2019

Numerical Report

NAME: AHMED ALI ABD EL MEGUID ALI HASSANEIN ID:5
NAME: SHERIF MOHAMED MOSTAFA MOHAMED ID:20
NAME: HAZEM MORSY HASSAN MORSY ID:16
NAME: BASSAM RAGEH EBARHEM ID:13

“Numerical analysis project”

“Part: 1”

Pseudocode:

1) Bisection:

Function Bisection(equ,iterations,es,xl,xu)

INPUTS: equation(equ) , number of iterations(itts) , epilson(es) and initial guesses(xl,xu)

OUTPUTS: function , root , array of iterations, array of relative errors , boundary condition, time.

```
For (i =1:iterations)
    xr = (xl + xu) / 2      // function calculates the next root
    xls(i,1) = xl           // array contains the lower guesses
    xus(i,1) = xu           // array contains the higher guesses
    xrs(i,1) = xr           // array contains the suggested roots
    if I >1 then
// calculates relative error starting from second iteration [ |xnew - xold| / xnew ] * 100
        ea = [ |xrs(i,1) – xrs(i-1,1) | / xrs(i,1) } ] * 100
        eas(i,1) = ea
    end if
    check = f(xr) * f(xl)     //get tighter limits to proceed to a new root
    fxls(i,1) = f(xl)         // array  contains f(x) for each xlower
    fxrs(i,1) = f(xr)         // array contains f(x) for each suggested root
    if check is positive then
        xl = xr
    else
```

```
        xu = xr
    end if
    // check for reaching the root or the allowed bound of error
    if (ea < es   and   i>1)  or  f(xr) = es  then
        get out of the loop
    end if
end for
timeelapsed = time taken by these iterations
arr = concatenation of (xrs,xls,xus,fxls,fxrs)
end function
```

2) False-Position

Function falseposition(equ,iterations,es,xl,xu)

INPUTS: equation(equ) , number of iterations(itts) , epilson(es) and initial guesses(xl,xu)

OUTPUTS: function , root , array of iterations, array of relative errors , time.

```
For (i =1:iterations)
    xr = ( xl*f(xu) – xu*f(xl) ) / (f(xu) – f(xl) )  // function calculates the next root
    xls(i,1) = xl       // array contains the lower guesses
    xus(i,1) = xu       // array contains the higher guesses
    xrs(i,1) = xr       // array contains the suggested roots
    if I >1 then
// calculates relative error starting from second iteration [ |xnew - xold| / xnew ] * 100
        ea =  [ |xrs(i,1) – xrs(i-1,1) | / xrs(i,1) } ] * 100
        eas(i,1) = ea
    end if
    check = f(xr) * f(xl)       //get tighter limits to proceed to a new root
    fxls(i,1) = f(xl)           // array  contains f(x) for each xlower
```

```
    fxrs(i,1) = f(xr)            // array contains f(x) for each suggested root
    if check is positive then
        xl = xr
    else
        xu = xr
    end if
    // check for reaching the root or the allowed bound of error
    if (ea < es   and   i>1)  or  f(xr) = es  then
        get out of the loop
    end if
end for
timeelapsed = time taken by these iterations
arr = concatenation of (xrs,xls,xus,fxls,fxrs)
end function
```

3) **Fixed-Point**

```
Function FixedPoint(equ,iterations,es,x0)
```

INPUTS: equation(equ) , number of iterations(itts) , epilson(es) and initial guess(x0)

OUTPUTS: function , root , array of iterations, array of relative errors , boundary condition, time.

```
numb = coefficient of (x) in the equation * (-1)
  // g(x) (magic function) is the equation (equ) divided by negative coefficient of x
f = (f(x) + numb*x ) / numb
xi = x0
For (i =1:iterations)
    arr(i,1) = xi       // array contains the guesses
```

```
    gs(i,1) = f(xi)    // array contains g(x) for each iteration
    if I >1 then
// calculates relative error starting from second iteration [ |xnew - xold| / xnew ] * 100
        ea =  [ |xrs(i,1) – xrs(i-1,1) | / xrs(i,1) } ] * 100
        eas(i,1) = ea
    end if
    // check for reaching the root or the allowed bound of error
    if (ea < es   and   i>1)  or  f(xi) = xi  then
        get out of the loop
    end if
    xi = f(xi)    // function calculates next root
end for
timeelapsed = time taken by these iterations
xr = xi
arr = concatenation of (arr,gs)
end function
```

4) <u>**Newton**</u>

Function Newton(equ,iterations,es,x0)

INPUTS: equation(equ) , number of iterations(itts) , epilson(es) and initial guess(x0)

OUTPUTS: function , root , array of iterations, array of relative errors , boundary condition, time.

```
df = derivative of function (equ)
fn = x – f(x) / df(x)
xi = x0
For (i =1:iterations)
    arr(i,1) = xi        // array  contains the guesses
```

```
ff(i,1) = f(xi)       // array contains f(x) for each guess
deriv(i,1) = df(xi)    // array contains the derivative of function for each guess
if I >1 then
// calculates relative error starting from second iteration [ |xnew - xold| / xnew ] * 100
    ea =   [ |xrs(i,1) – xrs(i-1,1) | / xrs(i,1) } ] * 100
    eas(i,1) = ea
end if
// check for reaching the root or the allowed bound of error
if (ea < es   and   i>1)  or  f(xi) = xi  then
    get out of the loop
end if
xi = fn(xi)    // function calculates next root
end for
timeelapsed = time taken by these iterations
xr = xi
arr = concatenation of (arr,ff,deriv)
end function
```

5) <u>secant</u>

Function secant(equ,iterations,es,x0,x1)

INPUTS: equation(equ) , number of iterations(itts) , epilson(es) and initial guesses(x0,x1)

OUTPUTS: function , root , array of iterations, array of relative errors , time.

```
 xi0 = xo
 xi1 = x1
For (i =1:iterations)
```

```
xi0s(i,1) = xi0          // array contains the first guesses
xi1s(i,1) = xi1          // array contains the second guesses
fxi0s(i,1) = f(xi0)     // array contains f(x) for each first guess
fxi1s(i,1) = f(xi1)     // array contains f(x) for each second guess
xi = xi1 – [f(xi1)*(xi0-xi1) / { f(xi0)-f(xi1) } ] // function calculates the next root
xrs(i,1) = xi    // array that contains the new guesses of each iteration
//  adjust the new guesses
xi0 = xi1
xi1 = xi
if I >1 then
// calculates relative error starting from second iteration [ |xnew - xold| / xnew ] * 100
    ea =   [ |xrs(i,1) – xrs(i-1,1) | / xrs(i,1) } ] * 100
    eas(i,1) = ea
end if
// check for reaching the root or the allowed bound of error
if (ea < es   and   i>1)  or  f(xr) = es  then
    get out of the loop
end if
end for
xr = xi
timeelapsed = time taken by these iterations
arr = concatenation of (xrs,xi0s,xi1s,fxi0s,fxi1s)
end function
```

6) Birge – Vieta

Function BirgeVeta(equ,iterations,es,xi)

INPUTS: equation(equ) , number of iterations(itts) , epilson(es) and initial guess(xi)

OUTPUTS: function , root , number of coefficients of polynomial,array of iterations, array contains (coeffcients,b,c),array of relative errors , time.

```
A = array  contains all coeffcients of the polynomial

For (i =1:iterations)
    arr(i,1) = xi
    for j = 2 : sizerow
        B(j,1) = B(j-1,1) * xi + A(j,1)
        C(j,1) = C(j-1,1) * xi – B(j,1)
    end for
   temparr = array concatenates (A,B,C)
   all = array concatenates (all,temparr)      // array  of (A,B,C) of every iteration
    if I >1 then
// calculates relative error starting from second iteration [ |xnew - xold| / xnew ] * 100
          ea =   [ |xrs(i,1) – xrs(i-1,1) | / xrs(i,1) } ] * 100
          eas(i,1) = ea
    end if
   // check for reaching the root or the allowed bound of error
    if (ea < es   and   i>1)  or  f(xi) = 0  then
         get out of the loop
   end if
   xi = xi – B(sizerow,1) / C(sizerow-1,1)    // function calculates the next root
end for
xr = xi
timeelapsed = time taken by these iterations
end function
```

General algorithm (Dekker's method)

Dekker's method combines bisection and secant. It finds two points (initial guess) xu, xl where f(xu)*f(xl)>0. This guarantees the existence of a root between the 2 points.

The next guess is calculated using secant but if it doesn't lie in the interval then bisection is used instead. This makes it converge faster than bisection in most cases.

This method finds multiple roots in the interval [-50, 50].

Pseudo code

Input: equation.

Output: array containing all roots.

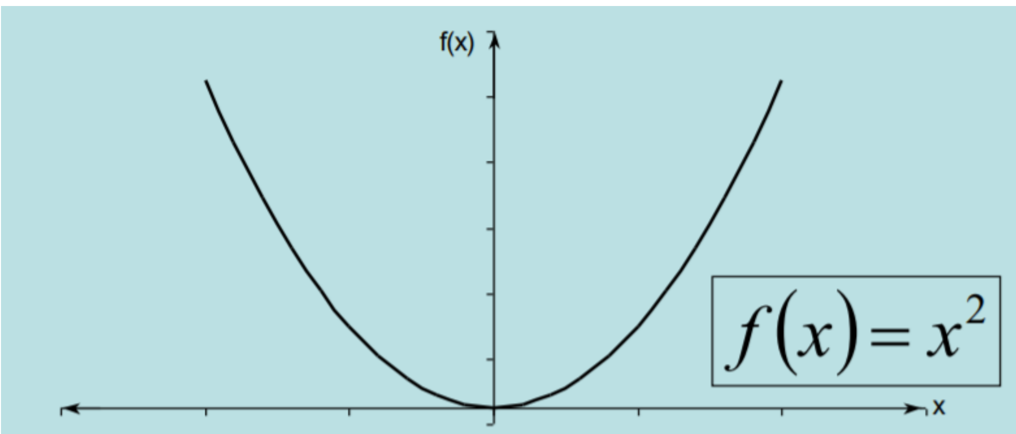
```
//initial guess
xl = -50
xu = -49
n=1       //number of roots
while xu<50    //loops from -50 to 50 with intervals of size 1
  if f(xl)*f(xu)>0  //checks valid interval
    xl = xl+1
    xu = xu+1
    continue
  end if
  temp=xu
  for max iterations
    if |f(xl)|<|f(xu)|     //checks that xu is the most recent guess
      temp=xl
      xl=xu
      xu=temp
    end if
    m=(xl+xu)/2   //bisection
    s= xu - (f(xu)*(xu-xl))/(f(xu)-f(xl))    //secant
    if ((s>xu&&s<xl)&&(xu<xl))||((s>xl&&s<xu)&&(xl<xu))
```

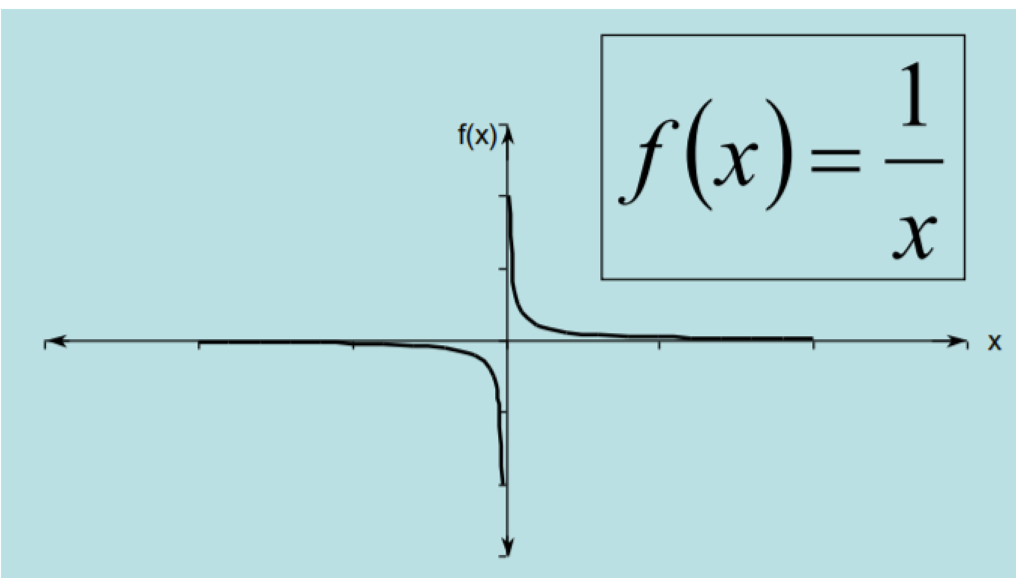
```
            xr=s
         else
            xr=m
         end if
         if f(xl)*f(xr)<0   //bracketing for next guess
            xu=xr
         else
            xl=xu
            xu=xr
         end if
         ea=xu-xl      //absolute error
         if abs(ea)<eps || f(xr)==0
            break;
         end
      end
      roots.add(xr)
      n=n+1
      xl=temp
      xu=xl+1
   end while
```

Problems with each method:

Bisection

If a function f(x) just touches the x-axis it will be unable to find the lower and upper guesses or if the function changes sign but has no root (not continuous).



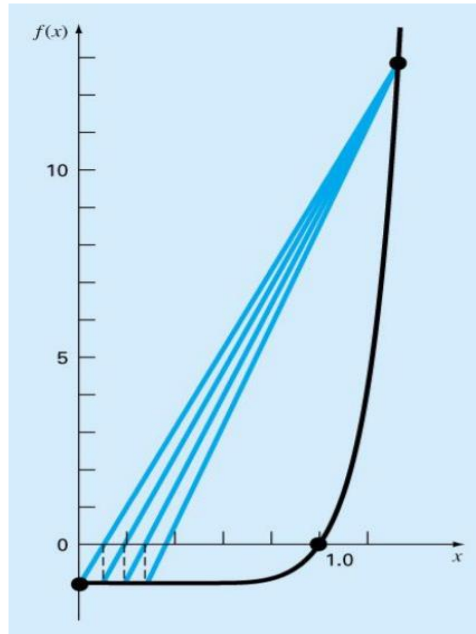


False position

Fails in the same conditions as bisection.

Faster than bisection except in special cases

Can be fixed by using bisection step for next guess if one of the bounds is stuck.



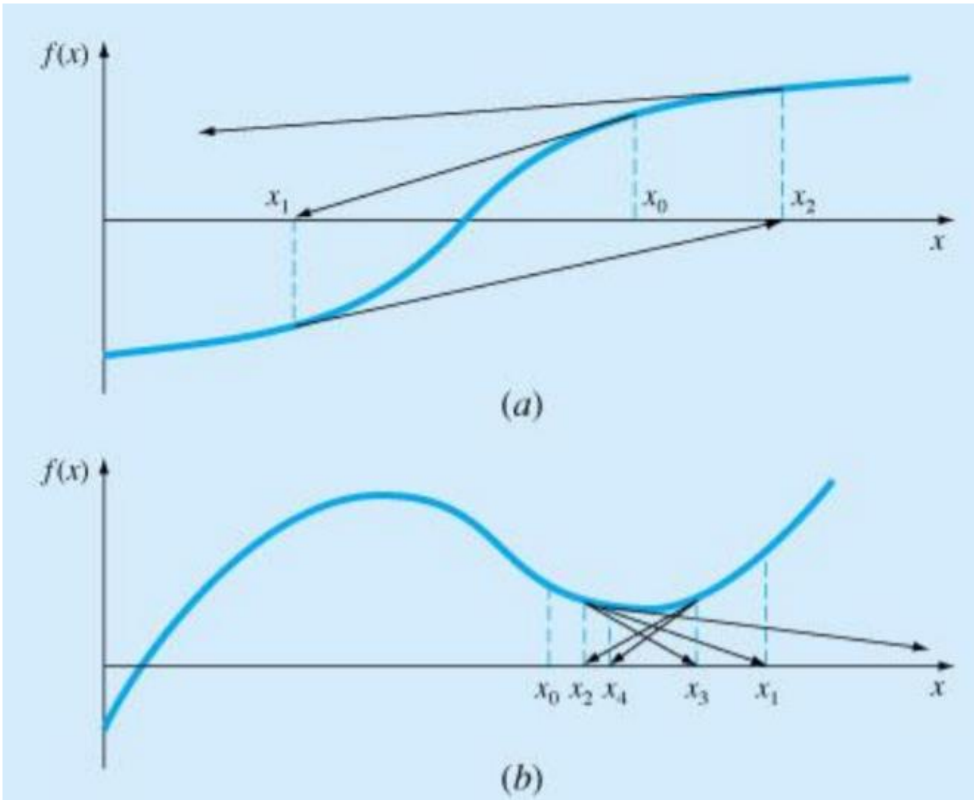
Fixed point

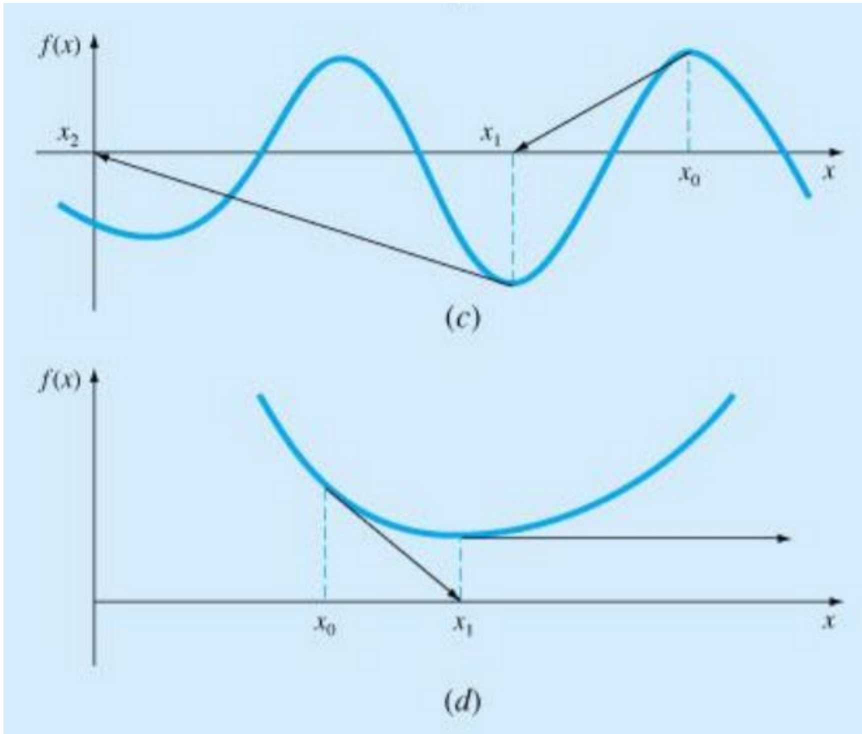
Doesn't always converge, diverges if $|g'(x)|>1$.

Newton

- Diverges at inflection points and local maximum or minimum point cause oscillation.
- May jump from 1 root to another

- Division by zero if f’(x)=0



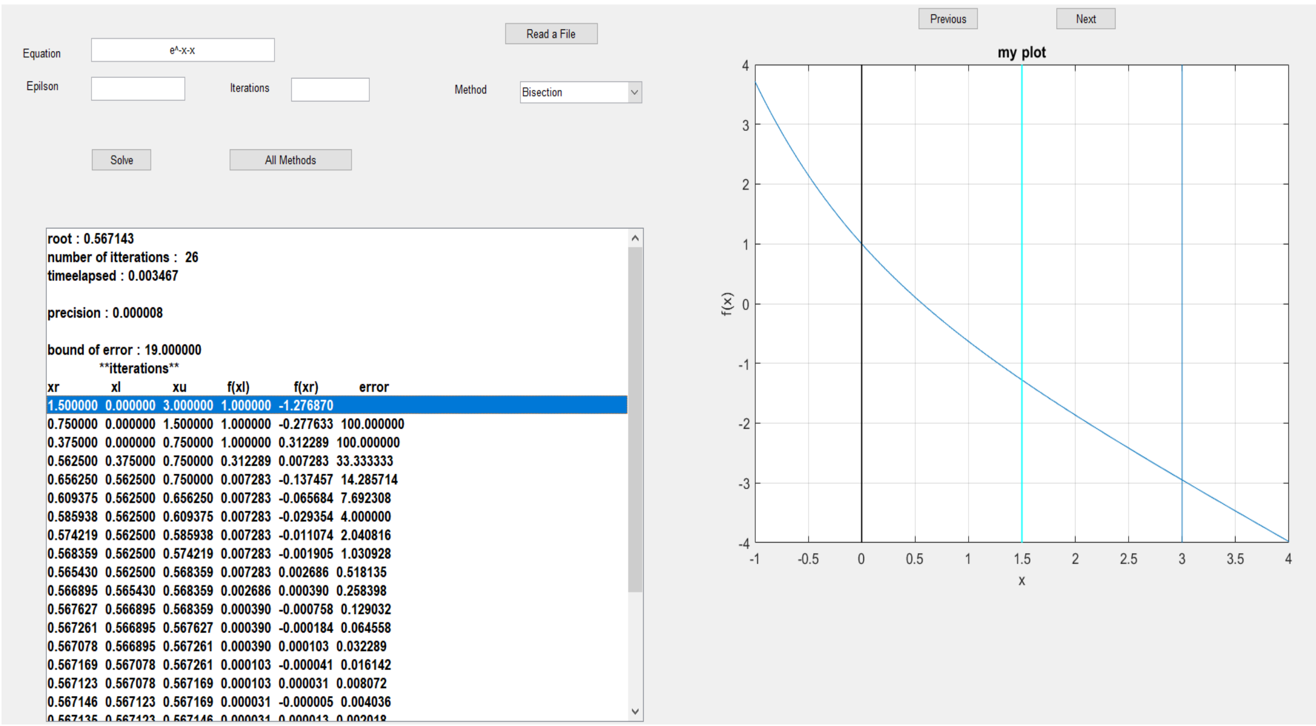


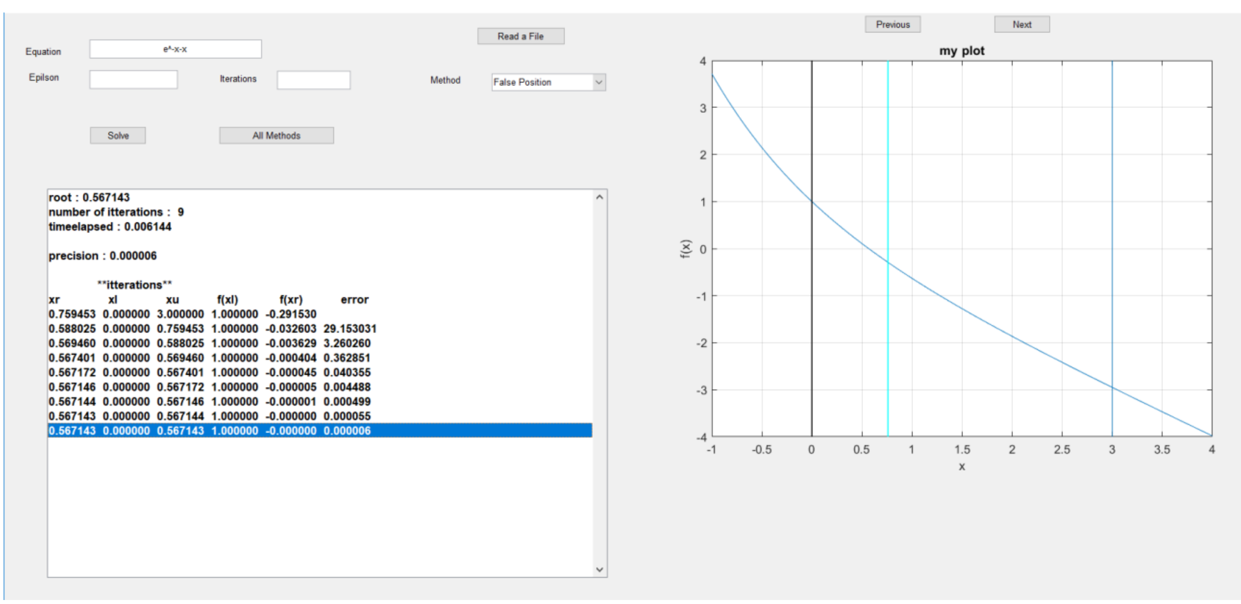
Secant

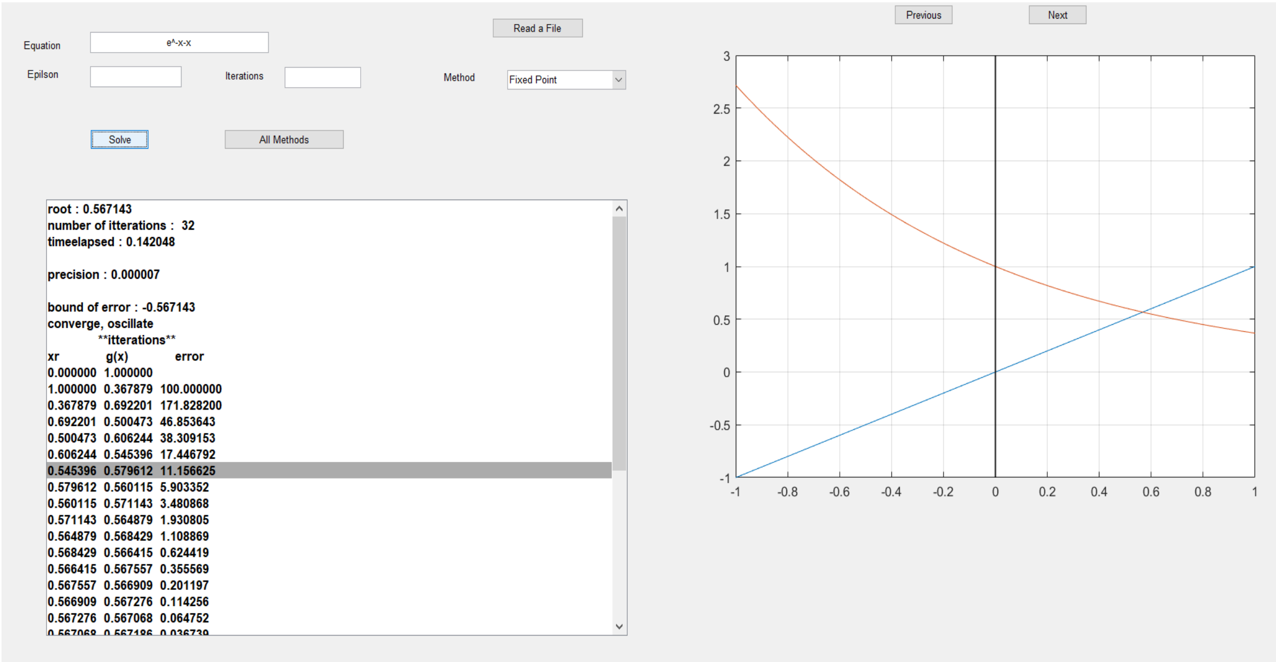
Since secant method is derived from Newton's method, it has similar drawbacks.

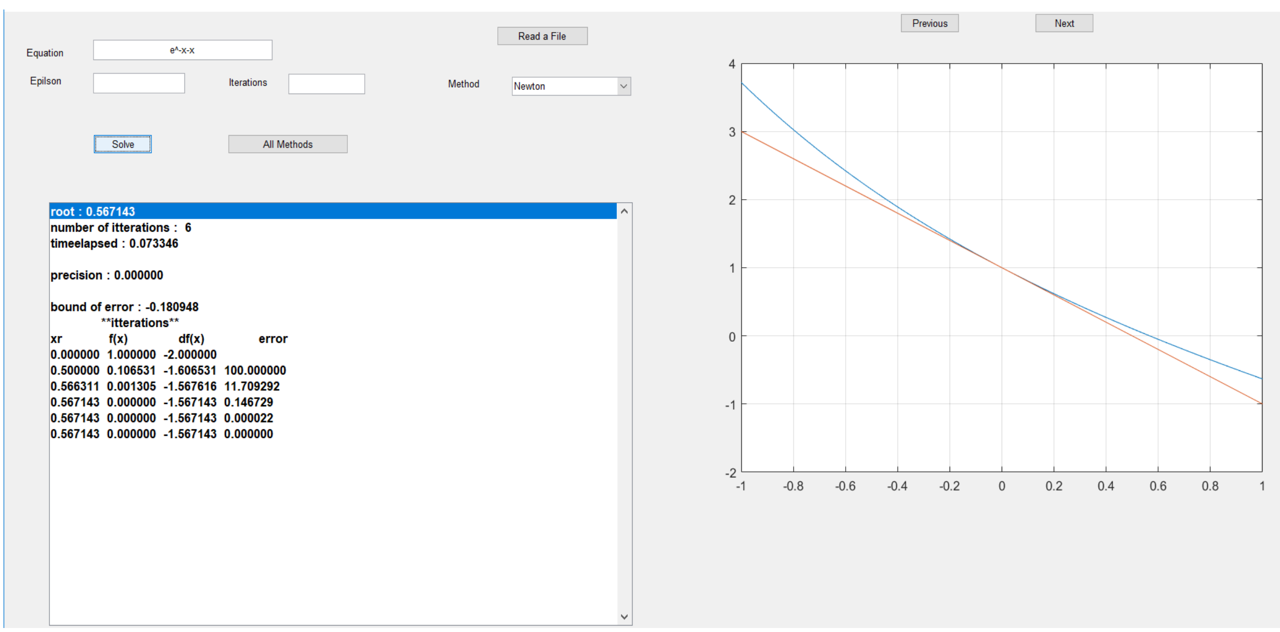
- Diverges at inflection points and local maximum or minimum point cause oscillation.
- May jump from 1 root to another
- Division by zero if f(xi)-f(xi-1)=0 (slope=0)

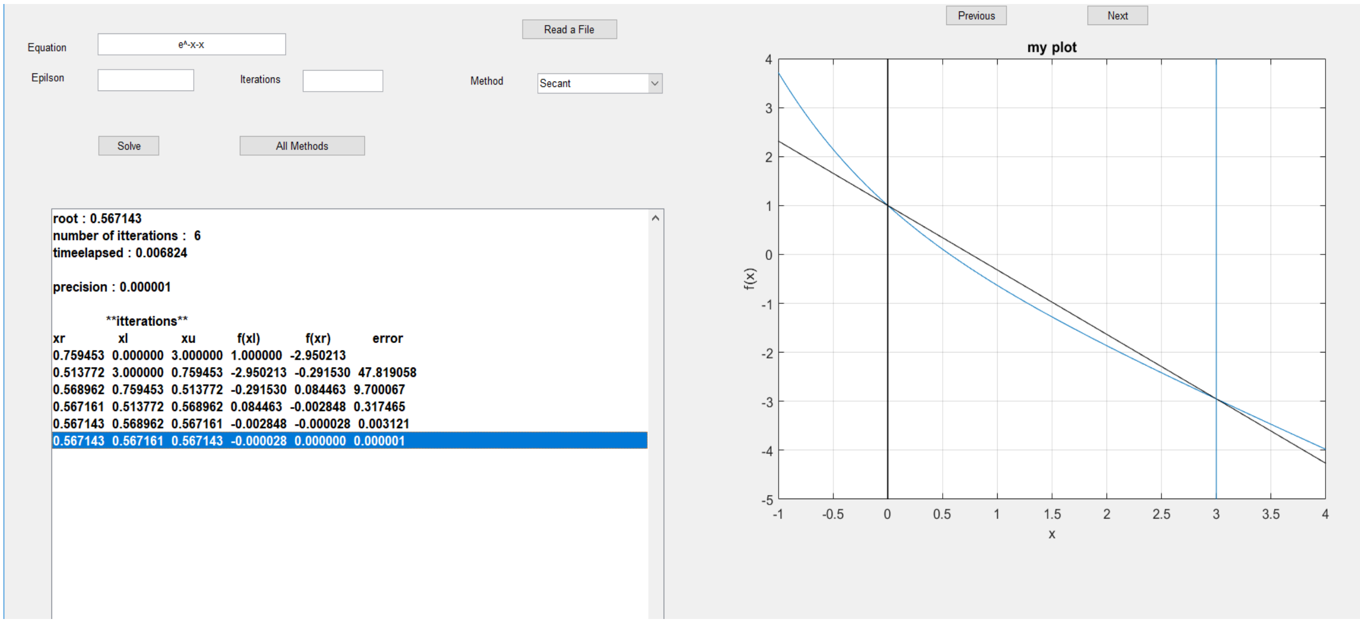
Analysis and Snapshots:






Read a File
Previous
Next
Equation
e^-x-x
Epilson
Iterations
Method
Fixed Point
Solve
All Methods
root : 0.567143
number of itterations : 32
timeelapsed : 0.142048
precision : 0.000007
bound of error : -0.567143
converge, oscillate
itterations
xr g(x) error
0.000000 1.000000
1.000000 0.367879 100.000000
0.367879 0.692201 171.828200
0.692201 0.500473 46.853643
0.500473 0.606244 38.309153
0.606244 0.545396 17.446792
0.545396 0.579612 11.156625
0.579612 0.560115 5.903352
0.560115 0.571143 3.480868
0.571143 0.564879 1.930805
0.564879 0.568429 1.108869
0.568429 0.566415 0.624419
0.566415 0.567557 0.355569
0.567557 0.566909 0.201197
0.566909 0.567276 0.114256
0.567276 0.567068 0.064752
3
2.5
2
1.5
1
0.5
0
-0.5
-1
-1
-0.8
-0.6
-0.4
-0.2
0
0.2
0.4
0.6
0.8
1


Read a File
Previous
Next
Equation
e^-x-x
Epilson
Iterations
Method
Newton
Solve
All Methods
root : 0.567143
number of itterations : 6
timeelapsed : 0.073346
precision : 0.000000
bound of error : -0.180948
itterations
xr f(x) df(x) error
0.000000 1.000000 -2.000000
0.500000 0.106531 -1.606531 100.000000
0.566311 0.001305 -1.567616 11.709292
0.567143 0.000000 -1.567143 0.146729
0.567143 0.000000 -1.567143 0.000022
0.567143 0.000000 -1.567143 0.000000
4
3
2
1
0
-1
-2
-1
-0.8
-0.6
-0.4
-0.2
0
0.2
0.4
0.6
0.8
1



By comparing all methods for the equation (e^-x-x) we found that the fastest was newton with initial guess(0) and secant with initial guesses (0,3) by 6 iterations and the slowest was fixed point with initial guess (0) by 32 iterations.

This proves that using open methods usually converges faster than bracketing methods IF they converge to the root of the equation.

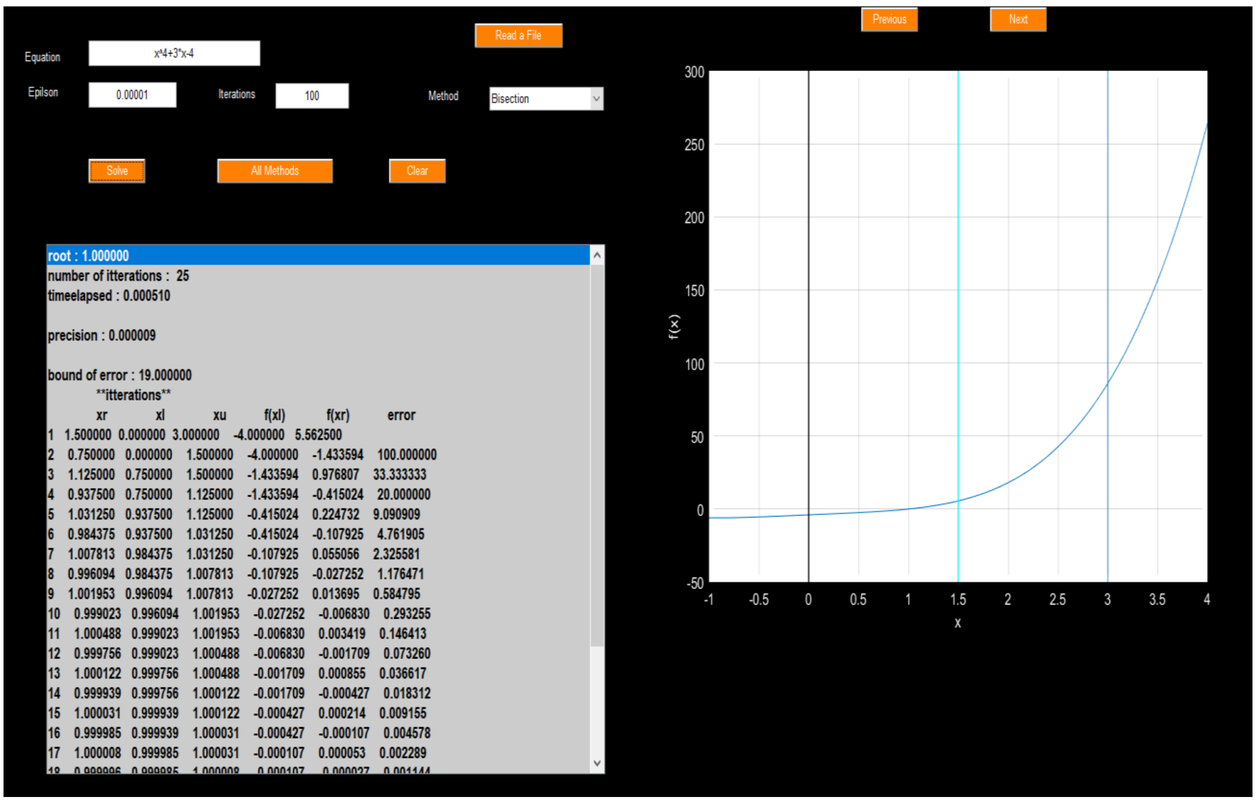
All the methods converges to the root (0.567143)

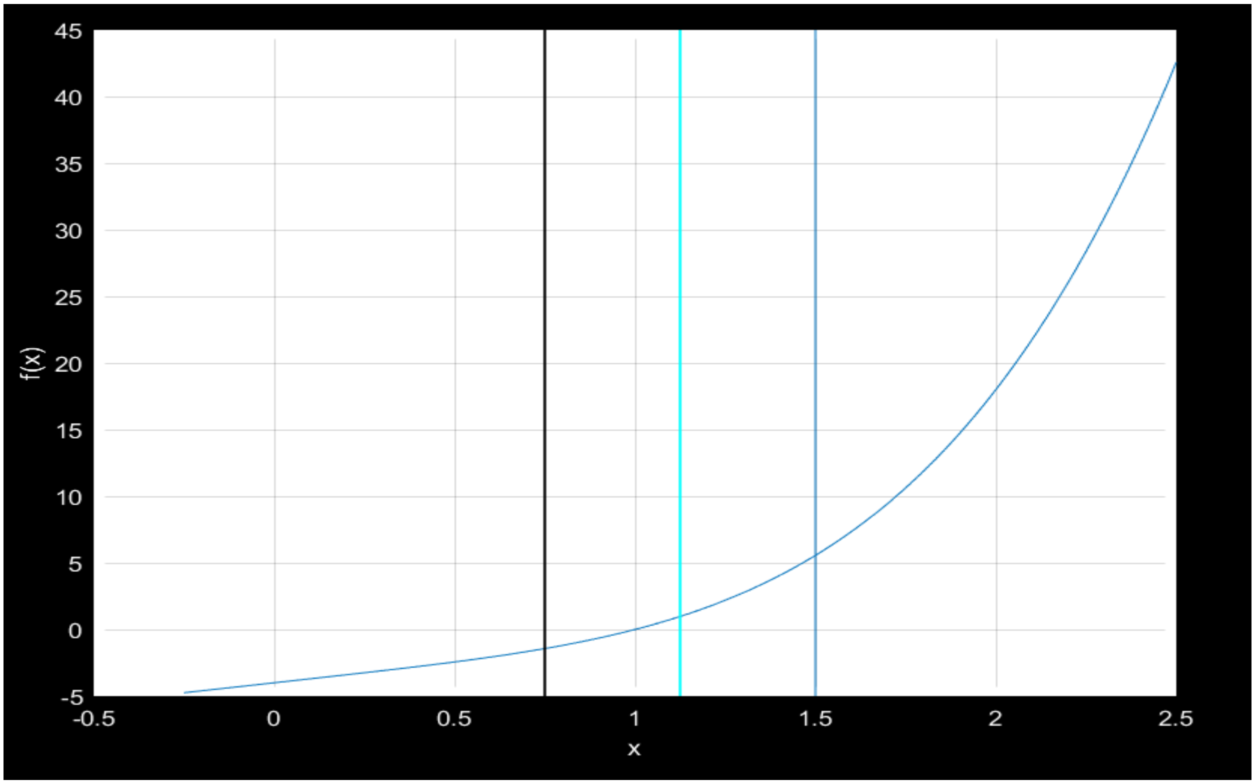
Bracketing Methods:

All bracketing methods has two initial guesses (xo,x1)

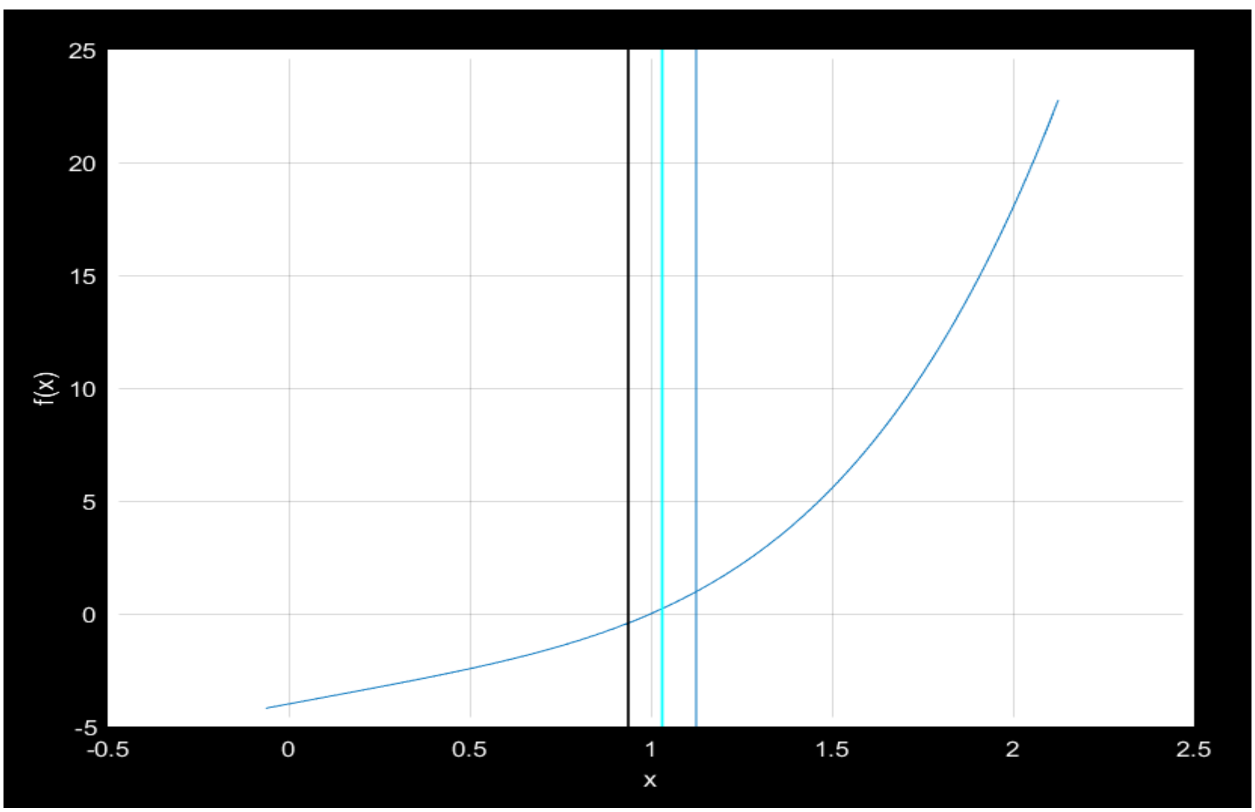
The condition for bracketing is $f(x0) * f(x1) < 0$ so the result of both functions must have different signs which allows these methods to diverge to a certain root which lies between (x0,x1)

1) Bisection

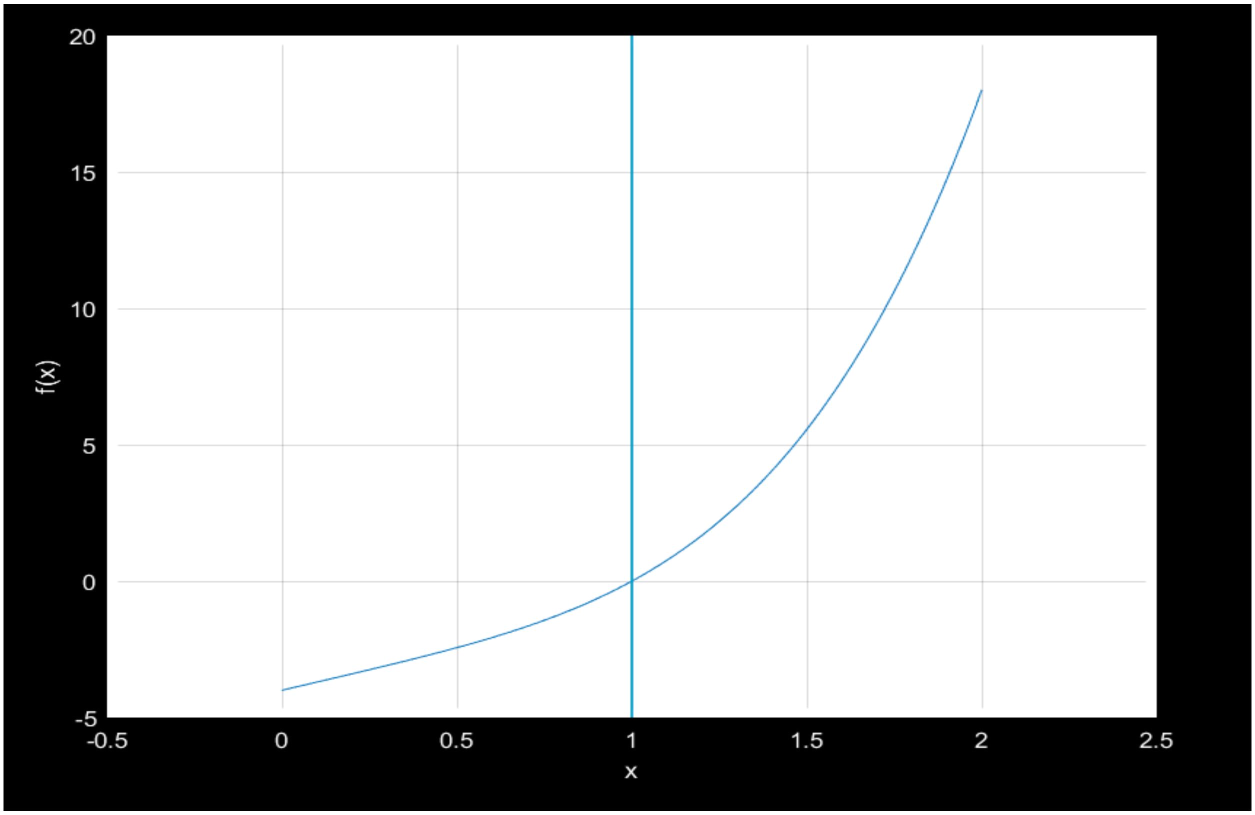




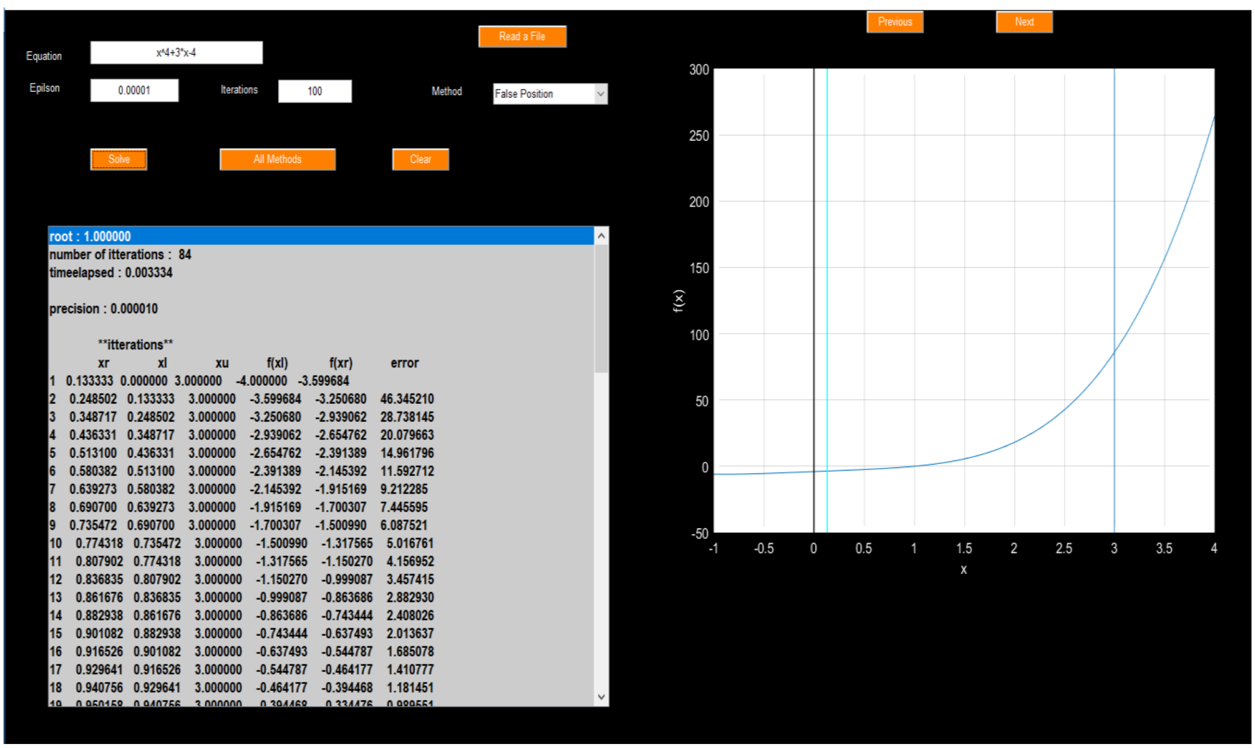
45
40
35
30
25
f(x)
20
15
10
5
0
-5
-0.5
0
0.5
1
1.5
2
2.5
x

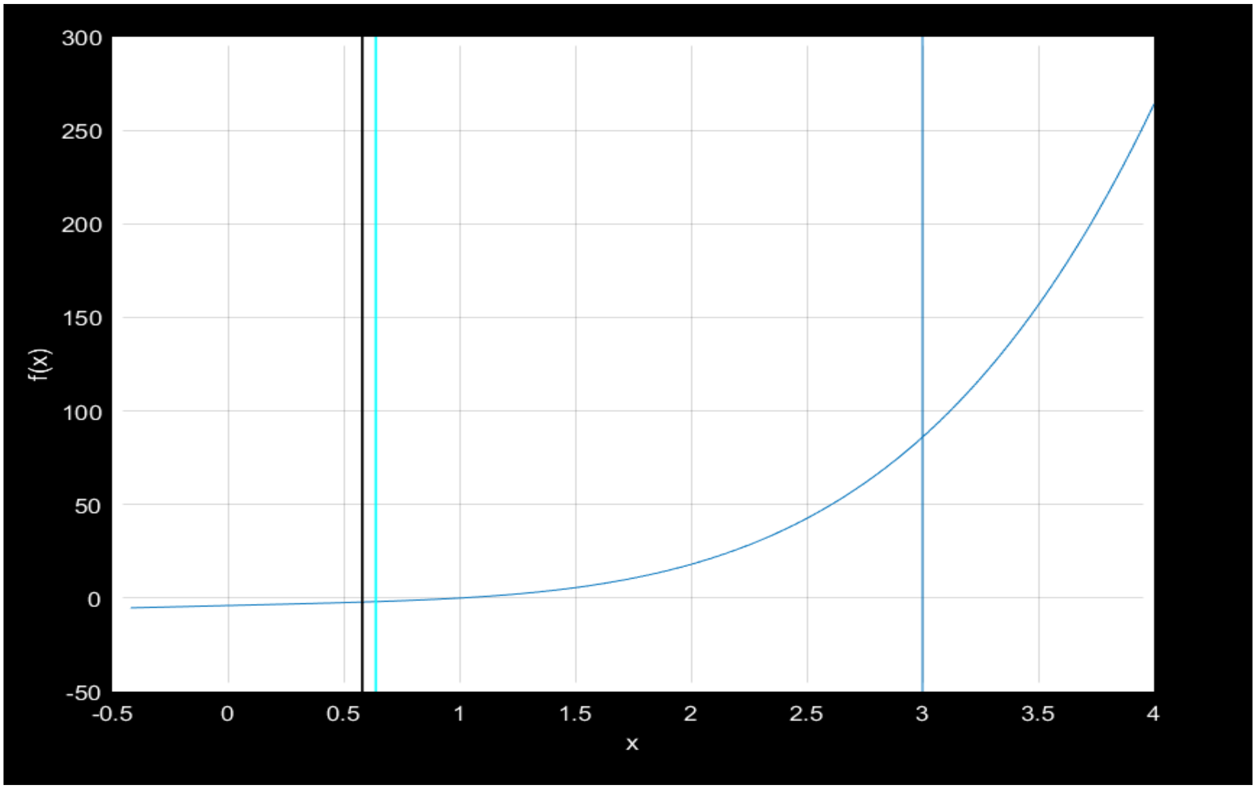


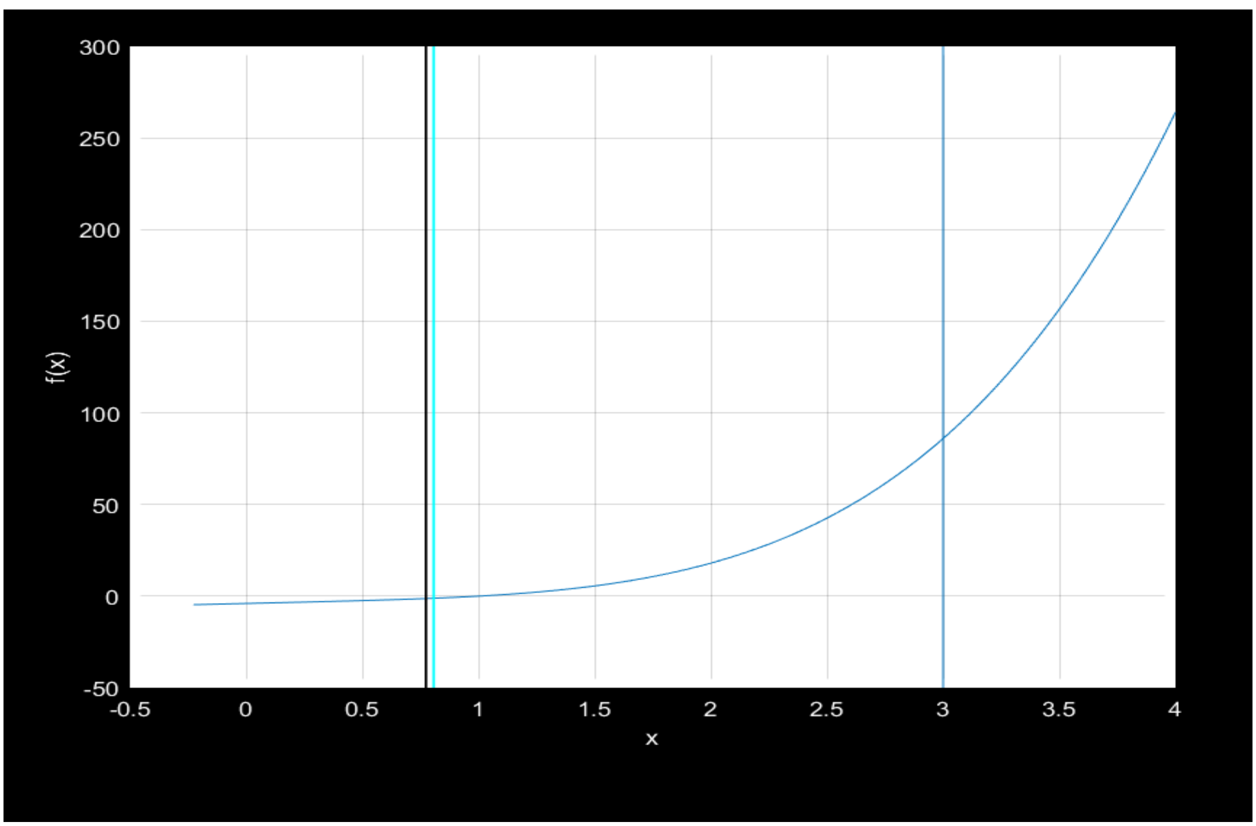
25
20
15
f(x)
10
5
0
-5
-0.5
0
0.5
1
1.5
2
2.5
x

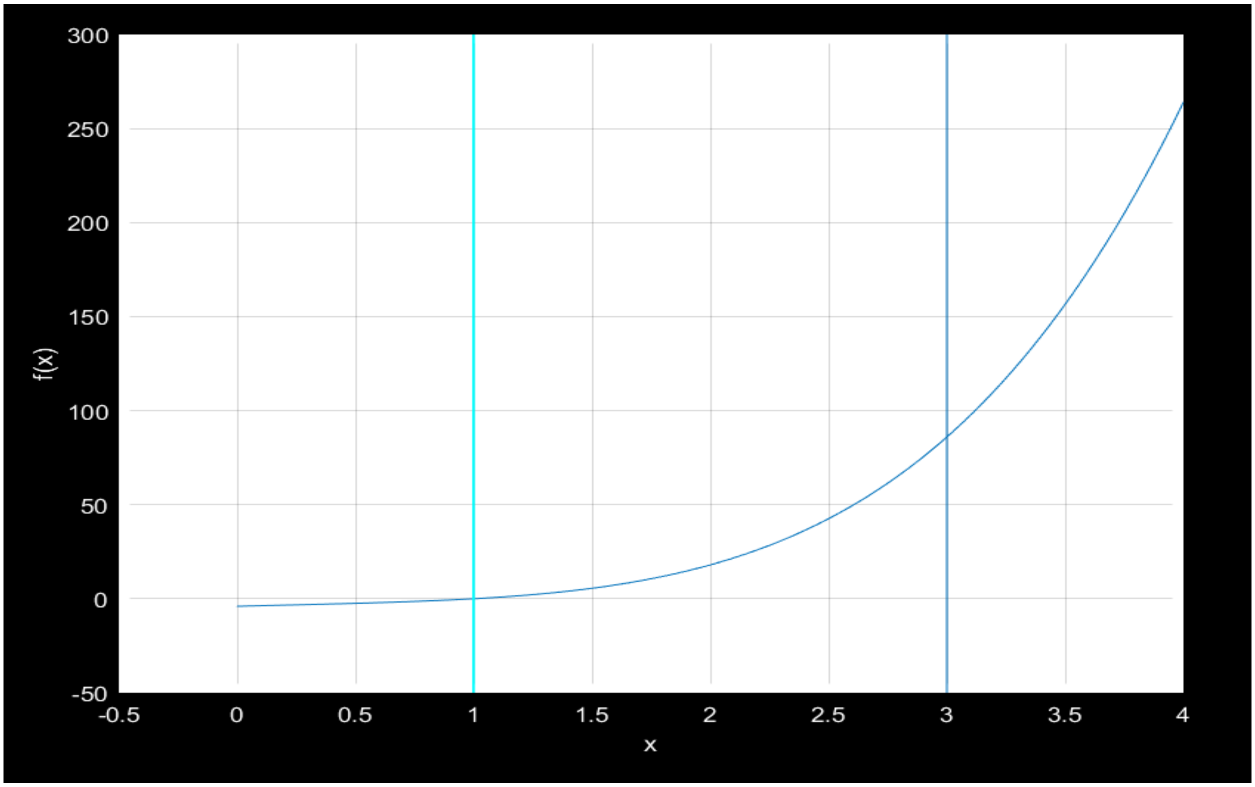


2) False-Position:




300
250
200
150
100
50
0
-50
f(x)
-0.5
0
0.5
1
1.5
2
2.5
3
3.5
4
x


300
250
200
150
100
50
0
-50
f(x)
-0.5
0
0.5
1
1.5
2
2.5
3
3.5
4
x

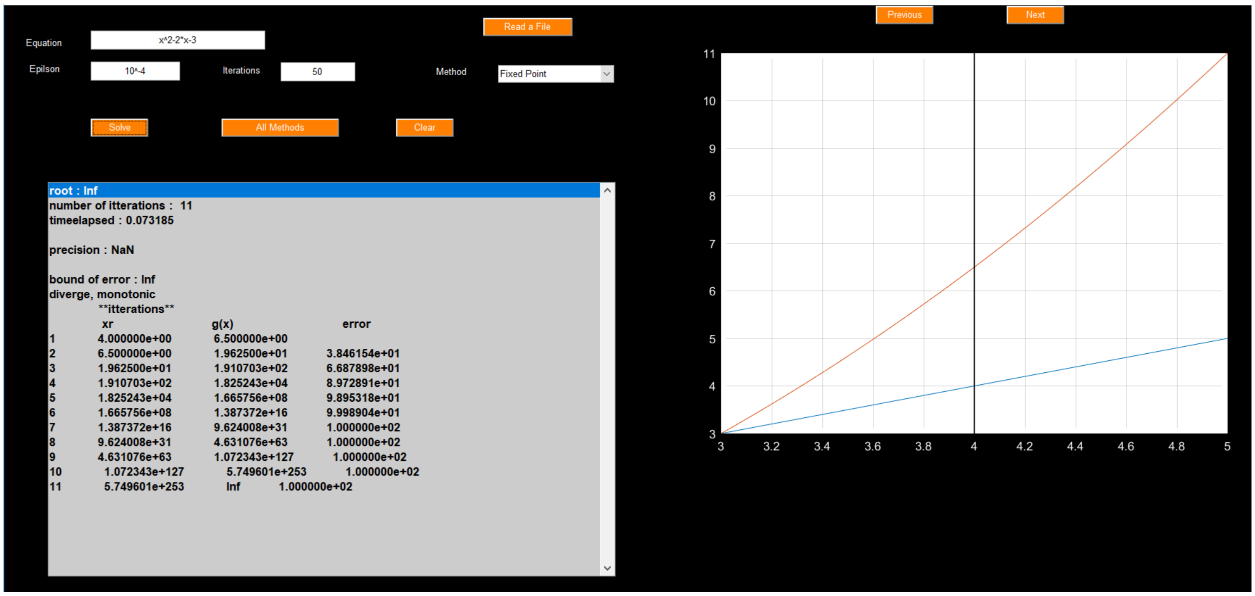


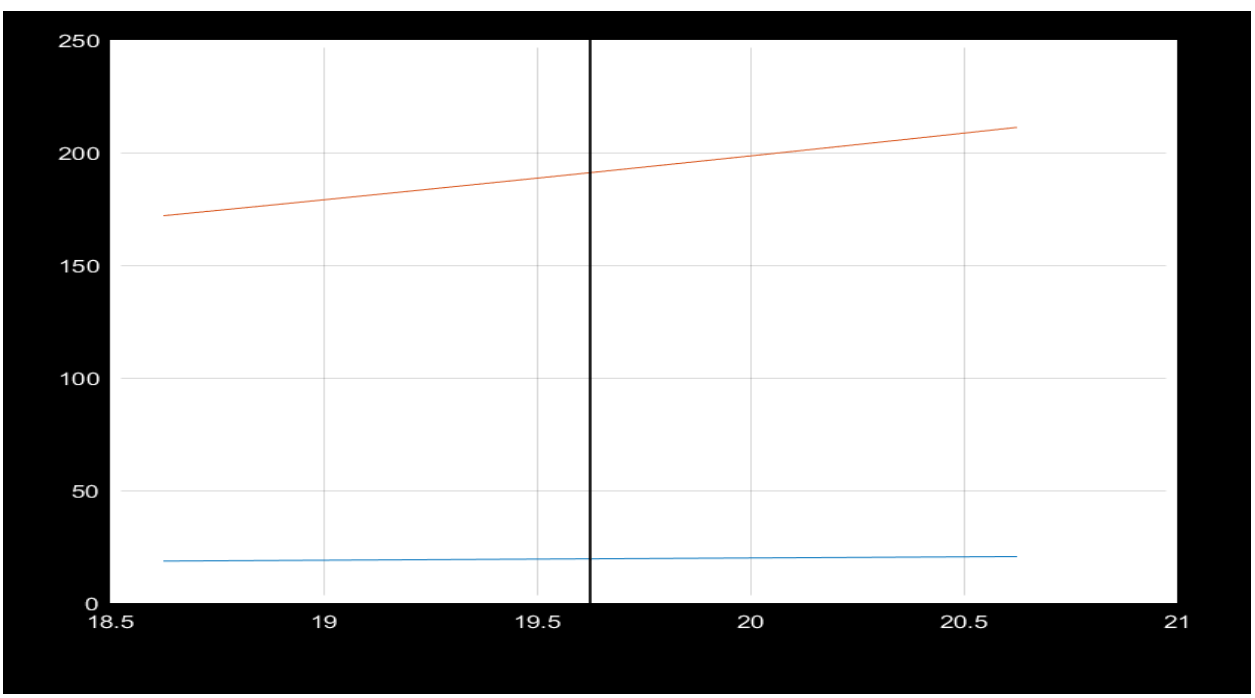
By comparing the two methods (bisection, false-position) we found that bisection which takes 25 iterations is faster than false-position which takes 84 iterations to converge to the same root which is (1). But we may change the guesses and found false-position is faster as in case of the equation (e^-x-x) .

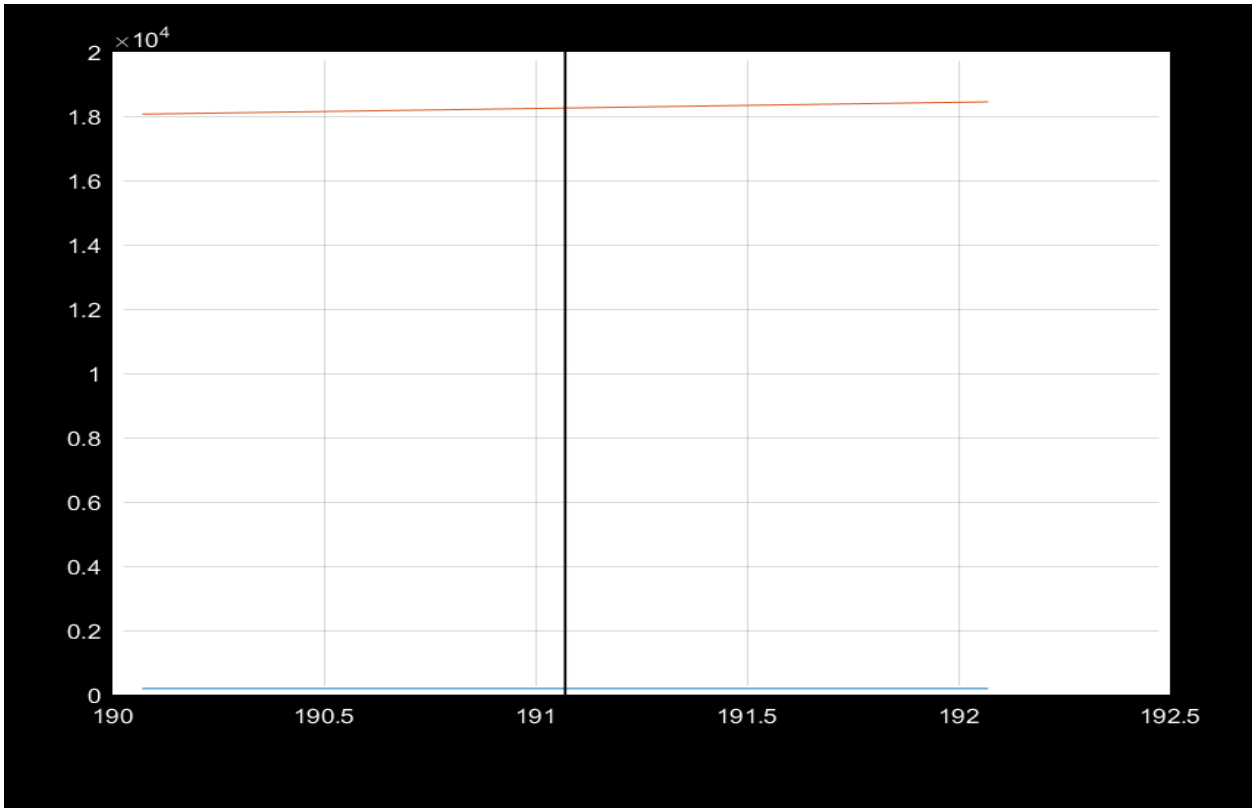
Open Methods:

Open methods have only one initial guess except secant which has two initial guesses but the only difference is that these methods don't have a certain condition which checks for convergence of the function so the function may converge or diverge.

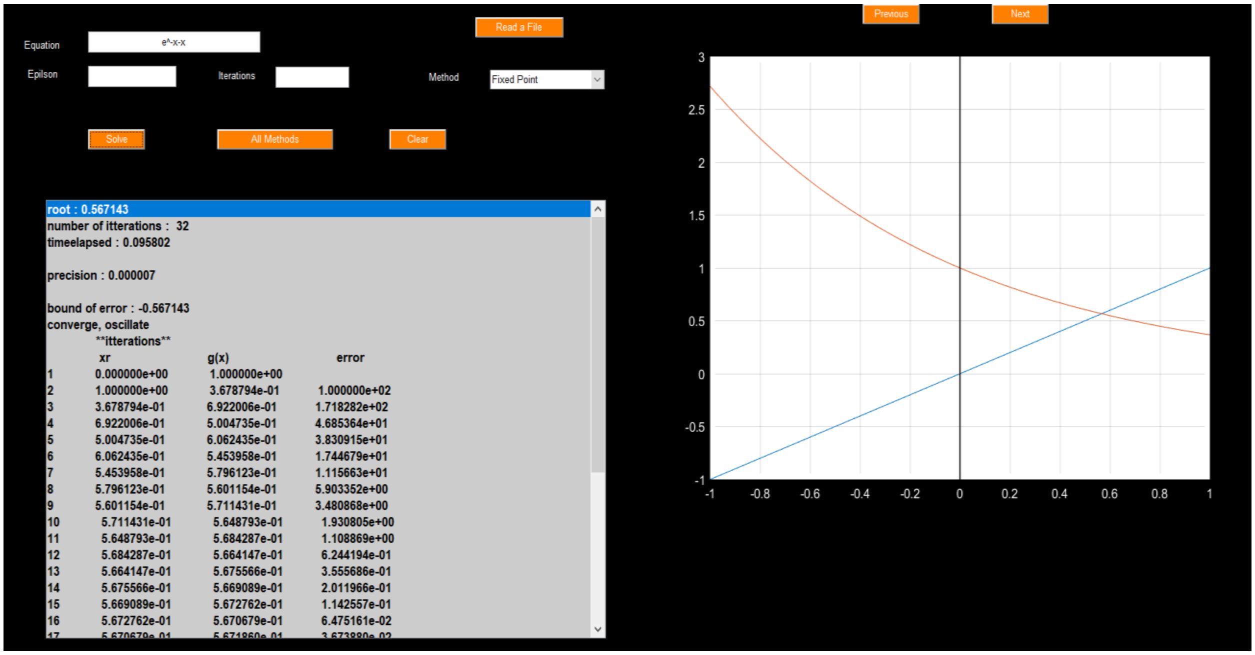
1) Fixed-point:



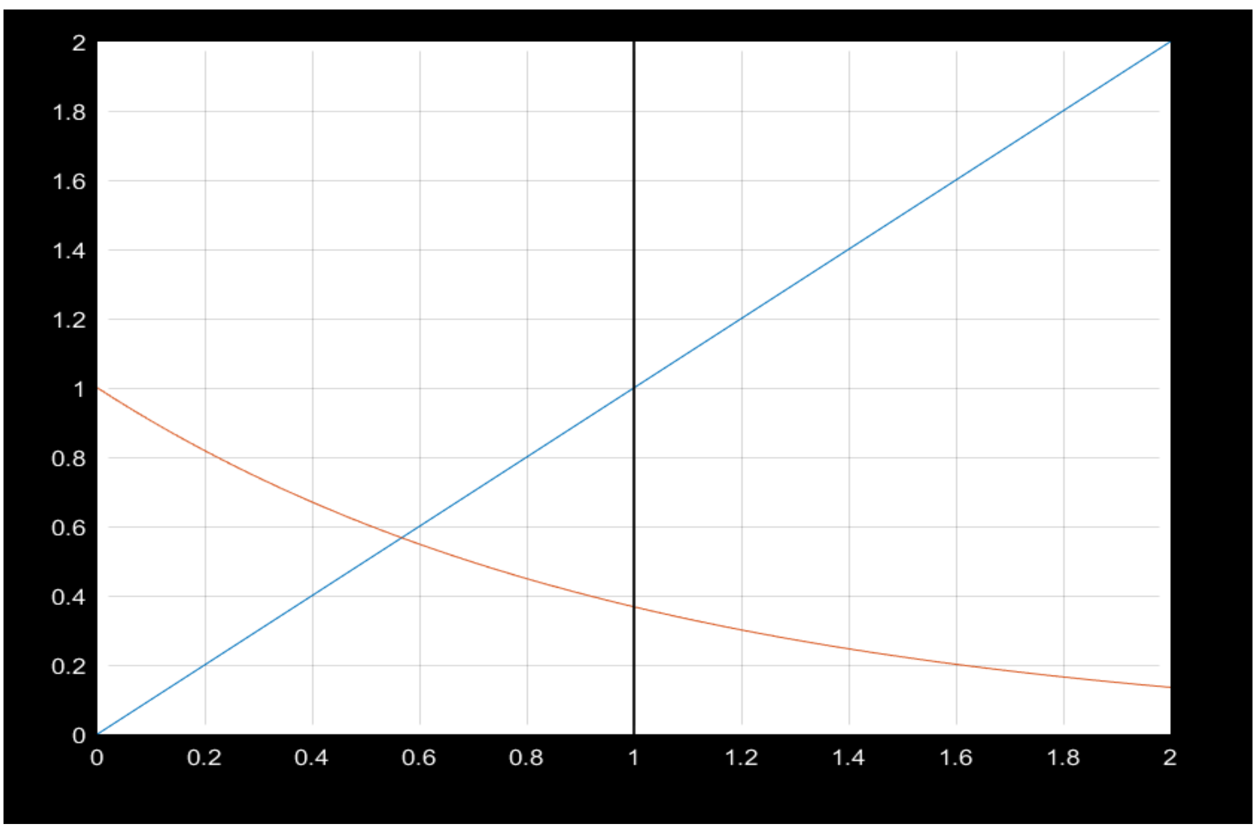




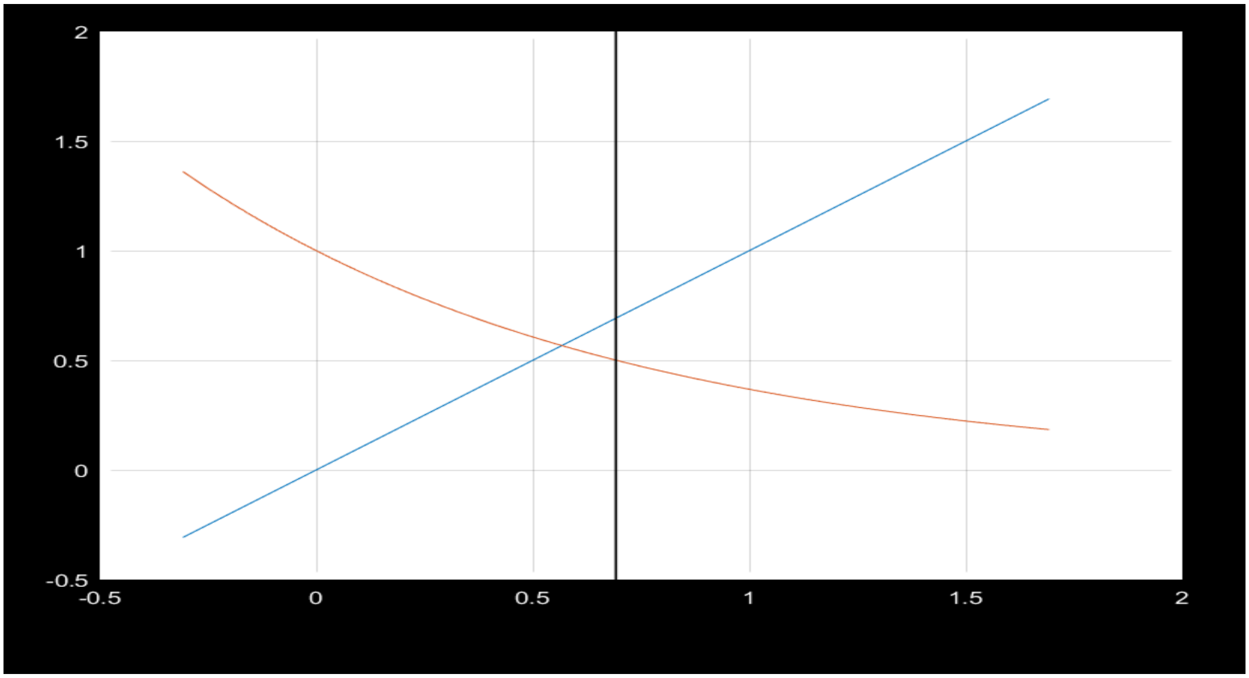
We found that this function diverges and guess for root will reach to infinity

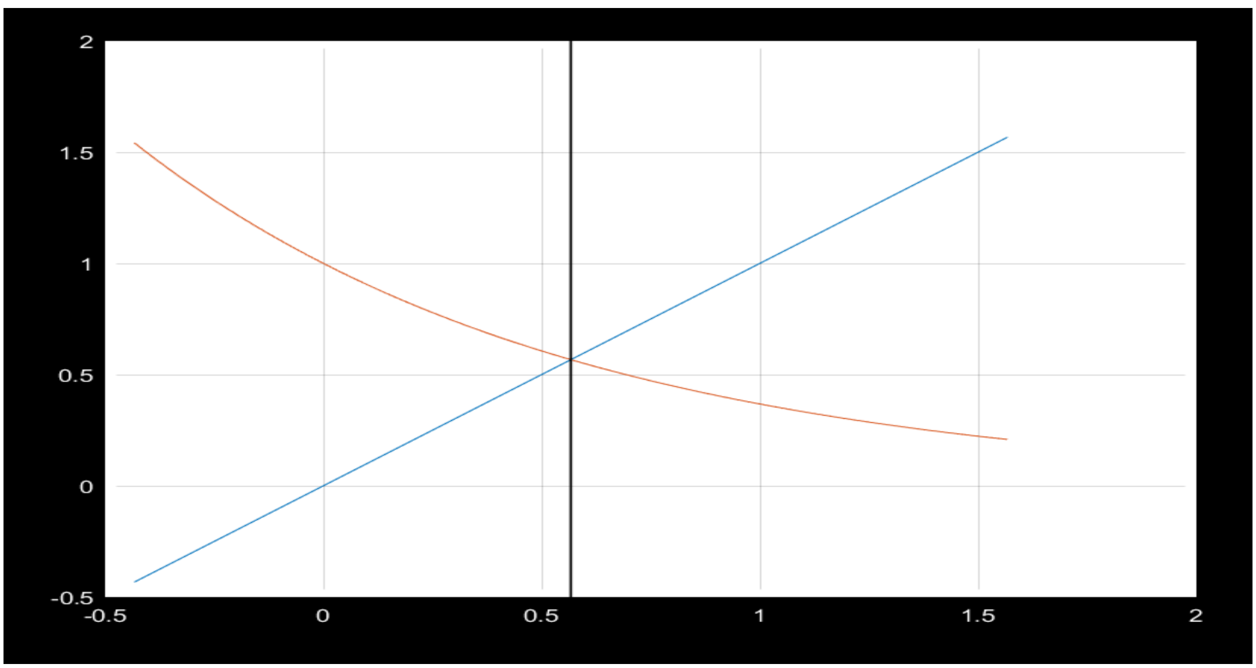


Equation
e^-x-x
Epilson
Iterations
Method
Fixed Point
Read a File
Solve
All Methods
Clear
Previous
Next
root : 0.567143
number of itterations : 32
timeelapsed : 0.095802
precision : 0.000007
bound of error : -0.567143
converge, oscillate
itterations
xr g(x) error
1 0.000000e+00 1.000000e+00
2 1.000000e+00 3.678794e-01 1.000000e+02
3 3.678794e-01 6.922006e-01 1.718282e+02
4 6.922006e-01 5.004735e-01 4.685364e+01
5 5.004735e-01 6.062435e-01 3.830915e+01
6 6.062435e-01 5.453958e-01 1.744679e+01
7 5.453958e-01 5.796123e-01 1.115663e+01
8 5.796123e-01 5.601154e-01 5.903352e+00
9 5.601154e-01 5.711431e-01 3.480868e+00
10 5.711431e-01 5.648793e-01 1.930805e+00
11 5.648793e-01 5.684287e-01 1.108869e+00
12 5.684287e-01 5.664147e-01 6.244194e-01
13 5.664147e-01 5.675566e-01 3.555686e-01
14 5.675566e-01 5.669089e-01 2.011966e-01
15 5.669089e-01 5.672762e-01 1.142557e-01
16 5.672762e-01 5.670679e-01 6.475161e-02
3
2.5
2
1.5
1
0.5
0
-0.5
-1
-1 -0.8 -0.6 -0.4 -0.2 0 0.2 0.4 0.6 0.8 1



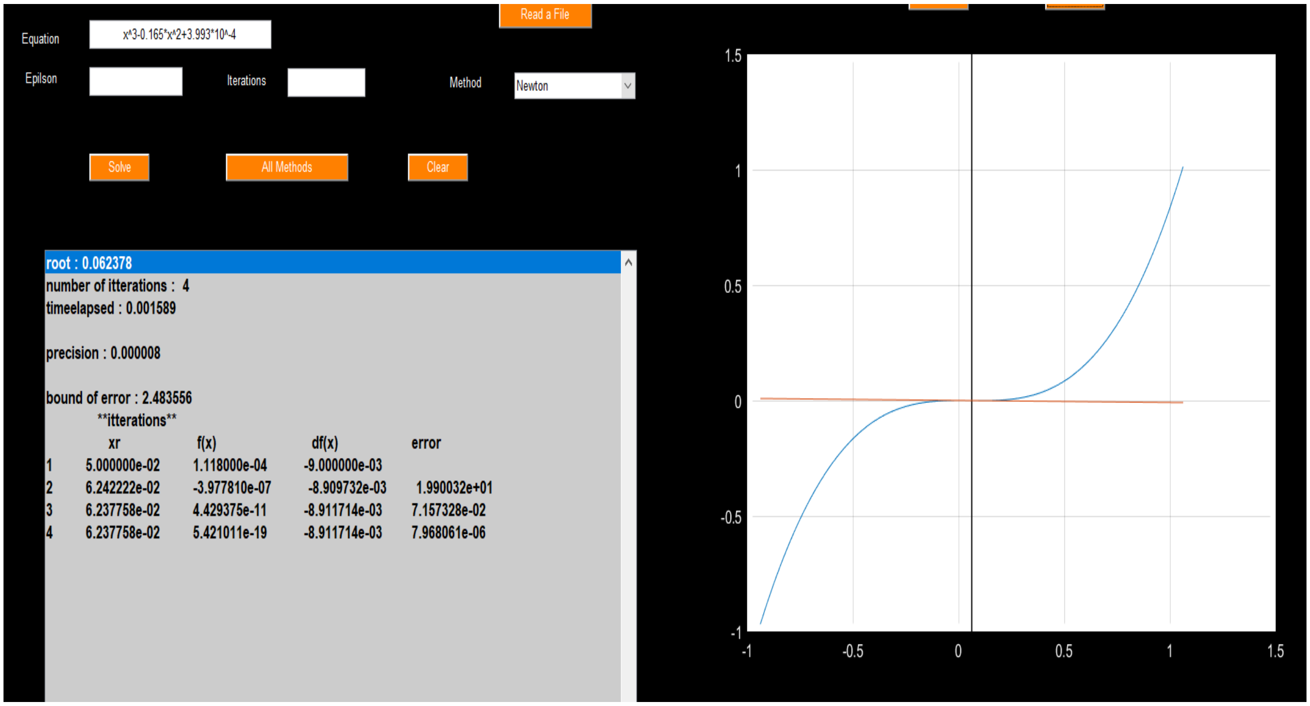
2
1.8
1.6
1.4
1.2
1
0.8
0.6
0.4
0.2
0
0 0.2 0.4 0.6 0.8 1 1.2 1.4 1.6 1.8 2

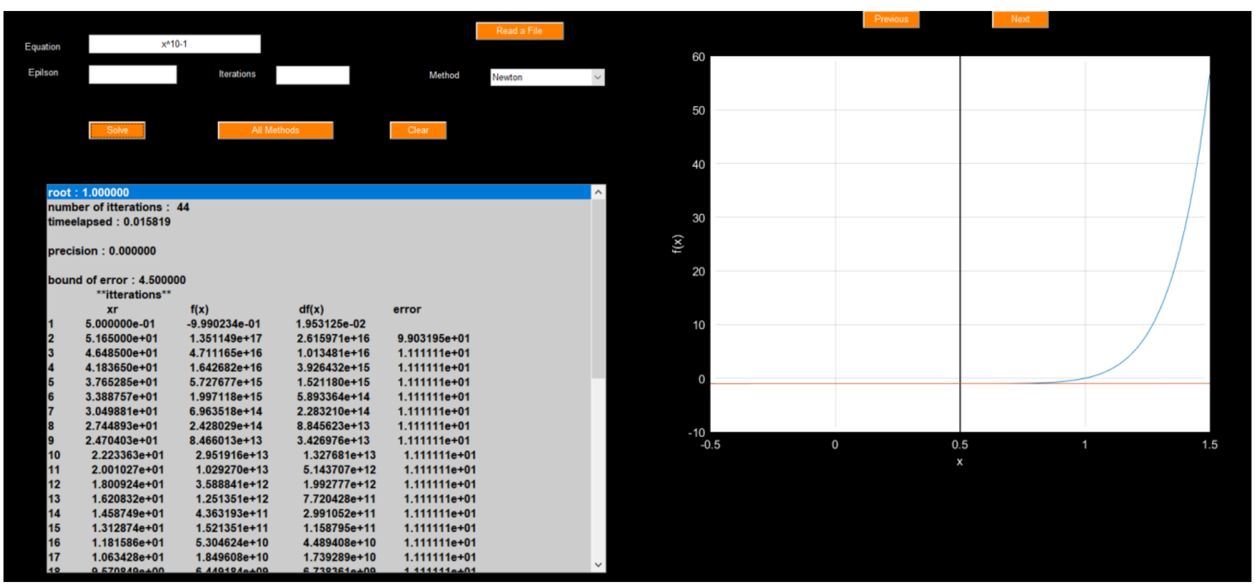


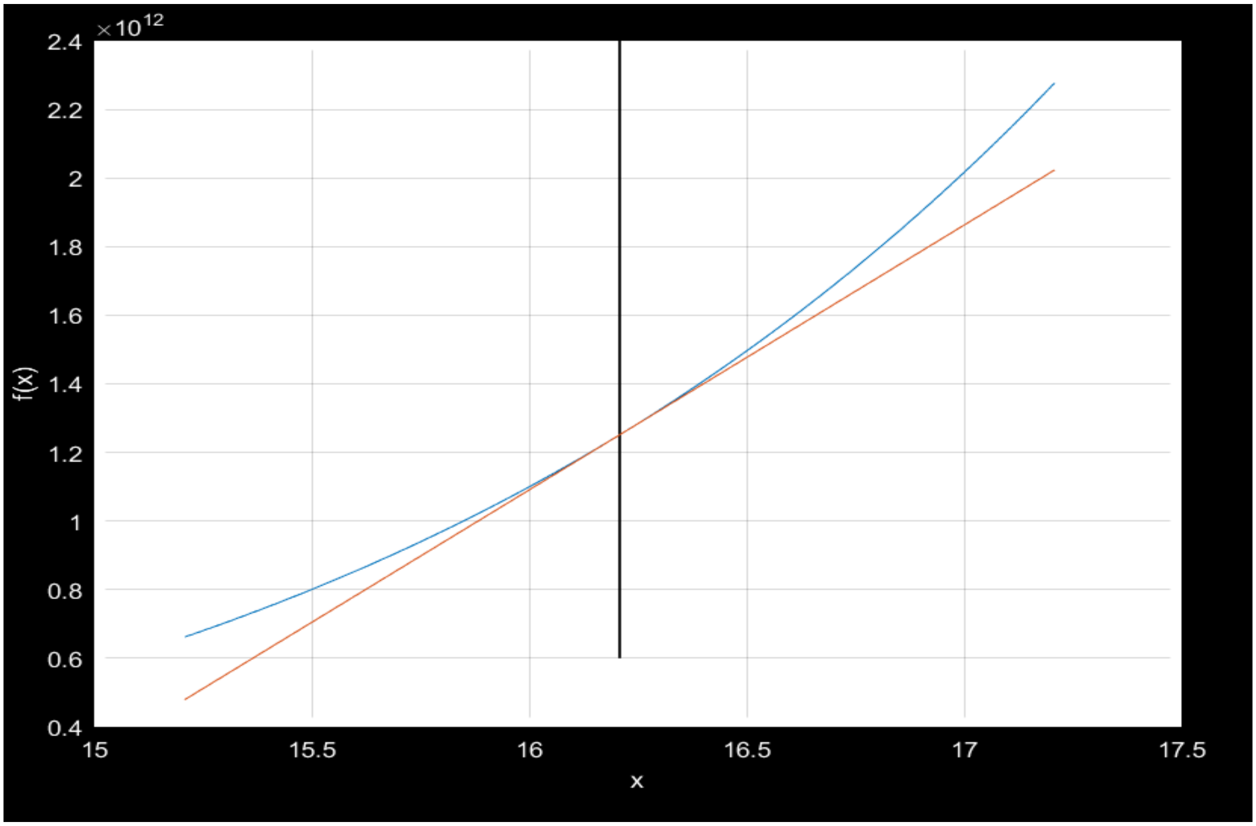


For this equation (e^-x-x) the function will converge to the root (0.567143)

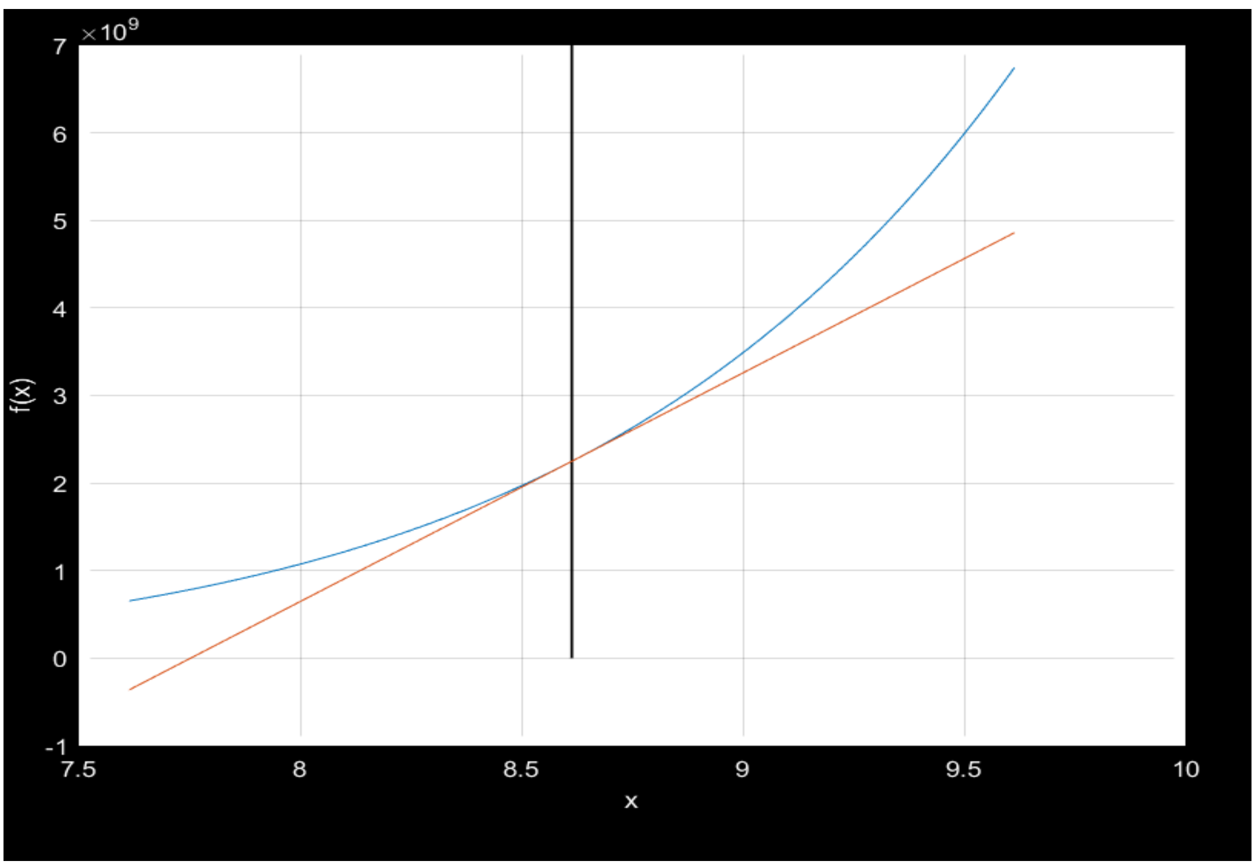
2) Newton:


Read a File
Equation
x^3-0.165*x^2+3.993*10^-4
Epilson
Iterations
Method
Newton
Solve
All Methods
Clear
root : 0.062378
number of itterations : 4
timeelapsed : 0.001589
precision : 0.000008
bound of error : 2.483556
itterations
xr f(x) df(x) error
1 5.000000e-02 1.118000e-04 -9.000000e-03
2 6.242222e-02 -3.977810e-07 -8.909732e-03 1.990032e+01
3 6.237758e-02 4.429375e-11 -8.911714e-03 7.157328e-02
4 6.237758e-02 5.421011e-19 -8.911714e-03 7.968061e-06
1.5
1
0.5
0
-0.5
-1
-1
-0.5
0
0.5
1
1.5

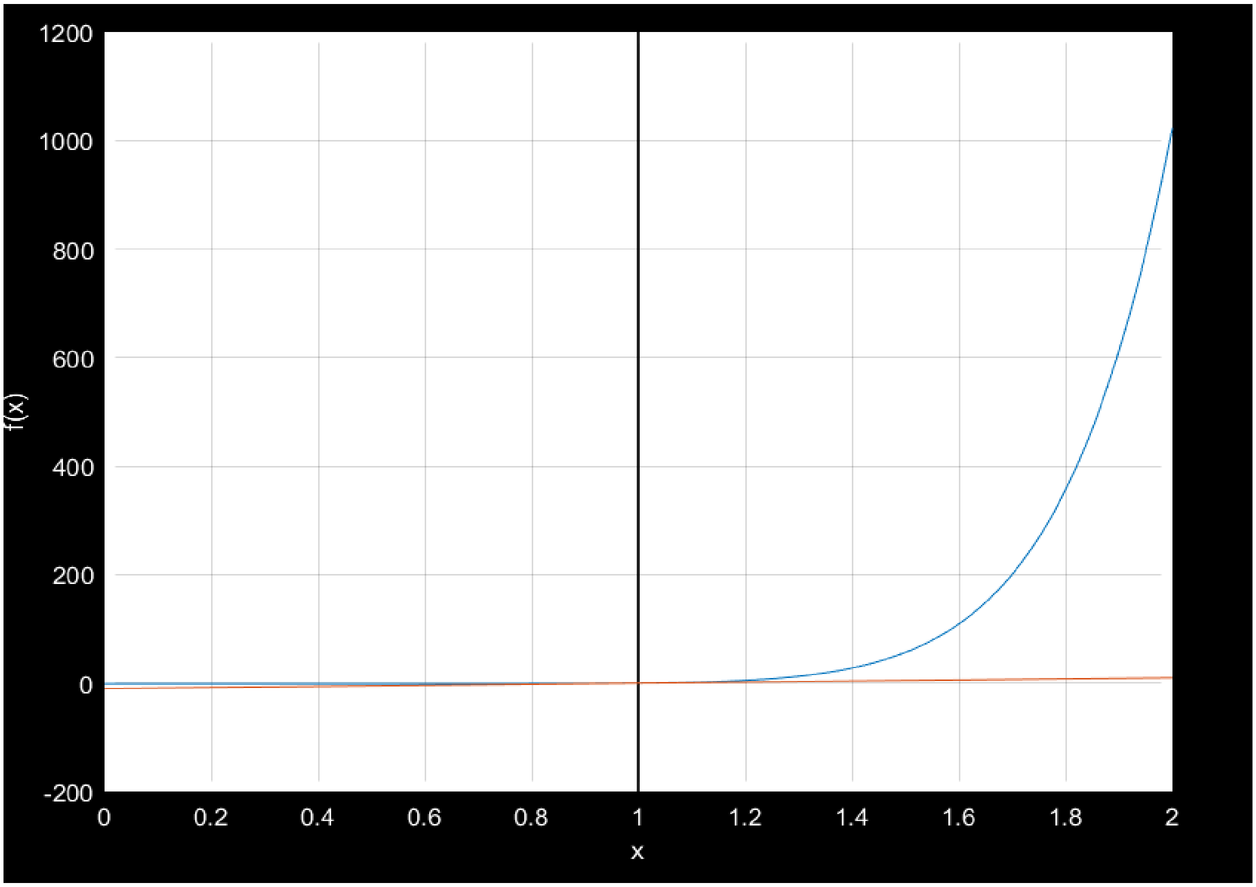

Read a File
Previous
Next
Equation
x^10-1
Epilson
Iterations
Method
Newton
Solve
All Methods
Clear
root : 1.000000
number of itterations : 44
timeelapsed : 0.015819
precision : 0.000000
bound of error : 4.500000
itterations
xr f(x) df(x) error
1 5.000000e-01 -9.990234e-01 1.953125e-02
2 5.165000e+01 1.351149e+17 2.615971e+16 9.903195e+01
3 4.648500e+01 4.711165e+16 1.013481e+16 1.111111e+01
4 4.183650e+01 1.642682e+16 3.926432e+15 1.111111e+01
5 3.765285e+01 5.727677e+15 1.521180e+15 1.111111e+01
6 3.388757e+01 1.997118e+15 5.893364e+14 1.111111e+01
7 3.049881e+01 6.963518e+14 2.283210e+14 1.111111e+01
8 2.744893e+01 2.428029e+14 8.845623e+13 1.111111e+01
9 2.470403e+01 8.466013e+13 3.426976e+13 1.111111e+01
10 2.223363e+01 2.951916e+13 1.327681e+13 1.111111e+01
11 2.001027e+01 1.029270e+13 5.143707e+12 1.111111e+01
12 1.800924e+01 3.588841e+12 1.992777e+12 1.111111e+01
13 1.620832e+01 1.251351e+12 7.720428e+11 1.111111e+01
14 1.458749e+01 4.363193e+11 2.991052e+11 1.111111e+01
15 1.312874e+01 1.521351e+11 1.158795e+11 1.111111e+01
16 1.181586e+01 5.304624e+10 4.489408e+10 1.111111e+01
17 1.063428e+01 1.849608e+10 1.739289e+10 1.111111e+01
60
50
40
30
20
10
0
-10
f(x)
-0.5
0
0.5
1
1.5
x



×10^12
2.4
2.2
2
1.8
1.6
1.4
1.2
1
0.8
0.6
0.4
f(x)
15
15.5
16
16.5
17
17.5
x

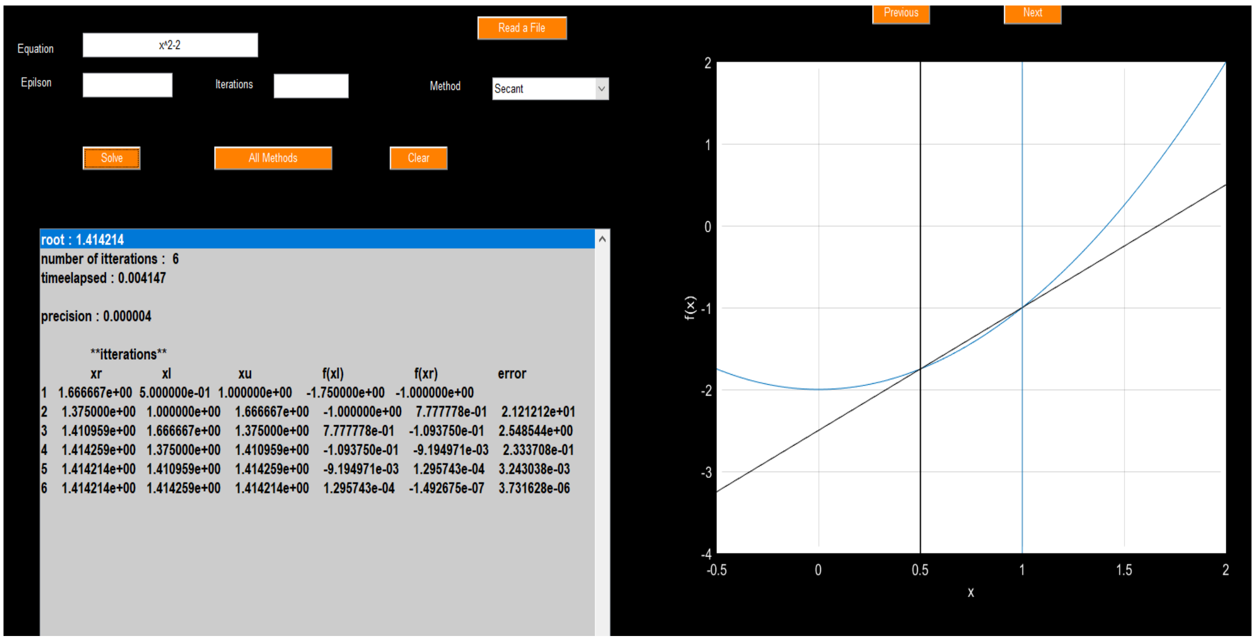


×10^9
7
6
5
4
3
2
1
0
-1
f(x)
7.5
8
8.5
9
9.5
10
x

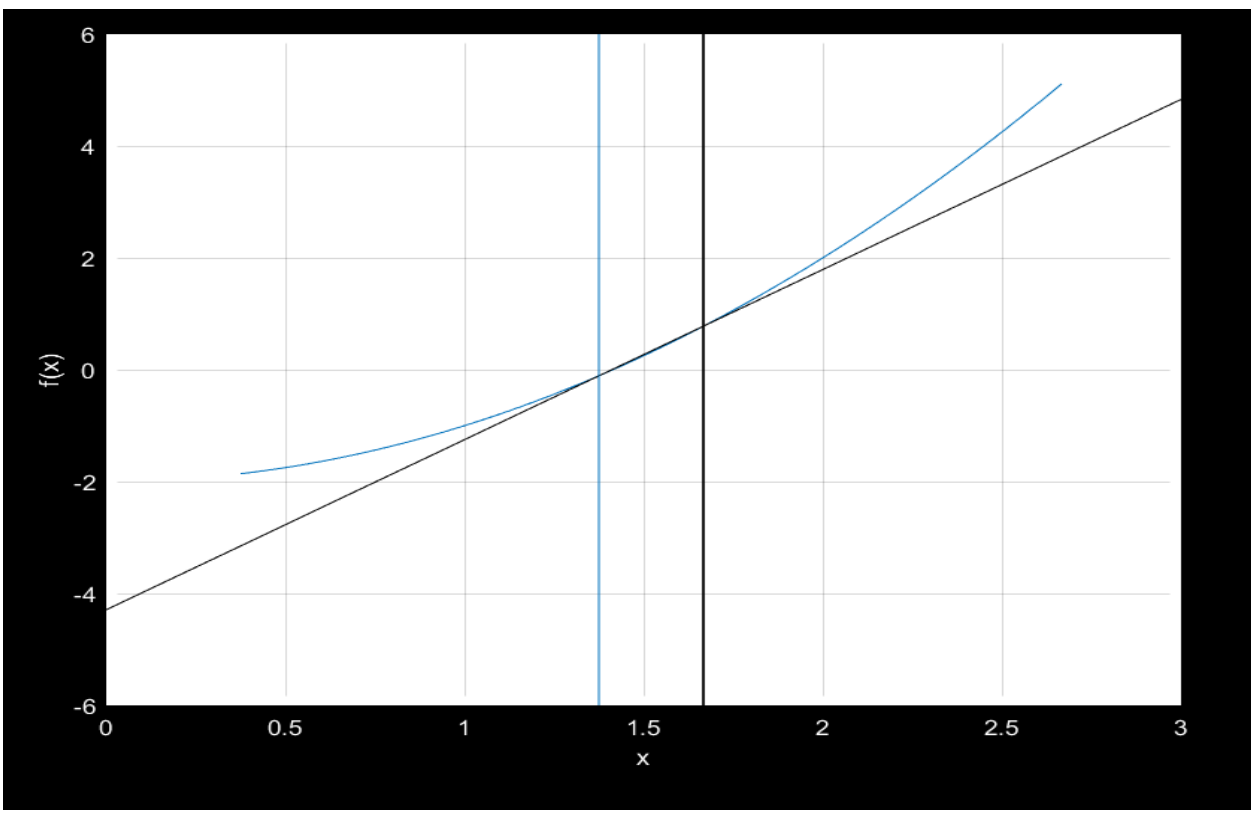


Here, we found that both functions of newton converges to a certain root but in the second example the function converges slowly to the root.

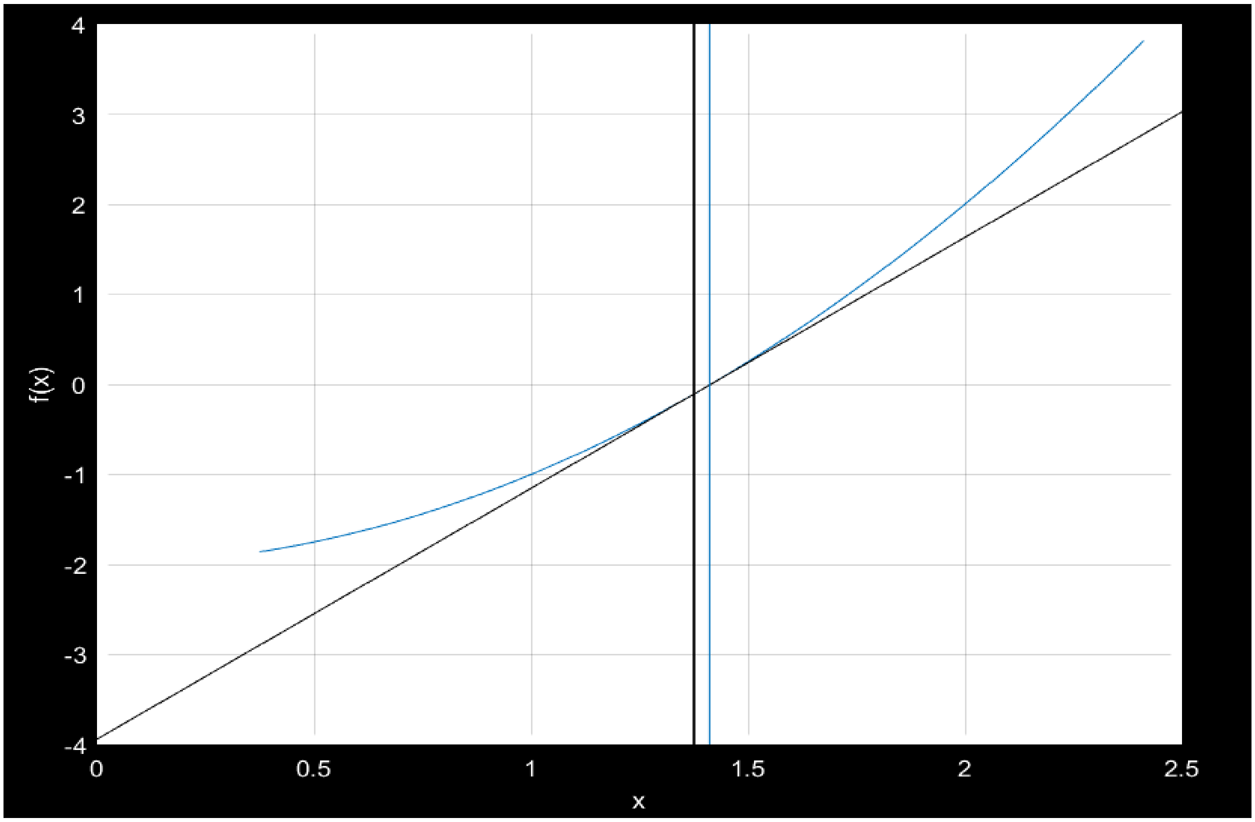
3) Secant:

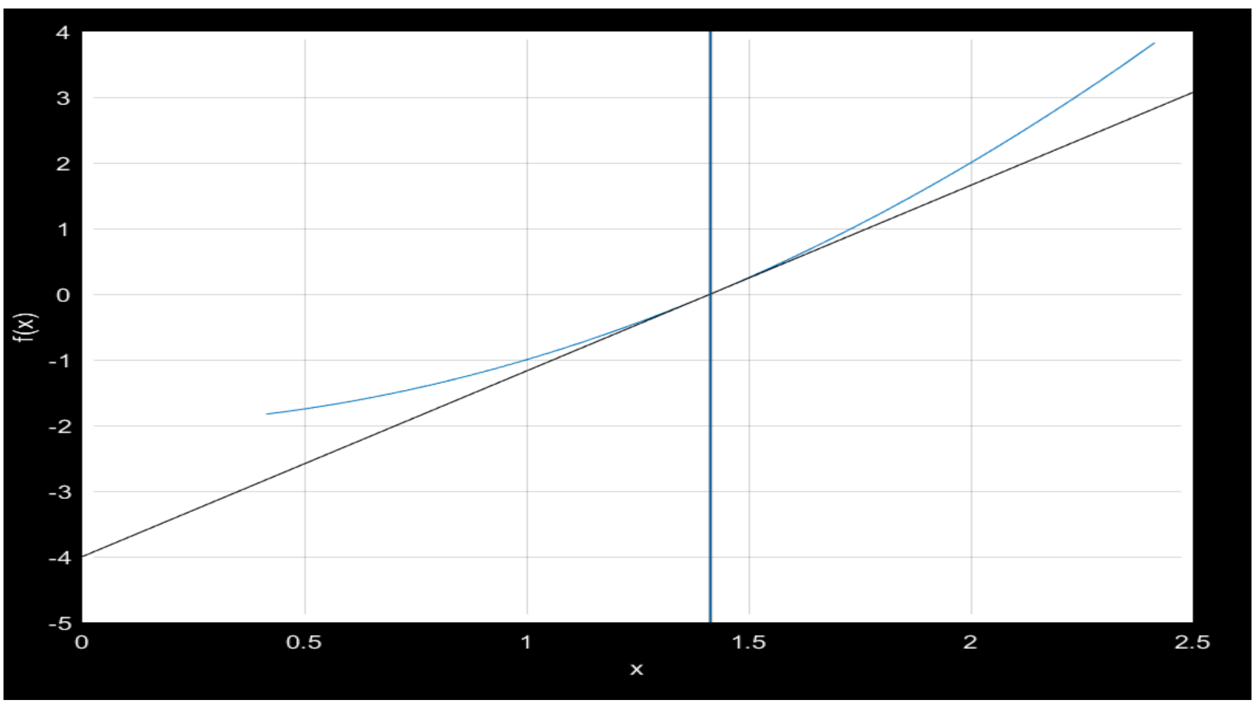


Read a File
Previous
Next
Equation
x^2-2
Epilson
Iterations
Method
Secant
Solve
All Methods
Clear
root : 1.414214
number of itterations : 6
timeelapsed : 0.004147
precision : 0.000004
itterations
xr xl xu f(xl) f(xr) error
1 1.666667e+00 5.000000e-01 1.000000e+00 -1.750000e+00 -1.000000e+00
2 1.375000e+00 1.000000e+00 1.666667e+00 -1.000000e+00 7.777778e-01 2.121212e+01
3 1.410959e+00 1.666667e+00 1.375000e+00 7.777778e-01 -1.093750e-01 2.548544e+00
4 1.414259e+00 1.375000e+00 1.410959e+00 -1.093750e-01 -9.194971e-03 2.333708e-01
5 1.414214e+00 1.410959e+00 1.414259e+00 -9.194971e-03 1.295743e-04 3.243038e-03
6 1.414214e+00 1.414259e+00 1.414214e+00 1.295743e-04 -1.492675e-07 3.731628e-06
f(x)
x



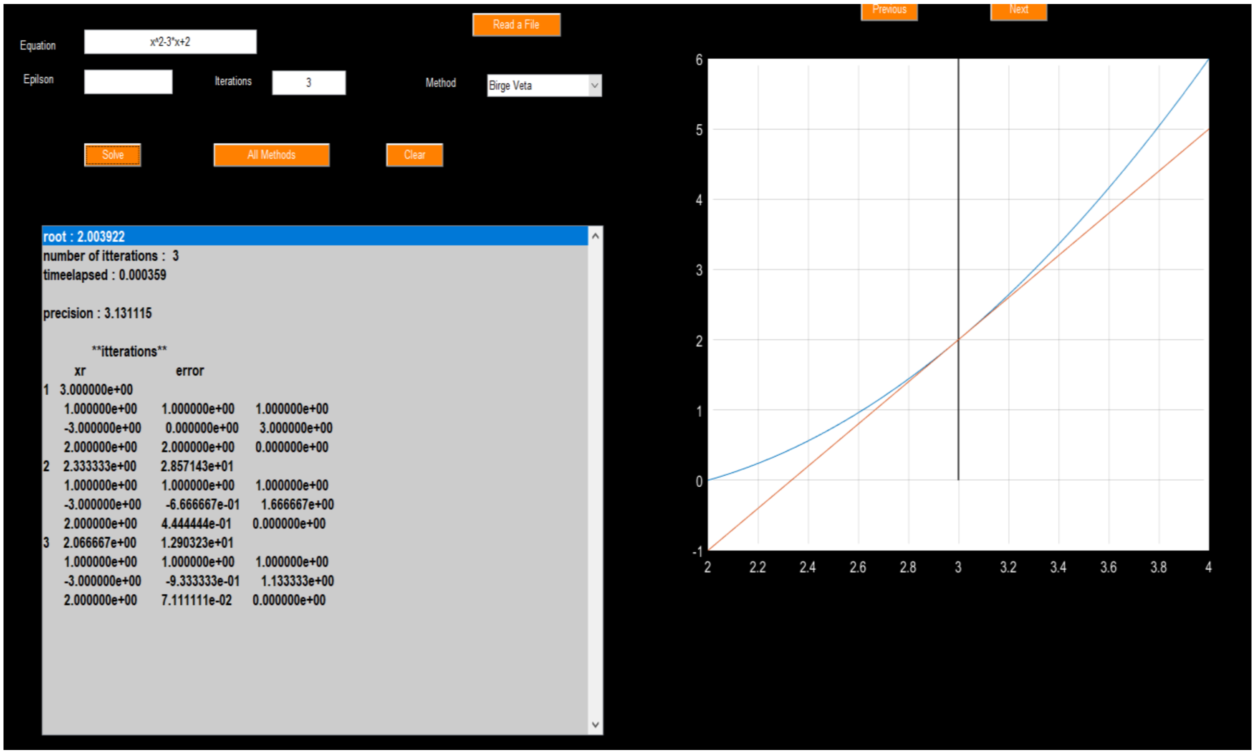
f(x)
x

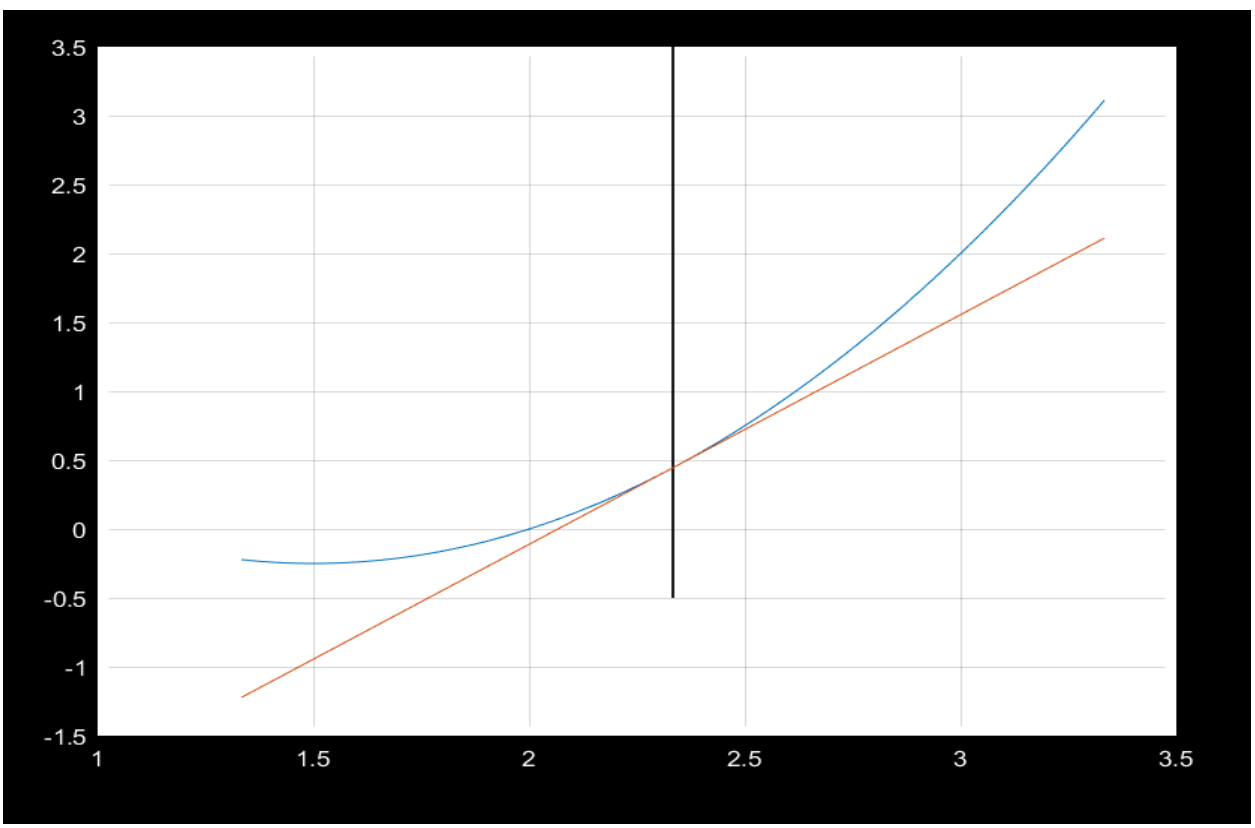


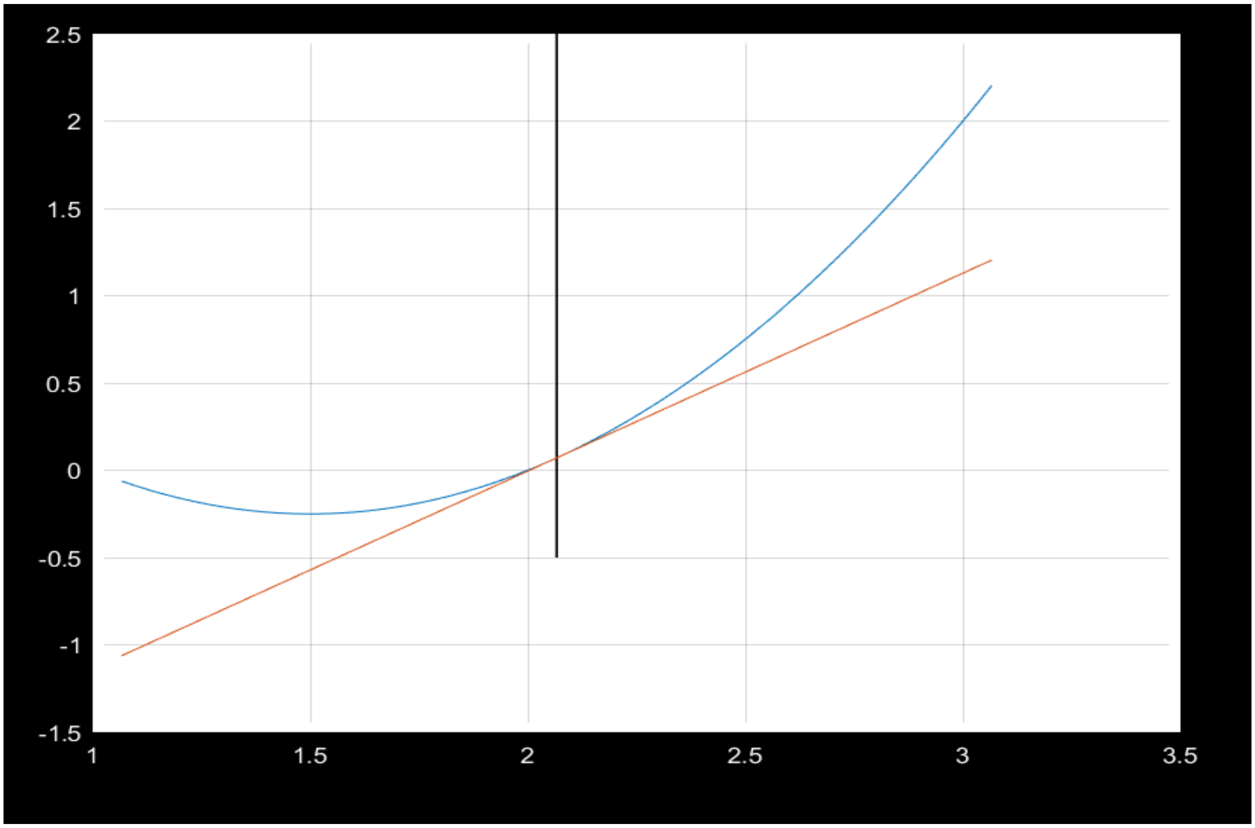


Secant converges to the root after 6 iterations

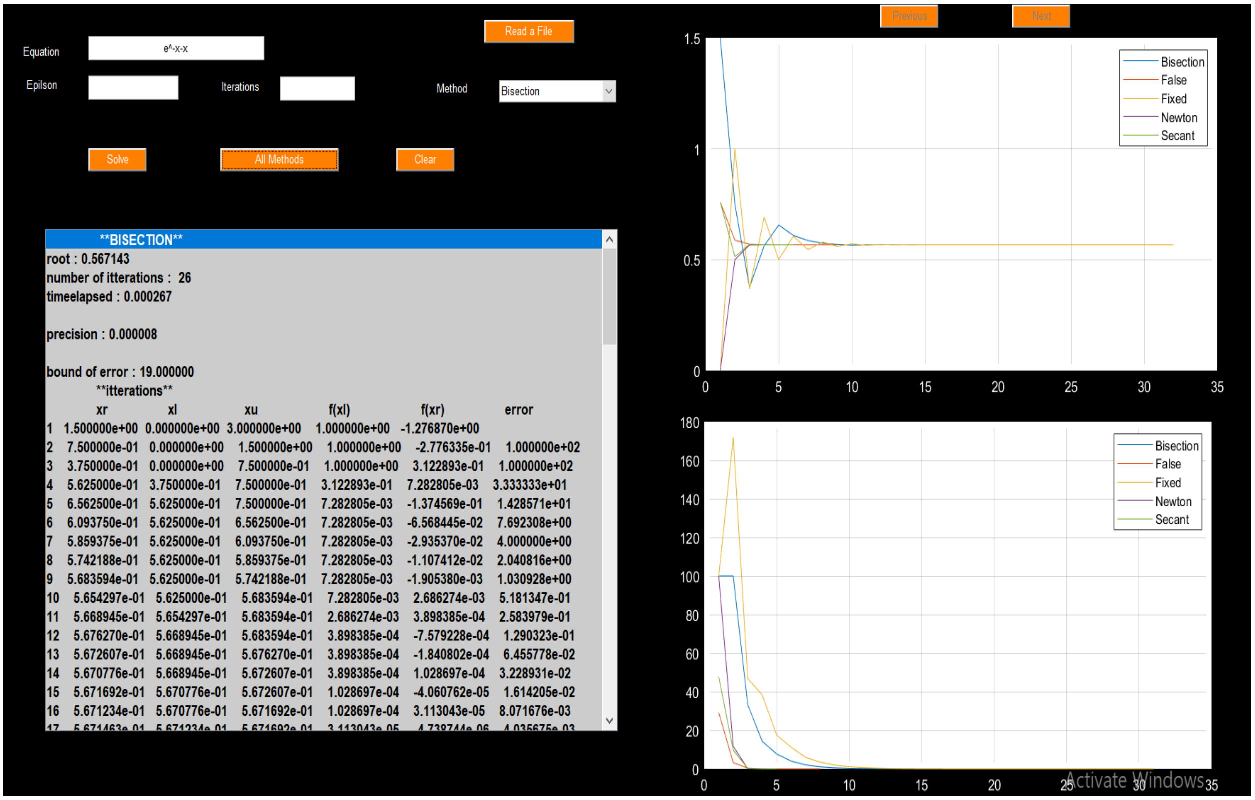
4) Birge-Veta


Read a File
Previous
Next
Equation
x^2-3*x+2
Epilson
Iterations
3
Method
Birge Veta
Solve
All Methods
Clear
root : 2.003922
number of itterations : 3
timeelapsed : 0.000359
precision : 3.131115
itterations
xr error
1 3.000000e+00
1.000000e+00 1.000000e+00 1.000000e+00
-3.000000e+00 0.000000e+00 3.000000e+00
2.000000e+00 2.000000e+00 0.000000e+00
2 2.333333e+00 2.857143e+01
1.000000e+00 1.000000e+00 1.000000e+00
-3.000000e+00 -6.666667e-01 1.666667e+00
2.000000e+00 4.444444e-01 0.000000e+00
3 2.066667e+00 1.290323e+01
1.000000e+00 1.000000e+00 1.000000e+00
-3.000000e+00 -9.333333e-01 1.133333e+00
2.000000e+00 7.111111e-02 0.000000e+00

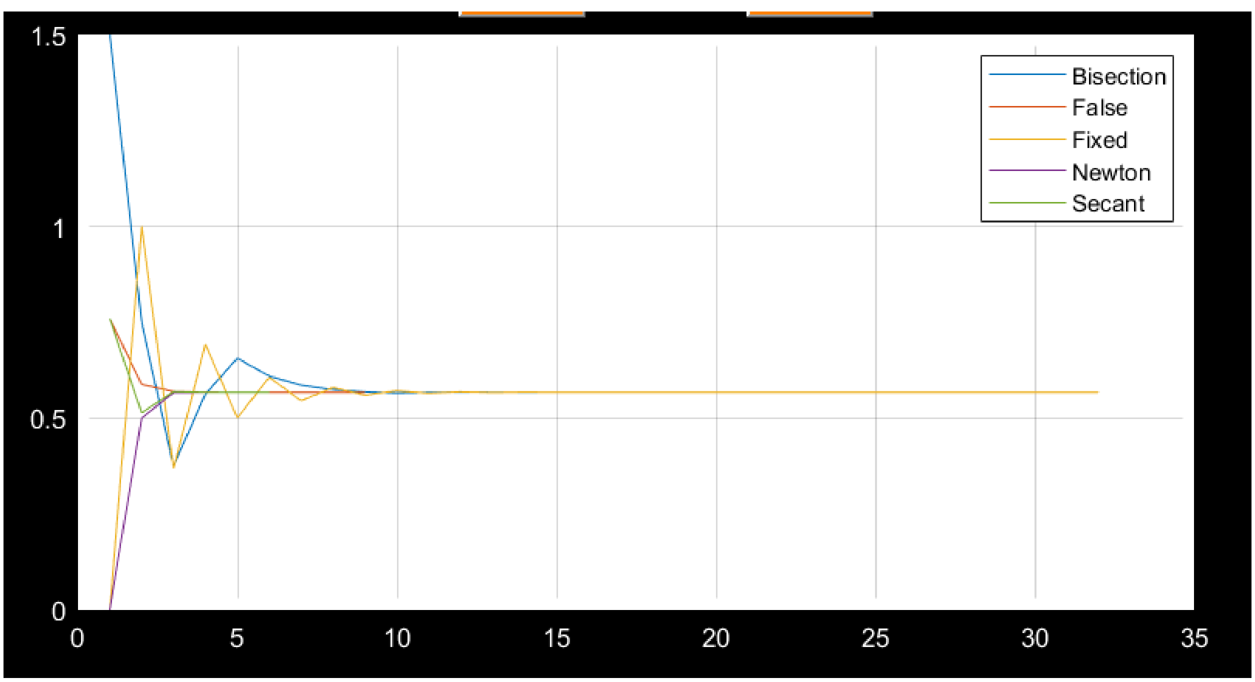



2.5
2
1.5
1
0.5
0
-0.5
-1
-1.5
1
1.5
2
2.5
3
3.5

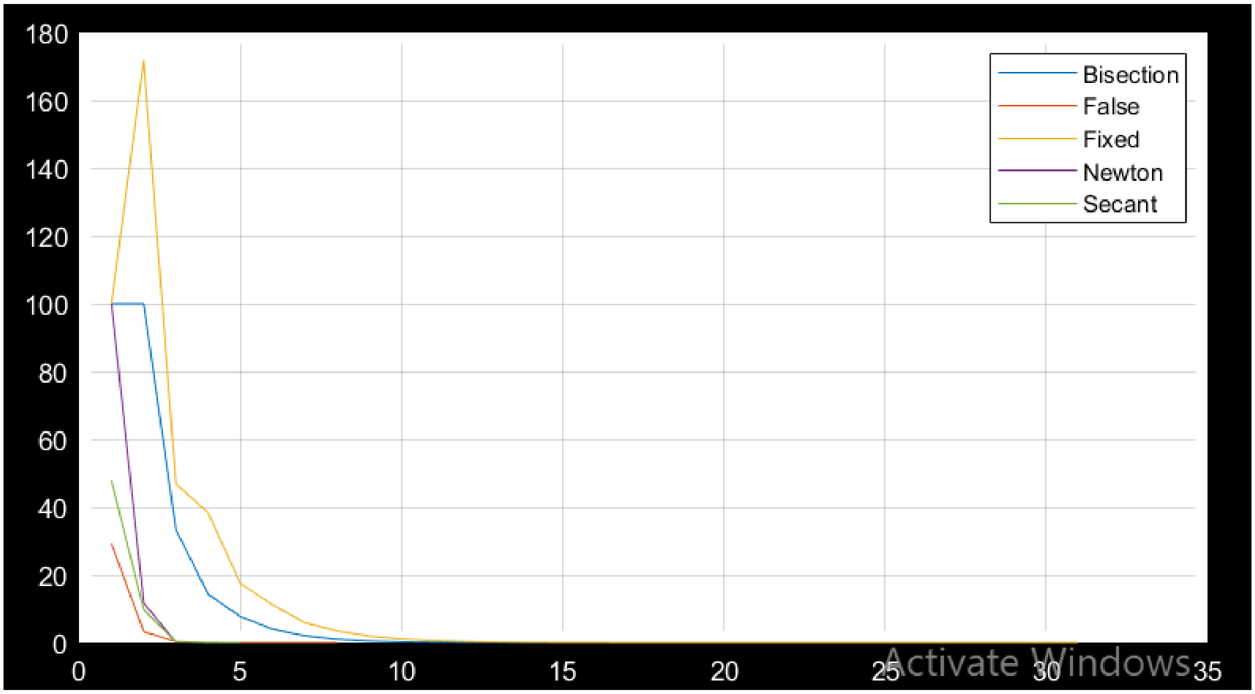
All Methods:



Iterations:



Errors:



Files in GUI:

1) All methods:

```
                 **BISECTION**
root : 0.567143
number of itterations :  26
timeelapsed : 0.000267

precision : 0.000008

bound of error : 19.000000
                **itterations**
```

	xr	xl	xu	f(xl)	f(xr)	error
1	1.500000e+00	0.000000e+00	3.000000e+00	1.000000e+00	-1.276870e+00	
2	7.500000e-01	0.000000e+00	1.500000e+00	1.000000e+00	-2.776335e-01	1.000000e+02
3	3.750000e-01	0.000000e+00	7.500000e-01	1.000000e+00	3.122893e-01	1.000000e+02
4	5.625000e-01	3.750000e-01	7.500000e-01	3.122893e-01	7.282805e-03	3.333333e+01
5	6.562500e-01	5.625000e-01	7.500000e-01	7.282805e-03	-1.374569e-01	1.428571e+01
6	6.093750e-01	5.625000e-01	6.562500e-01	7.282805e-03	-6.568445e-02	7.692308e+00
7	5.859375e-01	5.625000e-01	6.093750e-01	7.282805e-03	-2.935370e-02	4.000000e+00
8	5.742188e-01	5.625000e-01	5.859375e-01	7.282805e-03	-1.107412e-02	2.040816e+00
9	5.683594e-01	5.625000e-01	5.742188e-01	7.282805e-03	-1.905380e-03	1.030928e+00
10	5.654297e-01	5.625000e-01	5.683594e-01	7.282805e-03	2.686274e-03	5.181347e-01
11	5.668945e-01	5.654297e-01	5.683594e-01	2.686274e-03	3.898385e-04	2.583979e-01

```
        **FALSE-POSITION**
root : 0.567143
number of itterations :   9
timeelapsed : 0.000202

precision : 0.000006

             **itterations**
          xr           xl             xu               f(xl)          f(xr)                   error
1   7.594527e-01  0.000000e+00  3.000000e+00    1.000000e+00   -2.915303e-01
2    5.880255e-01   0.000000e+00    7.594527e-01    1.000000e+00    -3.260260e-02    2.915303e+01
3    5.694596e-01   0.000000e+00    5.880255e-01    1.000000e+00    -3.628510e-03    3.260260e+00
4    5.674008e-01   0.000000e+00    5.694596e-01    1.000000e+00    -4.035463e-04    3.628510e-01
5    5.671719e-01   0.000000e+00    5.674008e-01    1.000000e+00    -4.487691e-05    4.035463e-02
6    5.671465e-01   0.000000e+00    5.671719e-01    1.000000e+00    -4.990552e-06    4.487691e-03
7    5.671436e-01   0.000000e+00    5.671465e-01    1.000000e+00    -5.549754e-07    4.990552e-04
8    5.671433e-01   0.000000e+00    5.671436e-01    1.000000e+00    -6.171616e-08    5.549754e-05
9    5.671433e-01   0.000000e+00    5.671433e-01    1.000000e+00    -6.863158e-09    6.171616e-06
        **FIXED-POINT**
root : 0.567143
number of itterations :   32
timeelapsed : 0.009741

precision : 0.000007

bound of error : -0.567143
converge, oscillate
```

2) **Birge_veta**

```
root : 2.003922
number of itterations :   3
timeelapsed : 0.000359

precision : 3.131115

              **itterations**
        xr                | error
1   3.000000e+00
      1.000000e+00        1.000000e+00       1.000000e+00
      -3.000000e+00        0.000000e+00       3.000000e+00
      2.000000e+00        2.000000e+00       0.000000e+00
2    2.333333e+00        2.857143e+01
      1.000000e+00        1.000000e+00       1.000000e+00
      -3.000000e+00        -6.666667e-01       1.666667e+00
      2.000000e+00        4.444444e-01       0.000000e+00
3    2.066667e+00        1.290323e+01
      1.000000e+00        1.000000e+00       1.000000e+00
      -3.000000e+00        -9.333333e-01       1.133333e+00
      2.000000e+00        7.111111e-02       0.000000e+00
```

3) Newton:

```
root : 0.062378
number of itterations :  4
timeelapsed : 0.001589

precision : 0.000008

bound of error : 2.483556
```

itterations

	xr	f(x)	df(x)	error
1	5.000000e-02	1.118000e-04	-9.000000e-03	
2	6.242222e-02	-3.977810e-07	-8.909732e-03	1.990032e+01
3	6.237758e-02	4.429375e-11	-8.911714e-03	7.157328e-02
4	6.237758e-02	5.421011e-19	-8.911714e-03	7.968061e-06

4) False-position:

```
root : 1.000000
number of itterations :  84
timeelapsed : 0.003334

precision : 0.000010
```

itterations

	xr	xl	xu	f(xl)	f(xr)	error
1	0.133333	0.000000	3.000000	-4.000000	-3.599684	
2	0.248502	0.133333	3.000000	-3.599684	-3.250680	46.345210
3	0.348717	0.248502	3.000000	-3.250680	-2.939062	28.738145
4	0.436331	0.348717	3.000000	-2.939062	-2.654762	20.079663
5	0.513100	0.436331	3.000000	-2.654762	-2.391389	14.961796
6	0.580382	0.513100	3.000000	-2.391389	-2.145392	11.592712
7	0.639273	0.580382	3.000000	-2.145392	-1.915169	9.212285
8	0.690700	0.639273	3.000000	-1.915169	-1.700307	7.445595
9	0.735472	0.690700	3.000000	-1.700307	-1.500990	6.087521
10	0.774318	0.735472	3.000000	-1.500990	-1.317565	5.016761
11	0.807902	0.774318	3.000000	-1.317565	-1.150270	4.156952
12	0.836835	0.807902	3.000000	-1.150270	-0.999087	3.457415
13	0.861676	0.836835	3.000000	-0.999087	-0.863686	2.882930
14	0.882938	0.861676	3.000000	-0.863686	-0.743444	2.408026
15	0.901082	0.882938	3.000000	-0.743444	-0.637493	2.013637
16	0.916526	0.901082	3.000000	-0.637493	-0.544787	1.685078
17	0.929641	0.916526	3.000000	-0.544787	-0.464177	1.410777

"Part: 2"

Requirements:

The required was a program for solving systems of linear equations. The program utilizes the following numerical methods in solving the given linear equations:

- Gaussian-elimination (direct).
- LU decomposition (direct).
- Gaussian-Jordan (direct).
- Gauss-Seidel (iterative).

Program features:

- An interactive GUI enabling the user to enter the equations, the required method and the method parameters (if required).
- The ability to read the input from files.
- The output is available in a file and in the GUI showing the answer, execution time and for iterative methods also shows the precision, number of iterations and the answer after each iteration.
- A plot is also available for the iterative method showing the result after each iteration.

Main algorithms:

Let the equations system be AX = B.

Direct methods:

Pivoting:

In each of the direct methods, pivoting (partial) was used during the elimination phase in order to:

- Decrease the round off errors.
- Avoid the threat of division by zero.

Pseudo code for pivoting(i):

- pivotLocation = location of row having element with maximum magnitude in ith row.
- Swap A(i) with A(pivoltLocation)
- Swap B(i) with B(pivotLocation)
- SignOfDeterminant = -1*SignOfDeterminant (due to row swapping we change the sign of the determinant in case the determinant was being calculated)

- **Gaussian-Elemination:**
  Takes as input the system of equations and the number of variables; outputs an augmented matrix [A B] after forward elimination, a matrix X having the results for each variable and the determinant value (to show whether this system has a valid solution or not).

  Pseudo code for Gaussian-Elemination:

1. Forward elimination phase:
   - M = [A B]
   - For i = 1 to numberOfVariables
   - Pivot(i)
   - Delta = delta * M(i,i) →update the determinant value with the new main diagonal element.
   - For j = i+1 to numberOfVariables
   - Multiplier = M(j,i) /M(i,i) →get the dividend from the pivot
   - For k = I to numberOfVariables+1
   - M(j,k) = M(j,k) – multiplier * M(I,k) → update the value of each coefficient in this iteration.
   - End for
   - End for
   - End for
   - Delta = delta * M(numberOfVariables,numberOfVariables) → to complete the determinant calculation.

This phase is **O(n^3)**.

2. Backward substitution phase:
    - For I = numberOfVariables downto 1
    - Sum = 0
    - For j = i+1 to numberOfVariables
    - Sum = sum + M(I,j) * X(j) → in first iteration this loop is not entered as X(n) was still not calculated.
    - End for
    - X(i) = (M(I,numberOfVariables+1)-sum)/M(i,i).
    - End for

This phase is **O(n^2)**.

So, Gaussian-Elemination is **O(n^3)** in total.

Code snippet:

```matlab
% algorithm for gaussian elemination returns augmented coeffecient matrix
% and the variables matrix after solution and also the determinant.
% parameters are the number of variables and a matrix containing the left
% hand side and a matrix containing the right hand side results of the system.
function [M,X,delta] = gaussianElemination(num,eq,res)
% assume three equations for trying the algorithm.
X = zeros(1,num);
M = zeros(num,num);
for i = 1 : num
    M(i,1 : num) = getcoefficients(char(eq(i)),num);
end
M = [M res];
delta = 1;
sign = 1;
% forward elemination phase and calculating the determinant.
for i = 1 : num-1
%pivoting part (partial pivoting to eleminate the threat of division by zero and decrease the round off errors).
[maxi MErow]=max(abs(M(i:num,i)));
MErow = MErow+i-1;
if(MErow ~=i)% a change occured.
    sign = -1*sign;
end
temp = M(MErow,1:num+1);
M(MErow,1:num+1) = M(i,1:num+1);
M(i,1:num+1) = temp;
  delta  =delta * M(i,i);
    for j = i+1 : num
        multiplier = M(j,i)/M(i,i);
         for k = i : num+1
             M(j,k) = M(j,k) - multiplier*M(i,k);
         end
     M(j,i) = 0;
    end
end
```

```matlab
delta = delta * M(num,num)*sign;
% backward substitution phase.
for i = num:-1:1
    sum = 0;
    for j = i+1 : num
        sum = sum + M(i,j)*X(j);
    end
    X(i)  = (M(i,num+1)-sum)/M(i,i);
end
```

- **Gauss-Jordan:**

  This method is similar to the Gaussian elimination, but here we have no backward substitution and the forward elimination is done on all the rows not just the successive ones, also scaling the main diagonal elements to be ones so the solution becomes the last column in the augmented matrix. The algorithm takes input the system and the number of variables then outputs the solution of the system and a flag indicating whether this system has a valid solution or not.

Pseudo code for Gauss-Jordan:

Has only one phase (elimination):

- Flag = 1
- A = [A B]
- For i=1 to num
- Pivot(i)
- Divisor = A(I,i)
- A(I,I to num+1) = A(I,I to num+1)/divisor → scale the row so that the main diagonal element becomes unity.
- For j = 1 to num
- If j != i
- Multiplier = A(j,i)/A(i,i) → elimination for all the rows except for that being the current pivot.
- For k = i to num+1
- A(j,k) = A(j,k) – Multiplier * A(I,k) → eliminate all the coefficients and results.
- End for
- End if
- End for
- Flag = flag * A(I,i) → 1 for solvable system
- End for
- Flag = flag * A(num,num)
- If flag = 1
- X = A(1 to num,num+1) →solution is last column
- Else
- No solution
- End else
- End if

this method is **O(n^3)**.

<u>Code snippet:</u>

algorithm for the gauss-jordan elemination method.

takes a matrix with the equations, a matrix with the results and the number of variables. returns a matrix with the solution for the system and a flag indicating the existence of this solution 1 if exists 0 if doesnt exist. can be used to get the inverse of the matrix by entering the identity matrix augmented with the coefficient matrix but would require more time complexity so it is not implemented as only the solution is required.

```
function [X,flag] = gaussJordan(num,eq,res)
X = zeros(num,1);
A = zeros(num,num);
for i = 1: num
     A(i,1 : num) = getcoefficients(char(eq(i)),num);
end
A = [A res];
flag = 1;
% elemination phase and calculating the determinant.
for i = 1 : num
%pivoting part (partial pivoting to eleminate the threat of division by zero and decrease th
[maxi MErow]=max(abs(A(i:num,i)));
MErow = MErow+i-1;
temp = A(MErow,1:num+1);
A(MErow,1:num+1) = A(i,1:num+1);
A(i,1:num+1) = temp;
    % scale the row
    divisor = A(i,i);
    A(i,i:num+1) = A(i,i:num+1)/divisor;
    for j = 1 : num
        if(j~=i)
            multiplier = A(j,i)/A(i,i);
         for k = i : num+1
             A(j,k) = A(j,k) - multiplier*A(i,k);
         end
        end

    end
    flag  = flag * A(i,i);
end
flag = flag * A(num,num);
% get rid of any round off errors.
for i = 1:num
    A(i,i+1:num) = 0;
end
% in case of the presence of a solution we return it else nans are
% returned.
if(flag == 1)
    X = A(1:num,num+1);
else
    X(1:num) = nan;
if(isnan(flag))
    flag = 0;
end
```

- **LU Decomposition:**

This method has three phases decomposition, forward substitution and backward substitution. Decomposition is done using the Doolittle method reusing the space to store both L and U matrices in a single compact matrix LU. After first phase we get A = LU so LU X = B. after the forward substitution phase we get Y such that L Y = B. after the backward substitution phase we get X such that U X = Y. the decomposition phase uses the Gaussian forward elimination algorithm described earlier with the coefficients as elements of U and the multipliers as the elements of L. The algorithm takes as input the system and the number of variables and outputs the LU compact matrix (U in upper triangle and L in lower triangle), the solution and the determinant of the system.

Pseudo code for LU Decomposition:

1. Doolittle decomposition:
   - Delta = 1
   - LU = A
   - For I = 1 to num
   - Pivot(i)
   - Delta = delta * LU(I,i)
   - For j = i+1 to num
   - Multiplier = LU(j,i)/LU(I,i)
   - For k = I to num
   - LU(j,k) = LU(j,k) – multiplier * LU(I,k) → U elements
   - End for
   - LU(j,i) = multiplier → L elements
   - End for
   - End for
   - Delta = delta * LU(num,num) → complete the calculation of the determinant.

   O(n^3).

2. Forward substitution:
    - Y = zeros
    - For I = 1 to num
    - Sum = B(i)
    - For j = 1 to i-1
    - Sum = sum – LU(I,j) * Y(j) → doesn't enter when I = 1 as Y(1) was still not calculated
    - End for
    - Y(i) = sum
    - End for

    O(n^2).

3. Backward substitution:
    - For I = num down to 1
    - Sum = 0
    - For j = i+1 to num
    - Sum = sum + LU(I,j) * X(j) → won't enter when I = num as X(num) was still not calculated.
    - End for
    - X(i) = (y(i) – sum) / LU(I,i)
    - End for

    O(n^2).

So LU decomposition is **O(n^3)** algorithm.

Code snippet:

algorithm for LU decomposition algorithm takes a matrix for the equations

a matrix for the results and the number of variables, returns a matrix with the values of each variable, the determinant and a compact matrix containing the lower and upper matrices.

```
function [LU,X,delta] = LUDecomposition(num,eq,res)
X = zeros(1,num);
LU = zeros(num,num);
for i = 1: num
     LU(i,1 : num) = getcoefficients(char(eq(i)),num);
end
delta = 1;
sign = 1;
% decomposition using doolittle method reusing the space in LU as a compact
% matrix containing both L and U matrices using gaussian elemination.
for i = 1 : num-1
%pivoting part (partial pivoting to eleminate the threat of division by zero and decrease the
[maxi MErow]=max(abs(LU(i:num,i)));
MErow = MErow+i-1;
if(MErow ~=i)% a change occured.
    sign = -1*sign;
end
temp = LU(MErow,1:num);
LU(MErow,1:num) = LU(i,1:num);
LU(i,1:num) = temp;
temp = res(MErow,1);
res(MErow,1) = res(i,1);
res(i,1) = temp;
  delta  =delta * LU(i,i);
    for j = i+1 : num
        multiplier = LU(j,i)/LU(i,i);
         for k = i : num
             LU(j,k) = LU(j,k) - multiplier*LU(i,k);
         end
     LU(j,i) = multiplier;
    end
end
delta = delta * LU(num,num) *sign;
% forward substitution phase Lb=res
b = zeros(1,num); % an intermediate matrix
for i = 1: num
    sum = res(i);
    for j = 1:i-1
        sum = sum - LU(i,j)*b(j);
    end
    b(i) = sum;
end
% backward substitution phase,similar to gaussian elemination UX = b.
for i = num:-1:1
    sum = 0;
    for j = i+1:num
```

```
        sum = sum + LU(i,j)*X(j);
    end
    X(i) = (b(i)-sum)/LU(i,i);
end
```

Iterative methods:

- **Gauss-Seidel:**

This iterative method takes the initial guesses for the solution of the system and uses mere substitution to get the next guess, taking into consideration that the newest guess is the one used in the substitution. The substitution stops when the maximum number of allowable iterations are reached or the required precision is achieved.

The algorithm takes as input the system, number of variables, the initial guesses, the maximum iterations allowed and the precision. It returns a matrix containing the solution after each iteration, a matrix containing the absolute relative error between each two consecutive iterations, the final solution and the total number of iterations made.

Pseudo code for Gauss-Seidel:

- For I = 1 to maxIterations
- Max = 0
- For k = 1 to num
- X(k,i+1) = B(k)
- For j = 1 to num
- If j > k →use the guess from previous iteration.
- X(k,i+1) = X(k,i+1) – A(k,j) * X(j,i)
- End if
- If(j<k) → use the guess from current iteration.
- X(k,i+1) = X(k,i+1) – A(k,j) * X(j,i+1)
- End if
- End for
- X(k,i+1) = X(k,i+1)/A(k,k) → complete calculation of current iteration.
- Error(k,i) = |X(k,i+1)-X(k,i)| → error
- If(Error(k,i) > max)

- Max = Error(k,i) → get maximum error to compare with the required precision.
- End if
- End for
- If(max<precision)
- Break
- End for
- If(i>maxIterations)
- Iterations = maxIterations
- I = maxIterations
- Else
- Iterations = i
- End else
- End if
- Final = X(1 to num,i+1) → set final solution value.

Gauss seidel is **O(n^2)** for each iteration.

Code snippet:

algorithm for the gauss- seidel iterative method.

```
takes a matrix containing the equations, a matrix containing the results
a matrix containing the initial guess for each variable column wise
,the number of variables, the precision (default value 0.00001) and the
maximum number of iterations (default value 50).
returns a matrix containing the value of each variable after each
iteration row wise, a matrix containing the relative error after each
iteration for each variable row wise, a matrix containig the final values
and the total number of iterations.
the matrix containing the values has its first column as the initial
guess.
```

```
function[X,Error,Final,Iterations] = gaussSeidel(num,eq,res,initial,precision,maxIterations
X = initial;
A = zeros(num,num);
for i = 1: num
     A(i,1 : num) = getcoefficients(char(eq(i)),num);
end
%a loop to make sure that the pivoting coefficients are not zeros
for i = 1:num
    if(A(i,i) ==0)
        for j=1:num
            if(A(j,i) ~= 0)
                temp = A(j,1:num);
                A(j,1:num) = A(i,1:num);
                A(i,1:num) = temp;
                temp = res(j,1);
                res(j,1) = res(i,1);
                res(i,1) = temp;
            end
        end
    end
end
disp(A);

% gauss seidel iterations claculating the maximum error each time to check
% the stopping precision.
for i = 1: maxIterations
    X = [X zeros(num,1)];
    max = 0;
    for k = 1:num
        X(k,i+1) = res(k);
    for j = 1: num
        if(j>k)
            X(k,i+1) = X(k,i+1) - A(k,j)*X(j,i);
        end
        if(j<k)
            X(k,i+1) = X(k,i+1) - A(k,j)* X(j,i+1);
        end
    end
    % we divide by the coefficient after calculating the summation to
```

```
    % decrease the round off error amount.
    X(k,i+1) = X(k,i+1) / A(k,k);
    Error(k,i) = abs(X(k,i+1) - X(k,i));
    if(Error(k,i) > max)
        max = Error(k,i);
    end
    end
    if(max<precision)
        break;
    end
end
if(i>maxIterations)
    Iterations = maxIterations;
    i = maxIterations;
else
Iterations = i;
end
Final = X(1:num,i+1);
```

Methods Analysis:

Direct methods:

First: number of computations needed for each method (order):

- Gauss elimination: O(n^3).
  Pivoting requires O(n) comparisons done n times so a total O(n^2).
  Forward elimination requires O(n^3) operations.
  Backward substitution requires O(n^2) operations.
- Gauss-Jordan: O(n^3).
  Same orders as Gauss elimination but with no backward substitution.
  However, as n increases Gauss-Jordan becomes more costly where more accurate calculations give that Gauss-Jordan has total order = 4*n^3/3 whereas Gauss elimination has a total order = 2*n^3/3+O(n^2) which is asymptotically less.
- LU Decomposition: O(n^3).
  To compute L,U once we have O(n^3) operations.
  Forward and backward substitution we have O(n^2) operations.
  Has nearly the same computational cost as Gaussian elimination.

Second: behavior analysis:

Each method of the direct methods should produce the correct solution directly with only a little error corresponding to any round off occurring, however neglecting the problematic systems that would be discussed later, some systems may be ill-conditioned causing faulty results.

We define an ill-conditioned system as a system whose coefficient matrix determinant is nearly zero.

First: for the well-conditioned systems, we will explore the output of each method for the same input system, this output would be true and won't change much by slightly changing the coefficients:

For the input system of equations:

20*a+15*b+10*c = 45.

-3*a-2.249*b+7*c = 1.751

5*a+b+3*c = 9.

We should finally get the solution a = 1, b = 1, c = 1.

Plugging this input into the methods gives the following output:

```
Gaussian-Elemination
5*a+b+3*c = 9
20*a+15*b+10*c = 45
-3*a-2.249*b+7*c-1 = 1.751

Augmented Matrix
20.0000    15.0000    10.0000    45.0000
0.0000     -2.7500    0.5000     -2.2500
0.0000     0.0000     8.5002     8.5002

a             b             c
1.0000     1.0000     1.0000

delta
467.5100
Time taken: 8.640000e-03 seconds
---------------------------------------------------------------
```

```
LU Decomposition
5*a+b+3*c = 9
20*a+15*b+10*c = 45
-3*a-2.249*b+7*c-1 = 1.751

Augmented Matrix
20.0000    15.0000     10.0000
0.2500     -2.7500     0.5000
-0.1500    -0.0004     8.5002

a             b              c
1.0000     1.0000      1.0000

delta
467.5100
Time taken: 1.660544e-02 seconds
---------------------------------------------------------------
 Gauss Jordan
 5*a+b+3*c = 9
 20*a+15*b+10*c = 45
 -3*a-2.249*b+7*c-1 = 1.751

 a          b           c
 1.0000     1.0000      1.0000

 Time taken: 1.408171e-02 seconds
 ---------------------------------------------------------------
```

As shown all the methods gave the correct solution and took nearly the same time for the computation (10^-2 seconds) as all have nearly the same computational time **O(n^3)**.

By slightly changing the coefficients:

20.1a+15*b+9.1*c = 45.

-3*a-2.25*b+6.9*c = 1.751

5.15*a+1.01b+3*c = 9.05.

We obtain the outputs:

```
Gaussian-Elemination
20.1a+15*b+9.1*c = 45
-3*a-2.25*b+6.9*c-1 = 1.751
5.15*a+1.01b+3*c- = 9.05

Augmented Matrix
20.1000    15.0000         9.1000          45.0000
0.0000     -2.8333         0.6684          -2.4799
0.0000     0.0000          8.2556          8.4772

a              b              c
0.9400     1.1175          1.0268

delta
470.1463
Time taken: 3.680427e-03 seconds
---------------------------------------------------------------
LU Decomposition
20.1a+15*b+9.1*c = 45
-3*a-2.25*b+6.9*c-1 = 1.751
5.15*a+1.01b+3*c- = 9.05

Augmented Matrix
20.1000    15.0000         9.1000
0.2562     -2.8333         0.6684
-0.1493    0.0040          8.2556

a              b              c
0.9400     1.1175          1.0268

delta
470.1463
Time taken: 2.889169e-01 seconds
---------------------------------------------------------------
Gauss Jordan
20.1a+15*b+9.1*c = 45
-3*a-2.25*b+6.9*c-1 = 1.751
5.15*a+1.01b+3*c- = 9.05

a                  b                c
0.9400     1.1175          1.0268

Time taken: 3.808427e-03 seconds
```

We can see how the outputs just slightly changed too with slightly changing the system.

Second: for the ill-conditioned system we will explore how the output severely differs when the input coefficients are slightly changed which may cause faults in the outputs:

For the system:

5a+7b = 12.

7a+10b = 17.

We have the true output : a = 1, b = 1.

Plugging into the methods we get:

```
Gaussian-Elemination
5a+7b = 12
7a+10b = 17

Augmented Matrix
7.0000     10.0000         17.0000
0.0000     -0.1429         -0.1429

a             b
1.0000     1.0000

delta
1.0000
Time taken: 5.603840e-03 seconds
---------------------------------------------------------------
LU Decomposition
5a+7b = 12
7a+10b = 17

Augmented Matrix
7.0000     10.0000
0.7143     -0.1429

a             b
1.0000     1.0000

delta
1.0000
Time taken: 2.048853e-03 seconds
---------------------------------------------------------------
Gauss Jordan
5a+7b = 12
7a+10b = 17

a                  b
1.0000     1.0000

Time taken: 1.789440e-03 seconds
---------------------------------------------------------------
```

As shown the solutions were true, however the determinant is significantly small as calculated (equals 1) and is near zero indicating an ill-conditioned system.

By slightly changing the coefficients:

5a+7b = 12.075

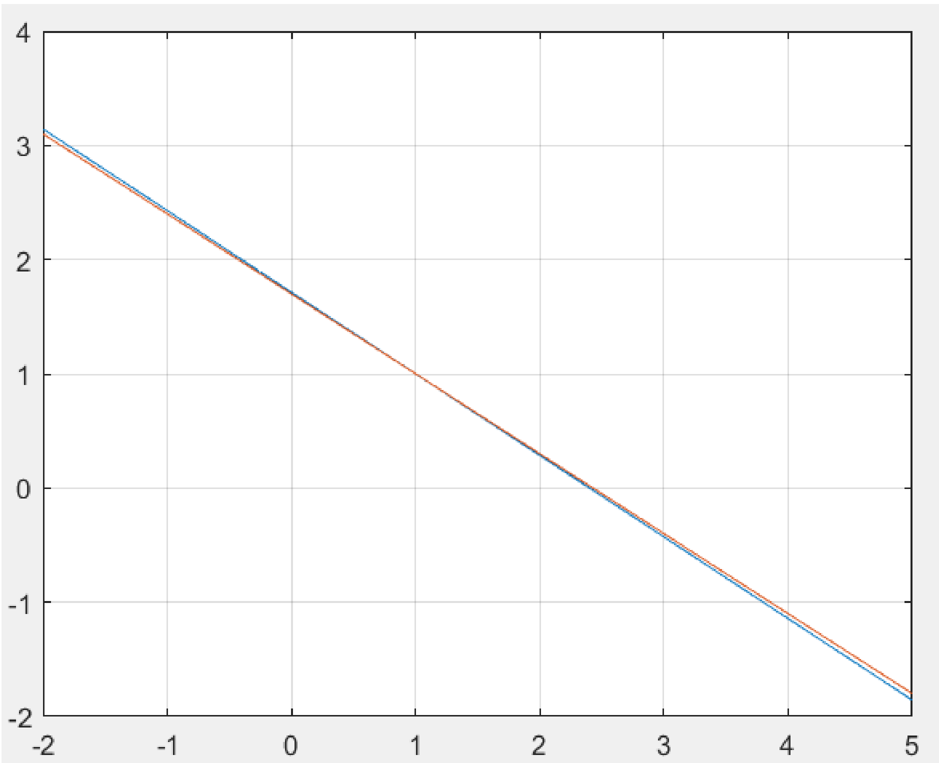
7a+10b = 16.905.

We obtain the outputs:

```
Gaussian-Elemination
5a+7b-12 = 12.075
7a+10b-16 = 16.905

Augmented Matrix
7.0000     10.0000         16.9050
0.0000     -0.1429         -0.0000

a             b
2.4150     0.0000

delta
1.0000
Time taken: 4.059307e-03 seconds
----------------------------------------------------------------
LU Decomposition
5a+7b-12 = 12.075
7a+10b-16 = 16.905

Augmented Matrix
7.0000     10.0000
0.7143     -0.1429

a             b
2.4150     0.0000

delta
1.0000
Time taken: 6.992640e-03 seconds
----------------------------------------------------------------
Gauss Jordan
5a+7b-12 = 12.075
7a+10b-16 = 16.905

a                  b
2.4150     -0.0000

Time taken: 6.827947e-03 seconds
----------------------------------------------------------------
```

As shown here the outputs differed completely although the system only slightly changed.

The graphical representation shows why the system is ill-conditioned with both the functions being nearly represented by the same straight line:



Iterative methods:

First: number of computations needed for each method (order):

- Gauss seidel:
  It takes O(n^2) computations for each iteration as the next guesses are obtained from substitutions into the system.

Second: behavior analysis:

- Iterative methods have significant advantage over direct in ill-conditioned systems solutions as they keep iterating till a certain precisions which ensures that the output is correct to a certain degree of error.
- However, unlike the direct methods the iterative methods may diverge as they don't produce the correct solution in a finite number of steps, on the contrary they theoretically require infinite number of steps to reach the correct solution.
- In case of the gauss seidel, a sufficient condition for convergence is diagonal dominance in which:
  Diagonally dominant: [A] in [A] [X] = [C] is diagonally dominant if:

$$\left|a_{ii}\right| \geq \sum_{\substack{j=1 \\ j \neq i}}^{n} \left|a_{ij}\right| \quad \text{for all 'i'} \qquad \text{and} \quad \left|a_{ii}\right| > \sum_{\substack{j=1 \\ j \neq i}}^{n} \left|a_{ij}\right| \quad \text{for at least one 'i'}$$

Now let's explore a converging and a diverging system for the same number of iterations each:

First for the system:

12a+3b-5c = 1

a+5b+3c = 28

3a+7b+13c = 76

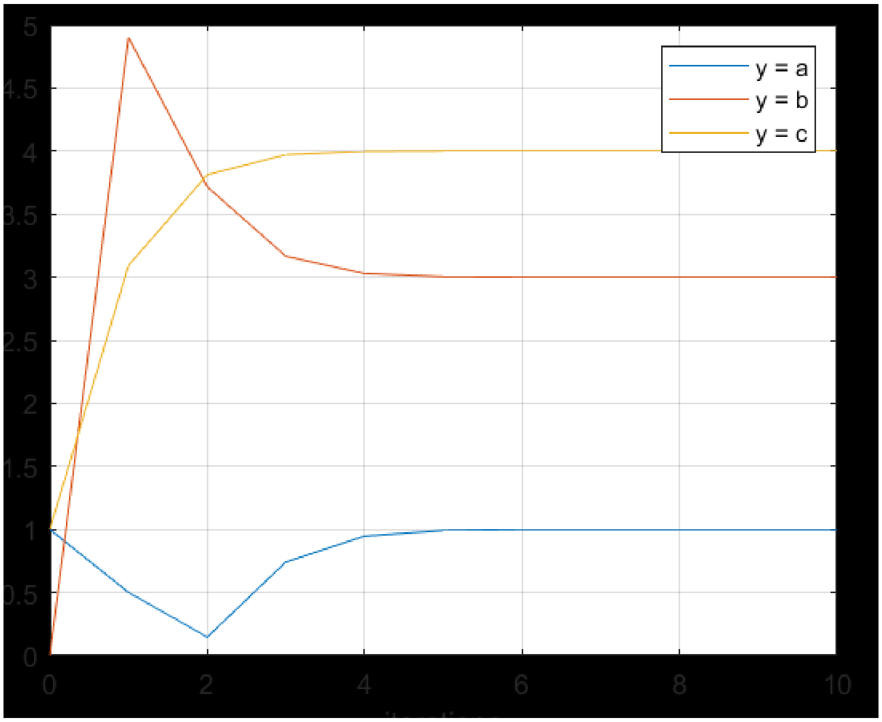
Having an exact solution: a=1,b=3,c=4.

This system is diagonally dominant and so it converges for the gauss seidel method.

We have the output for 10 iterations and an initial guess of a = 1,b = 0,c = 1:

```
Iteration        a        aError      b        bError       c        cError
1           0.5000      0.5000     4.9000      4.9000     3.0923      2.0923
2           0.1468      0.3532     3.7153      1.1847     3.8118      0.7194
3           0.7428      0.5960     3.1644      0.5509     3.9708      0.1591
4           0.9468      0.2040     3.0281      0.1363     3.9971      0.0263
5           0.9918      0.0450     3.0034      0.0248     4.0001      0.0030
6           0.9992      0.0074     3.0001      0.0033     4.0001      0.0000
7           1.0000      0.0008     2.9999      0.0002     4.0000      0.0001
8           1.0000      0.0000     3.0000      0.0001     4.0000      0.0000
9           1.0000      0.0000     3.0000      0.0000     4.0000      0.0000
10          1.0000      0.0000     3.0000      0.0000     4.0000      0.0000
Final Answers
1.0000     3.0000     4.0000
Number of iterations
10
Time taken: 3.021568e-02 seconds
---------------------------------------------------------------
```

That converged to the correct solution as shown.



However for the diverging system:

a-5b = -4

7a-b = 6

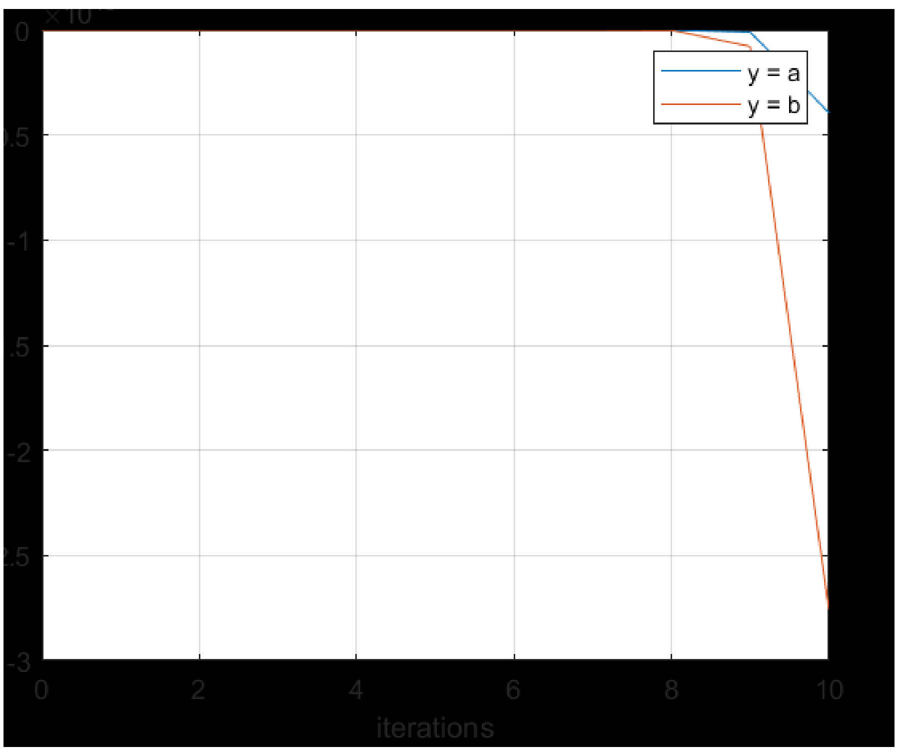
Having an exact solution of:

a=1,b=1 (done by a direct method)

we have the output of gauss seidel for 10 iterations and an initial guess of a = 0 and b = 0 as:

```
---------------------------------------------------------------
Gauss-Seidel
a-5b = -4
7a-b = 6

Iteration           a       aError          b       bError
1               -4.0000    4.0000          -34.0000        34.0000
2               -174.0000          170.0000         -1224.0000       1190.0000
3               -6124.0000         5950.0000        -42874.0000      41650.0000
4               -214374.0000       208250.0000      -1500624.0000           1457750.0000
5               -7503124.0000      7288750.0000     -52521874.0000          51021250.0000
6               -262609374.0000    255106250.0000          -1838265624.0000        1785743750.0000
7               -9191328124.0000           8928718750.0000         -64339296874.0000       62501031250.0000
8               -321696484374.0000         312505156250.0000       -2251875390624.0000     2187536093750.0000
9               -11259376953124.0000       10937680468750.0000     -78815638671874.0000    76563763281250.0000
10              -394078193359374.0000      382818816406250.0000    -2758547353515624.0000          2679731714843750.0000
Final Answers
-394078193359374.0000      -2758547353515624.0000
Number of iterations
10
Time taken: 2.301867e-03 seconds
---------------------------------------------------------------
```

which is cleary diverging.



Note that just interchanging the equations order would result in a converging system with a diagonally dominant coefficient matrix:

```
Gauss-Seidel
7a-b = 6
a-5b = -4

Iteration           a       aError          b       bError
1               0.8571     0.8571       0.9714          0.9714
2               0.9959     0.1388       0.9992          0.0278
3               0.9999     0.0040       1.0000          0.0008
4               1.0000     0.0001       1.0000          0.0000
5               1.0000     0.0000       1.0000          0.0000
6               1.0000     0.0000       1.0000          0.0000
7               1.0000     0.0000       1.0000          0.0000
8               1.0000     0.0000       1.0000          0.0000
9               1.0000     0.0000       1.0000          0.0000
10              1.0000     0.0000       1.0000          0.0000
Final Answers
1.0000     1.0000
Number of iterations
10
Time taken: 8.648107e-03 seconds
---------------------------------------------------------------
```

Problematic systems:

For all the three direct methods we would have the following problematic systems:

- Systems which would lead to divisions by zero and systems which may lead to large round off errors as:

  b = 0

  a+b = -1

  would lead to a division by zero during elimination or decomposition phases.

  Solution: use the pivoting strategy as implemented in the methods.

- Ill-conditioned systems in which outputs change drastically when the system is slightly changed as:

5a+7b=12

7a+10b=17

Solution: this case would be indicated by the determinant resulting after the method in which case to ensure the precision take the outputs as initial guesses for an iterative method and place a certain required degree of precision to obtain higher accuracy.

For the gauss seidel method we have the following problematic systems:

- Systems that would diverge as:

  a-5b = -4

  7a-b = 6

  Solution: try interchanging the equations till a system with a diagonally dominant coefficient matrix is obtained (converging system).

  a-5b = -4 → 7a-b = 6

  7a-b = 6 → a-5b = -4
- Systems that may cause division by zero:

B = 0

A+B = 1

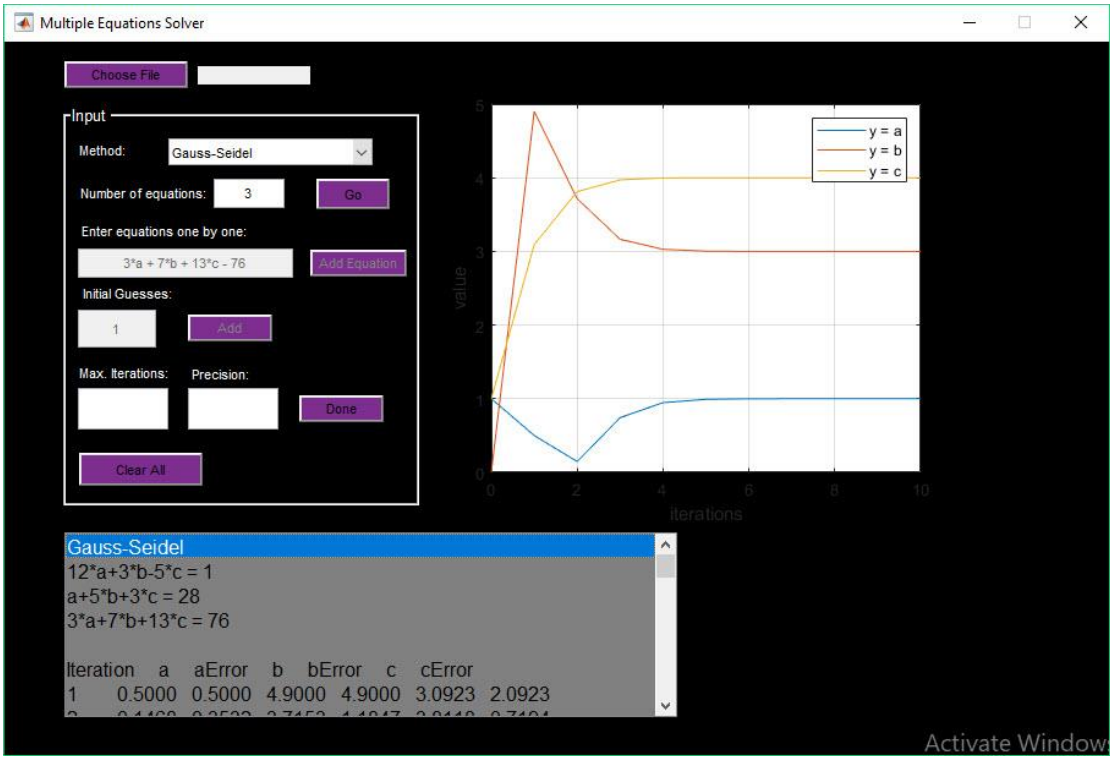
Solution: here we must interchange the equations till each line has its corresponding variable present.

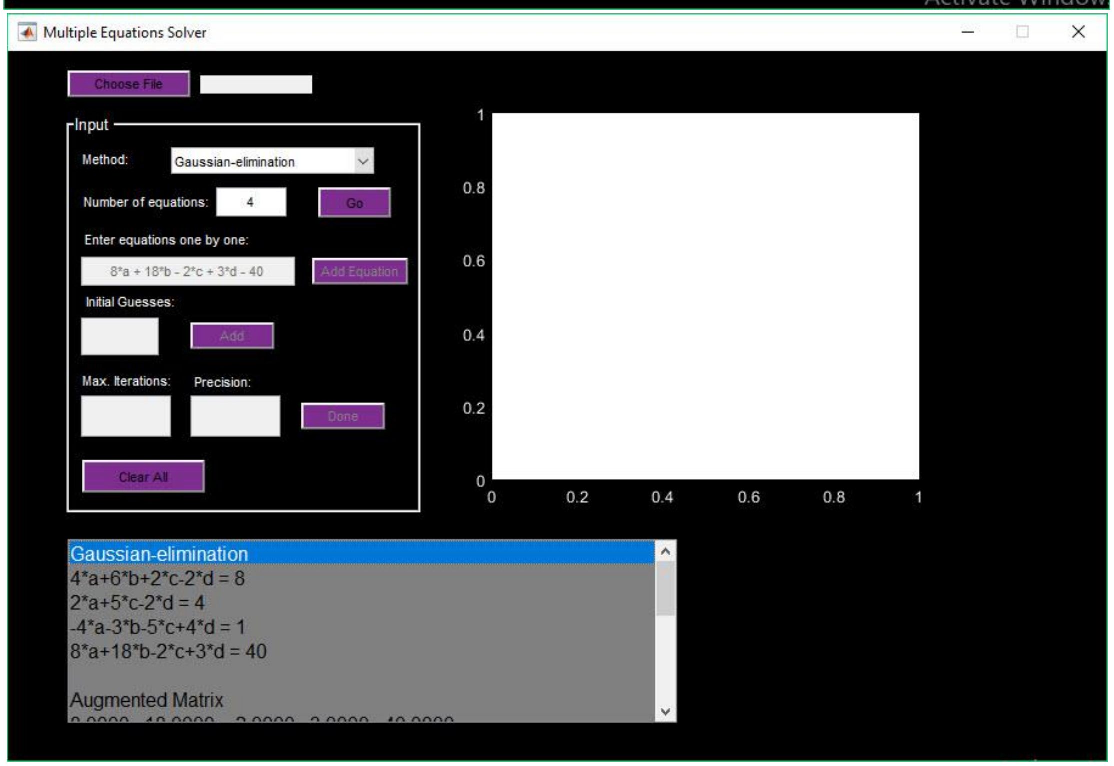
Note: no other systems may cause problems as this method is iterative which would eventually terminate after certain precision or maximum number of iterations.

Systems having no solution can be indicated by their determinant value of the coefficient matrix such that if it equals zero then this system can't have a single unique solution (has infinite solutions or no solution).

Screenshots:

Running examples using interface to solve single example





Using input file for solving multiple or single examples

Provided the screens for the input file and the output file for examples

```
3
3*a + 2*b + c - 6
2*a + 3*b - 7
2*c - 4
Gaussian-elimination
```

```
Gaussian-elimination
3*a+2*b+c = 6
2*a+3*b = 7
2*c = 4

Augmented Matrix
3.0000     2.0000     1.0000     6.0000
0.0000     1.6667     -0.6667    3.0000
0.0000     0.0000     2.0000     4.0000

a              b              c
-0.4000    2.6000     2.0000

delta
10.0000
Time taken: 2.154427e-02 seconds
----------------------------------------------------------------
```

```
4
4*a + 6*b + 2*c - 2*d - 8
2*a + 5*c - 2*d - 4
-4*a - 3*b - 5*c + 4*d - 1
8*a + 18*b - 2*c + 3*d - 40
Gaussian-elimination
```

```
Gaussian-elimination
4*a+6*b+2*c-2*d = 8
2*a+5*c-2*d = 4
-4*a-3*b-5*c+4*d = 1
8*a+18*b-2*c+3*d = 40

Augmented Matrix
4.0000      6.0000      2.0000      -2.0000     8.0000
0.0000      -3.0000     4.0000      -1.0000     0.0000
0.0000      0.0000      1.0000      1.0000      9.0000
0.0000      0.0000      0.0000      3.0000      6.0000

a             b             c             d
-13.5000      8.6667      7.0000      2.0000

delta
-36.0000
Time taken: 7.076509e-03 seconds
---------------------------------------------------------------
```

```
3
a + b + 2*c - 8
-a - 2*b + 3*c - 1
3*a + 7*b + 4*c - 10
Gaussian-jordan
```

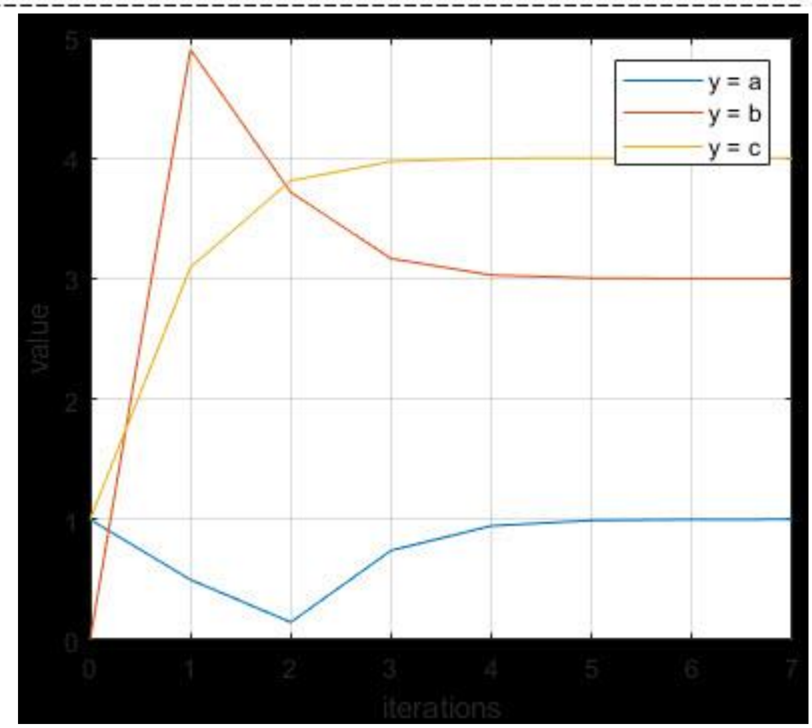
```
Gaussian-jordan
a+b+2*c = 8
-a-2*b+3*c = 1
3*a+7*b+4*c = 10

a           b           c
8.4444      -2.8889     1.2222

Time taken: 2.149694e-02 seconds
---------------------------------------------------------------
```

```
3
12*a + 3*b - 5*c - 1
a + 5*b + 3*c - 28
3*a + 7*b + 13*c - 76
gauss-seidel
1 0 1
20
0.001
```

```
gauss-seidel
12*a+3*b-5*c = 1
a+5*b+3*c = 28
3*a+7*b+13*c = 76
```

Iteration	a	aError	b	bError	c	cError
1	0.5000	0.5000	4.9000	4.9000	3.0923	2.0923
2	0.1468	0.3532	3.7153	1.1847	3.8118	0.7194
3	0.7428	0.5960	3.1644	0.5509	3.9708	0.1591
4	0.9468	0.2040	3.0281	0.1363	3.9971	0.0263
5	0.9918	0.0450	3.0034	0.0248	4.0001	0.0030
6	0.9992	0.0074	3.0001	0.0033	4.0001	0.0000
7	1.0000	0.0008	2.9999	0.0002	4.0000	0.0001

```
Final Answers
1.0000     2.9999      4.0000
Number of iterations
7
Time taken: 8.088579e-03 seconds
-----------------------------------------------------------------
```

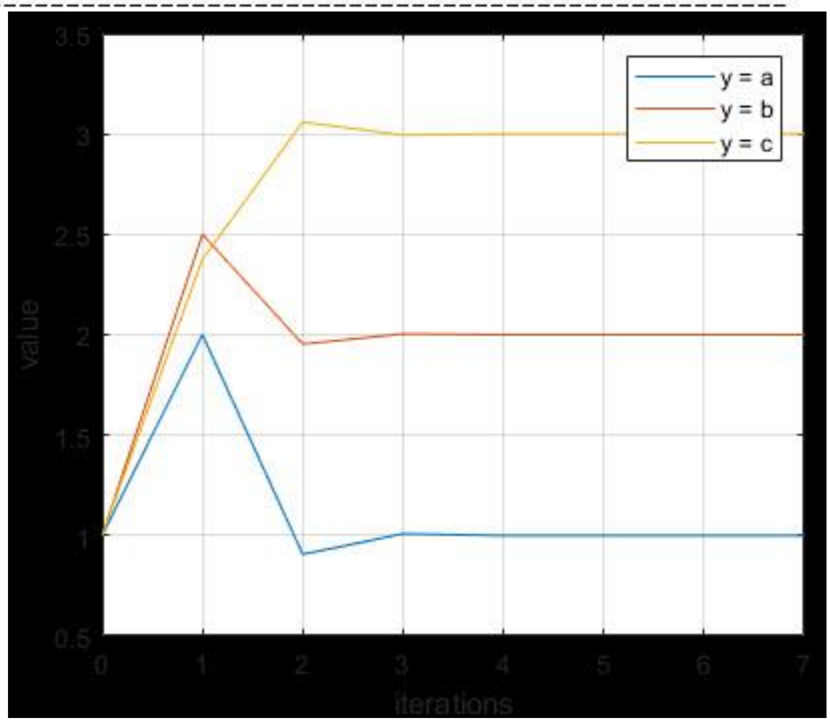


```
3
4*a + 2*b + c - 11
-a + 2*b - 3
2*a + b + 4*c - 16
gauss-seidel
1 1 1
50
0.00001
```

```
gauss-seidel
4*a+2*b+c = 11
-a+2*b = 3
2*a+b+4*c = 16
```

Iteration	a	aError	b	bError	c	cError
1	2.0000	1.0000	2.5000	1.5000	2.3750	1.3750
2	0.9063	1.0938	1.9531	0.5469	3.0586	0.6836
3	1.0088	0.1025	2.0044	0.0513	2.9945	0.0641
4	0.9992	0.0096	1.9996	0.0048	3.0005	0.0060
5	1.0001	0.0009	2.0000	0.0005	3.0000	0.0006
6	1.0000	0.0001	2.0000	0.0000	3.0000	0.0001
7	1.0000	0.0000	2.0000	0.0000	3.0000	0.0000

```
Final Answers
1.0000     2.0000      3.0000
Number of iterations
7
Time taken: 4.612189e-03 seconds
--------------------------------------------------------------
```



```
3
25*a + 5*b + c - 106.8
64*a + 8*b + c - 177.2
144*a + 12*b + c - 279.2
LU Decomposition
```

```
LU Decomposition
25*a+5*b+c-1 = 106.8
64*a+8*b+c-1 = 177.2
144*a+12*b+c-2 = 279.2

Augmented Matrix
25.0000    5.0000     1.0000
2.5600     -4.8000    -1.5600
5.7600     3.5000     0.7000

a             b            c
0.2905     19.6905    1.0857

delta
-84.0000
Time taken: 1.886157e-02 seconds
----------------------------------------------------------------
```