University *of* Alexandria
Faculty of Engineering
Computer and Systems Engineering Department

NUMERICAL ANALYSIS

Assignment

Shady Abdel Aziz (27)
Omar Khaled (41)
Mohamed Ahmed Abd-ElTwab (52)
Mohamed Samir Shaaban (56)
Mahmoud Abdel Latif (64)

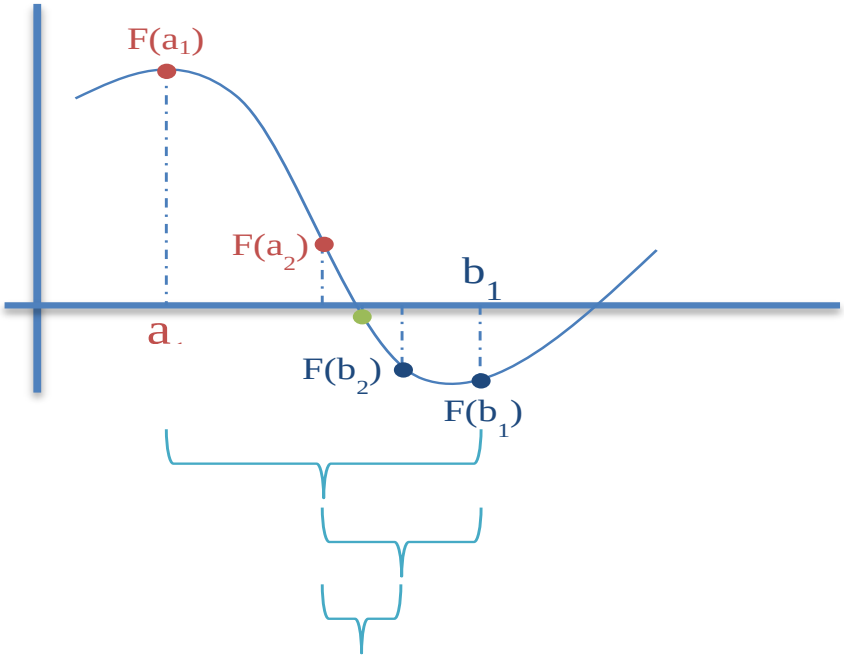
Assignment 1 - Part 1

- **Bisection Method :**

1-Pseduo Code:

```
%Bisection Method: bracketing methodin which the interval is always divided in half.
%Input:
%        func   :  name of the function.
%        xl, xu  :  lower and upper guesses
%        maxit  : maximum allowable iterations (default = 50)
%        eps     : desired relative error
%Output:
%        root : the root of the function if  founded
bisection(func, xl, xu, maxit, eps ):
        If func( xl ) * func( xu ) > 0 then
                Error( "No sign change" );
        for  i←1 to maxit
                xr = ( xl  +  xu ) / 2;
                er = abs( xu - xl / 2 )
                if  f( xr ) = 0 or  er<eps   then
                        returnxr; // The root
                if  sign( f(xr) ) = sign( f(xl) ) then  xl = xr;
                else xl = xr;
        End for;
Error( 'No root was found.' );
End bisection;
```

2 - General Analysis :



The bisection method gives only a range where the root exists, rather than a single estimate for the root's location. Without using any other information, the best estimate for the location of the root is the midpoint of the smallest bracket found. In that case, the absolute error after n steps is at most

$$\frac{|b-a|}{2^{n+1}}.$$

If either endpoint of the interval is used, then the maximum absolute error is

$$\frac{|b-a|}{2^{n}},$$

the entire length of the interval.

These formulas 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. For, using the second formula for the error, the number of iterations n has to satisfy

$$n > \log_2 \frac{b-a}{\epsilon}$$

to ensure that the error is smaller than the tolerance ε.

3- Analysis for the behavior of different examples:

- ✓ $f(x) = x^3 - x - 2, x_l = 1, x_u = 2, eps = 10^{-2}$

x_l	x_u	x_r	$f(x_r)$	e_a
1	2	1.5	-0.125	
1.5	2	1.75	1.6094	0.14286
1.5	1.75	1.625	0.66602	-0.076923
1.5	1.625	1.5625	0.2522	-0.04
1.5	1.5625	1.5313	0.059113	-0.020375
1.5	1.5313	1.5156	-0.034054	-0.010359
1.5156	1.5313	1.5234	0.01225	5.120110^{-3}

- ✓ $f(x) = x^2 - sin(x) - 0.5, x_l = 0, x_u = 2, eps = 10^{-2}$

x_l	x_u	x_r	$f(x_r)$	e_a
0	2	1	-0.34147	
1	2	1.5	0.7525	0.33333
1	1.5	1.25	0.11351	-0.2
1	1.25	1.125	-0.13664	-0.11111
1.125	1.25	1.1875	-0.01728	0.05263

1.1875	1.25	1.2188	0.046787	0.02568
1.1875	1.2188	1.2031	0.014291	-0.01305
1.1875	1.2031	1.1953	-0.001583	-6.5256$10^{-3}$

✓ $f(x) = e^x - 2, x_l = 0, x_u = 2, eps = 10^{-2}$

x_l	x_u	x_r	$f(x_r)$	e_a
0	2	1	0.71828	-1
0	1	0.5	-0.35128	0.33333
0.5	1	0.75	0.117	-0.2
0.5	0.75	0.625	-0.13175	0.09091
0.625	0.75	0.6875	-0.01126	0.04348
0.6875	0.75	0.71875	0.051867	-0.022222
0.6875	0.71875	0.703125	0.020056	-0.01124

4- Problematic functions:

Functions that tangent the x-axis so that no interval in which sign change bracket that root (for example: $f(x) = x^2 - 2x + 1$).

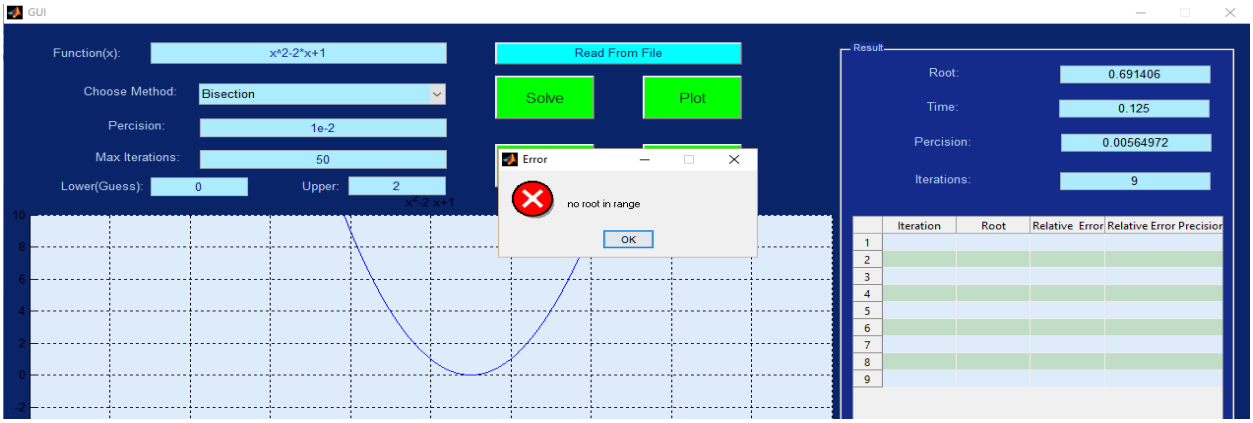
5- Theoretical bound:

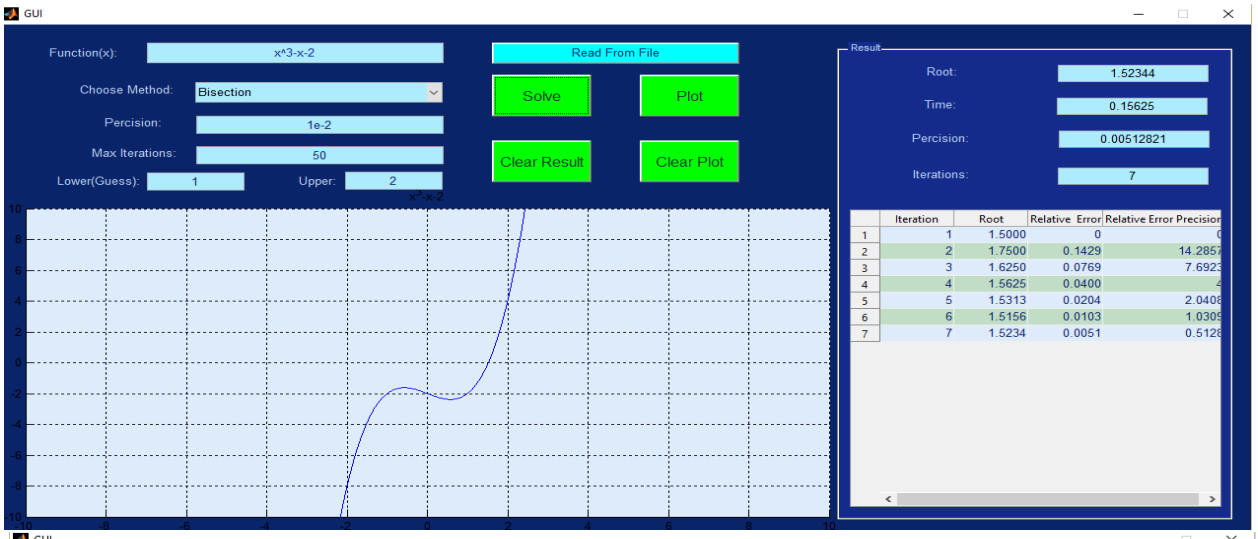
$$\left|\frac{L_k}{x_i}\right| \le error\_tolerance$$

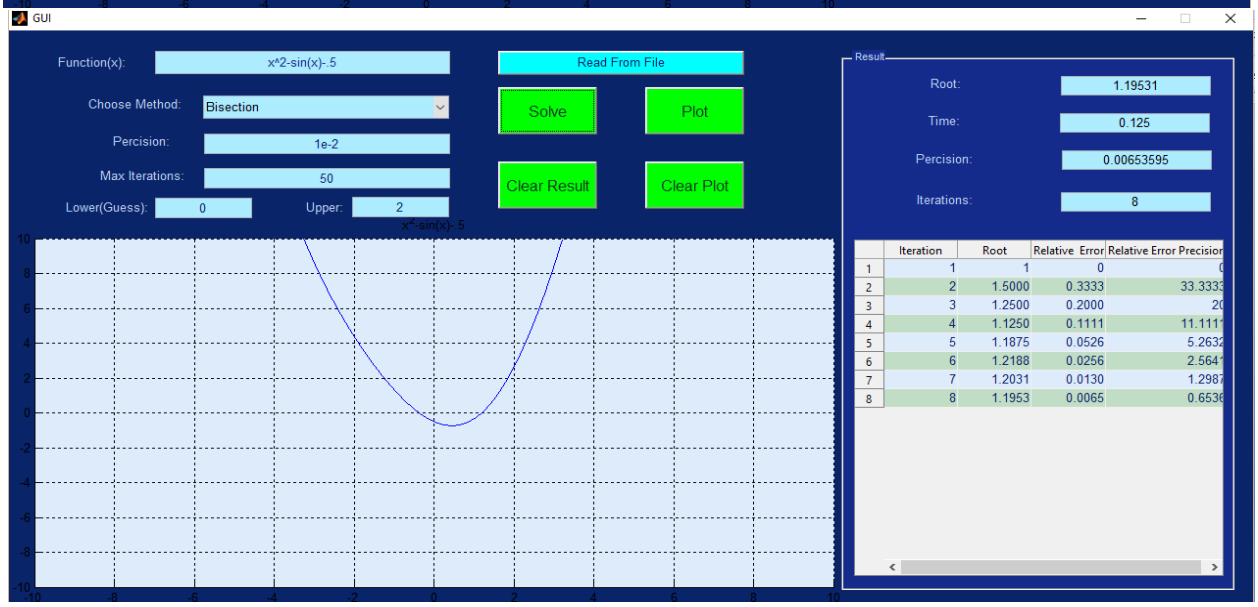
$$\left|\frac{L_0}{2^k}\right| \le \left|x_i * \varepsilon_{es}\right|$$

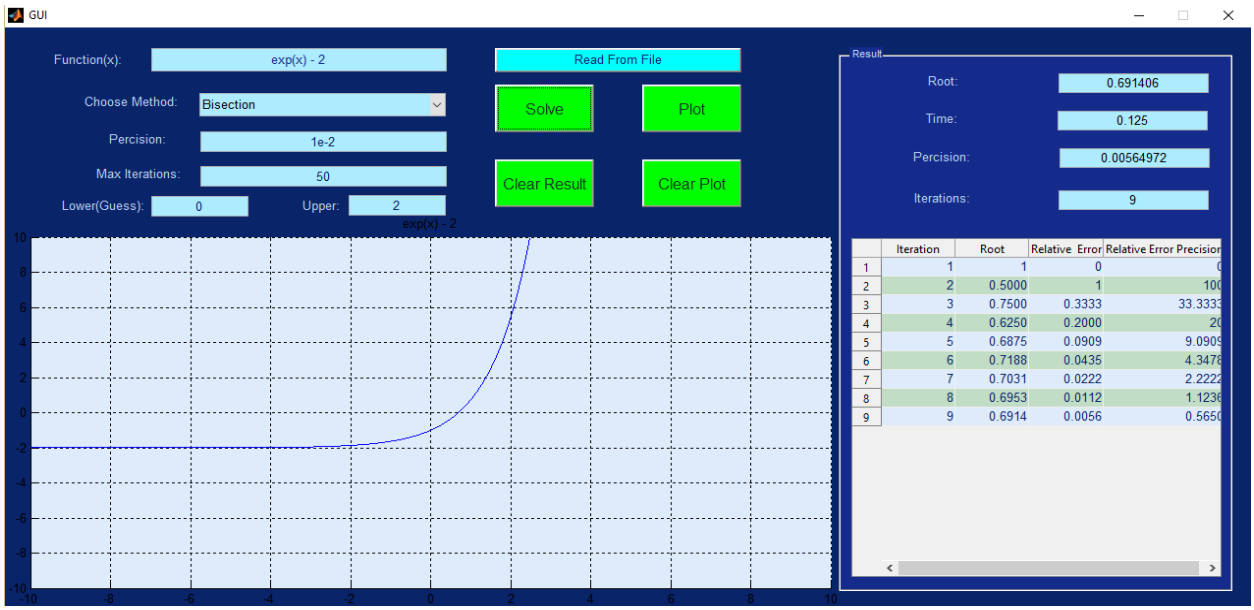
$$2^k \ge \left|\frac{L_0}{x_i * \varepsilon_{es}}\right| \Rightarrow k \ge \log_2\left(\left|\frac{L_0}{x_i * \varepsilon_{es}}\right|\right)$$

6- Sample Run:


GUI
Function(x):
x^2-2*x+1
Read From File
Choose Method:
Bisection
Solve
Plot
Percision:
1e-2
Max Iterations:
50
Lower(Guess):
0
Upper:
2
x^2-2 x+1
Error
no root in range
OK
Result
Root:
0.691406
Time:
0.125
Percision:
0.00564972
Iterations:
9
Iteration
Root
Relative Error
Relative Error Precision


GUI
Function(x):
x^3-x-2
Choose Method:
Bisection
Percision:
1e-2
Max Iterations:
50
Lower(Guess):
1
Upper:
2
Read From File
Solve
Plot
Clear Result
Clear Plot
x^3-x-2
Result
Root:
1.52344
Time:
0.15625
Percision:
0.00512821
Iterations:
7
Iteration
Root
Relative Error
Relative Error Precision
1 1 1.5000 0
2 2 1.7500 0.1429 14.285
3 3 1.6250 0.0769 7.692
4 4 1.5625 0.0400 4
5 5 1.5313 0.0204 2.040
6 6 1.5156 0.0103 1.030
7 7 1.5234 0.0051 0.512


GUI
Function(x):
x^2-sin(x)- 5
Choose Method:
Bisection
Percision:
1e-2
Max Iterations:
50
Lower(Guess):
0
Upper:
2
Read From File
Solve
Plot
Clear Result
Clear Plot
x^2-sin(x)-5
Result
Root:
1.19531
Time:
0.125
Percision:
0.00653595
Iterations:
8
Iteration
Root
Relative Error
Relative Error Precision
1 1 1 0
2 2 1.5000 0.3333 33.333
3 3 1.2500 0.2000 20
4 4 1.1250 0.1111 11.111
5 5 1.1875 0.0526 5.263
6 6 1.2188 0.0256 2.564
7 7 1.2031 0.0130 1.298
8 8 1.1953 0.0065 0.653

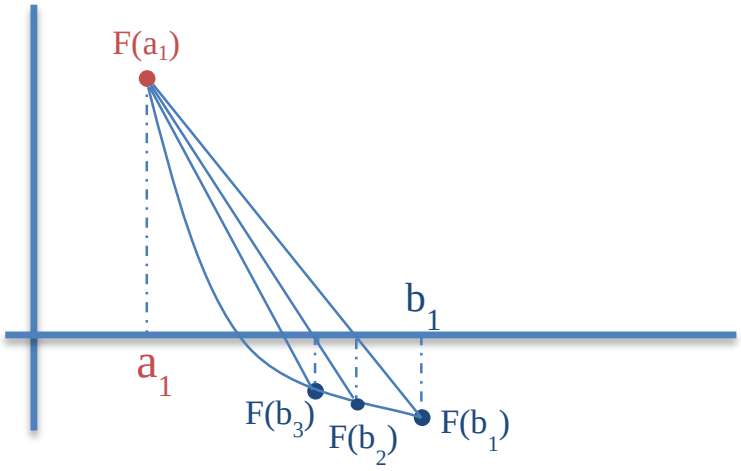

GUI
Function(x):
exp(x) - 2
Choose Method:
Bisection
Percision:
1e-2
Max Iterations:
50
Lower(Guess):
0
Upper:
2
Read From File
Solve
Plot
Clear Result
Clear Plot
exp(x) - 2
Result
Root:
0.691406
Time:
0.125
Percision:
0.00564972
Iterations:
9
Iteration
Root
Relative Error
Relative Error Precision
1 1 1 0
2 2 0.5000 1 100
3 3 0.7500 0.3333 33.333
4 4 0.6250 0.2000 20
5 5 0.6875 0.0909 9.090
6 6 0.7188 0.0435 4.347
7 7 0.7031 0.0222 2.222
8 8 0.6953 0.0112 1.123
9 9 0.6914 0.0056 0.565

- **False Position Method :**

1-Pseduo Code:

```
% False-Position Method: bracketing methodin which the root located by joining f( xl ) and
% f( xu ) with a straight line.
%Input:
%         func   :  name of the function.
%         xl, xu  :  lower and upper guesses
%         maxit  : maximum allowable iterations (default = 50)
%         eps     : desired relative error
%Output:
%         root : the root of the function if  founded
falsePosition(func, xl, xu, maxit, eps ):
         If func( xl ) * func( xu ) > 0 then
                  Error( "No sign change" );
         for  i←1 to maxit
                  xr = ( func(xu)  * xl - func( xl ) * xu  ) / ( func( xu ) – func( xl )  );
                  er = max( abs( xr - xl ), abs( xu – xr  ) );
                  if  f( xr ) = 0 or  er<eps   then
                           returnxr; // The root
                  if  sign( f(xr) ) = sign( f(xl) ) then  xl = xr;
                  else xl = xr;
         End for;
Error( 'No root was found.' );
End falsePosition;
```

2 - General Analysis :



At iteration number k, the number

$$c_k = b_k - f(b_k)\frac{(b_k - a_k)}{f(b_k) - f(a_k)}$$

is computed. As explained below, c_k is the root of the secant line through $(a_k, f(a_k))$ and $(b_k, f(b_k))$. If $f(a_k)$ and $f(c_k)$ have the same sign, then we set $a_{k+1} = c_k$ and $b_{k+1} = b_k$, otherwise we set $a_{k+1} = a_k$ and $b_{k+1} = c_k$. This process is repeated until the root is approximated sufficiently well.

The above formula is also used in the secant method, but the secant method always retains the last two computed points, while the false position method retains two points which certainly bracket a root. On the other hand, the only difference between the false position method and the bisection method is that the latter uses $c_k = (a_k + b_k) / 2$.

3- Analysis for the behavior of different examples:

✓ $f(x) = xcos\frac{x}{x-2}, x_l = 1, x_u = 1.5, eps = 10^{-4}$

x_l	x_u	x_r	$f(x_r)$	e_a
1	1.5	1.1333888	0.295	
1.1333888	1.5	1.19408058	0.106	0.0508272
1.19408058	1.5	1.21452024	0.03	0.01682941
1.21452024	1.5	1.22014602	0.008	0.00461074
1.22014602	1.5	1.22156754	0.002	0.00116369
1.22156754	1.5	1.22191762	5E-04	0.0002865
1.22191762	1.5	1.22200327	1E-04	7.0085E-05

✓ $f(x) = e^{x^2-1} - 10sin(2x) - 5, x_l = 0, x_u = 0.5, eps = 10^{-5}$

x_l	x_u	x_r	$f(x_r)$	e_a
0	0.5	0.27186369	0.569	
0	0.272	0.24210362	0.045	-0.12292286
0	0.242	0.23976629	0.003	-0.00974836
0	0.24	0.23959596	2E-04	-0.0007109
0	0.24	0.23958362	2E-05	-5.1498E-05

✓ $f(x) = e^x - 3x^2, x_l = 3, x_u = 4, eps = 10^{-4}$

x_l	x_u	x_r	$f(x_r)$	e_a
3	4	3.51170436	-3.491	
3.51170436	4	3.68065826	-0.969	0.04590317
3.68065826	4	3.72155975	-0.221	0.01099042
3.72155975	4	3.73059212	-0.048	0.00242116
3.73059212	4	3.73254421	-0.01	0.00052299

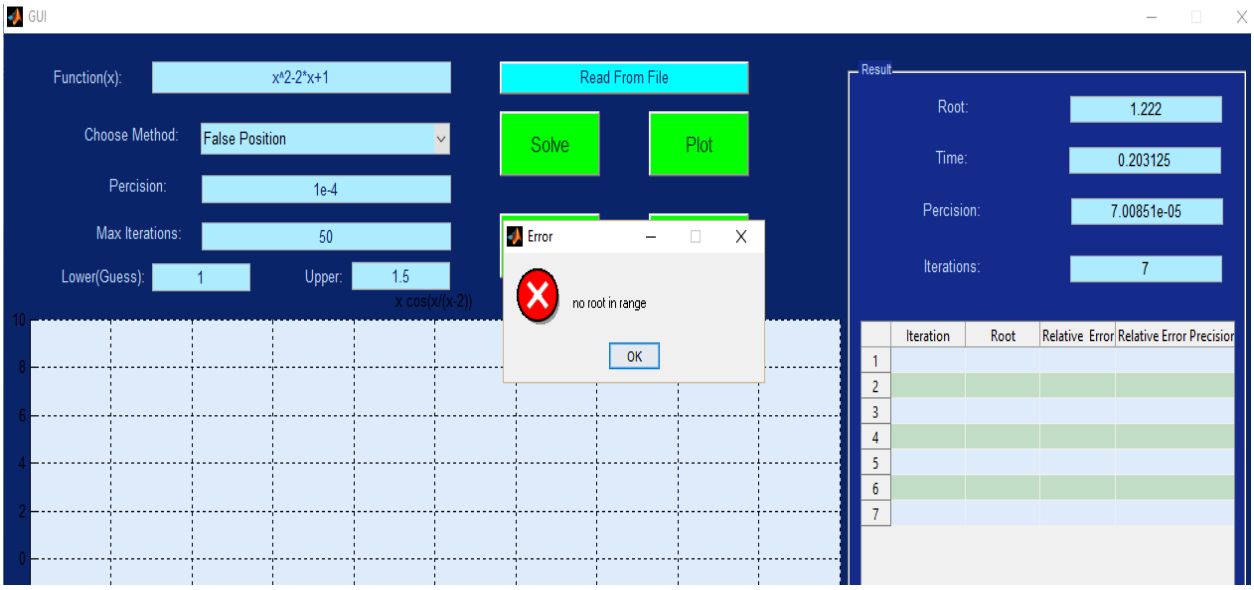
3.73254421	4	3.73296411	-0.002	0.00011248
3.73296411	4	3.73305434	-5E-04	2.417E-05

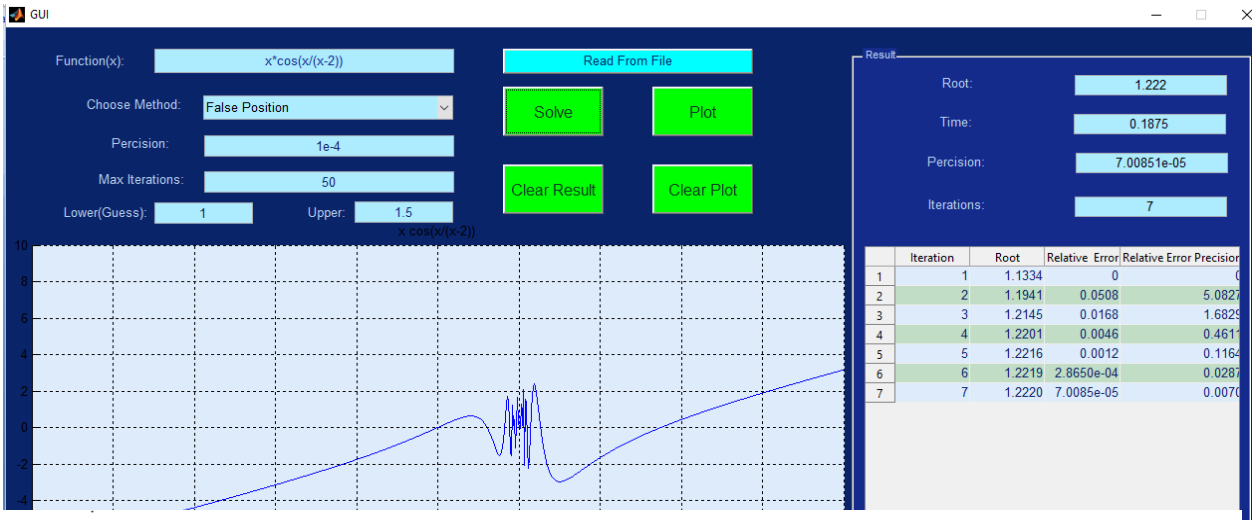
4- Problematic functions:

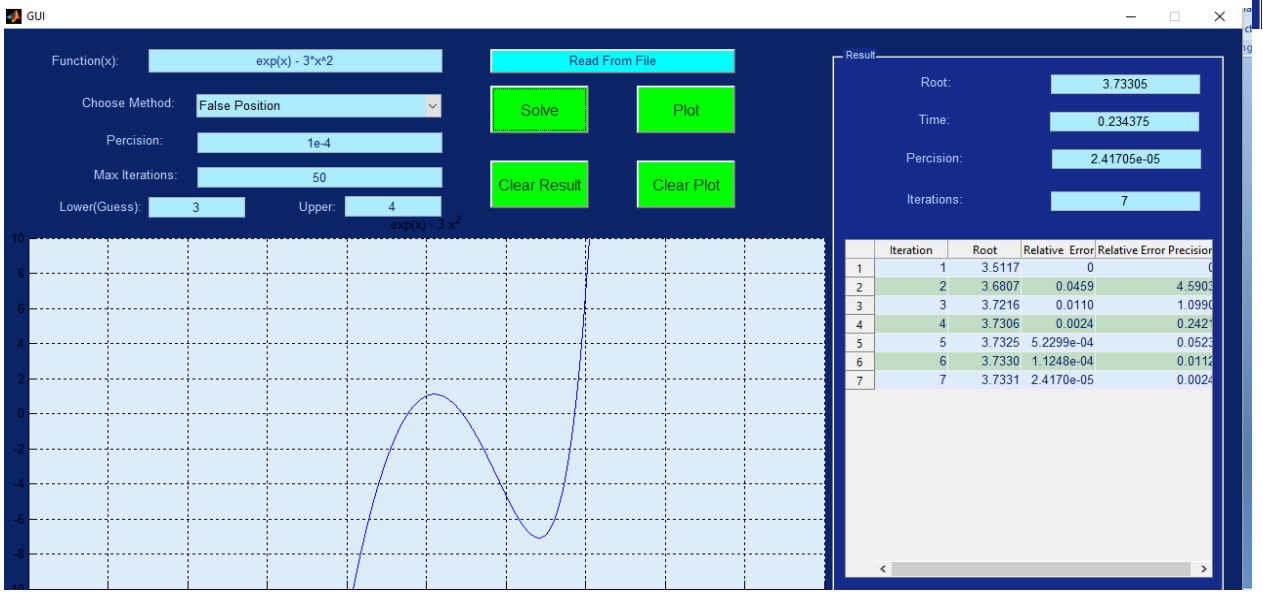
1. Functions that tangent the x-axis so that no interval in which sign change bracket that root (for example: $f(x) = x^2 - 2x + 1$).
2. Slow convergence in some functions(for example:

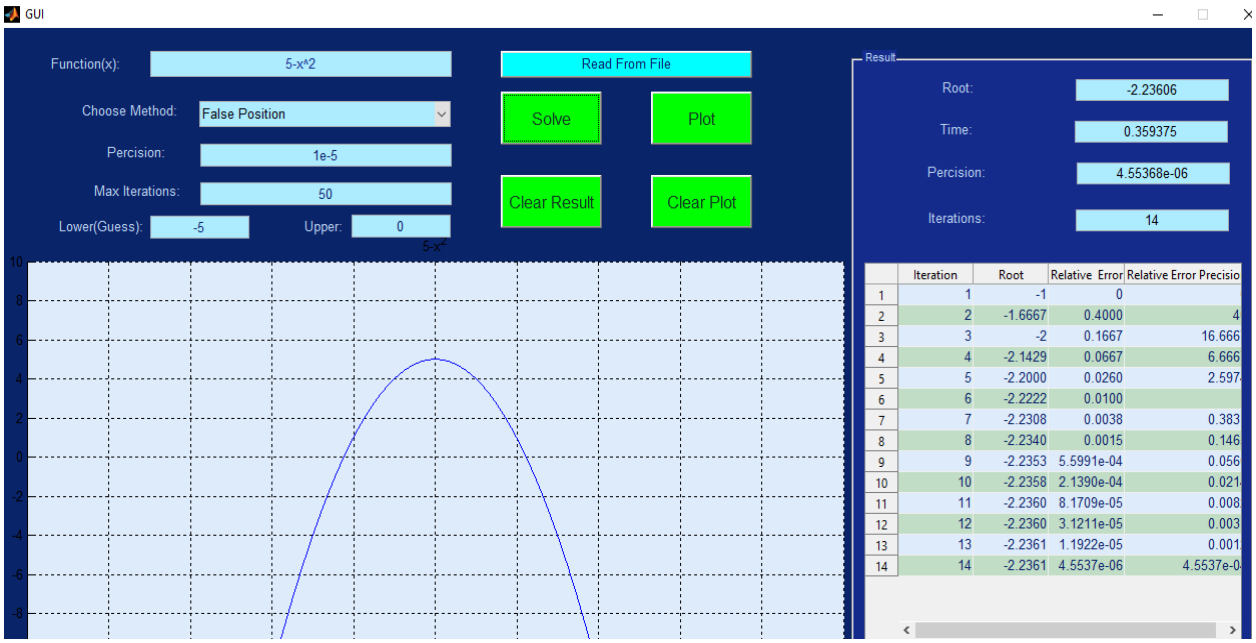
$f(x) = x^{10} - 1$.

5- Sample Run:




GUI
Function(x):
x*cos(x/(x-2))
Read From File
Choose Method:
False Position
Solve
Plot
Percision:
1e-4
Clear Result
Clear Plot
Max Iterations:
50
Lower(Guess):
1
Upper:
1.5
x*cos(x/(x-2))
Result
Root:
1.222
Time:
0.1875
Percision:
7.00851e-05
Iterations:
7
Iteration
Root
Relative Error
Relative Error Precision
1 1 1.1334 0
2 2 1.1941 0.0508 5.0827
3 3 1.2145 0.0168 1.6829
4 4 1.2201 0.0046 0.4611
5 5 1.2216 0.0012 0.1164
6 6 1.2219 2.8650e-04 0.0287
7 7 1.2220 7.0085e-05 0.007


GUI
Function(x):
exp(x) - 3*x^2
Read From File
Choose Method:
False Position
Solve
Plot
Percision:
1e-4
Clear Result
Clear Plot
Max Iterations:
50
Lower(Guess):
3
Upper:
4
exp(x) - 3 x^2
Result
Root:
3.73305
Time:
0.234375
Percision:
2.41705e-05
Iterations:
7
Iteration
Root
Relative Error
Relative Error Precision
1 1 3.5117 0
2 2 3.6807 0.0459 4.5903
3 3 3.7216 0.0110 1.0990
4 4 3.7306 0.0024 0.2421
5 5 3.7325 5.2299e-04 0.0523
6 6 3.7330 1.1248e-04 0.0112
7 7 3.7331 2.4170e-05 0.0024

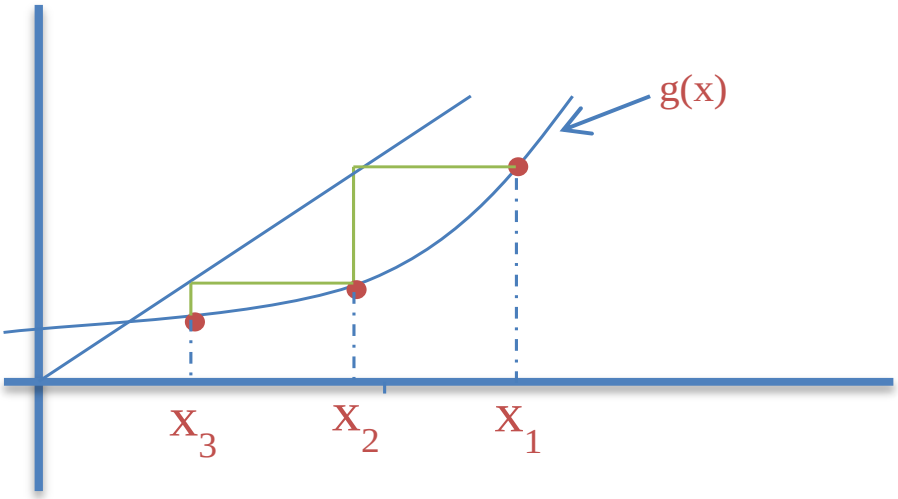

GUI
Function(x):
5-x^2
Read From File
Choose Method:
False Position
Solve
Plot
Percision:
1e-5
Clear Result
Clear Plot
Max Iterations:
50
Lower(Guess):
-5
Upper:
0
5-x^2
Result
Root:
-2.23606
Time:
0.359375
Percision:
4.55368e-06
Iterations:
14
Iteration
Root
Relative Error
Relative Error Precisio
1 1 -1 0
2 2 -1.6667 0.4000 4
3 3 -2 0.1667 16.666
4 4 -2.1429 0.0667 6.666
5 5 -2.2000 0.0260 2.597
6 6 -2.2222 0.0100
7 7 -2.2308 0.0038 0.383
8 8 -2.2340 0.0015 0.146
9 9 -2.2353 5.5991e-04 0.056
10 10 -2.2358 2.1390e-04 0.021
11 11 -2.2360 8.1709e-05 0.008
12 12 -2.2360 3.1211e-05 0.003
13 13 -2.2361 1.1922e-05 0.001
14 14 -2.2361 4.5537e-06 4.5537e-0

- **Fixed Point Method :**

1-Pseduo Code:

```
% Fixed Point Method: open methodthat predict the root by rearranging the function f (x) = 0 so that
% x is on the left-handside of the equation
%Input:
%       g    :  a function such that the original function f(x) to x = g(x)
%       x       :  initial guess
%       maxit  : maximum allowable iterations (default = 50)
%       eps     : desired relative error
%Output:
%       root : the root of the function if  founded
fixedPoint( g, x, maxit, eps ):
        for  i←1 to maxit
                xold = x;
                x = g(xold );
                er = abs( (x - xold) / x ) ;
                if  f( x ) = 0 or  er<eps   then
                        return x; // The root
        End for;
Error( 'The equation is diverge.' );
End fixedPoint;
```

2 - General Analysis :



- Newton's method for finding roots of a given differentiable function $f(x)$ is

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}.$$

If we write $g(x) = x - \frac{f(x)}{f'(x)}$, we may rewrite the Newton iteration as the fixed-point iteration

$$x_{n+1} = g(x_n).$$

If this iteration converges to a fixed point x of g, then

$$x = g(x) = x - \frac{f(x)}{f'(x)}, \text{ so } f(x)/f'(x) = 0.$$

3- Analysis for the behavior of different examples:

✓ $f(x) = x^4 - x - 10, g(x) = \sqrt[4]{x+10}, x = 4, eps = 10^{-5}$

x	e_a
4	
1.93433642	-1.06789262
1.85865836	-0.0407165
1.85570479	-0.00159161
1.85558923	-6.2276E-05
1.85558471	-2.4368E-06

✓ $f(x) = x - e^{-x}, x = 3, g(x) = e^{-x}, eps = 10^{-2}$

x_l	e_a
3	
0.04978707	-59.2566108
0.95143199	0.947671438
0.38618761	-1.4636523
0.67964301	0.431778734
0.50679788	-0.34105336
0.60242152	0.158732102
0.54748429	-0.10034485
0.57840308	0.05345544
0.56079319	-0.03140174
0.57075616	0.017455735
0.56509797	-0.01001276
0.56830447	0.005642215

✓ $f(x) = x - sin(x) - 0.5, x = 2, g(x) = sin(x) + 0.5, eps = 10^{-2}$

x	e_a

3	
0.64112001	-3.67931115
1.09809342	0.416151671
1.39034092	0.210198447
1.48376206	0.062962343

4- Problematic functions:

Functions where $|\frac{dg(x)}{dx}|$> 1 is diverged because the error is increase.

chose got g(x).

5- Theoretical bound:

Convergence of Fixed Point Iteration

According to the derivative mean-value theorem, if $g(x)$ and $g'(x)$ are continuous over an interval $x_i \le x \le \alpha$, there exists a value $x = c$ within the interval such that

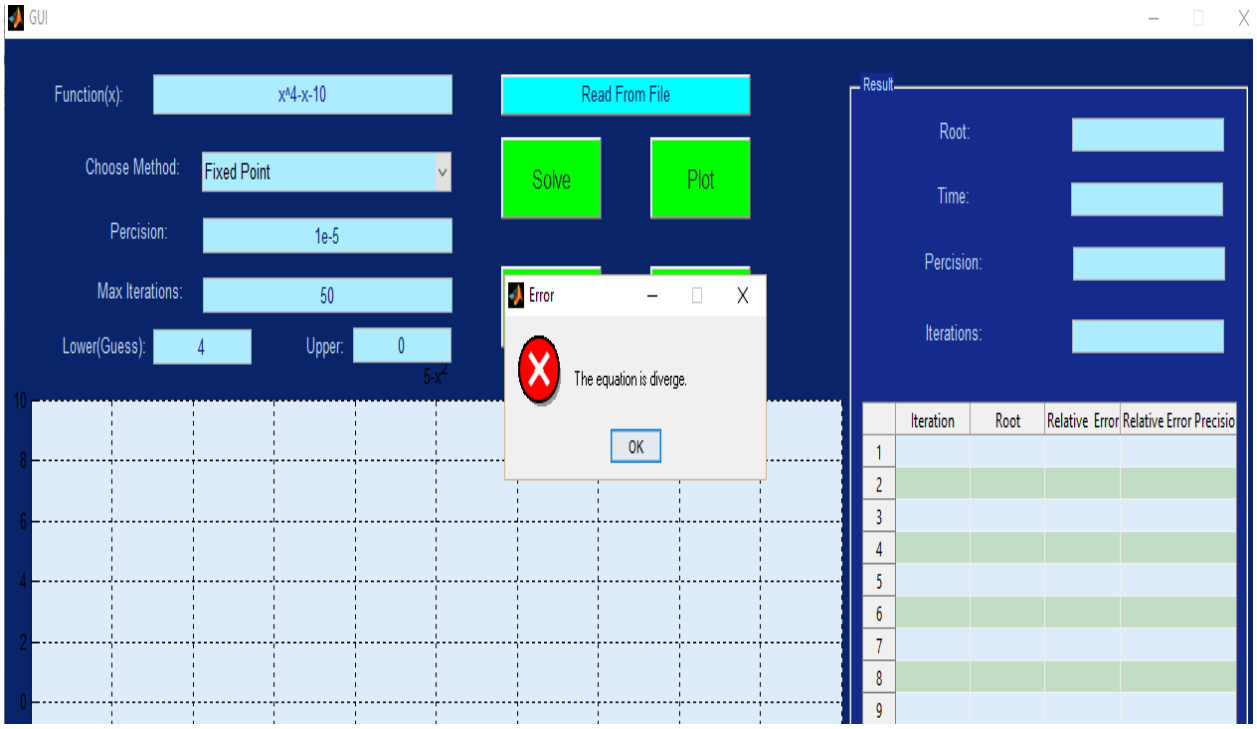
$$g'(c) = \frac{g(\alpha) - g(x_i)}{\alpha - x_i} \qquad (7)$$

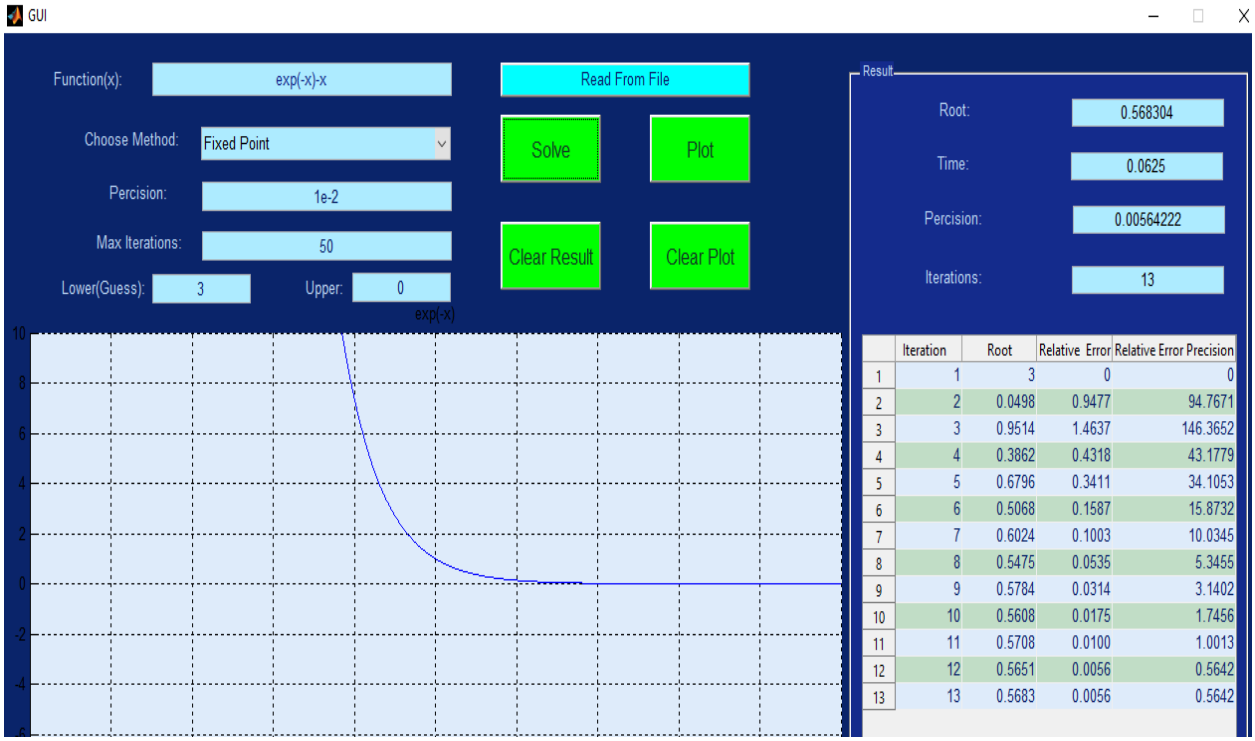
From (1) and (6), we have $\delta_i = \alpha - x_i$ and $\delta_{i+1} = g(\alpha) - g(x_i)$

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

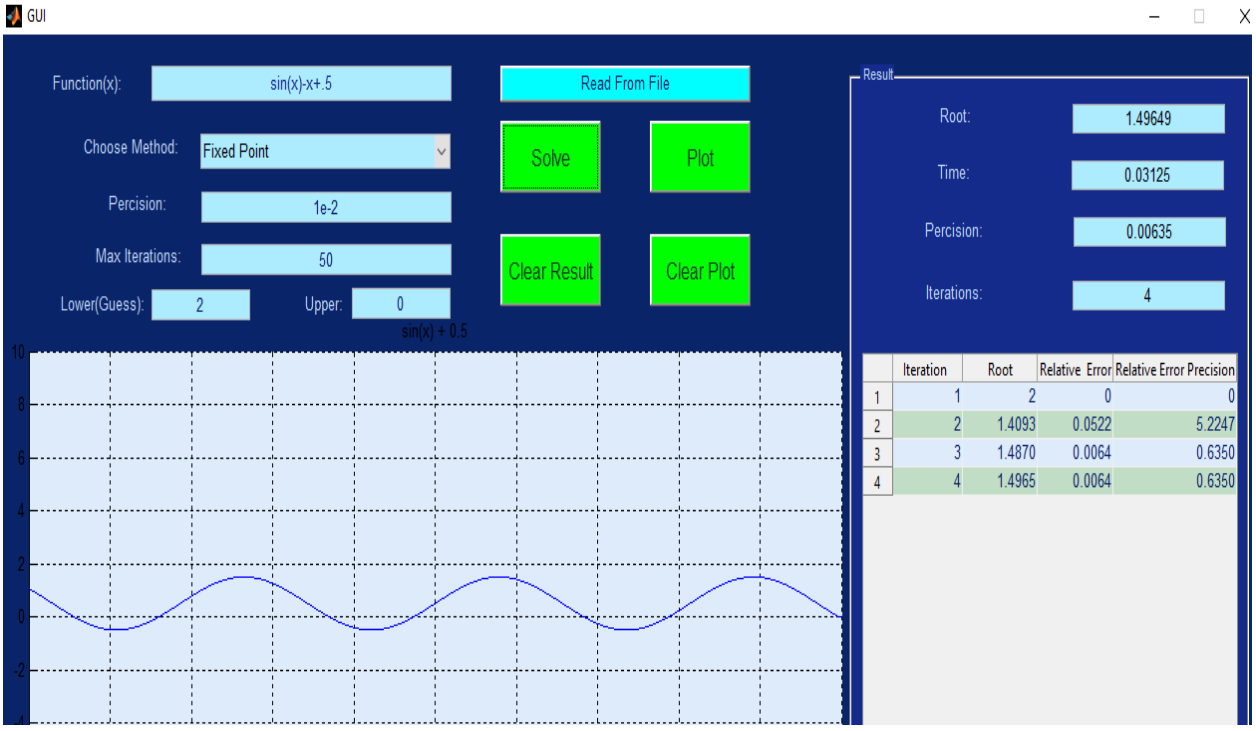
- Therefore, if $|g'(c)| < 1$, the error decreases with each iteration. If $|g'(c)| > 1$, the error increase.
- If the derivative is positive, the iterative solution will be monotonic.
- If the derivative is negative, the errors will oscillate.

6- Sample Run:


GUI
Function(x):
x^4-x-10
Read From File
Choose Method:
Fixed Point
Solve
Plot
Percision:
1e-5
Max Iterations:
50
Lower(Guess):
4
Upper:
0
5-x^4
Error
The equation is diverge.
OK
Result
Root:
Time:
Percision:
Iterations:
Iteration
Root
Relative Error
Relative Error Precisio



	Iteration	Root	Relative Error	Relative Error Precision
1	1	3	0	0
2	2	0.0498	0.9477	94.7671
3	3	0.9514	1.4637	146.3652
4	4	0.3862	0.4318	43.1779
5	5	0.6796	0.3411	34.1053
6	6	0.5068	0.1587	15.8732
7	7	0.6024	0.1003	10.0345
8	8	0.5475	0.0535	5.3455
9	9	0.5784	0.0314	3.1402
10	10	0.5608	0.0175	1.7456
11	11	0.5708	0.0100	1.0013
12	12	0.5651	0.0056	0.5642
13	13	0.5683	0.0056	0.5642



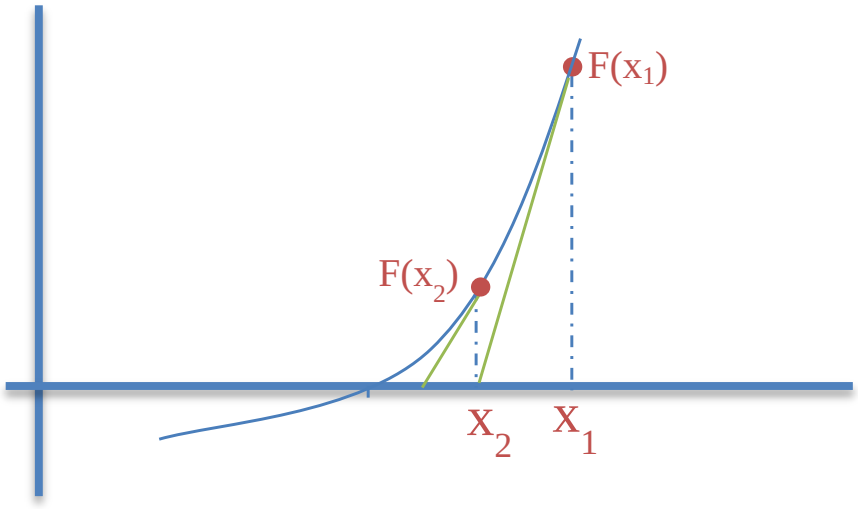
	Iteration	Root	Relative Error	Relative Error Precision
1	1	2	0	0
2	2	1.4093	0.0522	5.2247
3	3	1.4870	0.0064	0.6350
4	4	1.4965	0.0064	0.6350

- **Newton-Raphson Method :**

1-Pseduo Code:

```
% Newton Method: open methodthat predict the root by extending the tangent of
% f (x) at the current point and the new point is where this tangent crosses x-axis.
%Input:
%       func:   The function name
%       x       :  initial guess
%       maxit  : maximum allowable iterations (default = 50)
%       eps     : desired relative error
%Output:
%       root : the root of the function if  founded
newtRaph ( func, x,  maxit, eps ):
        dfunc = the differentiation of the function
        for  i ←1 to maxit
                xold = x;
                x = xold – func( xold ) / dfunc( xold );
                er = abs( (x - xold) / x ) ;
                if  f( x ) = 0 or  er < eps   then
                        return x; // The root
        End for;
Error( 'The equation is diverge.' );
End newtRaph;
```

2 - General Analysis :



3- Analysis for the behavior of different examples:

- ✓ $f(x) = cos(x) - xe^x, x = 2, eps = 10^{-4}$

x	e_a
2	
1.34156906	-0.49079169
0.84770056	-0.58259783
0.58755675	-0.44275521
0.52158097	-0.12649192
0.51776956	-0.00736121
0.51775736	-2.3556E-05

- ✓ $f(x) = e^{-x}(x^2 + 5X + 2) + 1, x = -2, eps = 10^{-10}$

x_l	e_a
-2	
-1.2270670	-0.62990277
-0.7756155	-0.58205573
-0.6029104	-0.28645246
-0.5795518	-0.0403046
-0.5791590	-0.0006782
-0.5791589	-1.8897E-07
-0.5791589	-1.4761E-14

- ✓ $f(x) = x - sin(x) - 0.5, x = 2, eps = 10^{-10}$

x	e_a
2	
1.58288042	-0.26351932
1.50091741	-0.05460861
1.4973074	-0.002411
1.49730039	-4.6849E-06
1.49730039	-1.7686E-11

4- Problematic functions:

1. Slow convergence in some functions(for example:

$$f(x) = x^{10} - 1$$

can use the following modified algorithm that preserves the quadratic convergence rate

$$x_{n+1} = x_n - m\frac{f(x_n)}{f'(x_n)}.$$

2. Poor intial estimate:

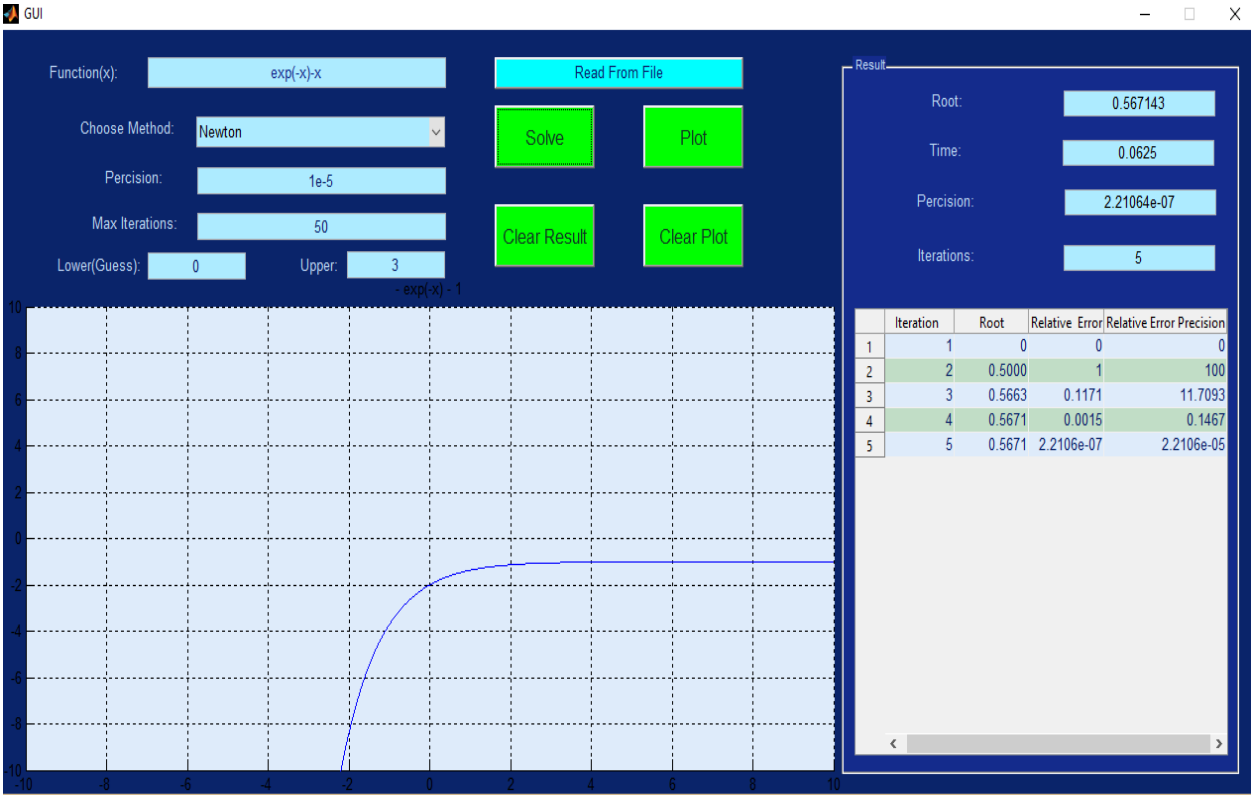
A large error in the initial estimate can contribute to non-convergence of the algorithm. To overcome this problem one can often linearise the function that is being optimized using calculus, logs, differentials, or even using evolutionary algorithms, such as the Stochastic Funnel Algorithm Good initial estimates lie close to the final globally optimal parameter estimate.

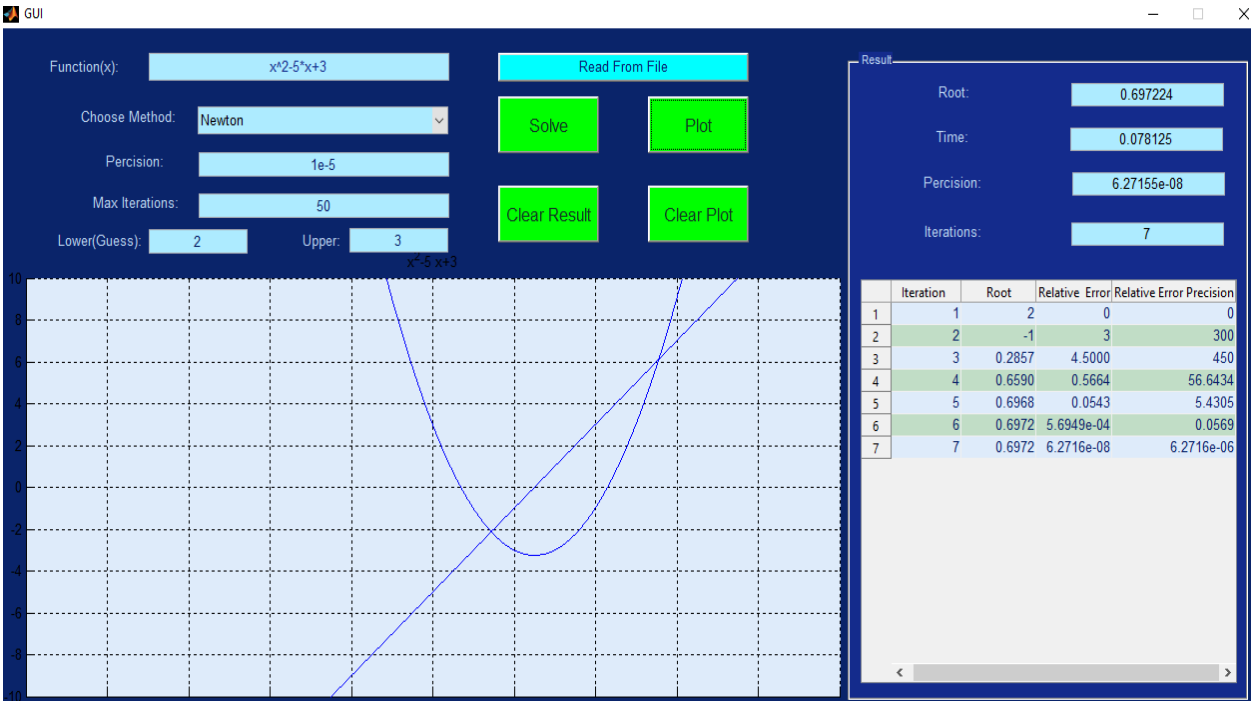
5- Theoretical bound:

Error of next iteration can be compute from this equation:

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

6- Sample Run:


GUI
Function(x):
exp(-x)-x
Read From File
Choose Method:
Newton
Solve
Plot
Percision:
1e-5
Max Iterations:
50
Clear Result
Clear Plot
Lower(Guess):
0
Upper:
3
- exp(-x) - 1
Result
Root:
0.567143
Time:
0.0625
Percision:
2.21064e-07
Iterations:
5
Iteration Root Relative Error Relative Error Precision
1 1 0 0 0
2 2 0.5000 1 100
3 3 0.5663 0.1171 11.7093
4 4 0.5671 0.0015 0.1467
5 5 0.5671 2.2106e-07 2.2106e-05

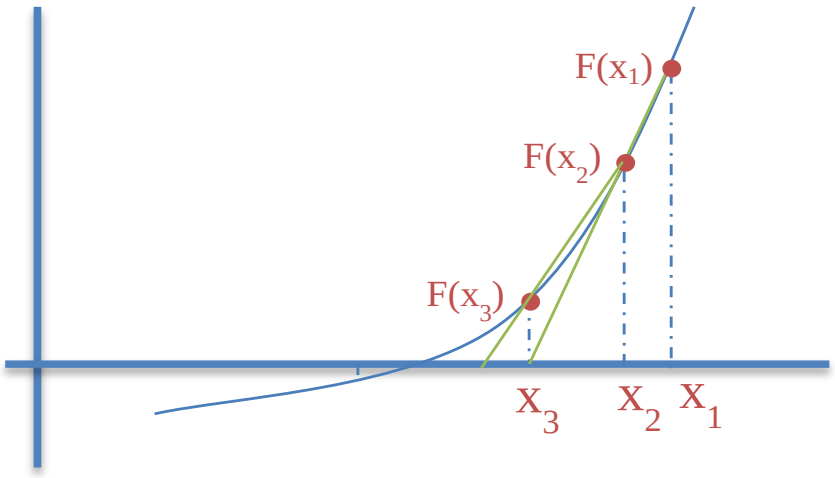

GUI
Function(x):
x^2-5*x+3
Read From File
Choose Method:
Newton
Solve
Plot
Percision:
1e-5
Max Iterations:
50
Clear Result
Clear Plot
Lower(Guess):
2
Upper:
3
x^2-5 x+3
Result
Root:
0.697224
Time:
0.078125
Percision:
6.27155e-08
Iterations:
7
Iteration Root Relative Error Relative Error Precision
1 1 2 0 0
2 2 -1 3 300
3 3 0.2857 4.5000 450
4 4 0.6590 0.5664 56.6434
5 5 0.6968 0.0543 5.4305
6 6 0.6972 5.6949e-04 0.0569
7 7 0.6972 6.2716e-08 6.2716e-06

- **Secant Method :**

1-Pseduo Code:

```
% Secant Method: open methodthat  that uses a succession of roots of secant lines to
% better approximate a root of a function
%Input:
%          func:   The function name
%          x0, x1        :  two initial guesses
%          maxit  : maximum allowable iterations (default = 50)
%          eps       : desired relative error
%Output:
%          root : the root of the function if  founded
secant ( func, x0, x1  maxit, eps ):
          for  i ←1 to maxit
                    x2 = x1- ( func(x1) * (x1 - x0) / ( func( x1) – func(x0) ) );
                    er = abs( (x2 – x1) / x2 ) ;
                    if  f( x2 ) = 0 or  er < eps   then
                              return x; // The root
                    else
                              x0 = x1;
                              x1 = x2;
          End for;
Error( 'The equation is diverge.' );
End secant;
```

2 - General Analysis :



3- Analysis for the behavior of different examples:

- ✓ $f(x) = 3x + sinx - e^x, x_0 = 0, x_1 = 1, eps = 10^{-6}$

x	e_a
0	
1	
0.47098959	-1.12318916
0.30750846	-0.53163133
0.36261324	0.151965716
0.36046148	-0.00596946
0.36042167	-0.00011045
0.3604217	8.65194E-08

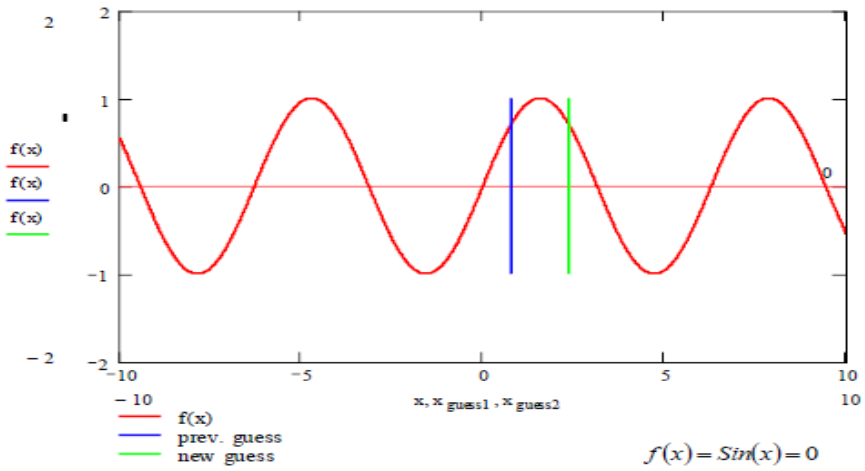
- ✓ $f(x) = x^4 - x - 10, x_0 = 1, x_1 = 2, eps = 10^{-4}$

x_i	e_a
1	
2	
1.71428571	-0.16666667
1.83853125	0.067578689
1.85777579	0.010358919
1.85555287	-0.00119799
1.85558447	1.70325E-05

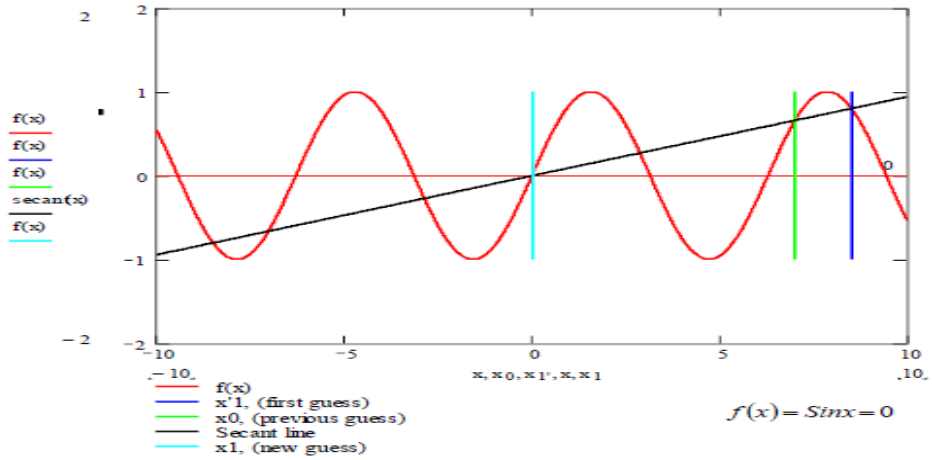
- ✓ $f(x) = x - e^{-x}, x_0 = 1, x_1 = 2, , eps = 10^{-10}$

x	e_a
1	
2	
0.48714165	-3.10558199
0.58377969	0.165538531
0.56738645	-0.02889254
0.56714256	-0.00043003
0.56714329	1.28739E-06
0.56714329	-5.664E-11

4- Problematic functions:



Division by zero



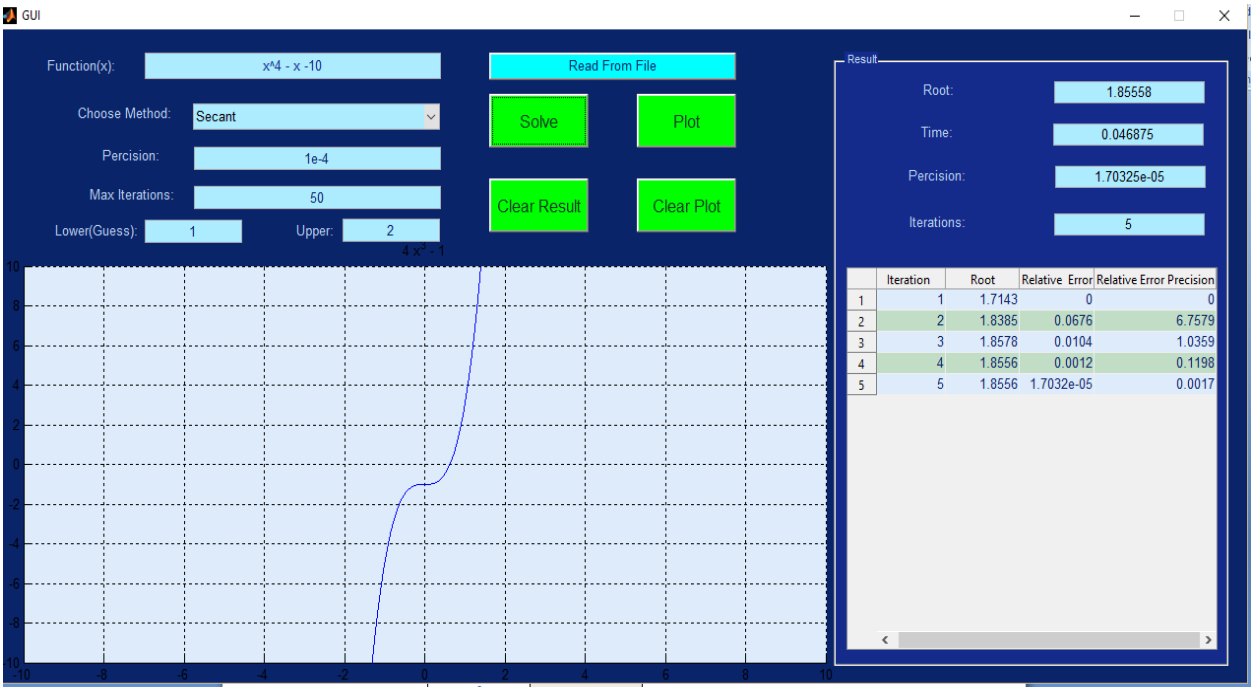
Root Jumping

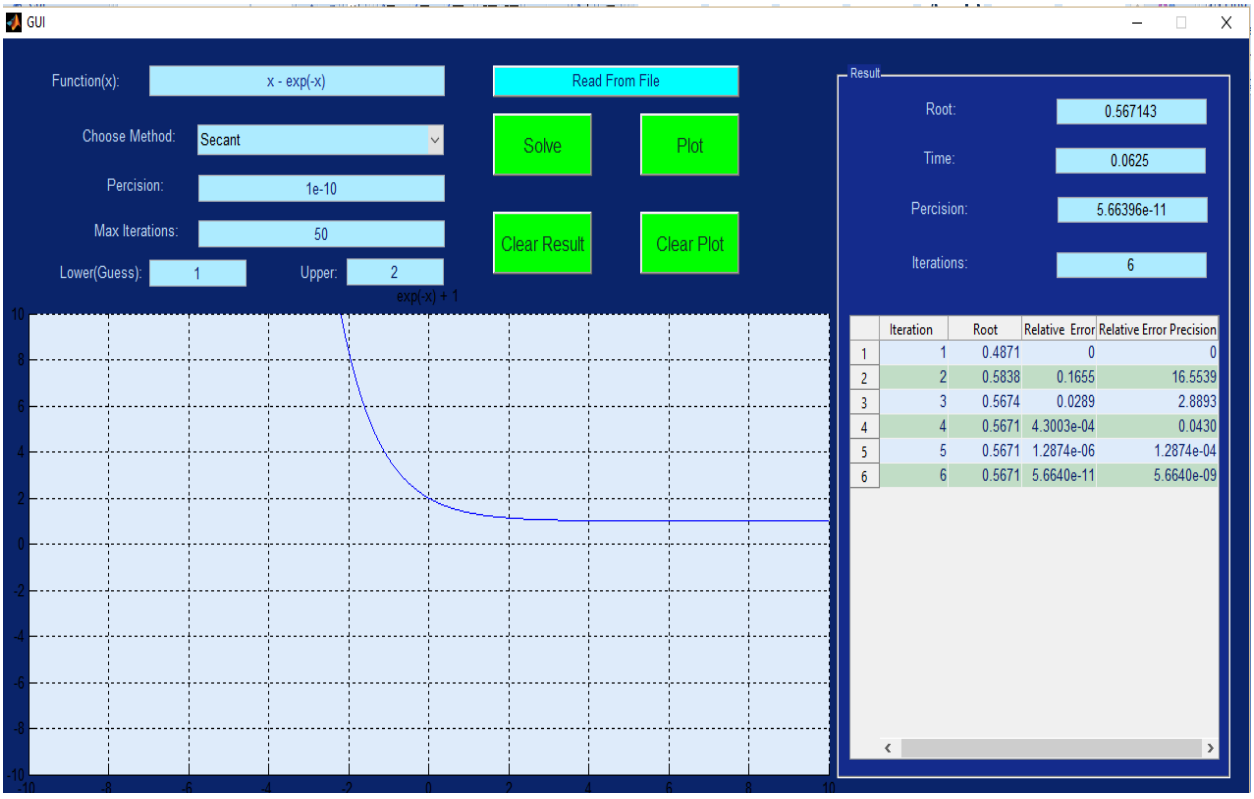
to modify it:

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

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

5- Sample Run:


GUI
Function(x):
x^4 - x -10
Read From File
Choose Method:
Secant
Solve
Plot
Percision:
1e-4
Max Iterations:
50
Clear Result
Clear Plot
Lower(Guess):
1
Upper:
2
4 x^3 - 1
Result
Root:
1.85558
Time:
0.046875
Percision:
1.70325e-05
Iterations:
5
Iteration Root Relative Error Relative Error Precision
1 1 1.7143 0 0
2 2 1.8385 0.0676 6.7579
3 3 1.8578 0.0104 1.0359
4 4 1.8556 0.0012 0.1198
5 5 1.8556 1.7032e-05 0.0017


GUI
Function(x):
x - exp(-x)
Read From File
Choose Method:
Secant
Solve
Plot
Percision:
1e-10
Max Iterations:
50
Clear Result
Clear Plot
Lower(Guess):
1
Upper:
2
exp(-x) + 1
Result
Root:
0.567143
Time:
0.0625
Percision:
5.66396e-11
Iterations:
6
Iteration Root Relative Error Relative Error Precision
1 1 0.4871 0 0
2 2 0.5838 0.1655 16.5539
3 3 0.5674 0.0289 2.8893
4 4 0.5671 4.3003e-04 0.0430
5 5 0.5671 1.2874e-06 1.2874e-04
6 6 0.5671 5.6640e-11 5.6640e-09

- **Bierge VietaMethod :**

1-Pseduo Code:

```
vieta( f , r , es , itrMax)
syms x;
symbol(x) ← sym(f);
coff ← sym2poly(symbol);
len ← length(coff);
itr ← 0 ;
error(1) ← 0;
x(1) ← r;
b(1) ← coff(1);
c(1) ← coff(1);
for j ← 1 : itrMax
    itr ← itr + 1 ;
    for i ← 1  :len-1
        b(i+1) ← coff(i+1) + (x(itr) * b(i));
        c(i+1) ← b(i+1)  + (x(itr) * c(i));
    end
    if(b(len) = 0)
        break;
    else
        if(itr > 1)
            ea ← abs(x(itr)-x(itr-1))/abs(x(itr)) ;
            error(itr) ← ea;
            if(ea < es)
                break;
            end
        end
        x(itr+1) ← x(itr) - (b(len)/c(len-1));
    end
end
end
```

2 - General Analysis :

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

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

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

$$c_m = b_m = a_m$$
$$b_j = a_j + x_i b_{j+1}$$
$$c_j = b_j + x_i c_{j+1} \qquad j = m-1, m-2, \ldots, 1$$
$$b_0 = a_0 + x_i b_1$$

3- Analysis for the behavior of different examples:

✓ $f(x) = x^2 - 3 * x + 2, x_0 = 3$

I	A	B	C	I	A	B	C	I	A	B	c
2	1	1	1	2	1	1	1	3	1	1	
1	-3	0	3	1	-3	.667	1.667	1	-3	-.933	1.133
0	2	2		0	2	.444		0	2	.071	

x1 = 3- 2/3 = 7/3

x2 = 2.333 - (.444/1.667) = 2.067

x3 = 2.067 - (.071/1.133) = 2.00

✓ $f(x) = x^4 - 9 * x^3 - 2 * x^2 + 120x - 130, xo = -3.$

iteration					x_r
0	i	a_i	b_i	c_i	-3
	4	1	1	1	
	3	-9	-12	-15	
	2	-2	34	79	
	1	120	18	-219	
	0	-130	-184		
1	i	a_i	b_i	c_i	-3.84018
	4	1	1	1	
	3	-9	-12.8402	-16.6804	
	2	-2	47.30865	111.3643	
	1	120	-61.6738	-489.333	
	0	-130	106.8388		
2	i	a_i	b_i	c_i	-3.62185
	4	1	1	1	
	3	-9	-12.6218	-16.2437	
	2	-2	43.7144	102.5466	
	1	120	-38.3269	-409.735	
	0	-130	8.814062		

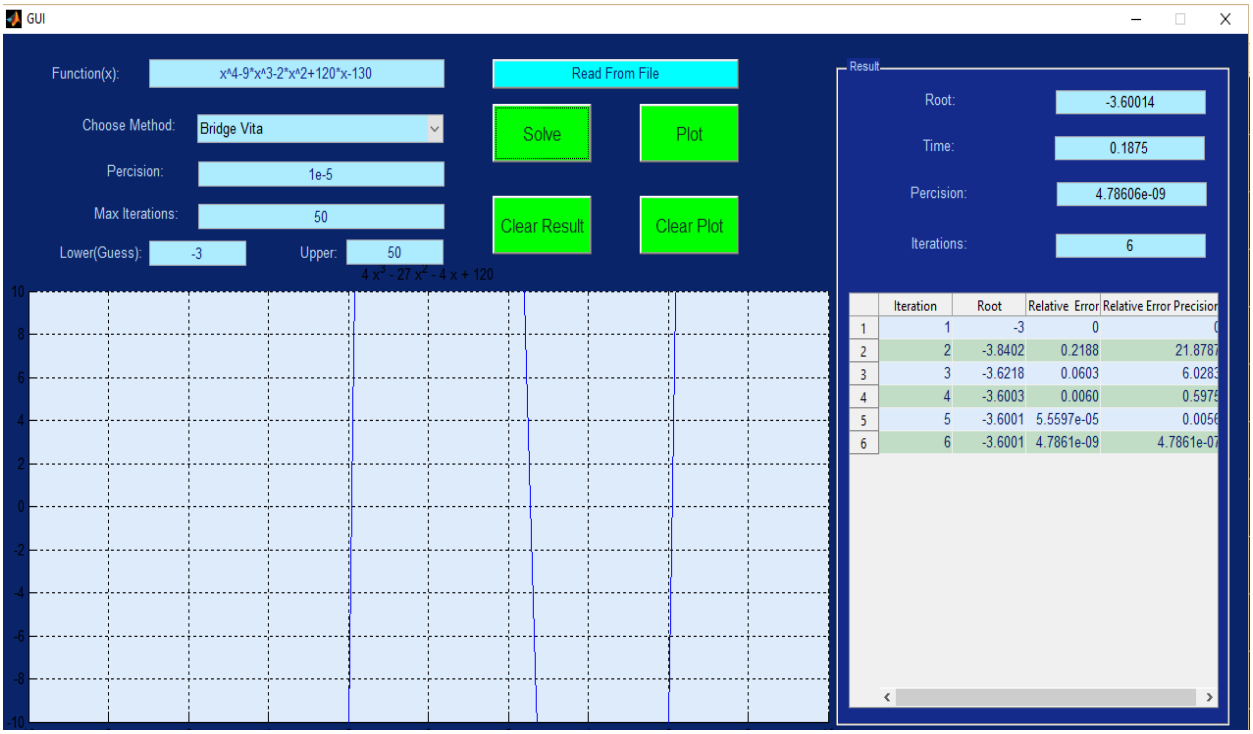
3	i	a_i	b_i	c_i	-3.60034
	4	1	1	1	
	3	-9	-12.6003	-16.2007	
	2	-2	43.36543	101.6933	
	1	120	-36.1301	-402.26	
	0	-130	0.080515		
4	i	a_i	b_i	c_i	-3.60014
	4	1	1	1	
	3	-9	-12.6001	-16.2003	
	2	-2	43.36219	101.6854	
	1	120	-36.1098	-402.191	
	0	-130	6.93E-06		
5	i	a_i	b_i	c_i	-3.60014
	4	1	1	1	
	3	-9	-12.6001	-16.2003	
	2	-2	43.36219	101.6854	
	1	120	-36.1098	-402.191	
	0	-130	0		

✓ $f(x) = x^3 - 12.6001 * x^2 + 43.36219 * x - 36.1098, xo = 1$

Sample Run:

iteration					x_r
0	i	a_i	b_i	c_i	1
	3	1	1	1	
	2	-12.6001	-11.6001	-10.6001	
	1	43.36219	31.76206	21.16192	
	0	-36.1098	-4.3477		
1	i	a_i	b_i	c_i	1.205449
	3	1	1	1	
	2	-12.6001	-11.3947	-10.1892	
	1	43.36219	29.62648	17.34387	
	0	-36.1098	-0.39654		
2	i	a_i	b_i	c_i	1.228313
	3	1	1	1	
	2	-12.6001	-11.3718	-10.1435	
	1	43.36219	29.39404	16.93463	
	0	-36.1098	-0.00468		

3	i	a_i	b_i	c_i	1.228589
	3	1	1	1	
	2	-12.6001	-11.3715	-10.143	
	1	43.36219	29.39123	16.9297	
	0	-36.1098	-6.8E-07		
4	i	a_i	b_i	c_i	1.228589
	3	1	1	1	
	2	-12.6001	-11.3715	-10.143	
	1	43.36219	29.39123	16.9297	
	0	-36.1098	0		



- **General Method (Secant):**

Justify and explain:

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

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

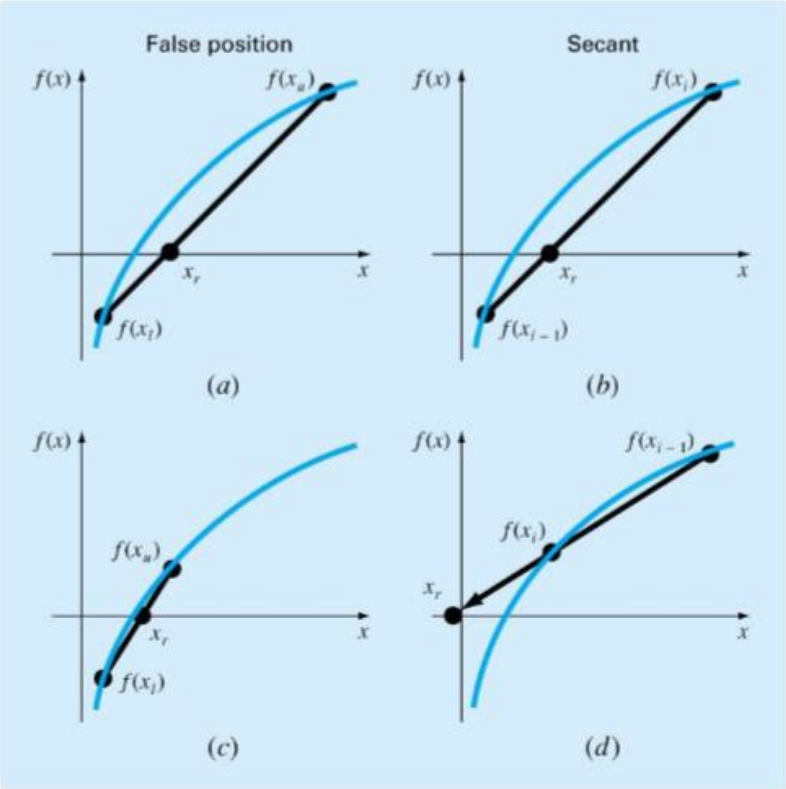
Comparison of the Secant and False-position method:

- Both methods use the same expression to compute x_r.

$$\text{Secant}: \quad x_{i+1} = x_i - \frac{f(x_i)(x_{i-1} - x_i)}{f(x_{i-1}) - f(x_i)}$$

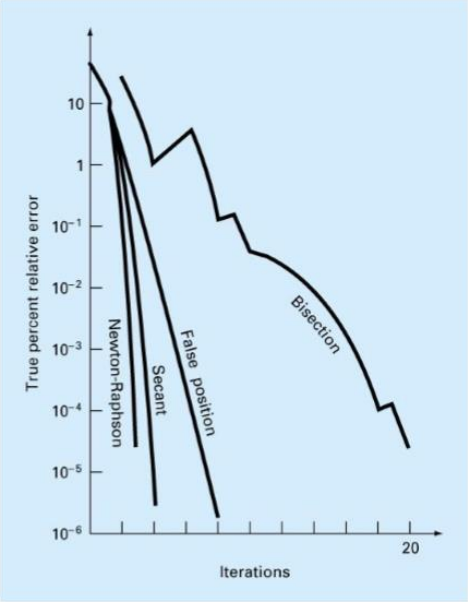
$$\text{False position}: \quad x_r = x_u - \frac{f(x_u)(x_l - x_u)}{f(x_l) - f(x_u)}$$

- They have different methods for the replacement of the initial values by the new estimate


False position
Secant
f(x)
f(x_u)
f(x_l)
x_r
x
(a)
f(x_i)
f(x_{i-1})
(b)
(c)
(d)

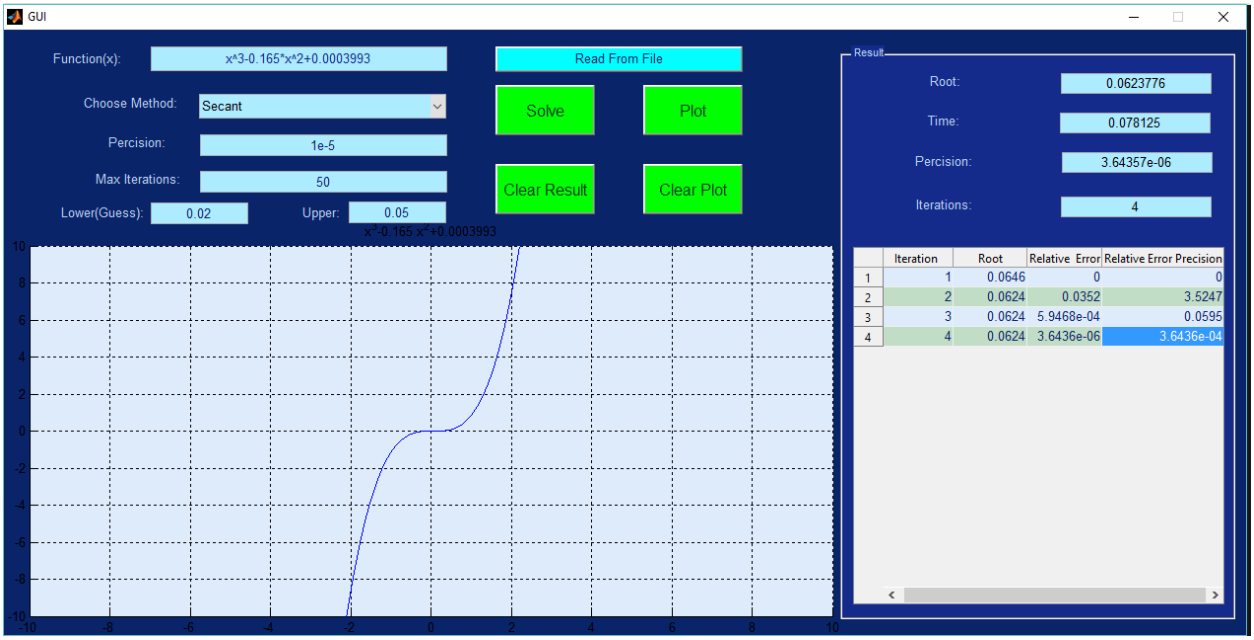
for

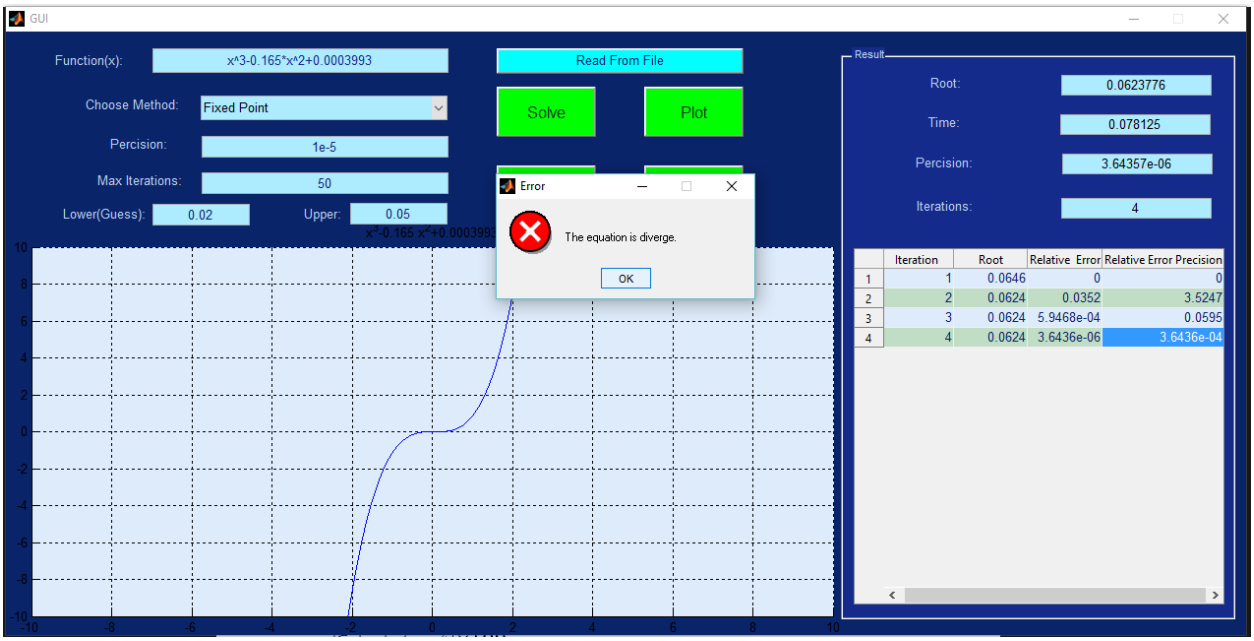
$$f(x) = e^{-x} - x$$

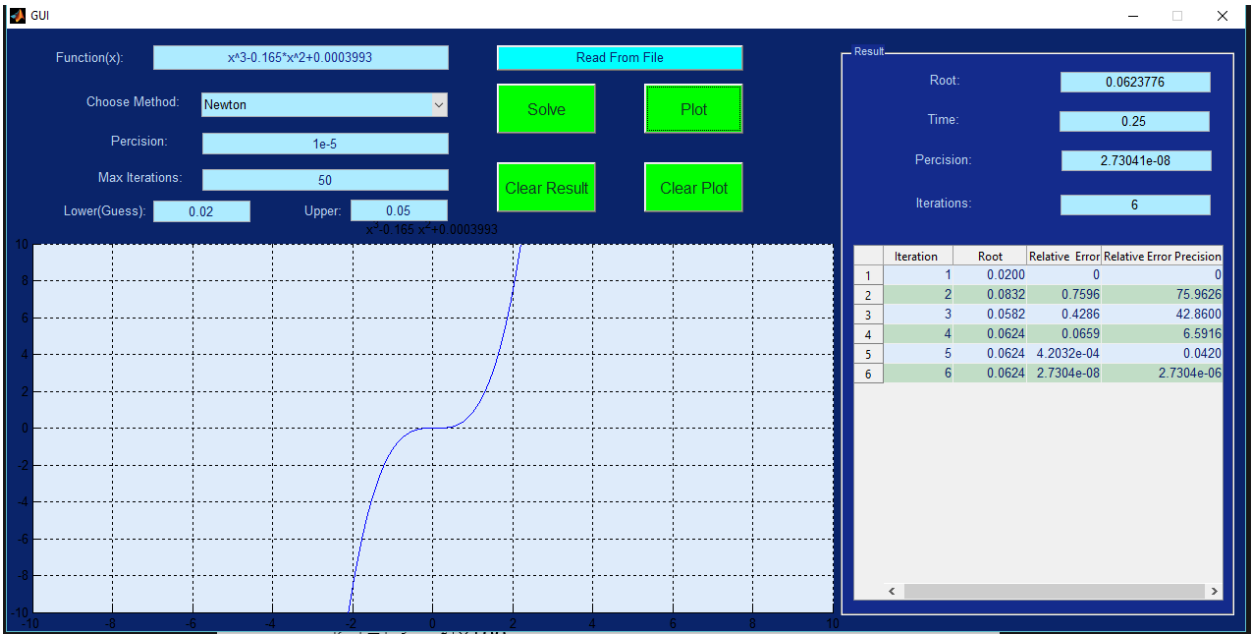

True percent relative error
10
1
10^-1
10^-2
10^-3
10^-4
10^-5
10^-6
Newton-Raphson
Secant
False position
Bisection
20
Iterations

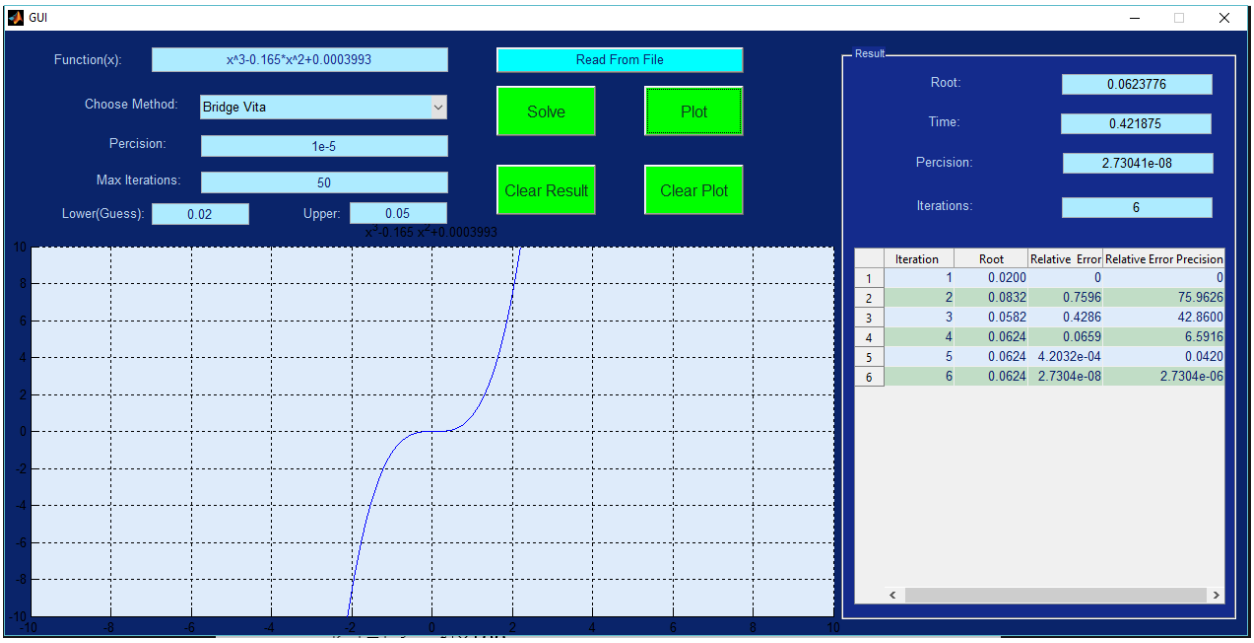
comparing secant method with all method:

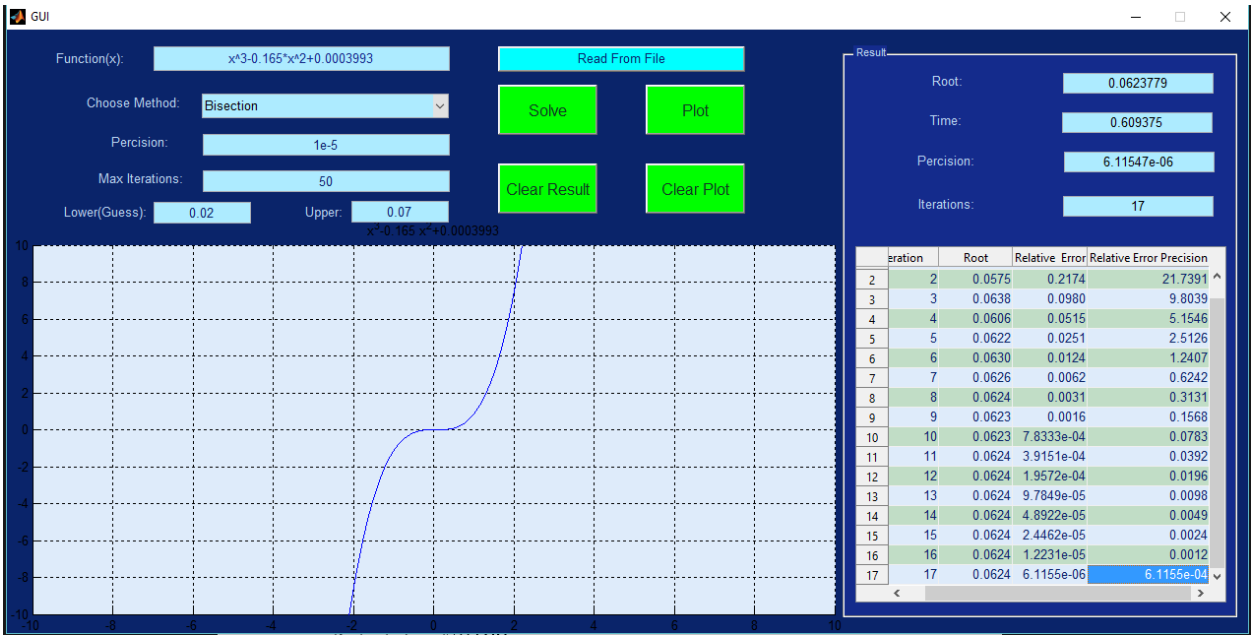
for $f(x) = x^3 - 0.165x^2 + 3.993\times10^{-4}$






GUI
Function(x):
x^3-0.165*x^2+0.0003993
Read From File
Choose Method:
Newton
Solve
Plot
Percision:
1e-5
Max Iterations:
50
Clear Result
Clear Plot
Lower(Guess):
0.02
Upper:
0.05
x^3-0.165 x^2+0.0003993
Result
Root:
0.0623776
Time:
0.25
Percision:
2.73041e-08
Iterations:
6
Iteration Root Relative Error Relative Error Precision
1 1 0.0200 0 0
2 2 0.0832 0.7596 75.9626
3 3 0.0582 0.4286 42.8600
4 4 0.0624 0.0659 6.5916
5 5 0.0624 4.2032e-04 0.0420
6 6 0.0624 2.7304e-08 2.7304e-06


GUI
Function(x):
x^3-0.165*x^2+0.0003993
Read From File
Choose Method:
Bridge Vita
Solve
Plot
Percision:
1e-5
Max Iterations:
50
Clear Result
Clear Plot
Lower(Guess):
0.02
Upper:
0.05
x^3-0.165 x^2+0.0003993
Result
Root:
0.0623776
Time:
0.421875
Percision:
2.73041e-08
Iterations:
6
Iteration Root Relative Error Relative Error Precision
1 1 0.0200 0 0
2 2 0.0832 0.7596 75.9626
3 3 0.0582 0.4286 42.8600
4 4 0.0624 0.0659 6.5916
5 5 0.0624 4.2032e-04 0.0420
6 6 0.0624 2.7304e-08 2.7304e-06



Advantage of Secant method:

- Converges fast, if it converges.
- Requires two guesses that do not need to bracket the root.

Newton Interpolation

Pseudo-Code:

Given n distinct points X0, X1,……., Xn

And the value of the function at these points Y0, Y1, Y2….Yn

And the point(P) at which the value of the function is required

We get f = f(p)

for $i = 0, 1, \ldots, n$ **do**
    $d_i = f(x_i)$
end
for $i = 1, 2, \ldots, n$ **do**
    for $j = n, n-1, \ldots, i$ **do**
        $d_j = (d_j - d_{j-1})/(x_j - x_{j-i})$
    end
end

A[] = diagonal(d)

For j = 2 , 3, …, n do

    Df(j, :) = (p – x(j - 2)).*Df(j – 1, :)

    C(j, :) = a(j).*Df(j, :) End

F = sum(c);

Data structure Used:

One-dimensional array :

1 - x[] as input

2 – y[] as input

Two-dimensional array:

1 – d[][] used for computing the f(x) at the point given

Lagrange Interpolation

Pseudo-Code :

INPUT: vector x; vector $y = f(x)$; a point to evaluate z
OUTPUT: Pz Lagrange polynomial $P(x)$ evaluated at z

Step 1 Initialize variables. Set Pz equal zero. Set n to the number of pairs of points (x, y). Set L to be the all ones vector of length n.

Step 2 For $i = 1$ to n do ...

Step 3 For $j = 1$ to n do Step 4.

Step 4 If $i \neq j$ then $L_i = (z - x_j)/(x_i - x_j) * L_i$

Step 5 $Pz = L_i * y_i + Pz$

Step 6 Output Pz. Stop.

Data Structure Used:

1 - Vector x given (x-points)

2 – vector y given (y - points)

3 – vector L used to compute the required point

Analysis for Lagrange

```
function [y, final, error]=lagrange(pointx,pointy,x,n)
final = strcat('', '0');
error = strcat('', '');
if (size(pointx,1)~=size(pointy,1))
    error = strcat('', 'ERROR!! X array and Y array must have the same number of elements');
    y=NaN;
    return;
end
if( (x<pointx(1)) || (x>pointx(size(pointx,1))) )
    error = strcat('', 'ERROR!! You want to find extrapolation and this method calculates only Interpolation!!');
    y=NaN;
    return;
end
if(n > size(pointx,1)-1)
    error = strcat('', 'ERROR!! Order of polynomial must be <= size of array - 1 !!');
    y=NaN;
    return;
end
p = 0;
for(i=1:size(pointx,1))
    if(pointx(i)>=x)
        p = i;
        break;
    end
end
```

```
while(1)
    if(p-ceil(n/2)>=1)
        if(p+floor(n/2)<=size(pointx,1))
            p = p-ceil(n/2);
            break;
        else
            p = p-1;
        end
    else
        p = p+1;
    end
end
L=ones(n+1);
y=0;
for i=1:(n+1)
  result = strcat('', int2str(1));
  for j=1:(n+1)
     if (i~=j)
         L(i)=L(i).*(x-pointx(j+p-1))/(pointx(i+p-1)-pointx(j+p-1));
         result = strcat(result, '*(1/', '(', num2str((pointx(i+p-1)-pointx(j+p-1))), ')', ')', '*(x-', '(' , num2str(pointx(j+p-1)), ')', ')');
     end
  end
  y=y+pointy(i+p-1)*L(i);
  final = strcat(final, '+', '(', num2str(pointy(i+p-1)), ')', '*', result);
end
```

The function takes 4 inputs:

1 - x-points

2 - f(x) correspond to x (y - points)

3 - the point at which f(x) is required

4 - the order of polynomial

The function produces 3 outputs :

1 - f(x) of the given point

2 - error In computation

3 – a string representing the function to be drawn

Details:

1 – the function make sure of valid inputs :

a - the size of x-point should be equal the size of y-points

b – the point given should be within the interval of point given

c – order of the polynomial should be less than or equal size of ((x- points) - 1) or ((y – points) – 1)

2 – the function declare a vector L of ones used for computation

3 – the pseudo code explained is to be executed to compute L

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

4 – error is calculated by the formula explained

5 – the point is calculated by substitution in the equation

6 – A string representing the function is formed to be drawn

Example(0) for analysis:

x-points = [0, 2, 3, 4, 5]

y- points = [2, 14, 74, 242, 602]

order = 4

x = 3.5

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	$f(x_4)$	$f[x_4,x_3]$	$f[x_4,x_3,x_2]$	$f[x_4,x_3,x_2,x_1]$

x_i	$f(x_i)$	$f[x_i,x_j]$	$f[x_i,x_j,x_k]$	$f[x,x,x,x]$	$f[x...x]$
x_0=0	2				
x_1=2	14	6			
x_2=3	74	60	18		
x_3=4	242	168	54	9	
x_4=5	602	360	96	14	1

b0 = 2, b1 = 6, b2 = 18, b3 = 9, b4 = 1

f(x) = 2 + 6x +18x(x-2) +9x(x-2)(x-3) +1x(x-2)(x-3)(x-4)

f(3.5) = 139.8125

Example (1):

x-points = [0, 10, 15, 20, 22.5, 30]

y- points = [0, 227.04, 362.78, 517.35, 602.97, 901.67]

order = 5

x = 6

find y correspond to x = 6

the function generates results as y = 392.071

and the time = 0.01

the string representing the function is

(0)*1*(1/(-10))*(x-(10))*(1/(-15))*(x-(15))*(1/(-20))*(x-(20))*(1/(-22.5))*(x-(22.5))*(1/(-30))*(x-(30))+(227.04)*1*(1/(10))*(x-(0))*(1/(-5))*(x-(15))*(1/(-10))*(x-(20))*(1/(-12.5))*(x-(22.5))*(1/(-20))*(x-(30))+(362.78)*1*(1/(15))*(x-(0))*(1/(5))*(x-(10))*(1/(-5))*(x-(20))*(1/(-7.5))*(x-(22.5))*(1/(-15))*(x-(30))+(517.35)*1*(1/(20))*(x-(0))*(1/(10))*(x-(10))*(1/(5))*(x-(15))*(1/(-2.5))*(x-(22.5))*(1/(-10))*(x-(30))+(602.97)*1*(1/(22.5))*(x-(0))*(1/(12.5))*(x-(10))*(1/(7.5))*(x-(15))*(1/(2.5))*(x-(20))*(1/(-7.5))*(x-(30))+(901.67)*1*(1/(30))*(x-(0))*(1/(20))*(x-(10))*(1/(15))*(x-(15))*(1/(10))*(x-(20))*(1/(7.5))*(x-(22.5))

Example(2):

x- points = [2, 4.25, 5.25, 7.81, 9.2, 10.6]

y- points = [7.2, 7.1, 6, 5, 3.5, 5]

order = 5

x = 4

the function generates results as y = 7.47497

and the time = 0.01

the string representing the function is

(7.2)*1*(1/(-2.25))*(x-(4.25))*(1/(-3.25))*(x-(5.25))*(1/(-5.81))*(x-(7.81))*(1/(-7.2))*(x-(9.2))*(1/(-8.6))*(x-(10.6))+(7.1)*1*(1/(2.25))*(x-(2))*(1/(-1))*(x-(5.25))*(1/(-3.56))*(x-(7.81))*(1/(-4.95))*(x-(9.2))*(1/(-6.35))*(x-(10.6))+(6)*1*(1/(3.25))*(x-(2))*(1/(1))*(x-(4.25))*(1/(-2.56))*(x-(7.81))*(1/(-3.95))*(x-(9.2))*(1/(-5.35))*(x-(10.6))+(5)*1*(1/(5.81))*(x-(2))*(1/(3.56))*(x-(4.25))*(1/(2.56))*(x-(5.25))*(1/(-1.39))*(x-(9.2))*(1/(-2.79))*(x-(10.6))+(3.5)*1*(1/(7.2))*(x-(2))*(1/(4.95))*(x-(4.25))*(1/(3.95))*(x-(5.25))*(1/(1.39))*(x-(7.81))*(1/(-1.4))*(x-(10.6))+(5)*1*(1/(8.6))*(x-(2))*(1/(6.35))*(x-(4.25))*(1/(5.35))*(x-(5.25))*(1/(2.79))*(x-(7.81))*(1/(1.4))*(x-(9.2))

Example(3):

x- points = [0, 2, 4, 6, 9, 10]

y- points = [-4, 0, 2, 7, 15, 20]

order = 5

x = 5

the function generates results as y = 4.2748

and the time = 0.03

the string representing the function is

(-4)*1*(1/(-2))*(x-(2))*(1/(-4))*(x-(4))*(1/(-6))*(x-(6))*(1/(-9))*(x-(9))*(1/(-10))*(x-(10))+(0)*1*(1/(2))*(x-(0))*(1/(-2))*(x-(4))*(1/(-4))*(x-(6))*(1/(-7))*(x-(9))*(1/(-8))*(x-(10))+(2)*1*(1/(4))*(x-(0))*(1/(2))*(x-(2))*(1/(-2))*(x-(6))*(1/(-5))*(x-(9))*(1/(-6))*(x-(10))+(7)*1*(1/(6))*(x-(0))*(1/(4))*(x-(2))*(1/(2))*(x-(4))*(1/(-3))*(x-(9))*(1/(-4))*(x-(10))+(15)*1*(1/(9))*(x-(0))*(1/(7))*(x-(2))*(1/(5))*(x-(4))*(1/(3))*(x-(6))*(1/(-1))*(x-(10))+(20)*1*(1/(10))*(x-(0))*(1/(8))*(x-(2))*(1/(6))*(x-(4))*(1/(4))*(x-(6))*(1/(1))*(x-(9))

Analysis for Newton

```
function [f, final, error] = Newton (x, y, p, n)
final = strcat('', '0');
error = strcat('', '');
if (size(x,1)~=size(y,1))
    error = strcat('', 'ERROR!! X array and Y array must have the same number of elements');
    f=NaN;
    return;
end
if( (p<x(1)) || (p>x(size(x,1))) )
    error = strcat('', 'ERROR!! You want to find extrapolation and this method calculates only Interpolation!!');
    f=NaN;
    return;
end
if(n > size(x,1)-1)
    error = strcat('', 'ERROR!! Order of polynomial must be <= size of array - 1 !!');
    f=NaN;
    return;
end
m = 0;
for(i=1:size(x,1))
    if(x(i)>=p)
        m = i;
        break;
    end
end
while(1)
    if(m-ceil(n/2)>=1)
        if(m+floor(n/2)<=size(x,1))
            m = m-ceil(n/2);
            break;
        else
            m = m-1;
        end
    else
        m = m+1;
    end
last = m+n;
n = n + 1;
d(:,1)=y(m:last)';
for j=2:n
     for i=j:n
         d(i,j)= ( d(i-1,j-1)-d(i,j-1)) / (x(i-j+m)-x(i+m-1));
     end
end
a = diag(d)';
Df(1,:) = repmat(1, size(p));
c(1,:) = repmat(a(1), size(p));
for j = 2 : n
   Df(j,:)=(p - x(j+m-2)) .* Df(j-1,:);
   c(j,:) = a(j) .* Df(j,:);
end
final  = strcat('', num2str(a(1)));
str1 = strcat('', '');
k = 1;
str2 = strcat('', '');
for i = 2:length(a)
  for j = 1:k
    if(j ~= k)
      str1 = strcat(str1, '(x - ', '(', num2str(x(j+m-1)), ')', ')', '*');
    else
      str1 = strcat(str1, '(x - ', '(', num2str(x(j+m-1)), ')', ')');
    end
  end
  k = k + 1;
  str2 = strcat(str2, '(', num2str(a(i)), ')', ' * ', str1, ' + ');
  str1 = strcat('', '');
end
f=sum(c);
final = strcat(final, ' + ', str2, ' 0 ');
```

The function takes 4 inputs:

1 - x-points

2 - f(x) correspond to x (y - points)

3 - the point at which f(x) is required

4 - the order of polynomial

The function produces 3 outputs :

1 - f(x) of the given point

2 - error In computation

3 – a string representing the function to be drawn

Details:

1 – the function make sure of valid inputs :

a - the size of x-point should be equal the size of y-points

b – the point given should be within the interval of point given

c – order of the polynomial should be less than or equal size of ((x- points) - 1) or ((y – points) – 1)

2 – the function declare a two dimensional array d, the first column is the y-points

3 – then a 1-dimentional array representing the coefficients [b0, b1, b2, …. bn]

Is calculated

4 – error is calculated by the formula explained

5 – the point is calculated by substitution in the equation

6 – then a string representing the function is estimated

Example (0) for analysis:

t-points = [0, 10, 15, 20, 22.5, 30]

v(t) = y- points = [0, 227.04, 362.78, 517.35, 602.97, 901.67]

order = 2

x = 16

$$v(t)=\left(\frac{t-t_1}{t_0-t_1}\right)\left(\frac{t-t_2}{t_0-t_2}\right)v(t_0)+\left(\frac{t-t_0}{t_1-t_0}\right)\left(\frac{t-t_2}{t_1-t_2}\right)v(t_1)+\left(\frac{t-t_0}{t_2-t_0}\right)\left(\frac{t-t_1}{t_2-t_1}\right)v(t_2)$$

$$\begin{aligned} v(16) &= \left(\frac{16-15}{10-15}\right)\left(\frac{16-20}{10-20}\right)(227.04)+\left(\frac{16-10}{15-10}\right)\left(\frac{16-20}{15-20}\right)(362.78)+\left(\frac{16-10}{20-10}\right)\left(\frac{16-15}{20-15}\right)(517.35) \\ &= (-0.08)(227.04)+(0.96)(362.78)+(0.12)(527.35) \\ &= 392.19 \text{ m/s} \end{aligned}$$

Example (1):

x-points = [0, 10, 15, 20, 22.5, 30]

y- points = [0, 227.04, 362.78, 517.35, 602.97, 901.67]

order = 5

x = 6

find y correspond to x = 6

the function generates results as y = 392.071

and the time = 0.01

the string representing the function is

(22.704) *(x -(0)) +(0.29627) *(x -(0))*(x -(10)) +(0.0040167) *(x -(0))*(x -(10))*(x -(15)) +(6.3022e-05) *(x -(0))*(x -(10))*(x -(15))*(x -(20)) +(1.4341e-06) *(x -(0))*(x -(10))*(x -(15))*(x -(20))*(x -(22.5))

Example(2):

x- points = [2, 4.25, 5.25, 7.81, 9.2, 10.6]

y- points = [7.2, 7.1, 6, 5, 3.5, 5]

order = 5

x = 4

the function generates results as y = 7.47497

and the time = 0.01

the string representing the function is

7.2 +(-0.044444) *(x -(2)) +(-0.32479) *(x -(2))*(x -(4.25)) +(0.090198) *(x -(2))*(x -(4.25))*(x -(5.25)) +(-0.023009) *(x -(2))*(x -(4.25))*(x -(5.25))*(x -(7.81)) +(0.0072923) *(x -(2))*(x -(4.25))*(x -(5.25))*(x -(7.81))*(x -(9.2)

Example(3):

x- points = [0, 2, 4, 6, 9, 10]

y- points = [-4, 0, 2, 7, 15, 20]

order = 5

x = 5

the function generates results as y = 4.2748

and the time = 0.01

the string representing the function is

-4 +(2) *(x -(0)) +(-0.25) *(x -(0))*(x -(2)) +(0.10417) *(x -(0))*(x -(2))*(x -(4)) +(-0.016997) *(x -(0))*(x -(2))*(x -(4))*(x -(6)) +(0.0034557) *(x -(0))*(x -(2))*(x -(4))*(x -(6))*(x -(9))

Conclusion

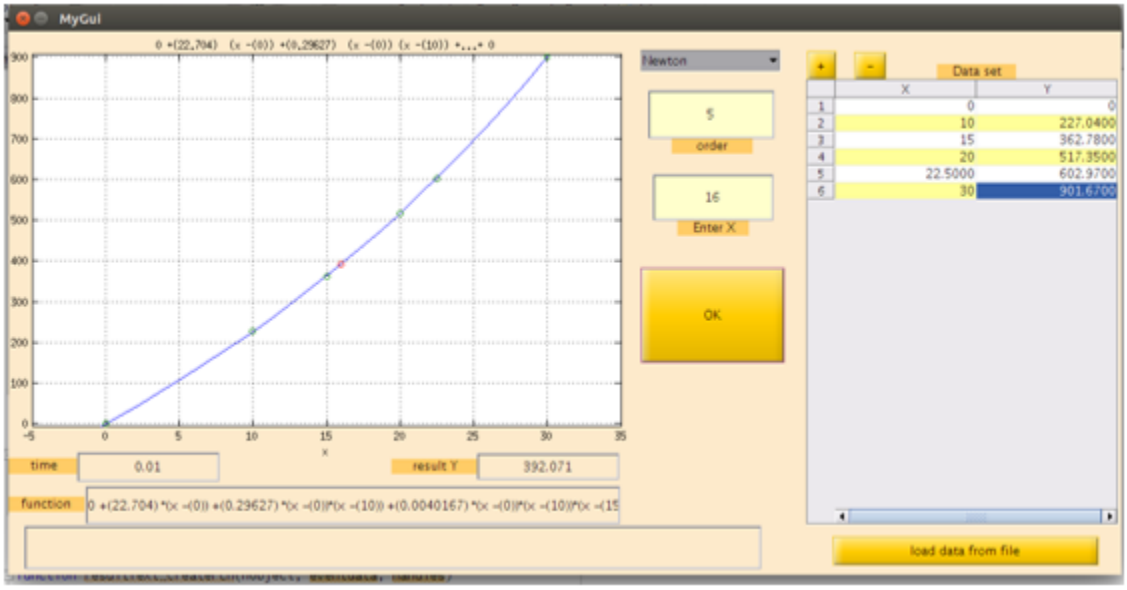
Lagrange interpolation is mostly just useful for theory. Actually computing with it requires huge numbers and catastrophic cancellations. In floating point arithmetic this is very bad. It does have some small advantages: for instance, the Lagrange approach amounts to diagonalizing the problem of finding the coefficients, so it takes only linear time to find the coefficients. This is good if you need to use the same set of points repeatedly. But all of these advantages do not make up for the problems associated with trying to actually evaluate a Lagrange interpolating polynomial.

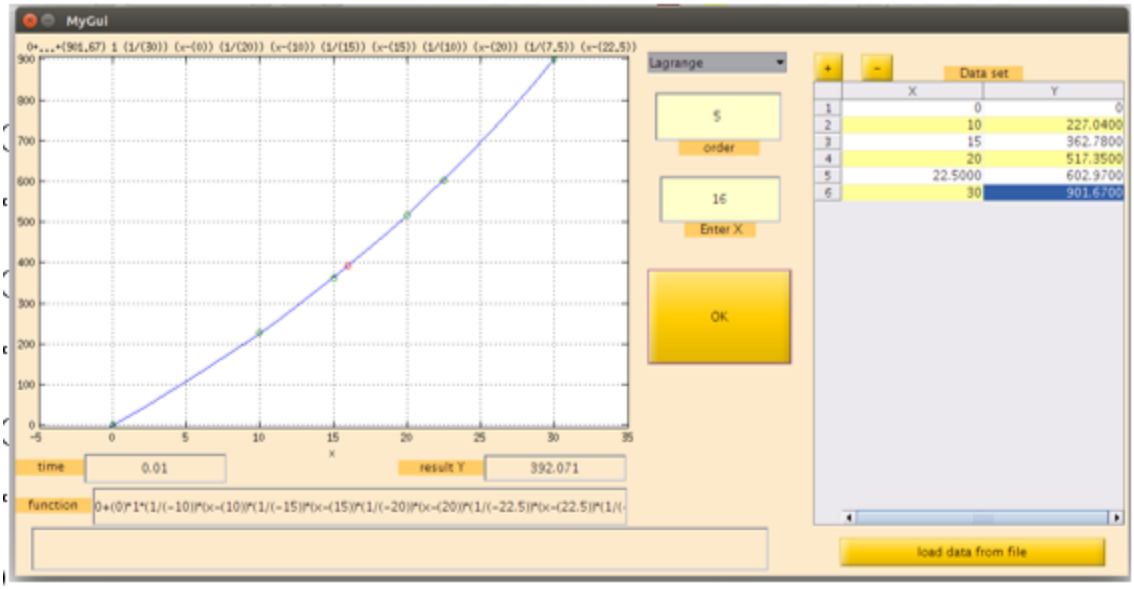
It needs about n^2 arithmetic operations to calculate coefficients

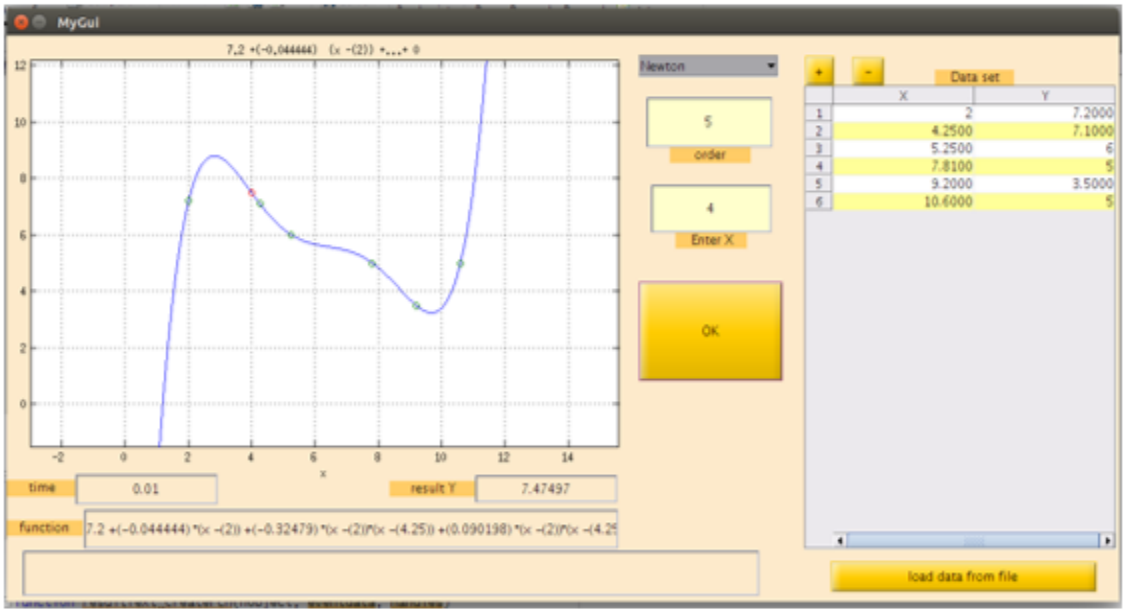
With Newton interpolation, you get the coefficients reasonably fast (quadratic time), the evaluation is much more stable (roughly because there is usually a single dominant term for a given xx), the evaluation can be done quickly and straightforwardly using Horner's method, and adding an additional node just amounts to adding a single additional term. It is also fairly easy to see how to interpolate derivatives using the Newton framework

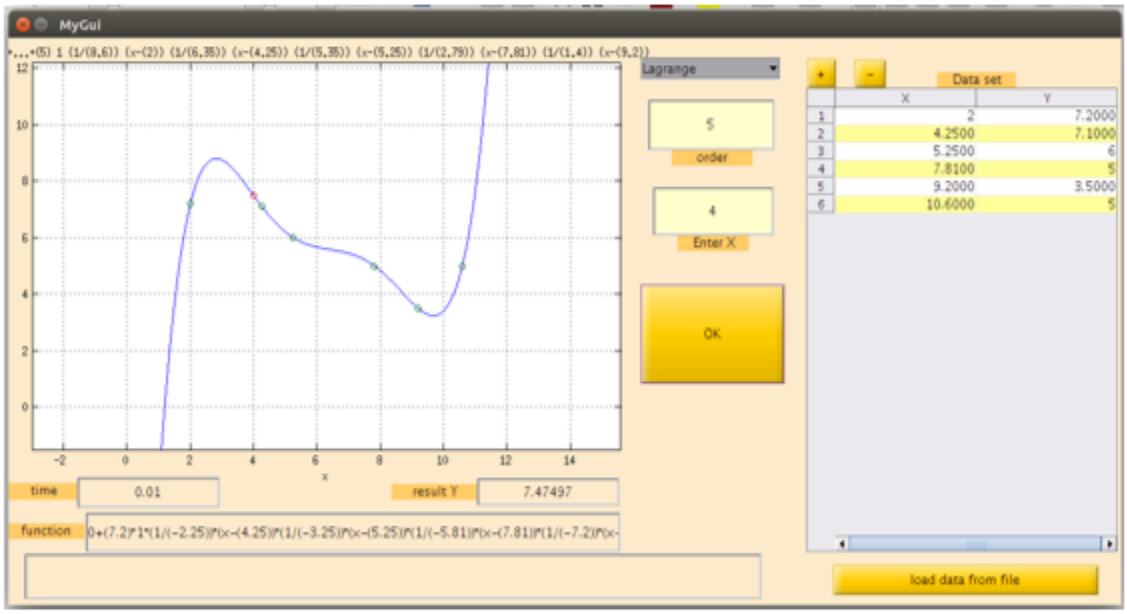
It needs about (n^2)/2 arithmetic operations to calculate coefficients

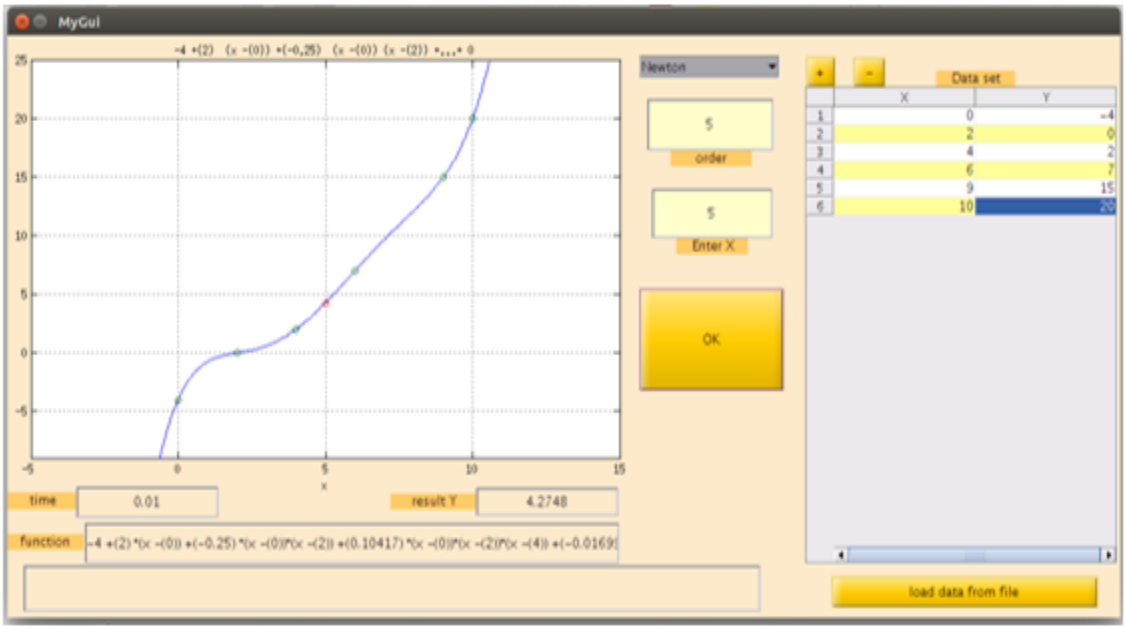
Sample Runs And Snap Shots

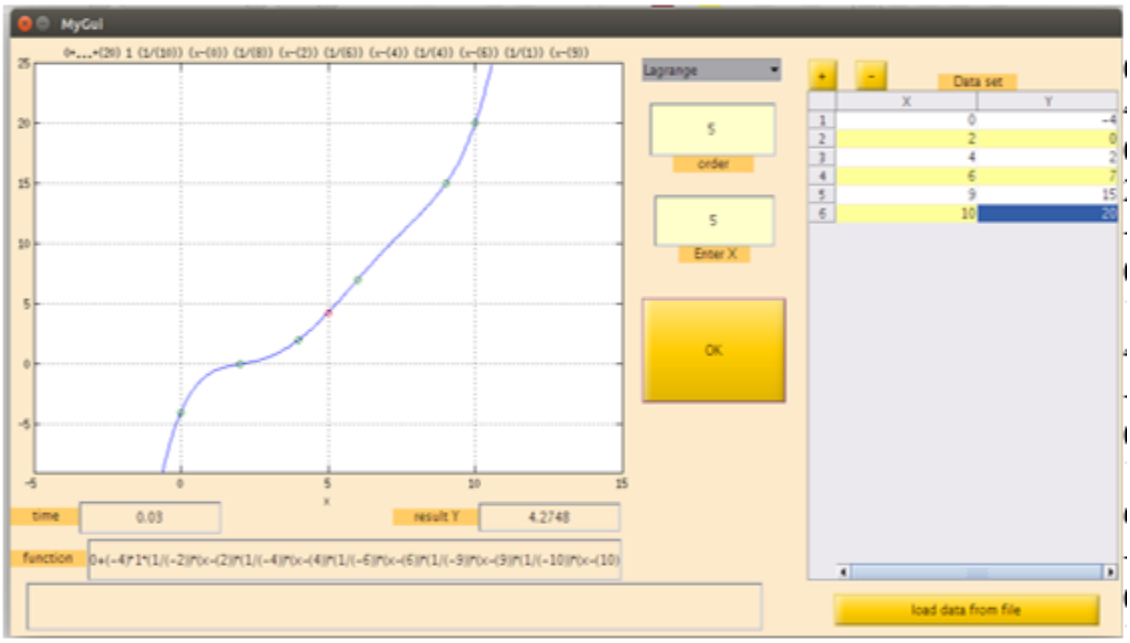



MyGui
Lagrange
5
order
16
Enter X
OK
Data set
X
Y
1 0 0
2 10 227.0400
3 15 362.7800
4 20 517.3500
5 22.5000 602.9700
6 30 901.6700
time
0.01
result Y
392.071
function
load data from file


MyGui
Newton
5
order
4
Enter X
OK
Data set
X
Y
1 2 7.2000
2 4.2500 7.1000
3 5.2500 6
4 7.8100 5
5 9.2000 3.5000
6 10.6000 5
time
0.01
result Y
7.47497
function
load data from file


MyGui
Lagrange
5
order
4
Enter X
OK
Data set
X
Y
1 2 7.2000
2 4.2500 7.1000
3 5.2500 6
4 7.8100 5
5 9.2000 3.5000
6 10.6000 5
time
0.01
result Y
7.47497
function
load data from file


MyGui
Newton
5
order
5
Enter X
OK
Data set
X
Y
1 0 -4
2 2 0
3 4 2
4 6 7
5 9 15
6 10 20
time
0.01
result Y
4.2748
function
load data from file


MyGui
Lagrange
5
order
5
Enter X
OK
Data set
X
Y
1 0 -4
2 2 0
3 4 2
4 6 7
5 9 15
6 10 20
time
0.03
result Y
4.2748
function
load data from file