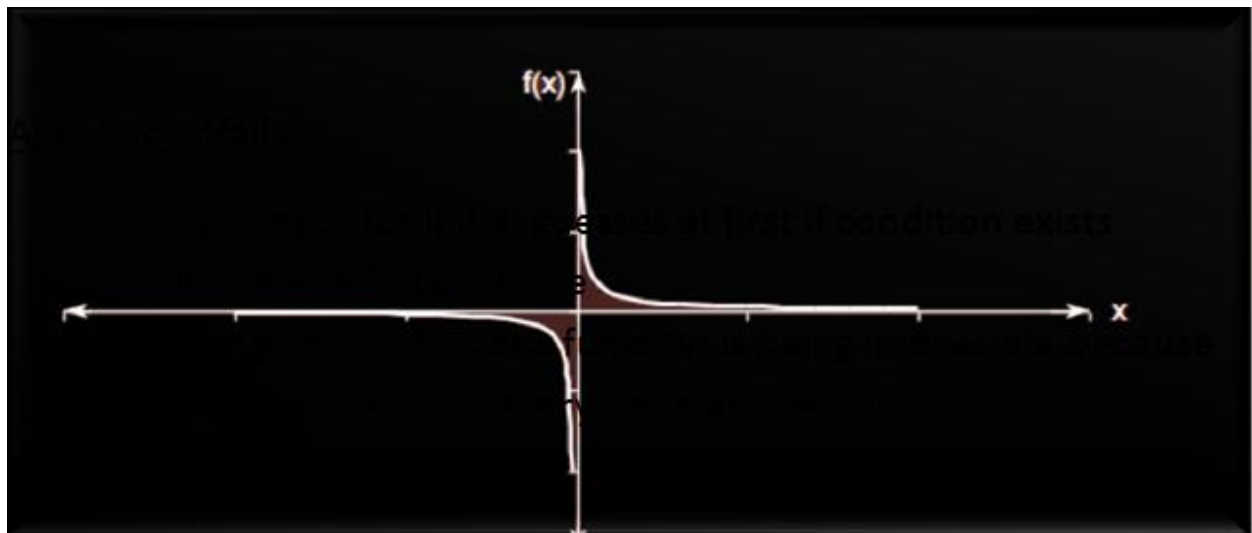
- Pseudo code :

```
function [ y ] = bisection( upper, lower, es,
    iMax, equation )
    y <= []
    fl <= f(lower , equation)
    fu <= f(upper , equation)
    ea <= 0.0
    xro <= 0.0
    if fu * fl > 0
        return
    else if fu * fl < 0
        xu <= upper
        xl <= lower
        for i <= 0:iMax
            xr <= (xu+xl)/2
            if i != 0
                ea <= abs(xr - xro)
                fxr <= f(xr , equation)
                test <= fxr * fl;
                y = [y;[xu,xl,xr,ea]]
                if test < 0
                    xu <= xr
                    fu <= fxr
                else if test > 0
                    xl <= xr
                    fl <= fxr
                else
                    break
                if ea < es && i != 0
                    break
                xro <= xr
        else
            if fu == 0
                xr <= upper
            else
                xr <= lower
            y = [ y ;[ upper , lower , xr , ea]]
```

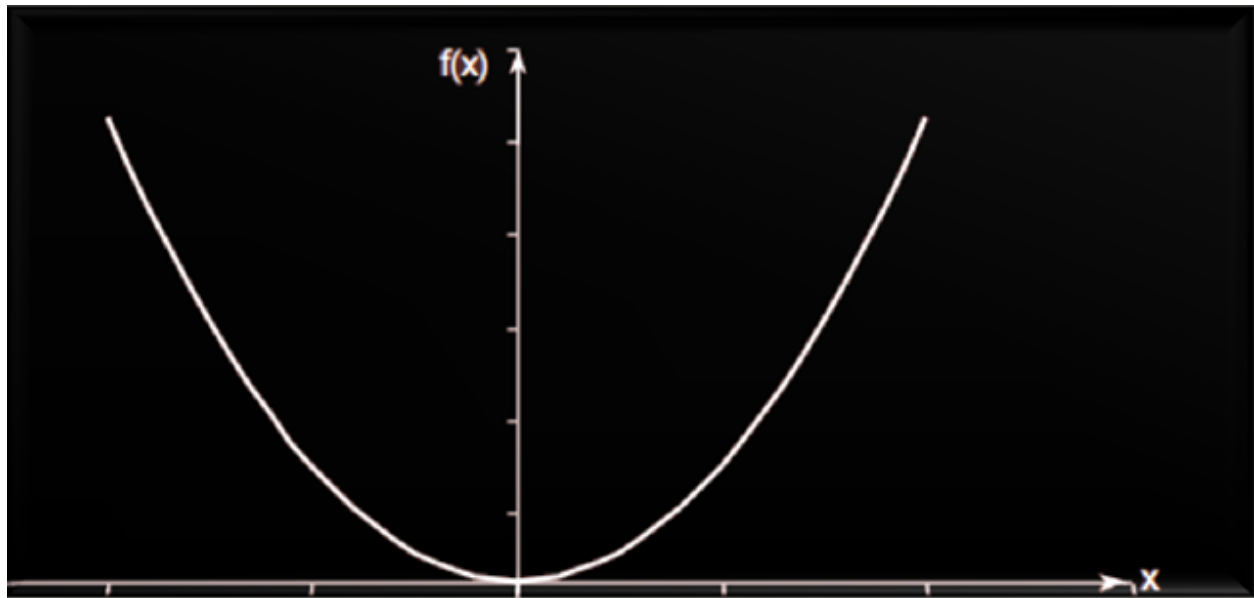
- This method returns an array containing upper limit , lower limit , xr and error .

- **Pitfalls:**

1. Slow method .
2. Need to find suitable initial guesses (x_l, x_u) . "sometimes it is impossible to find the initial guesses "
3. No account is taken of the fact that $f(x_1)$ is closer to zero ,it is likely that the root is closer to x_1 .
4. Doesn't work with non-continuous functions even the initial guesses are correct .



- Functions that changes sign but does not have roots.



-If the function just touches the x-axis , we will be unable to find the initial guesses.

- **Avoiding pitfalls :**

1. we can check for initial guesses at first . If condition exists continue ,else notify the user of the error.
2. Avoiding non-continuous functions is impossible because we have to check for every point at the given interval.

- **Divergence and convergence :**

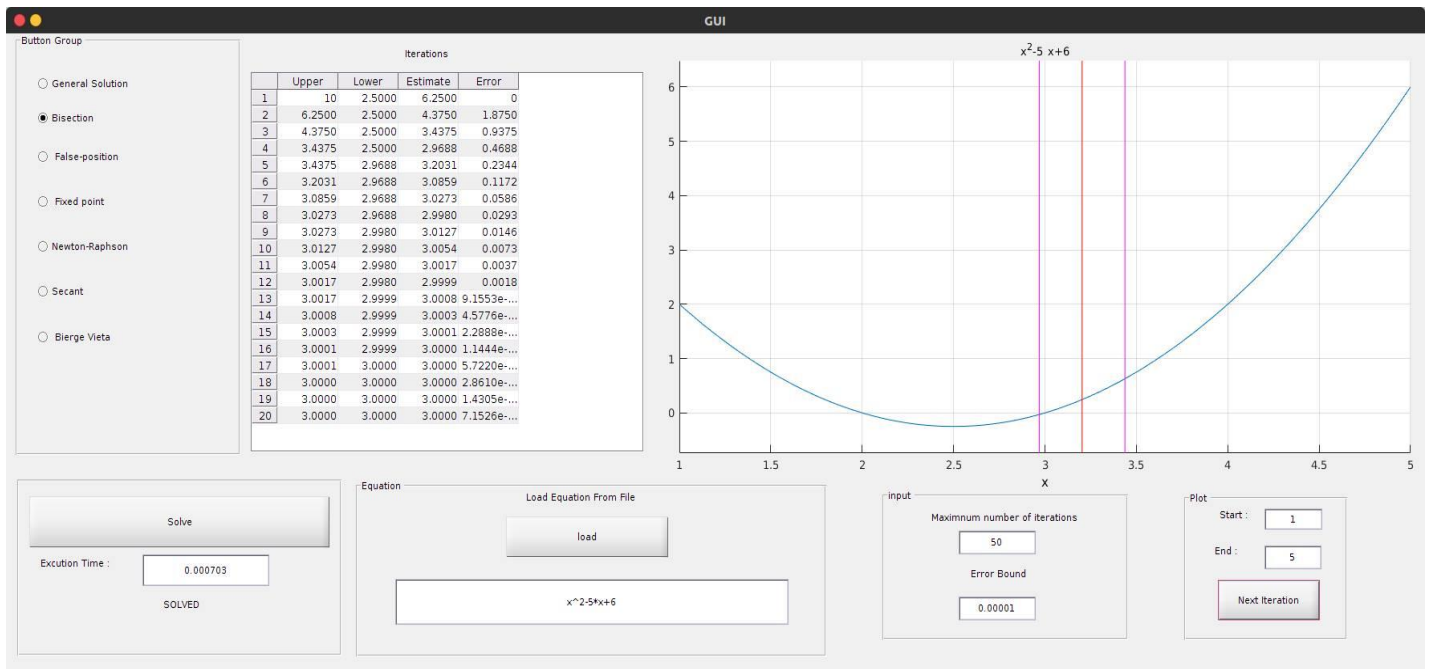
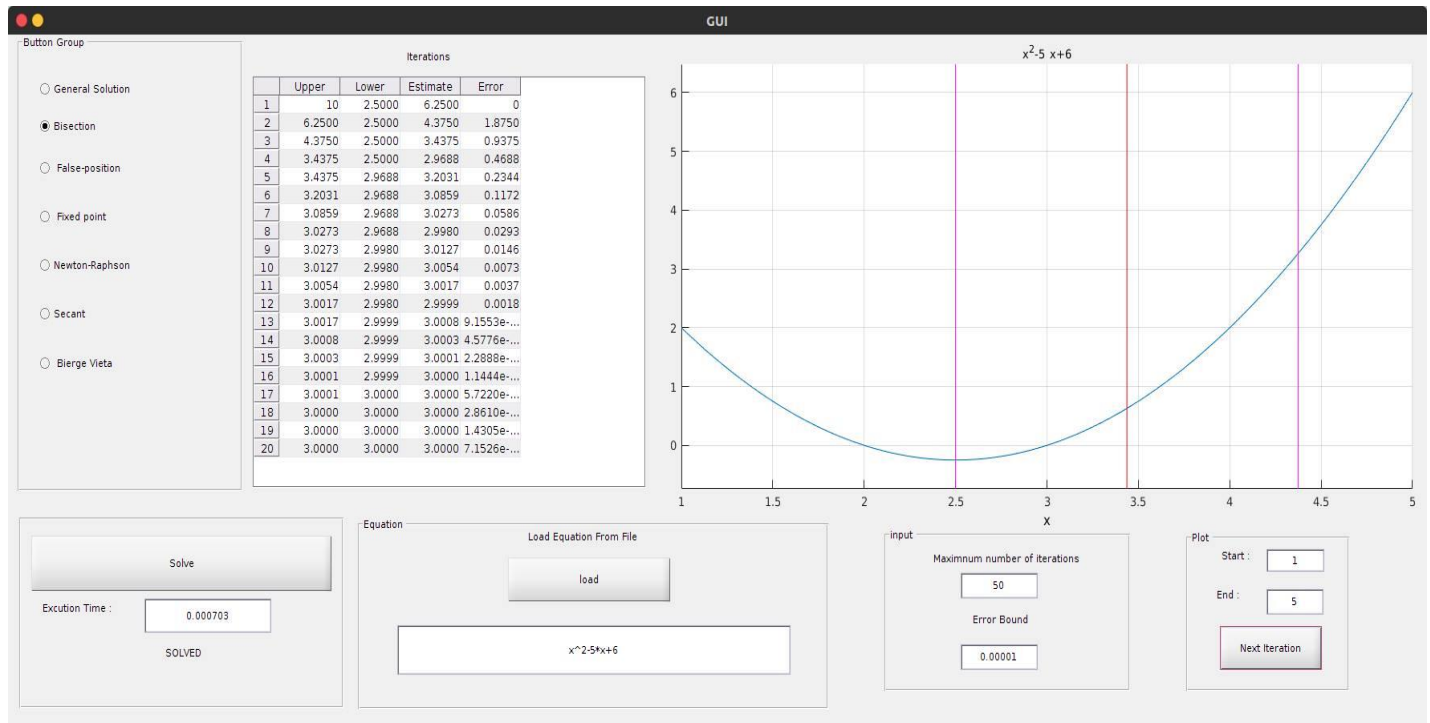
-It depends on the choice of the initial guesses :

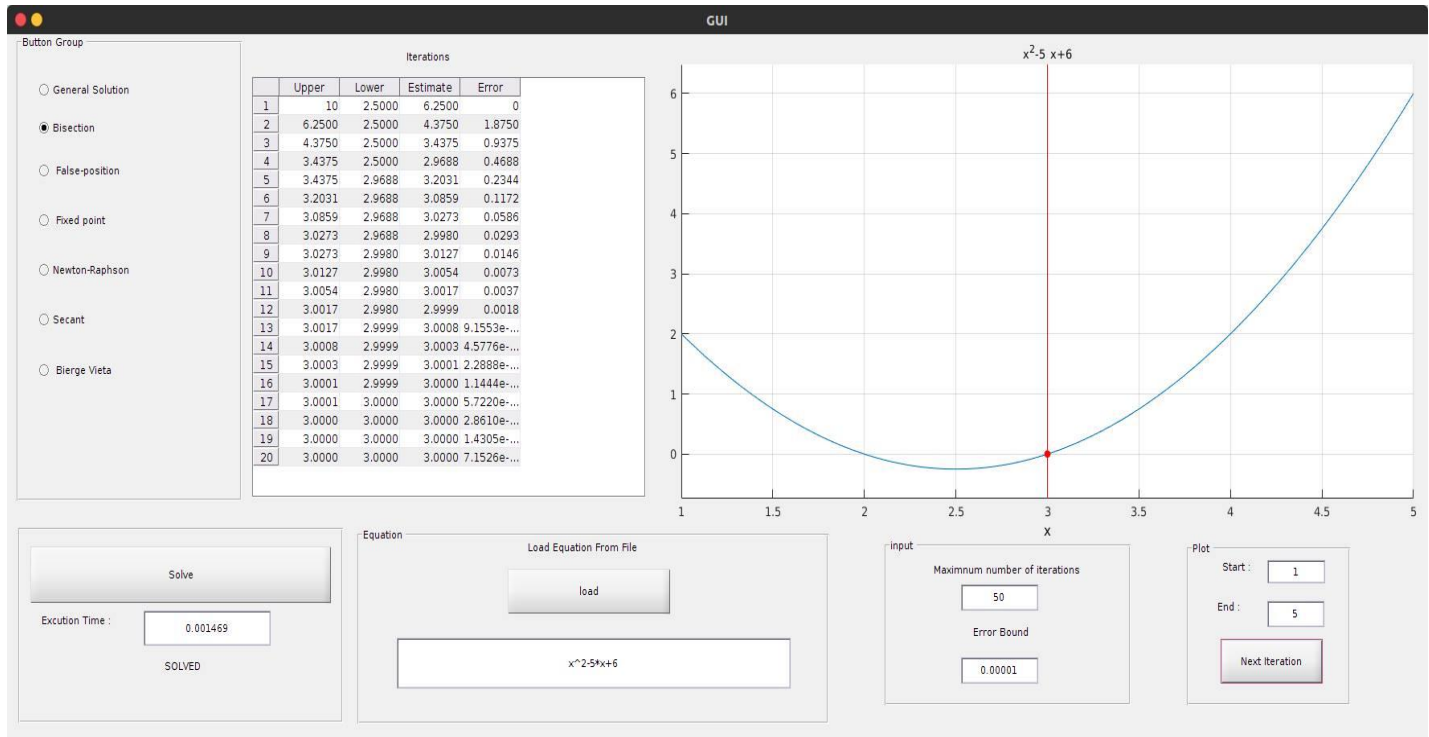
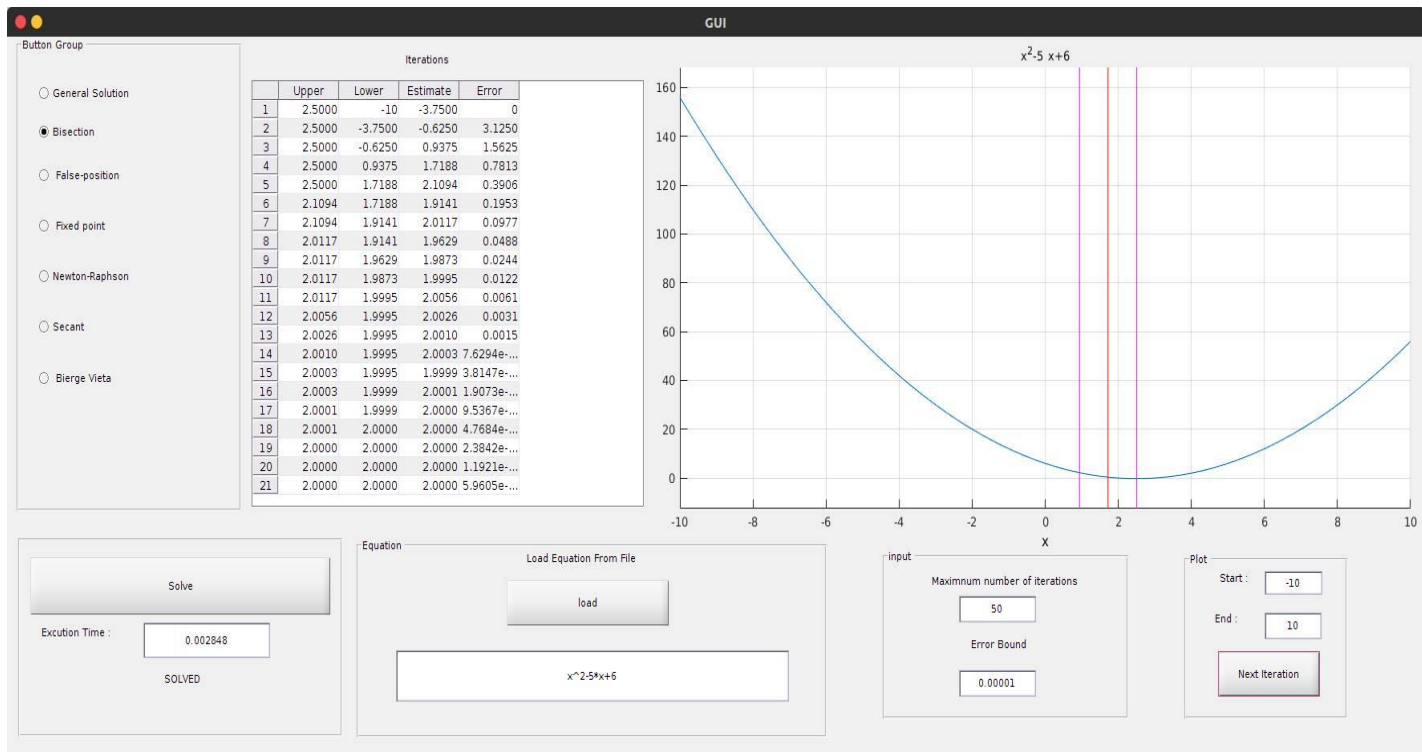
-If initial guesses are correct then, it converges with number of iterations equals :

$$k \geq \log_2 \left| \frac{L_o}{x_l * \epsilon_{es}} \right|$$

-If initial guesses are not correct then, it diverges.

● Sample Runs :





❖ False-position:

• Algorithm:

1. Find a pair of values of x , x_l and x_u such that $f_l = f(x_l) < 0$ and $f_u = f(x_u) > 0$.

2. Estimate the value of the root from the following formula

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

and evaluate $f(x_r)$.

3. Replace one of the original points with the new point :

If $f(x_r) < 0$ then $x_l = x_r \rightarrow f_l = f(x_r)$

If $f(x_r) > 0$ then $x_u = x_r \rightarrow f_u = f(x_r)$

If $f(x_r) = 0$ then the root is found .

4. Check if the new x_u and x_l are close enough for convergence to be declared . If they are not then return to step 2 .

- Pseudo code :

```
function [ y ] = falseposition( upper, lower, es, iMax, equation )
y <= []
fl <= f(lower,equation)
fu <= f(upper,equation)
if fu * fl > 0
    disp('there is no root in this interval')
elseif fu * fl < 0
    lcounter <= 0
    ucounter <= 0
    xu <= upper
    xl <= lower
    xro<=0.0
    ea <= 0.0
    for i = 0:iMax
        if fu == fl
            disp('division by zero')
            return
        end
        xr =((fu * xl) - (fl * xu))/ (fu - fl)
        if i != 0
            ea <= abs(xr - xro)
        end
        fxr <= f(xr,equation)
        test <= fxr * fl
        y <= [y:[xu, xl, xr, ea]]
        if test < 0
            xu <= xr
            fu <= fxr
        end
    end
end
```

```

        lcounter <= lcounter + 1
    else if test > 0
        xl <= xr
        fl <= fxr
        ucounter <= ucounter +1
    else
        break
    if ea < es && il!= 0
        break
    if abs(ucounter-lcounter) > 2
        ucounter <= 0
        lcounter <= 0
        [xu , xl , fu , fl] <= modify_falseposition(xu,xl,fu,fl,equation)
        xro <= xr
else
    if fu == 0
        xr <= upper
    else
        xr <= lower
y <= [y:[upper,lower,xr,ea]]

```

-modifications :

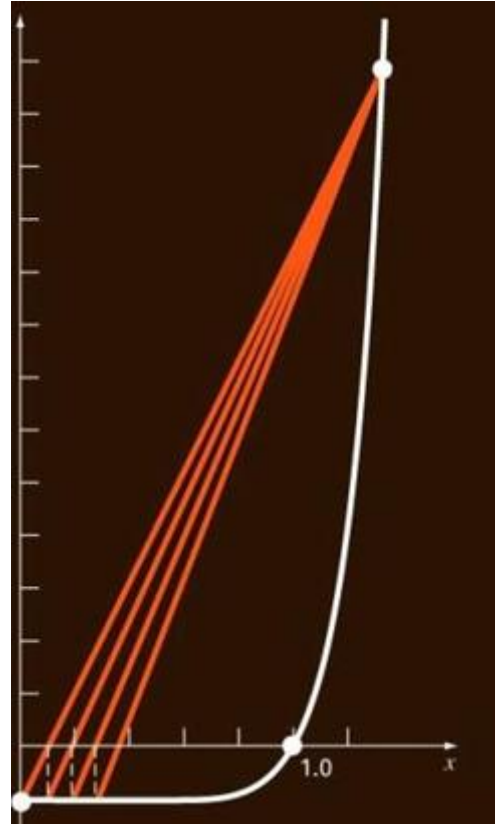
```
function [ xu , xl, fu, fl ] = modify_falseposition( xu, xl, fu, fl,equation)  
    xr <=(upper+lower)/2  
    fxr <= f(xr,equation)  
    test <= fxr * fl  
    if test < 0  
        xu <= xr  
        fu <= fxr  
    else if test > 0  
        xl <= xr  
        fl <= fxr
```

- **Pit falls :**

- Works well but not always .
- Cannot detect continuous functions .
- Initial guesses are required .
- one of bounds might get stuck .

- **Avoiding Pit falls :**

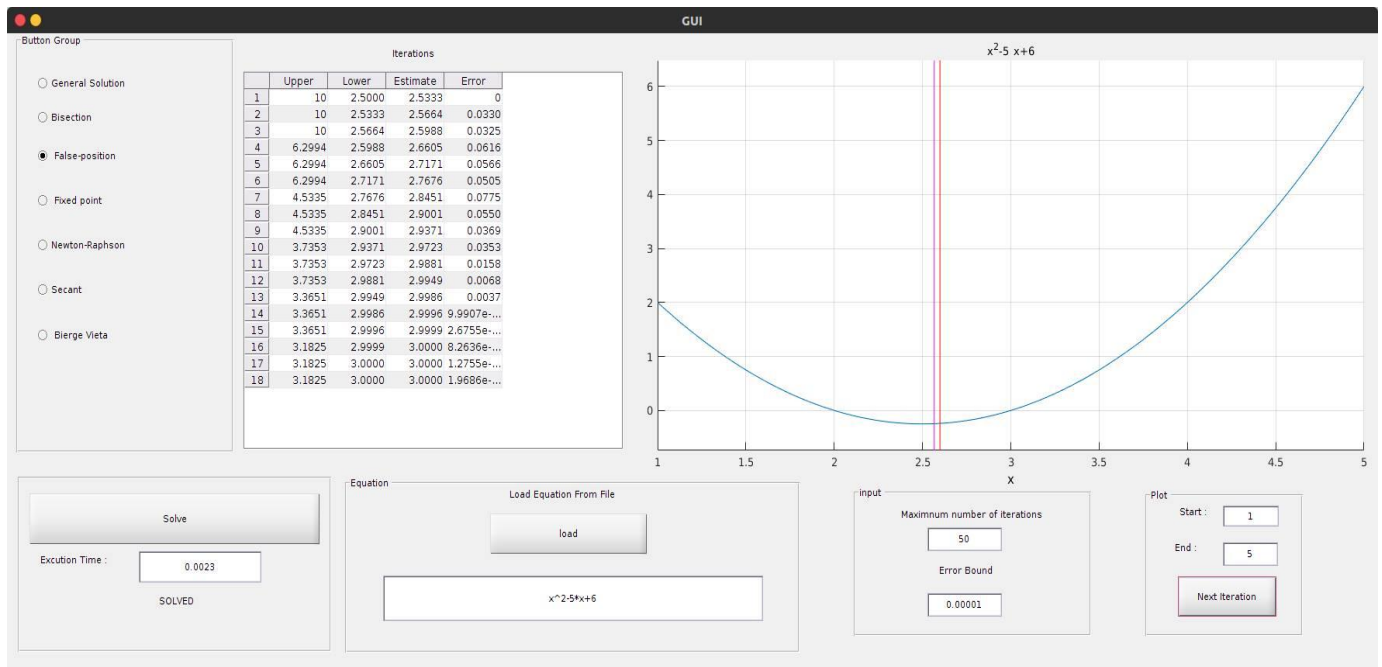
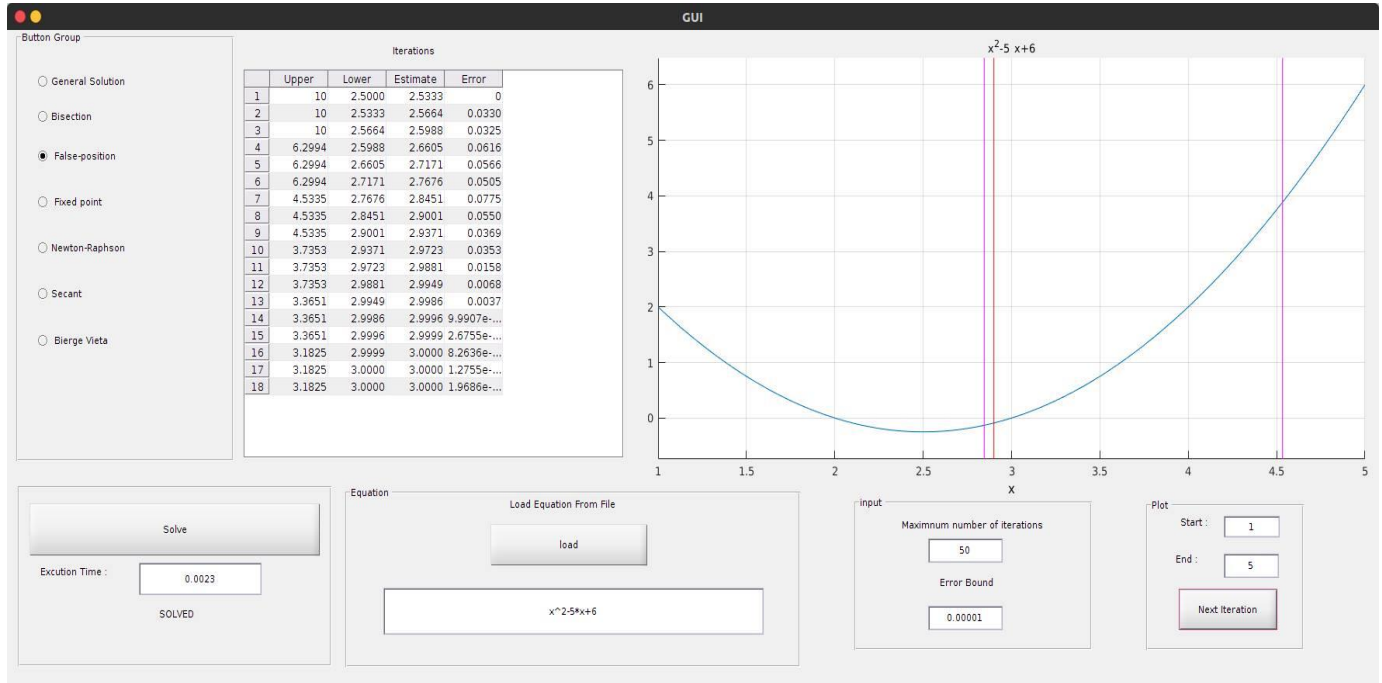
One way to mitigate the "one-sided" nature of the false position is to have the algorithm detect when one of the bounds is stuck. If this occurs, then the original formula of bisection can be used.



- **Divergence and convergence:**

- It depends on the choice of the initial guesses :
 - If initial guesses are correct then, it converges.
 - But number of iterations cannot be detected.
 - If initial guesses are not correct then, it diverges.

● Sample Runs :



❖ Fixed - point :

• Algorithm :

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

$$f(x) = 0 \quad \langle \text{-----} \rangle \quad g(x) = x$$

- If we are able to solve $g(x) = x$, we solve $f(x) = 0$

- We solve $g(x) = x$ by computing $g(x_i) = x_{i+1}$ with x_0 given until x_{i+1} converges to x .

• Pseudo code :

```
function [ y ] <= fixedpoint( xr, es, iMax, equation )
y<= []
syms x
df <= diff(equation, x)
if f(xr,df) > 1
    disp('the function diverges')
    return
for i = 0:iMax
    x_old <= xr
    xr <= f(xr,equation)
    ea <= abs(xr-x_old)
    y <= [y;[xr, ea]]
    if ea < es
        break
```

● Pit falls :

- Finding the magical formula that will converge is the only pit fall .

● Avoiding Pit falls :

-By finding the first derivative then substitute by the initial guess ,If the result is greater than 1 then it will converge .

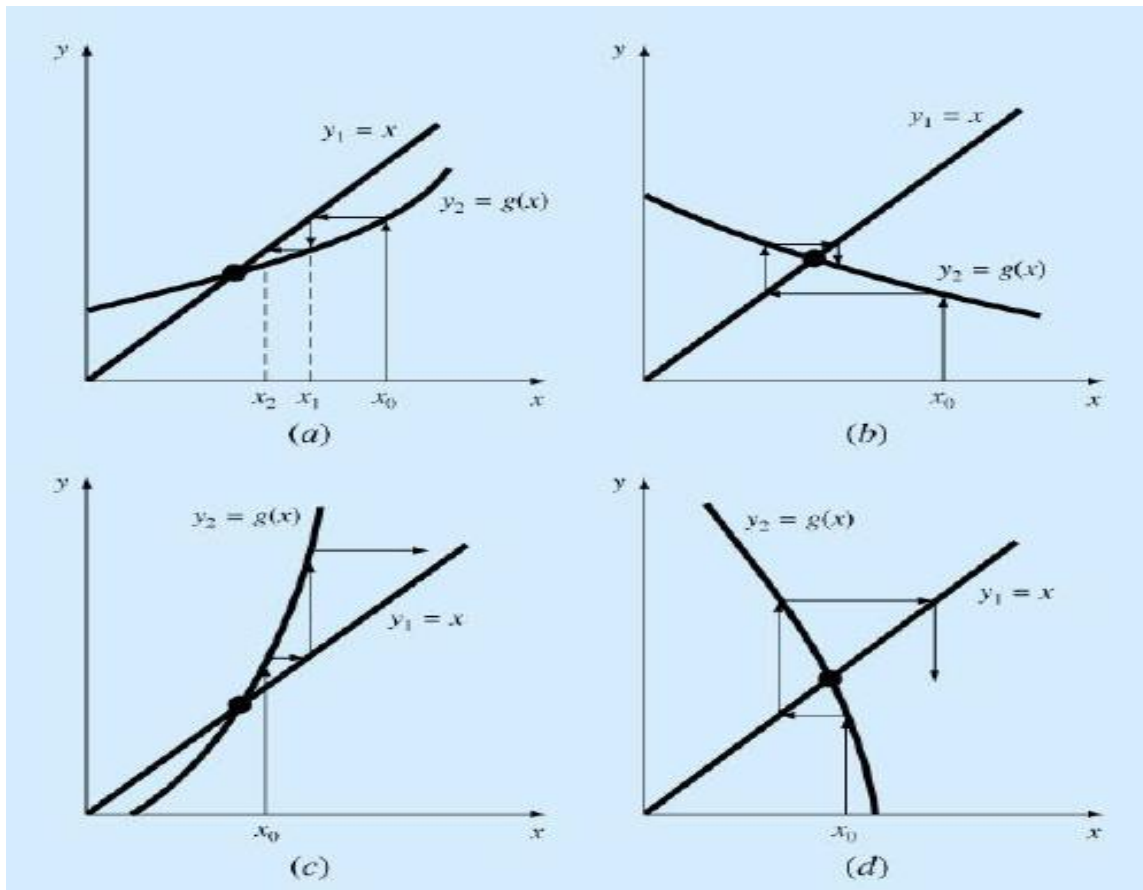
● Divergence and convergence :

a. $|g'(x)| < 1$, $g'(x)$ is positive . -----> converge , monotonic.

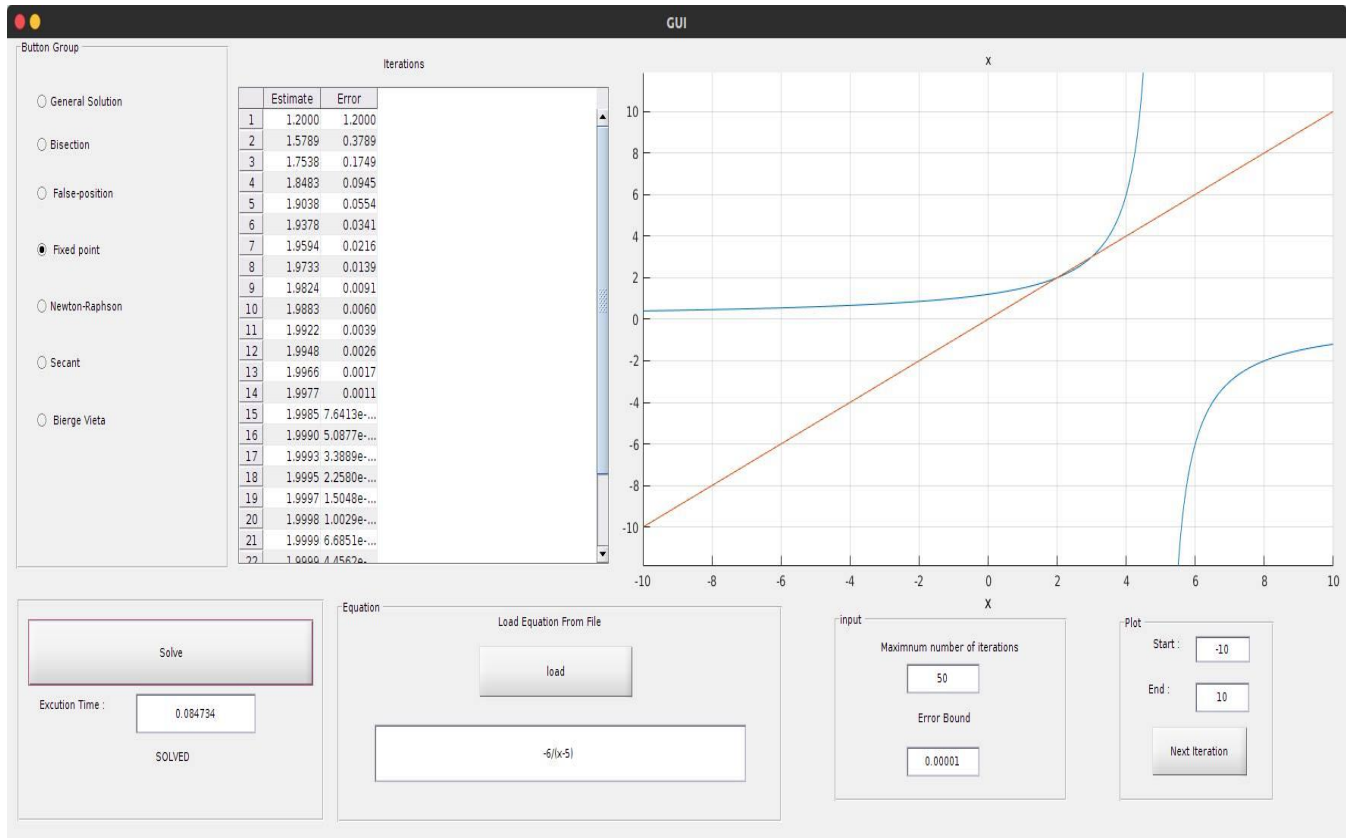
b. $|g'(x)| < 1$, $g'(x)$ is negative -----> converge , oscillate.

c. $|g'(x)| > 1$, $g'(x)$ is positive . -----> diverge , monotonic.

d. $|g'(x)| > 1$, $g'(x)$ is negative . -----> diverge , monotonic.



• Sample Runs :



❖ Newton - Raphson :

- 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} , as the following :

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

3. Find the absolute relative approximate error $|\mathcal{E}_a|$, as the following :

$$|\mathcal{E}_a| = \left| \frac{x_{i+1} - x_i}{x_{i+1}} \right| * 100$$

4. compare the absolute relative approximate error with the pre-specified relative error tolerance \mathcal{E}_s .

$|\mathcal{E}_a| > \mathcal{E}_s$: yes -----> return to step2

No -----> stop .

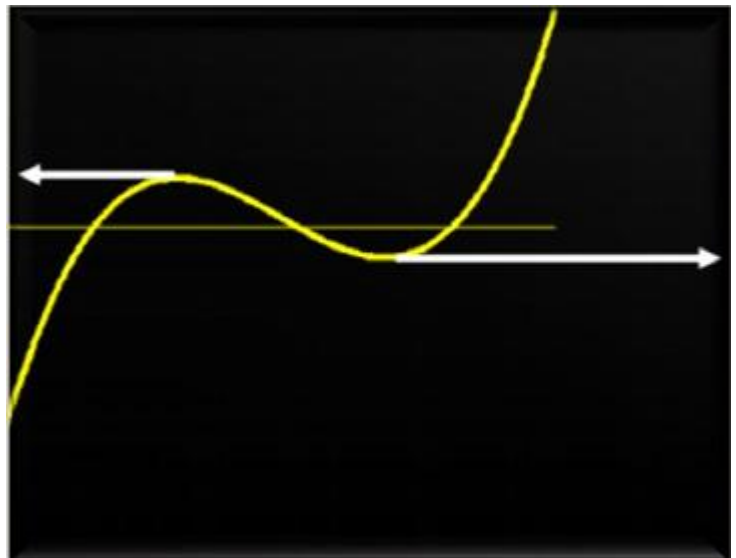
-Also , check if the number of iterations has exceeded the maximum number of iterations allowed ,if so stop the algorithm and notify the user .

- Pseudo code :

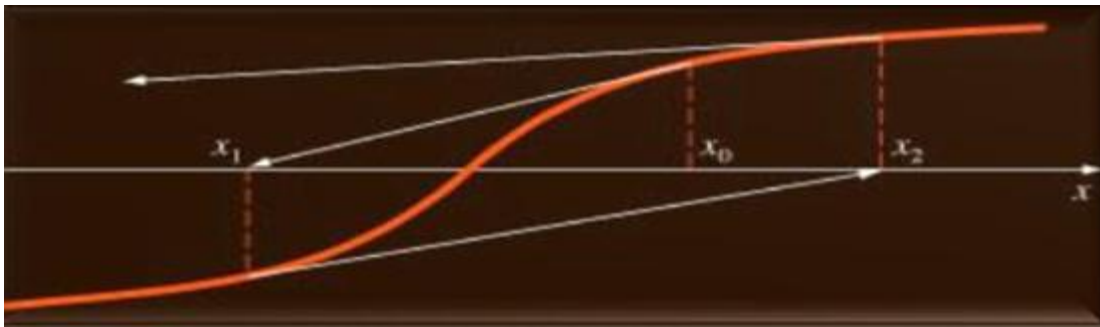
```
function [ y ] = newtonraphson( xr, es, iMax, equation )
y <= []
syms x
df <= diff(equation,x)
for i = 0:iMax
    x_old <= xr
    xprime <= f(x_old,df)
    if xprime == 0
        disp('Division by zero')
        break
    xr <= x_old - f(x_old,equation)/xprime
    ea <= abs(xr-x_old)
    y <= [y:[xr, ea]]
    if ea < es
        break
    end
end
```

- Pit Fall :

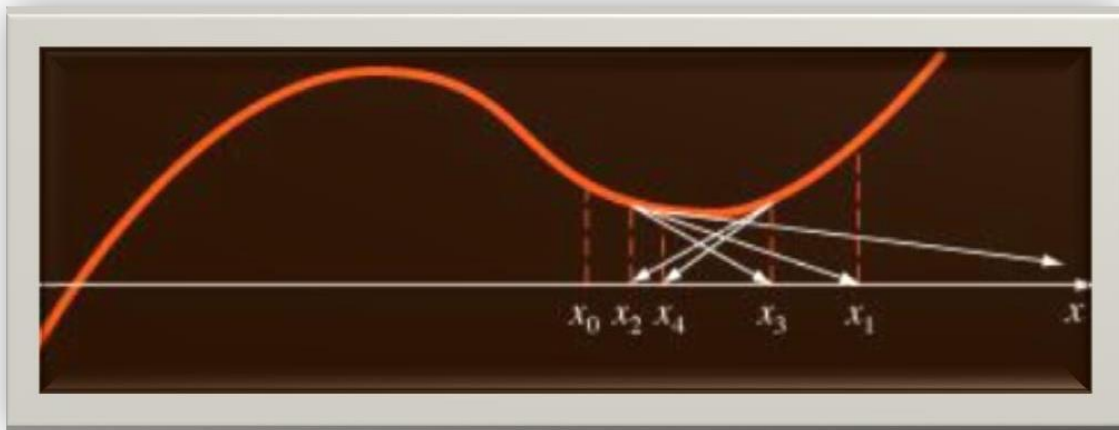
- Division by zero .



- An inflection point ($f''(x)=0$) at the vicinity of a root.



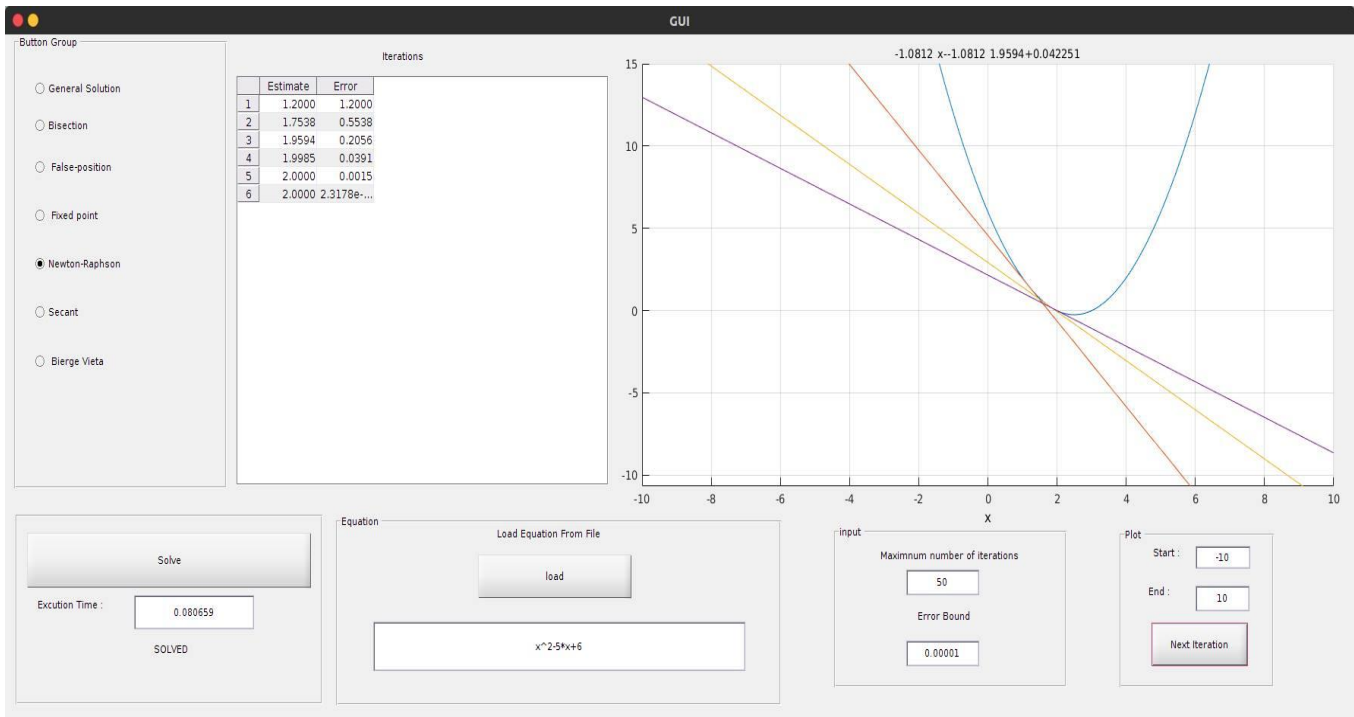
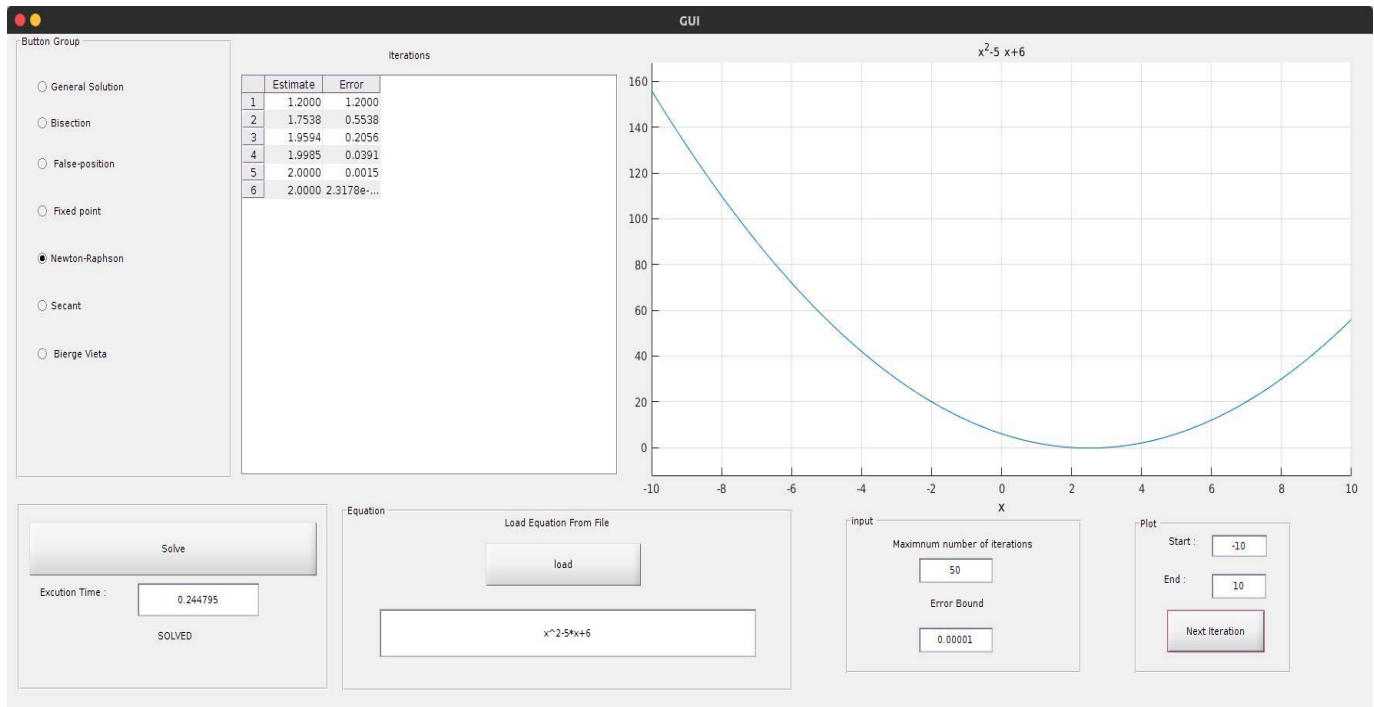
- A local maximum or minimum causes oscillations.

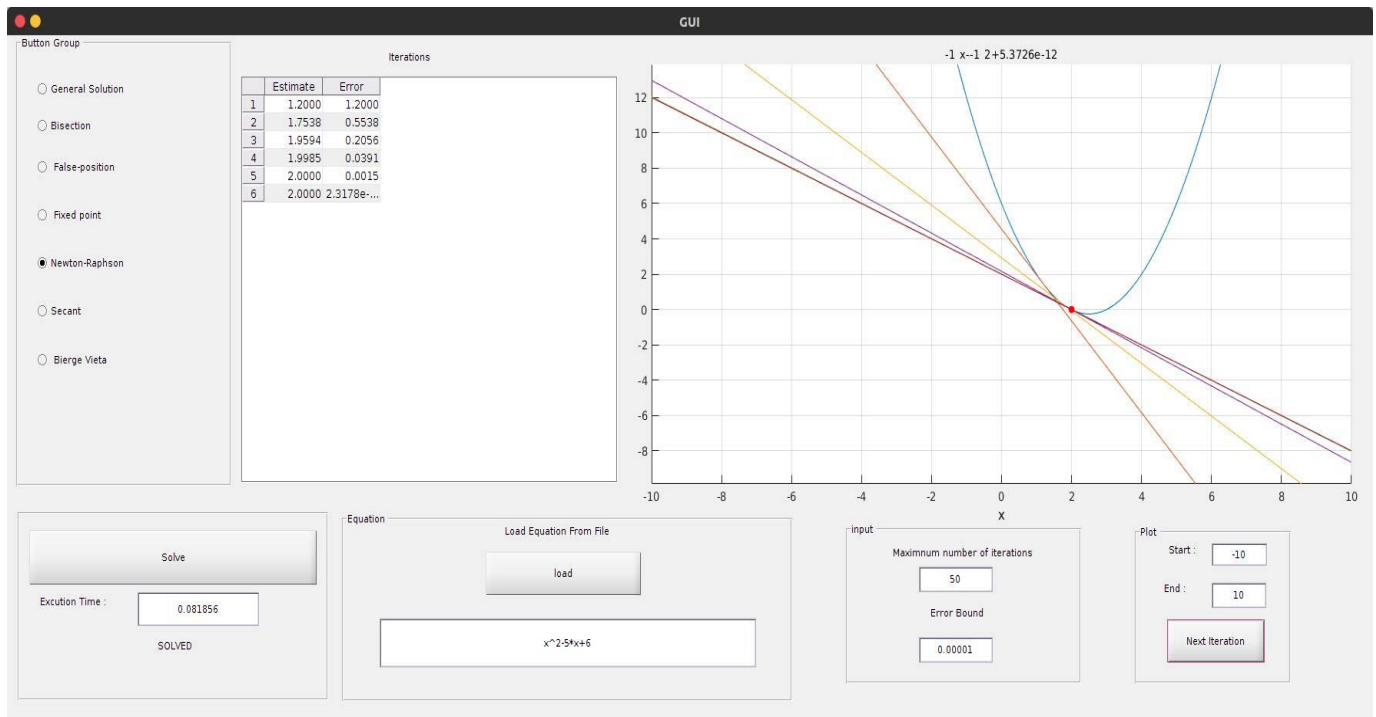


- **Divergence and convergence :**

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.

• Sample Runs :





❖ Secant :

- Algorithm :

-Newton - Raphson method needs to compute the derivatives . But the secant method approximate the derivatives by finite divided difference .

- Pseudo code :

```
function [ y ] = secant( x0, x1, es, iMax, equation )
```

```
y <= []
```

```
for i = 0:iMax
```

```
    fx1 <= f(x1,equation)
```

```
    fx0 <= f(x0,equation)
```

```
    if fx0 == fx1
```

```
        disp('division by zero')
```

```
        return
```

```
    xr <= x1 - ((fx1 * (x0-x1))/(fx0-fx1))
```

```
    ea <= abs(xr-x1)
```

```
    y <= [y;[xr, x1, x0, ea]]
```

```
    x0 <= x1
```

```
    x1 <= xr
```

```
    if ea < es
```

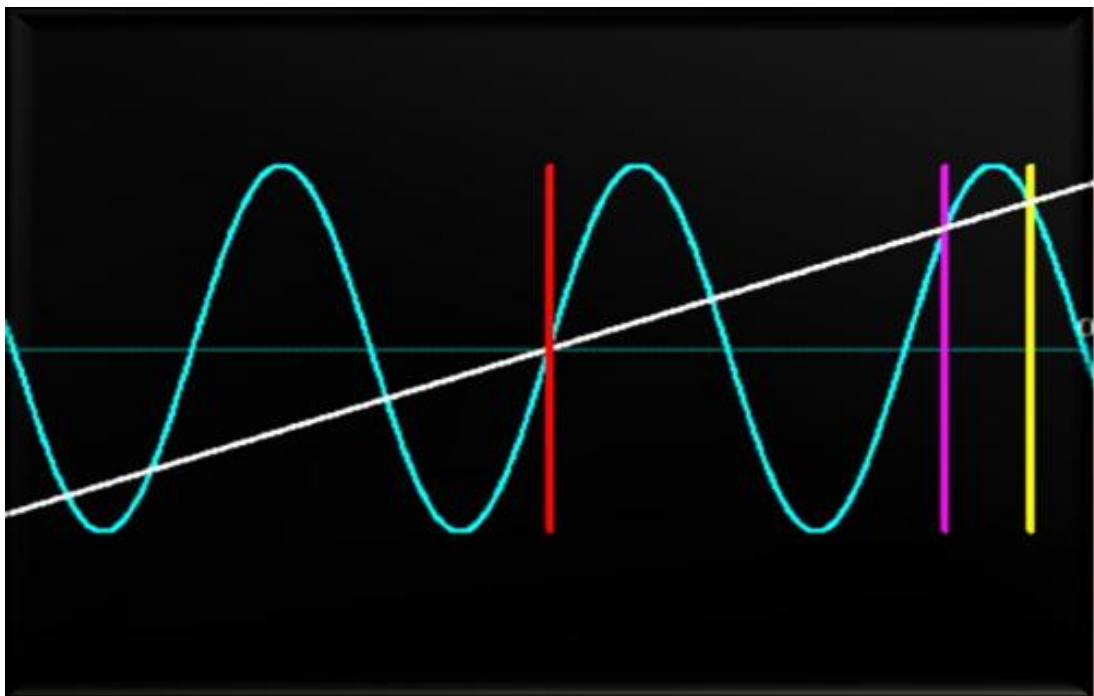
```
        break
```

❖ Pit falls :

- Division by zero .



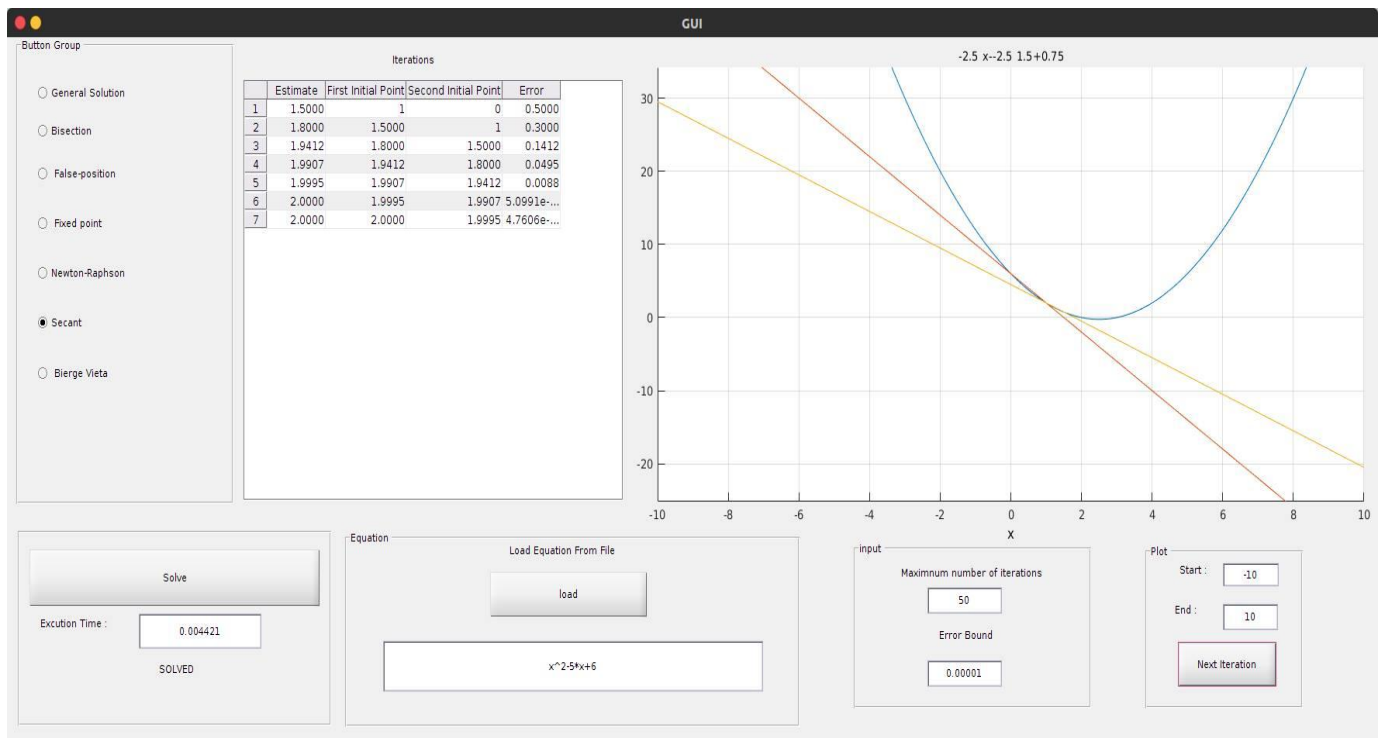
-Root jumping .

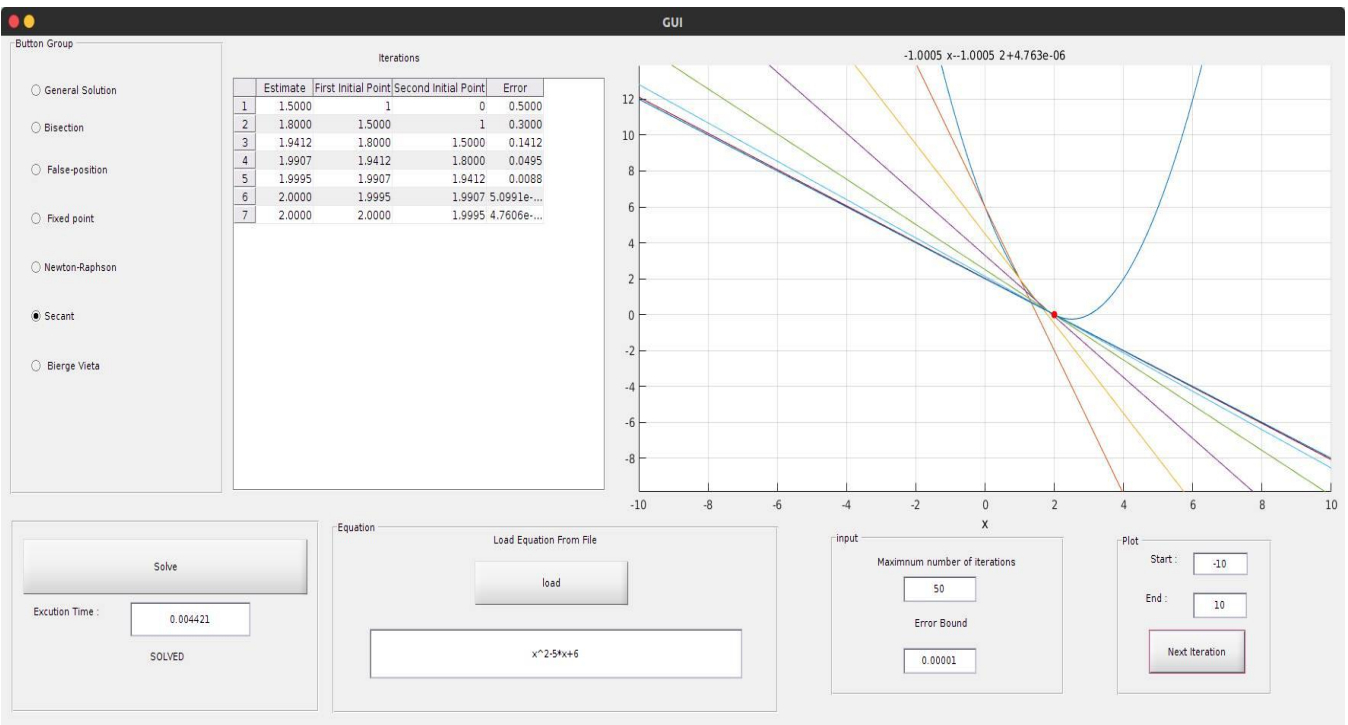
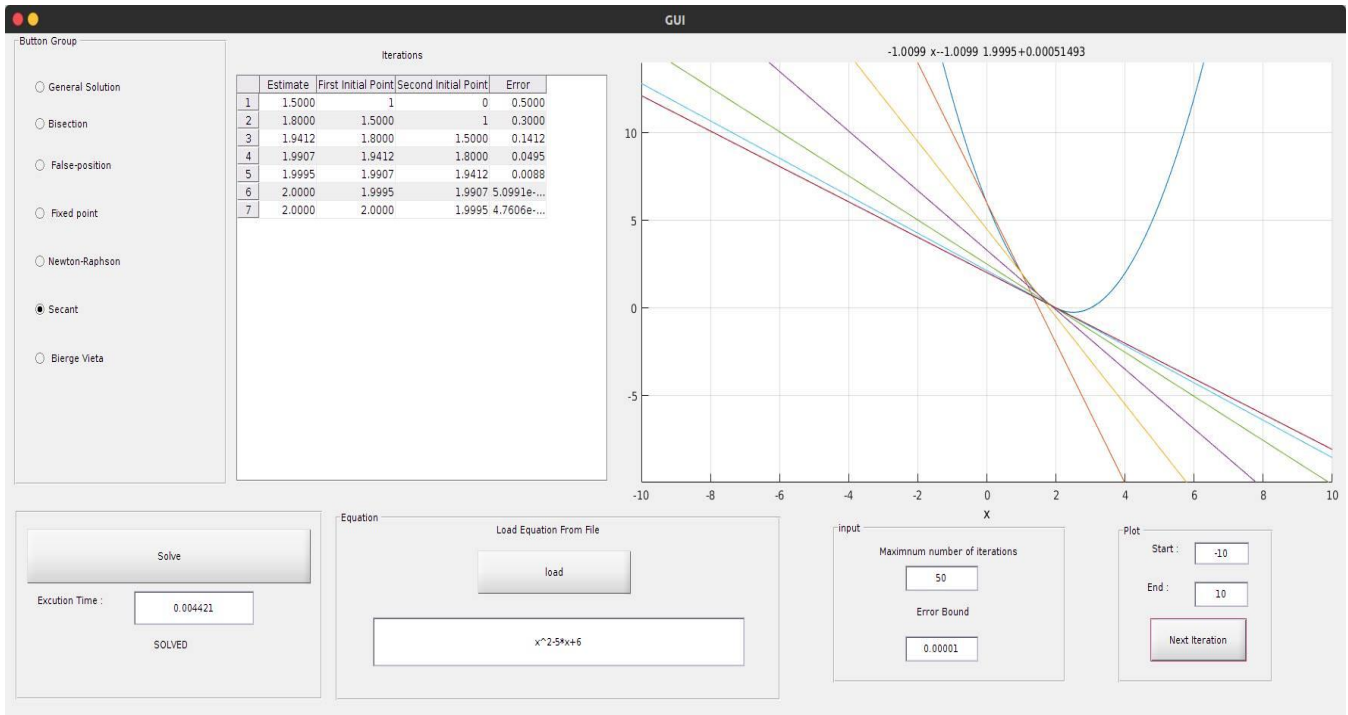


❖ Divergence and convergence :

- If it converged it will converge at quadratic rate.
- No guarantee for convergence.

❖ Sample Runs :





❖ Bierge vieta :

❖ Algorithm :

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

$$f(x) = a_0 + a_1x + a_2x^2 + \dots + a_mx^m = (x-r)(b_1 + b_2x + \dots + b_mx^{m-1}) + b_0 = (x-r)h(x) + b_0$$

$$b_m = a_m$$

$$b_j = a_j + rb_{j+1} \quad j = m-1, m-2, \dots, 1, 0$$

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

$$f'(r) = h(r)$$

$$h(x) = b_1 + b_2x + \dots + b_mx^{m-1} = (x-r)(c_2 + c_3x + \dots + c_mx^{m-2}) + c_1$$

$$c_m = b_m$$

$$c_i = b_i + rc_{i+1} \quad j = m-1, m-2, \dots, 1$$

$$f(r) = b_0$$

$$f'(r) = h(r) = c_1$$

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

$$f(r) = b_0$$

$$f'(r) = h(r) = c_1$$

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

$$c_m = b_m = a_m$$

$$b_j = a_j + x_ib_{j+1}$$

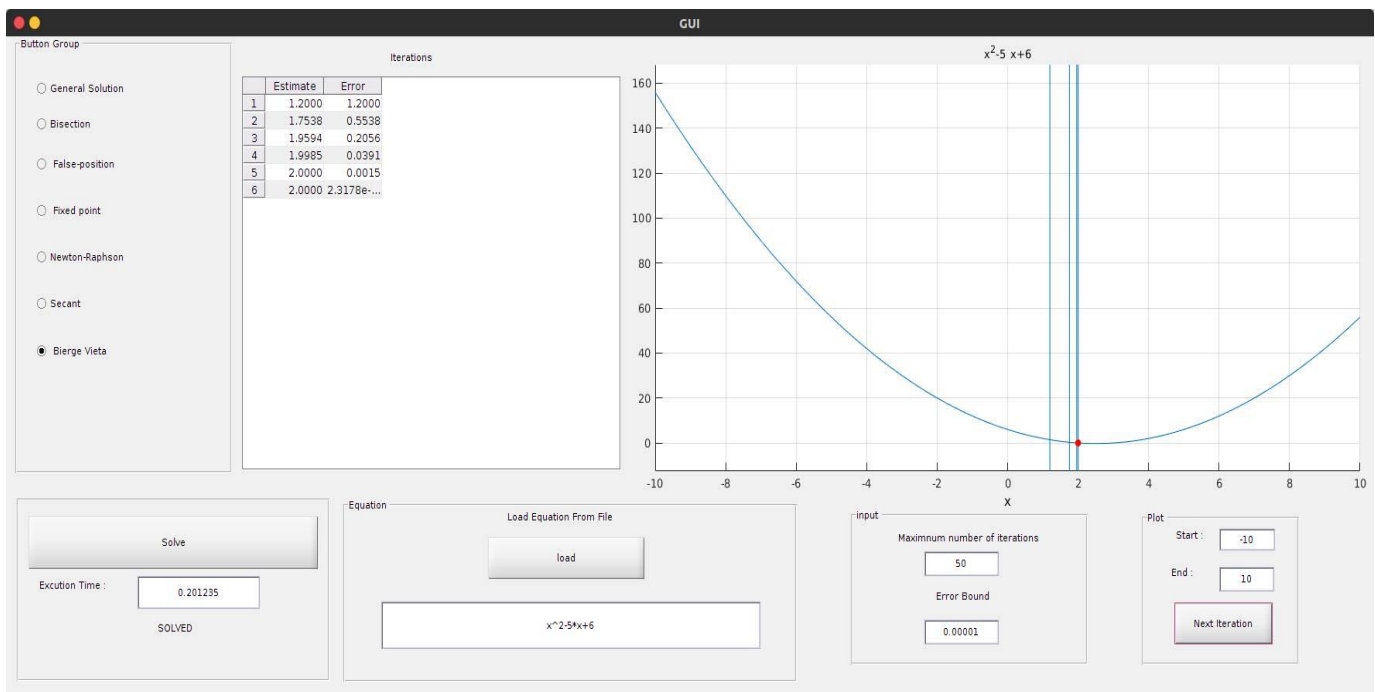
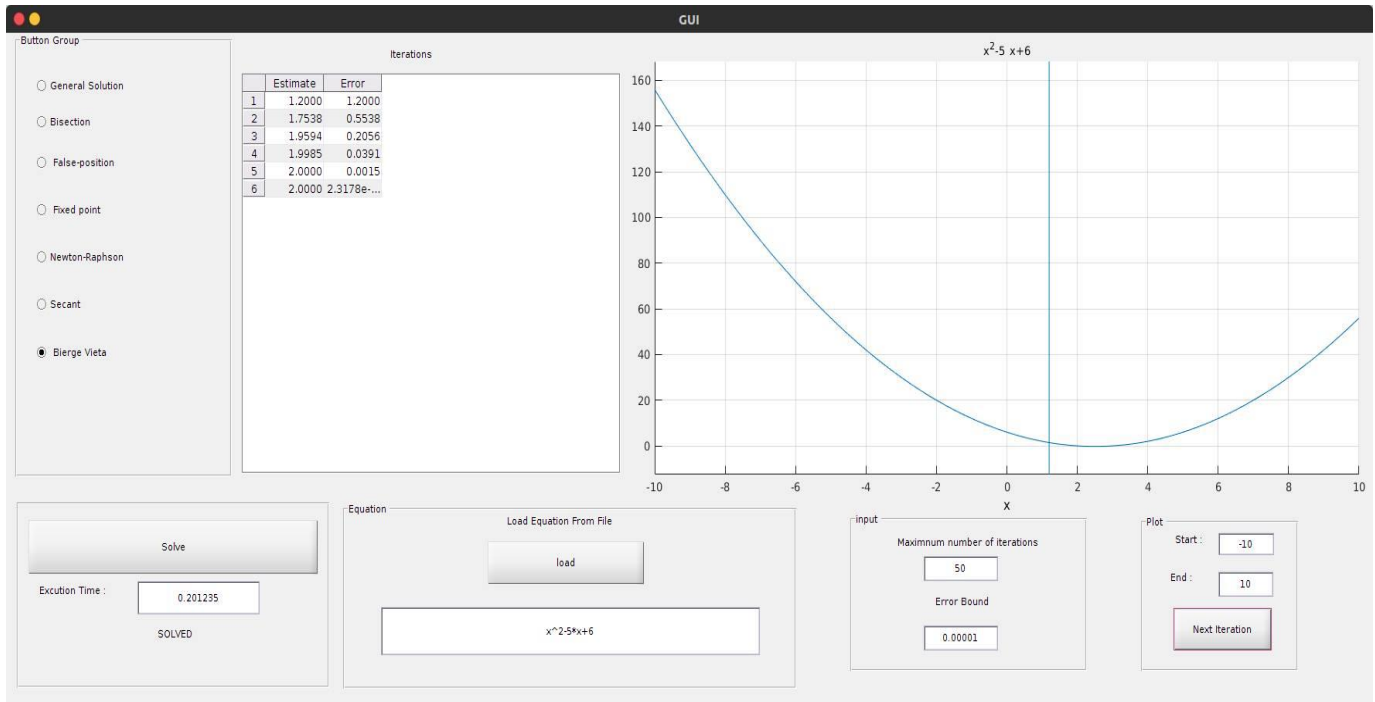
$$c_j = b_j + x_ic_{j+1} \quad j = m-1, m-2, \dots, 1$$

$$b_0 = a_0 + x_ib_1$$

❖ Pseudo code :

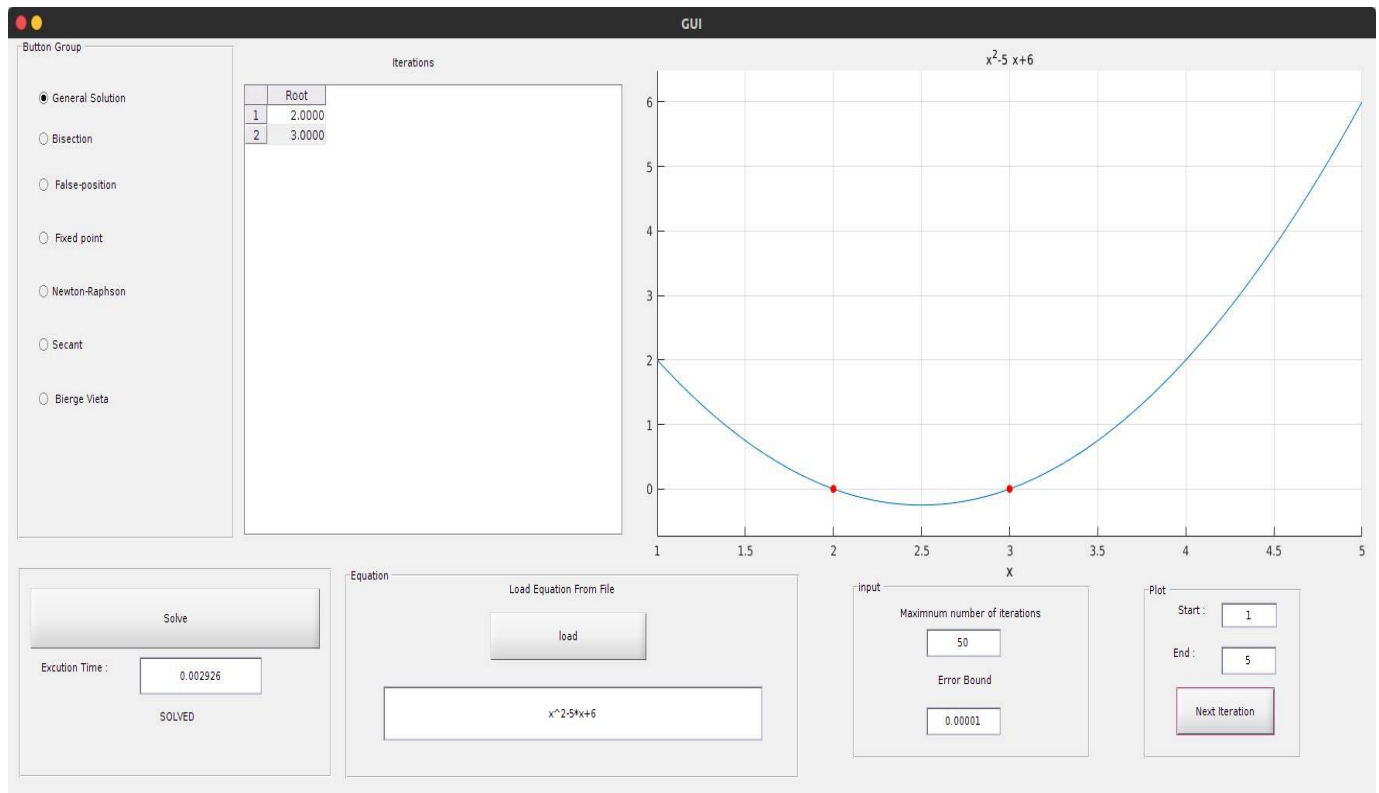
```
function [ y ] = biergevieta( xr, es, iMax, equation )
y = []
syms x
p = inline(equation)
table = transpose(sym2poly(p(x)))
n = size(table,1)
table = [table,zeros(n,2)]
for k = 0:iMax
    table(1,2) = table(1,1)
    table(1,3) = table(1,1)
    for j = 2:3
        for i = 2:n
            table(i,j) = xr * table(i-1,j) + table(i,j-1)
        end
    end
    fx = double(table(n,2))
    dfx = double(table(n-1,3))
    if dfx == 0
        disp('division by zero!')
    end
    return
    x_old = xr
    xr = x_old - fx/dfx
    ea = double(abs(xr-x_old))
    y = [y;[xr,ea]]
    if ea < es
        break
    end
end
```

• Sample runs :



❖ General solution :

Sample runs :



Part 2

❖ User manual :

GUI

Number of points Number of queries

Attach points file Attach query file

	x	y
1	0	0
2	0	0

Query	F(Query)
0	0

Methods
☒ Newton ☐ Lagrange

Calculate

Execution Time

Interpolated Formula

PLOT

f(x)

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

1. Enter number of points to be given .

2. Enter the points.

Number of points

	x	y
1	0	0
2	0	0

3. Enter the number of queries

Number of queries

4. Enter the queries required to be calculated.

Query	F(Query)
0	0

Query (input) →

→ F(Query) (Output)

5. Choose the method of solving(Newton or Lagrange).

Methods

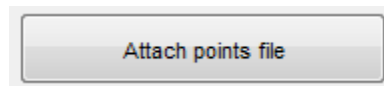
☒ Newton ☐ Lagrange

6. Submit your inputs and begin the calculations by pressing the calculate button .

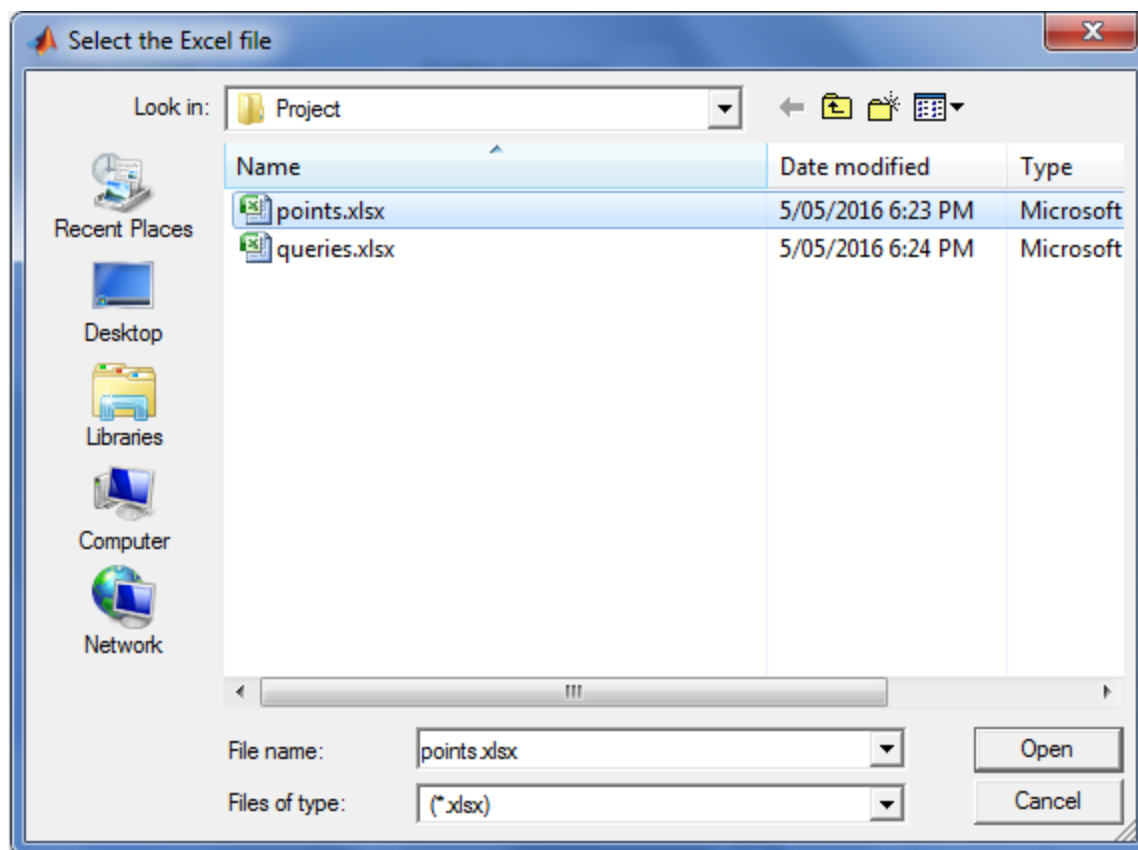
Calculate

⇒ IF it is required to enter the points and queries through out an external files:

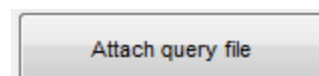
1. Press "Attach point File" button .



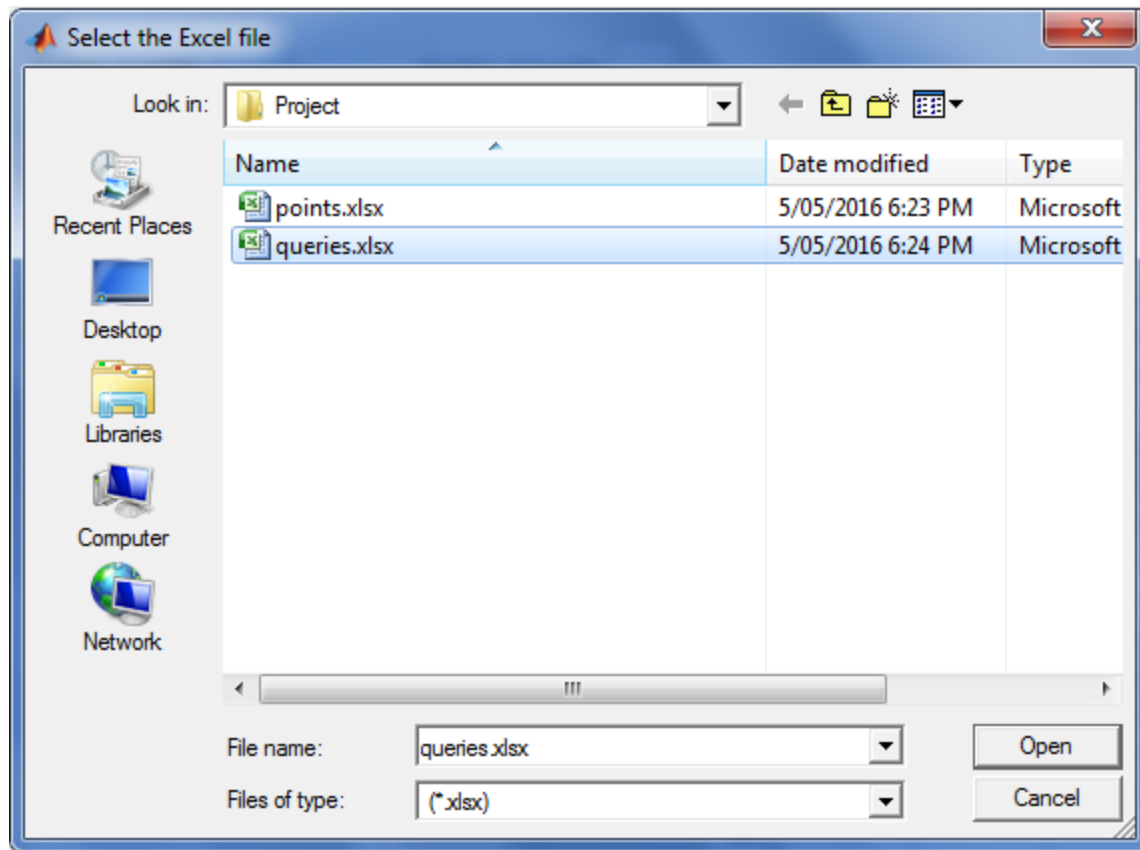
2. Select the file from the file chooser .



3. Press " Attach query file" button.



4. Select the file from the file chooser .



5. Complete as the previous , from step 5.

❖ Algorithm:

⇒ Newton :

we can simply describe the algorithm using a matrix (Divided Difference Table).

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

$$\begin{aligned} & f[x_n, x_{n-1}, \dots, x_1, x_0] \\ &= \frac{f[x_n, x_{n-1}, \dots, x_1] - f[x_{n-1}, x_{n-2}, \dots, x_0]}{x_n - x_0} \end{aligned}$$

$$\begin{aligned} f_n(x) = & b_0 + b_1(x - x_0) + b_2(x - x_0)(x - x_1) \\ & + \dots + b_n(x - x_0)(x - x_1) \dots (x - x_{n-1}) \end{aligned}$$

⇒ La Grange:

- By applying these 2 main laws :

$$f_n(x) = \sum_{i=0}^n L_i(x) f_n(x_i)$$

$$L_i(x) = \prod_{\substack{j=0 \\ j \neq i}}^n \frac{x - x_j}{x_i - x_j}$$

❖ Pseudo code :

⇒ Newton :

○ buildArray:

- It is responsible for building the (Divided Difference Table) and returns it 'values'.

```
function values = buildArray(Data, x)
```

```
    values = zeros(x, x + 1) //initialize the matrix 'values' by zero
```

```
    values(:, 1 : 2) = Data(1 : x, 1 : 2)
```

```
    for i = 1 : (x - 1)
```

```
        for j = 1 : (x - i)
```

```
            values(j, i + 2) = (values(j + 1, i + 1) - values(j, i + 1)) / (values(i + j, 1) - values(j, 1))
```

○ Interpolation :

- It is responsible for constructing the $f_n(x)$ equation without substitution by the query point 'fx'.

```
function fx = Interpolation(x, values)
    fx <= num2str(values(1,2))
    for i <= 1 : (x - 1)
        multiplication <= num2str(values(1, i + 2))
        for j <= 1 : i
            multiplication = [multiplication ' * (x - ' (num2str(values(j, 1))) ')']
        end
        fx <= [fx ' + ' multiplication]
    end
    fx <= sym(fx)
```


⇒ La Grange :

○ lagrange:

- It is responsible for constructing the $f_n(x)$ equation without substitution by the query point 'fx'.

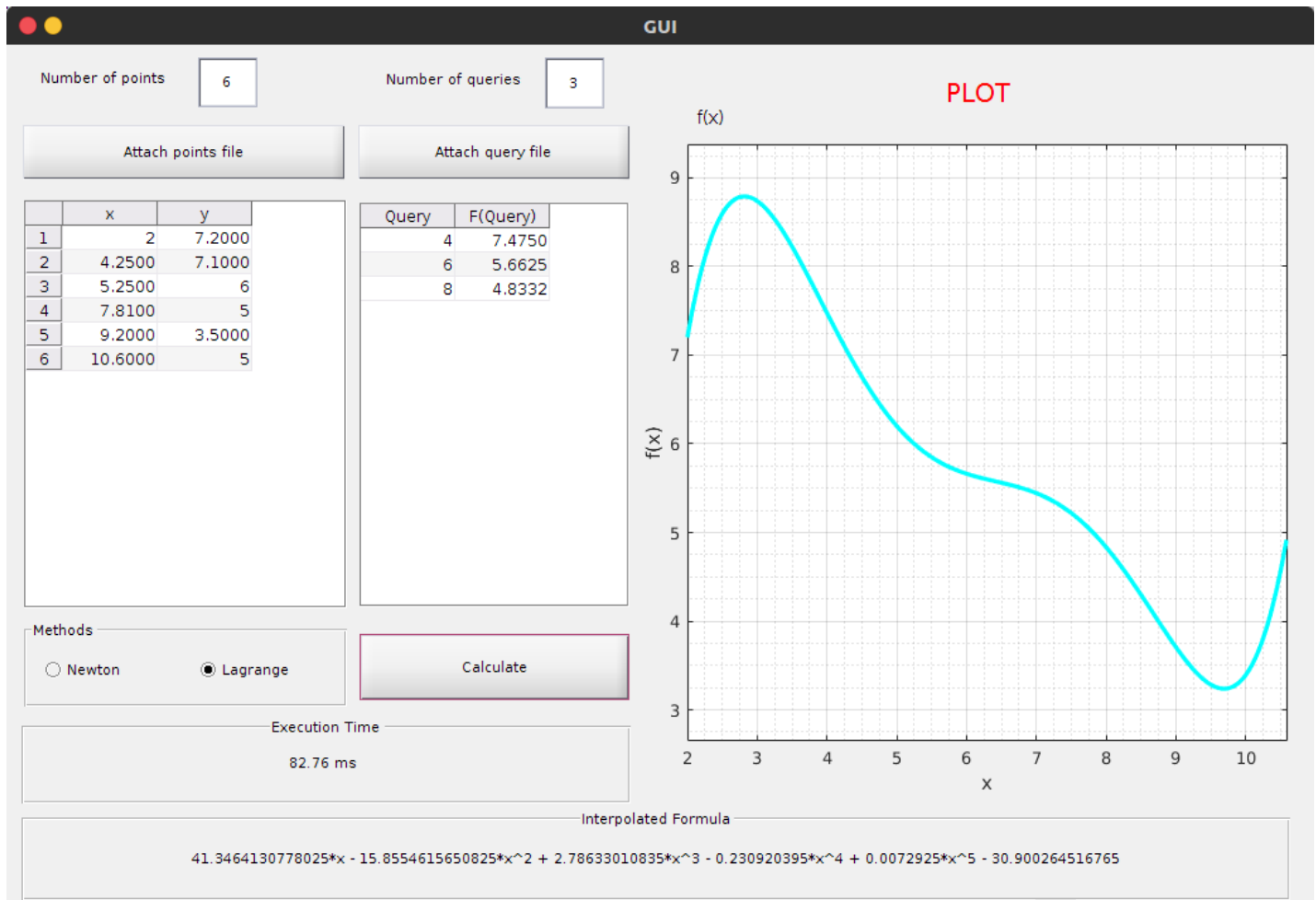
```
function fx = lagrange(Data, n)
    fx <= []
    for i <= 1 : n
        l <= []
        den <= 1
        for j <= 1 : n
            if i ~= j
                den <= den * (Data(i, 1) - Data(j, 1))
            size(l)
            if (size(l) == 0)
                l <= ['(x - ' (num2str(Data(j, 1))) ')']
            else
                l <= [l ' * (x - ' (num2str(Data(j, 1))) ')']
            if (size(fx) == 0)
                fx <= [num2str(Data(i, 2)/den) ' * ' l]
            else
                fx <= [fx ' + ' num2str(Data(i, 2)/den) ' * ' l]
        end
    end
    fx <= sym(fx);
```



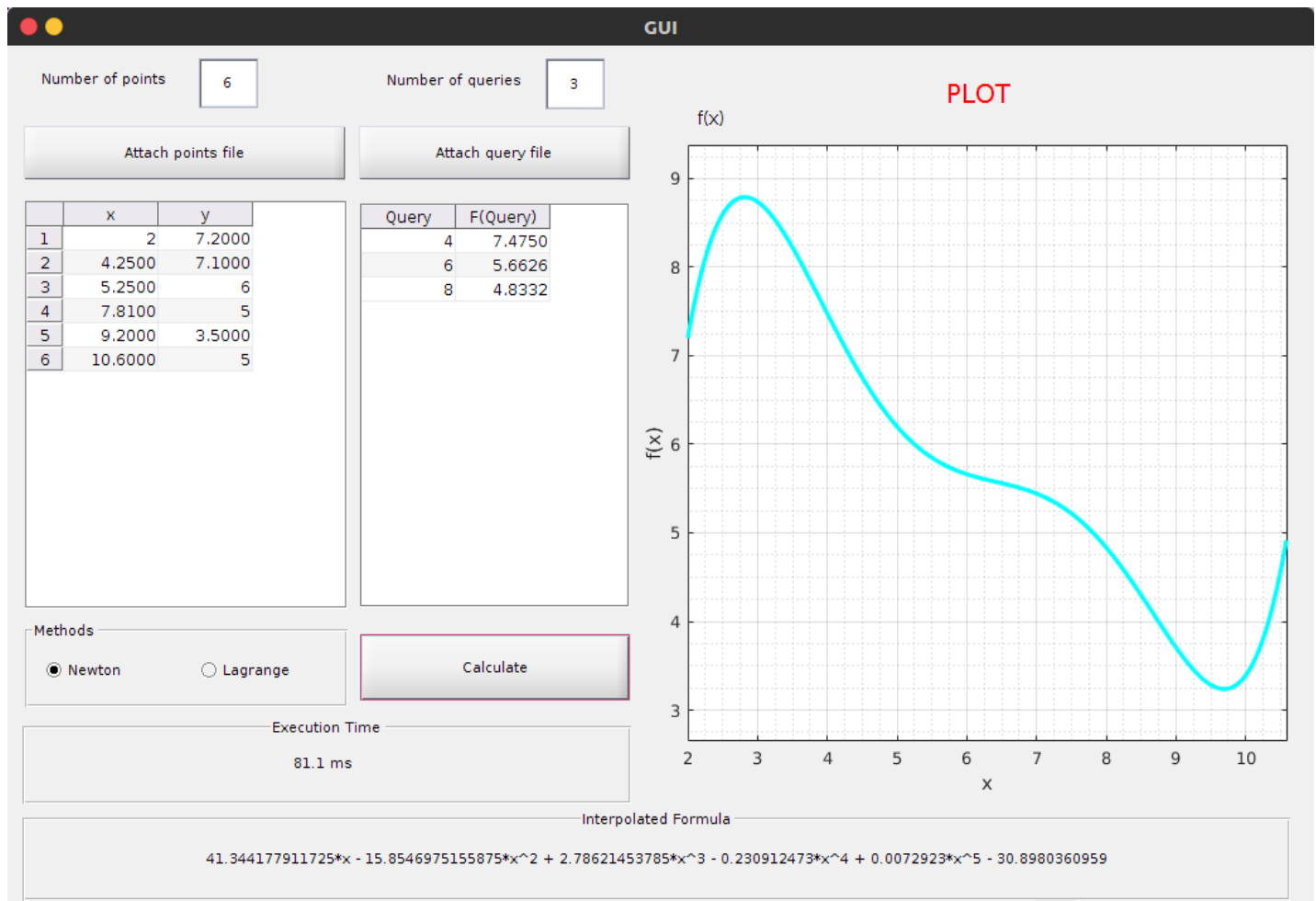
Sample Runs :

- sample run (1) :

⇒ la grange method:

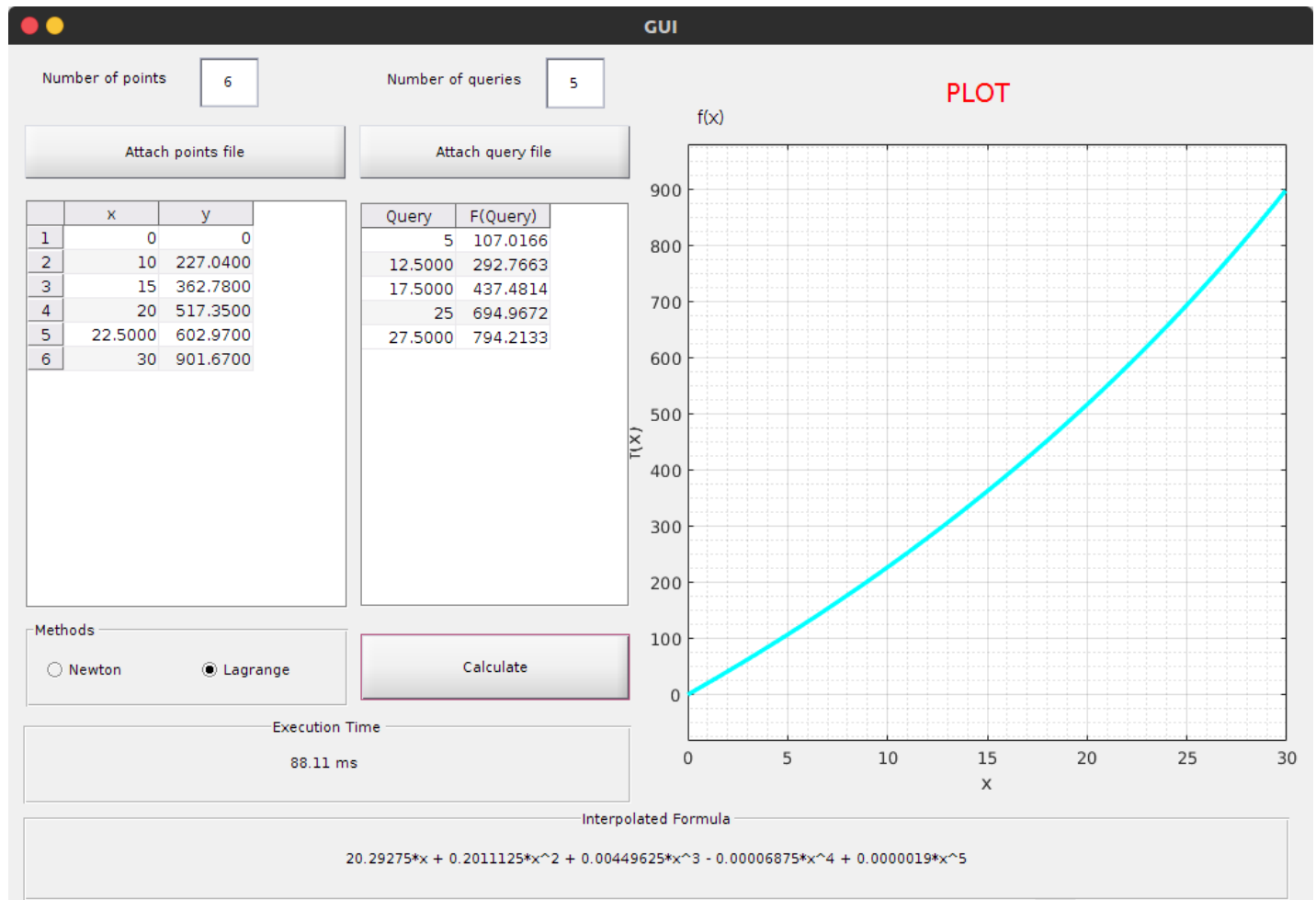


⇒ Newton method:

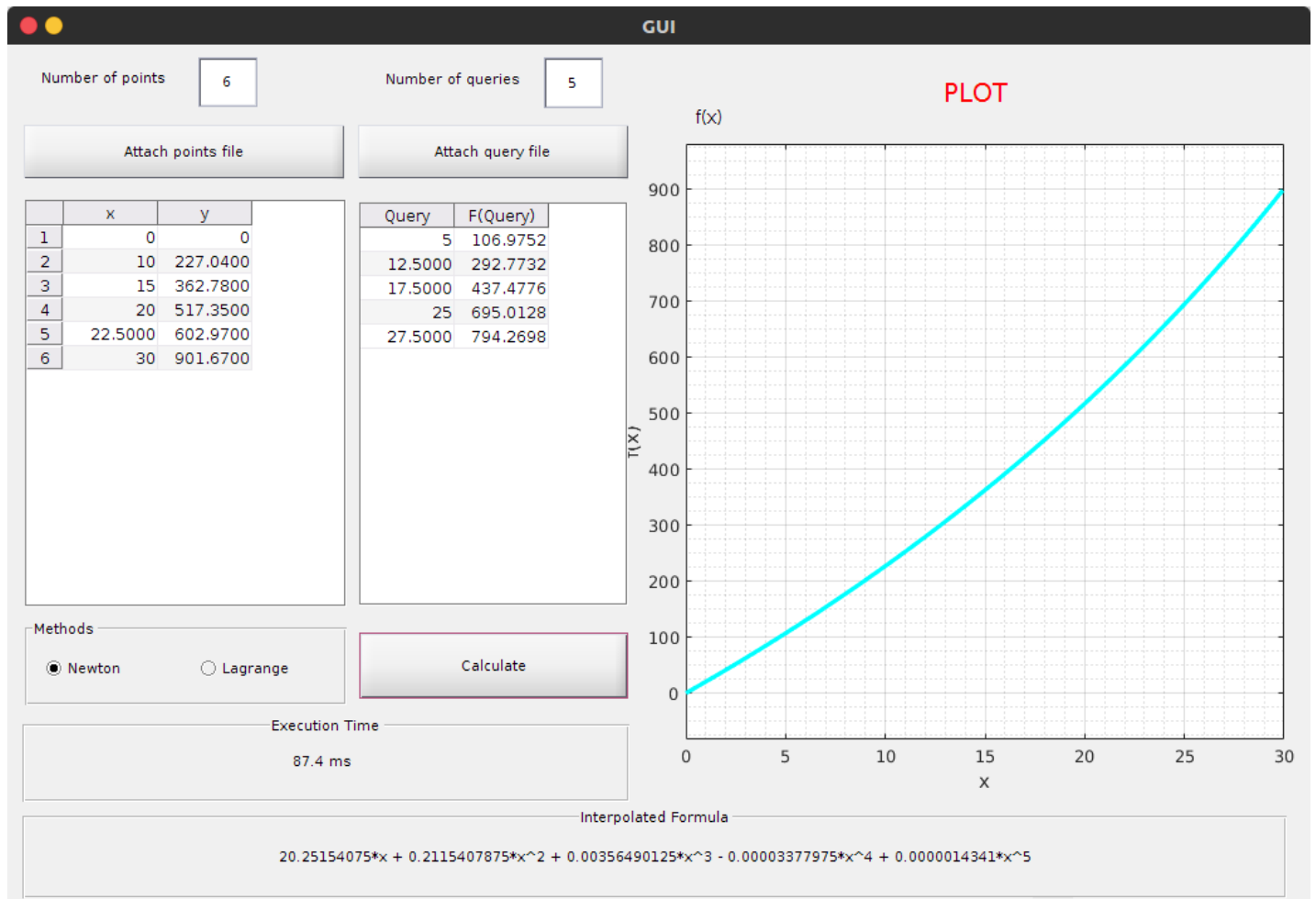


- sample run (2):

⇒ La grange method :

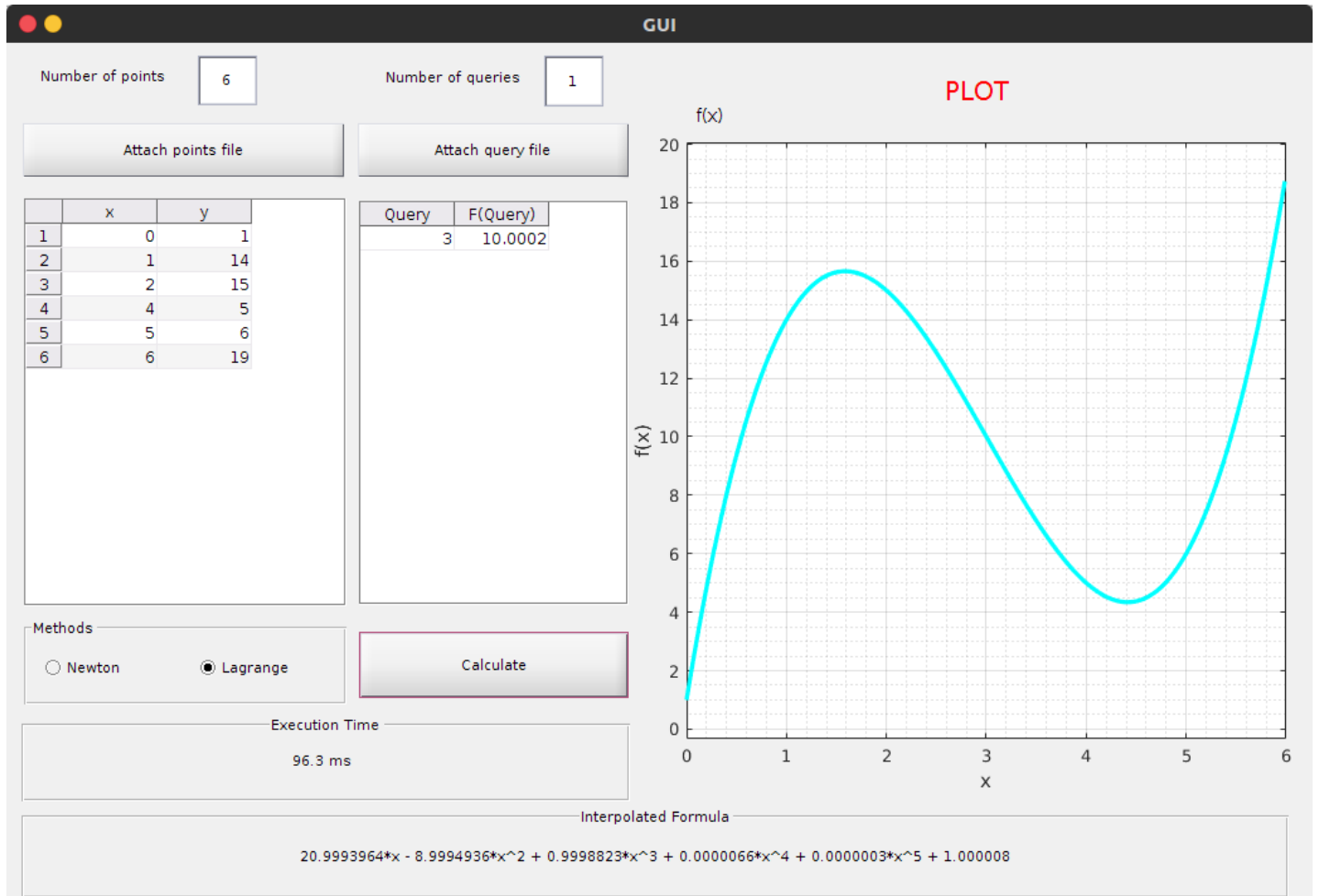


⇒ Newton method :

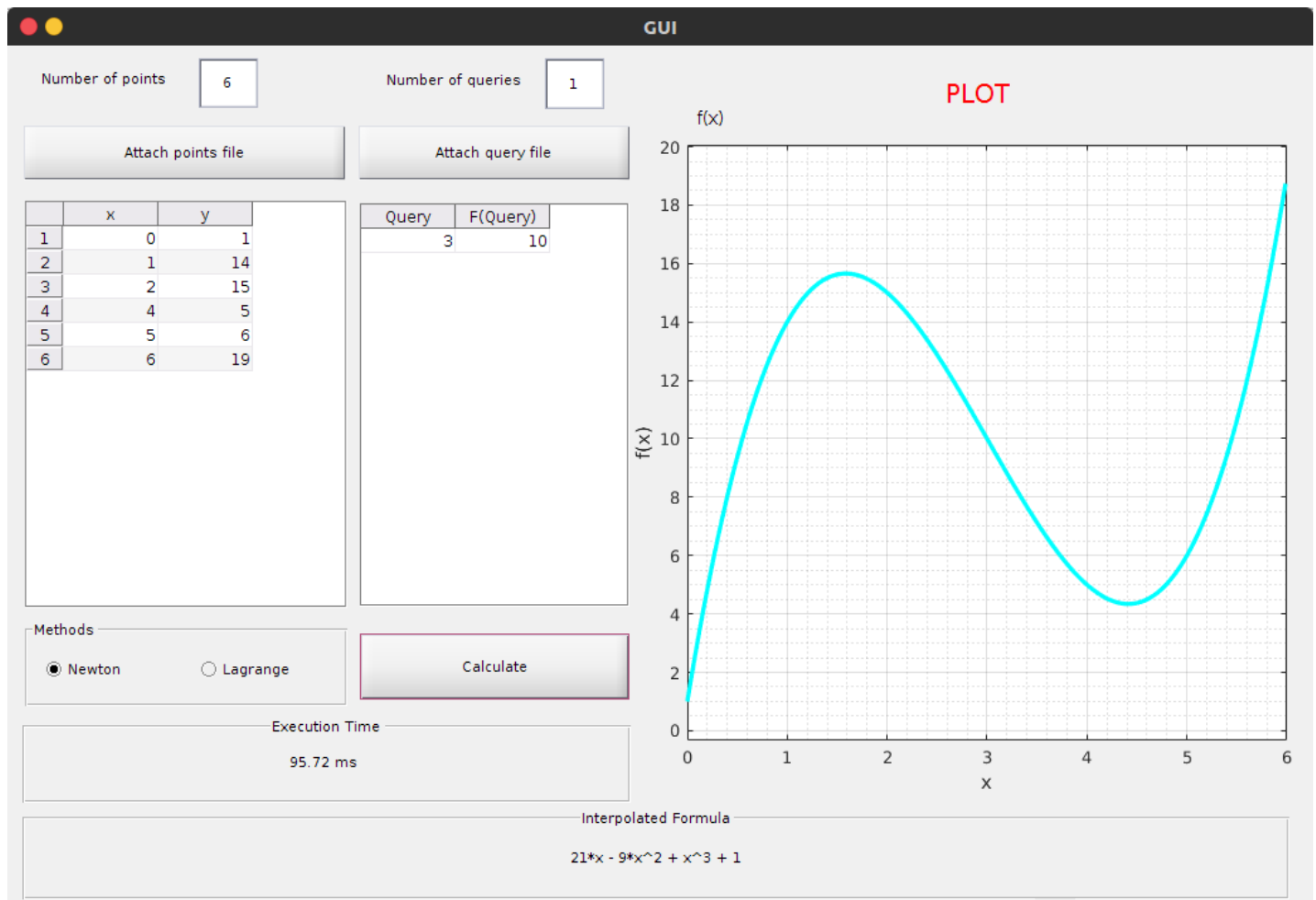


- sample run (3):

⇒ La grange method :

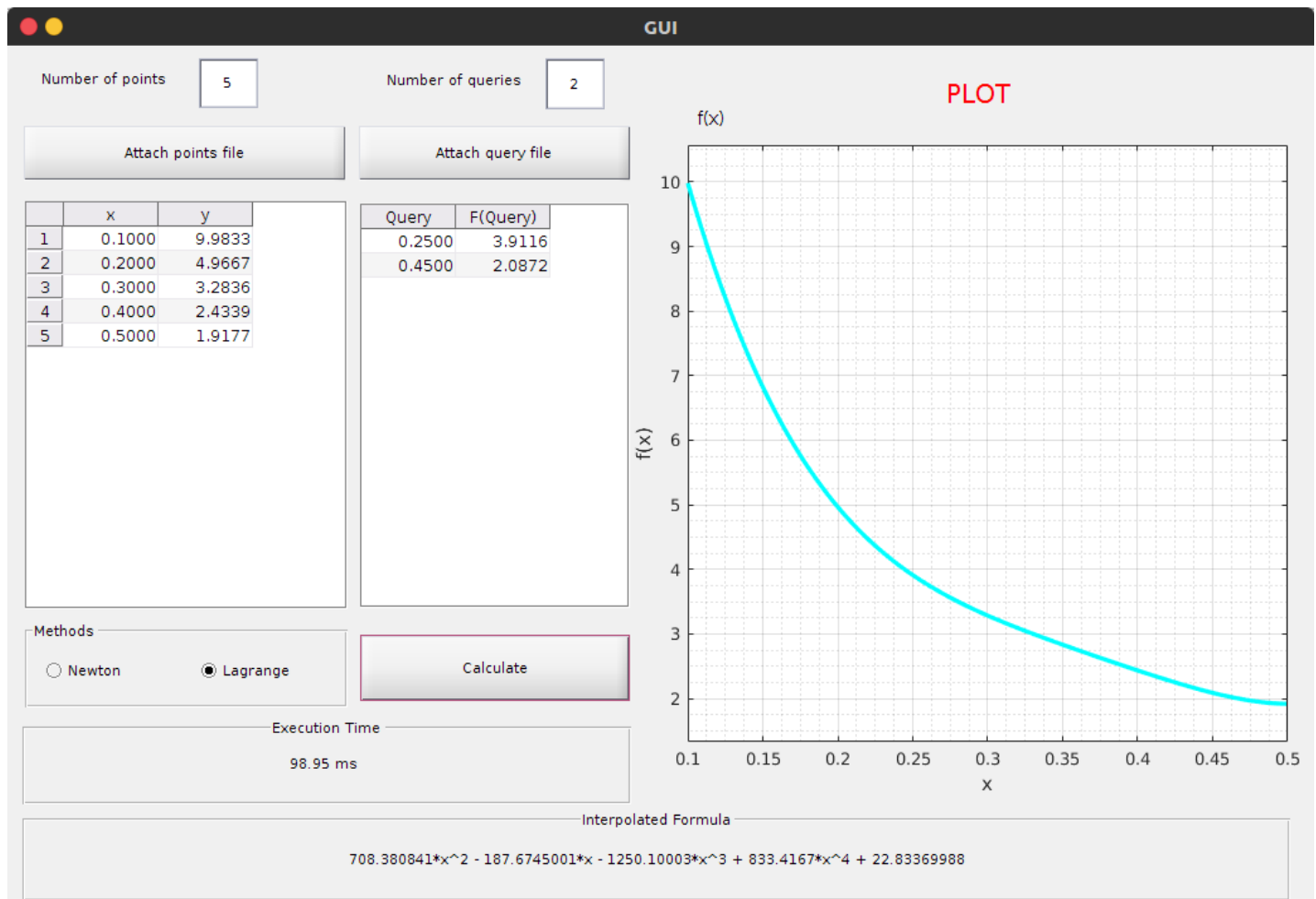


⇒ Newton method :



- sample run (4) :

⇒ La grange method :



⇒ Newton method :

