

Root Finder

Numerical Methods

Final project report

# Main Idea

The aim of the numerical analysis approximation methods is to solve equations is not easy to be solved manually. By moving all of an equation’s terms to one side, we can get an equation that says: f(x) = 0, where f(x) is some function of the unknown variable “x”. That transforms the problem into one of finding the x-value at which f(x) = 0. That x-value is the root of this equation. In Bracketing Methods, we use to guess two initial points whether to make us converge quickly or slowly or the root might not be bracketed by those points then we have to choose another two values. But the open methods make it simpler by choosing only one initial point from which we will get the approximated root.

Takes a function as an input, along with its initial points, tolerance and number of iterations.

If tolerance is not given it is set as 0.00001 & number of iterations 50 by default.

The function will be drawn in the GUI as an output along with the iterations, the values of the points, absoluter error, relative error.

The elapsed time will be shown as well.

# Methods:

## Secant Method

Pseudocode:

**1-Normal Mode:**

Function and data (iterations, initial pt1,initial pt2,tolerance) are passed to Secant method function.

initialize array for A and F(A) points;

initialize array for B and F(B) points;

initialize array ffor Xn and, absolute and relative error;

start timer;

start Iteration loop

calculate Xn by the equation Xn=b-((f(b)\*(a-b))/(f(a)-f(b)));

check value of the absolute error reached tolerance or iteration reached max value.

if true

break;

else

continue loop till tolerance reached.

after function is done [a,b,fa,fb,xn,tol,rele,time] are returned and displayed in GUI.

**2-Single Step Mode:**

As normal Mode , but in displaying data,

iteration by iteration is being displayed on clicking Single Solve button, till all iteration are displayed

and by that a message appears to announce that tolerance reached and iterations are all displayed.

Data Structure USED:

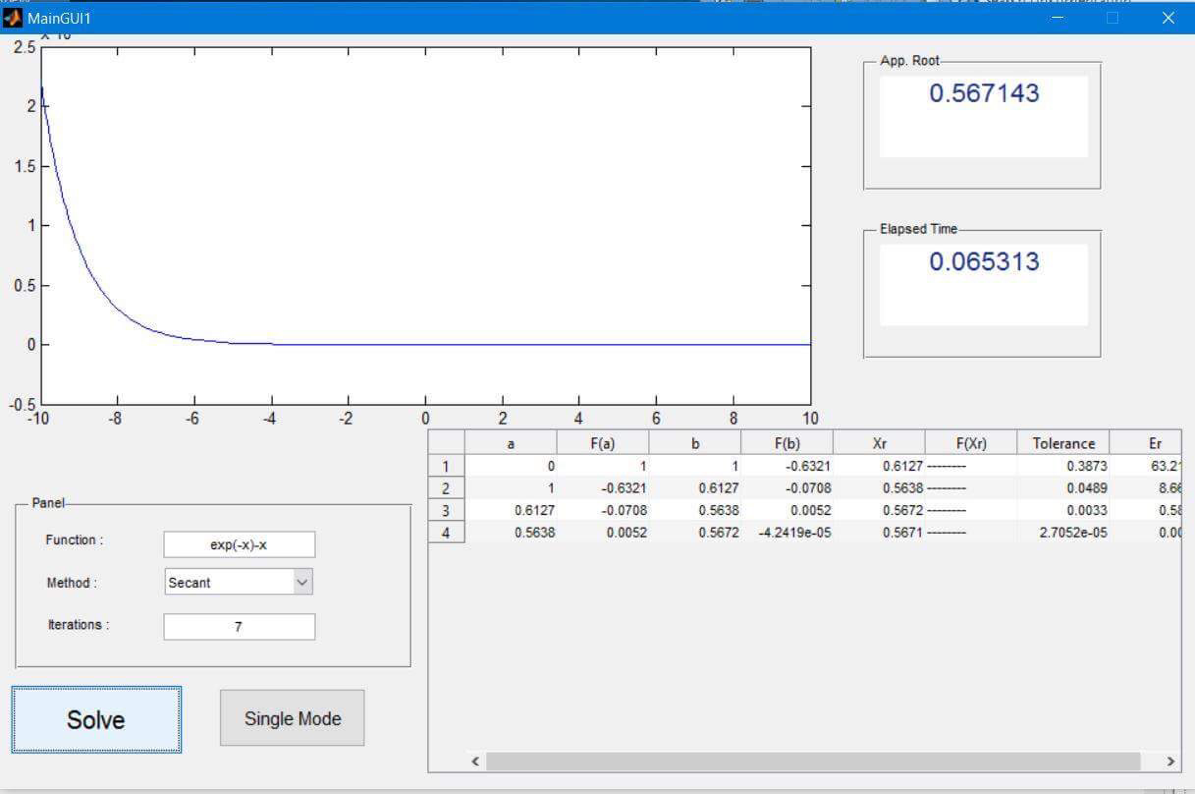
arrays are used to store data and to extract and deal with it, you should use cell2mat function to be able to store data.

arrays are easily used to extract the required data by indexing.

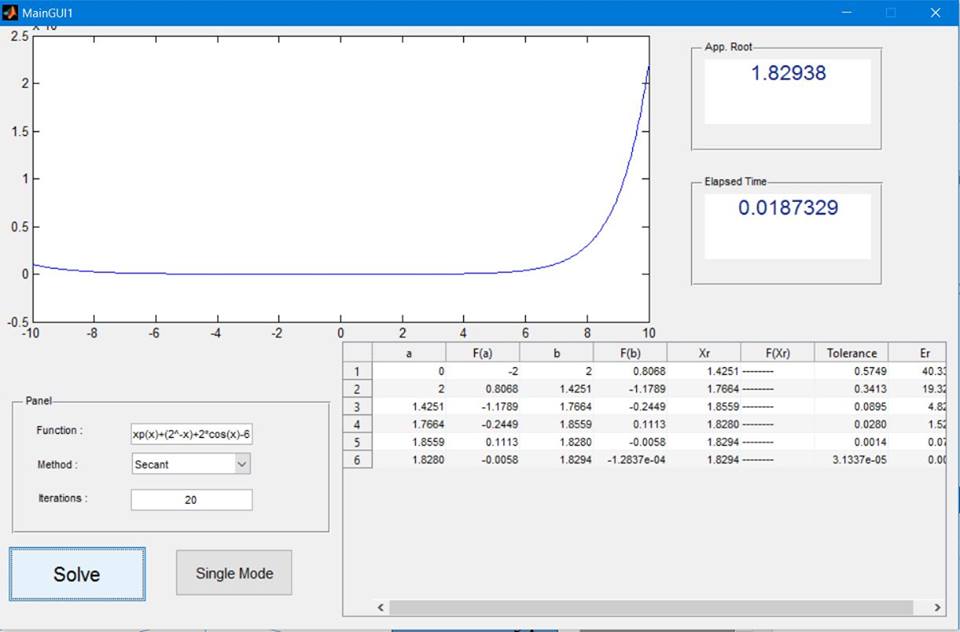
Vectors are also used in storing internal data and data being passed from function to main GUI.

**Examples:**

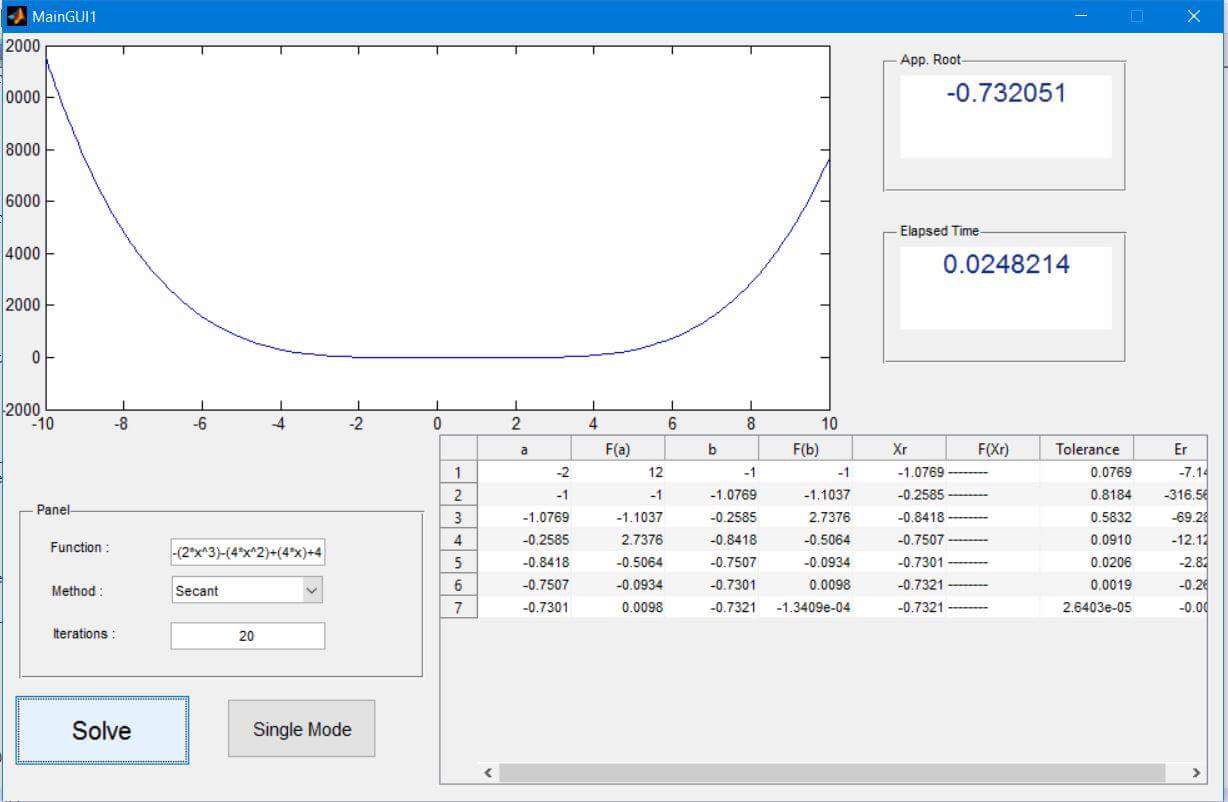
1-exp(-x)-x >>> i1=0 , i2=1 , tolerance= 0.001 .



2-exp(x)+(2^-x)+2\*cos(x)-6=0 , i1= -3 i2= 6 ,tolerance=0.0001.



3-(x^4)-(2\*x^3)-(4\*x^2)+(4\*x)+4 , i1= -2,-1 , tolerance=0.01



## Newton Method

Pseudocode:

**1-Normal Mode:**

* The array of data that includes the initial point and tolerance are passed to newton function along with the function and number of iterations.
* Initialize seven arrays which are: the initial point(the old point) xold[], the new point used xnew[], fxold[],fxnew[], iterations iters[], absoluter error Ees[], relative error Erel[].
* Start timer;
* Start Iteration loop
* Calculate xnew 🡪 xnew= xold-(f(xold)/fdiff(xold))

where f(xold) is the value of xold at the given function f(x).

and fdiff(xold) is the value of xold at the derivative of the given function f(x).

* Check value of the absolute error reached tolerance or iteration reached max value.

if true

break;

else

continue loop till tolerance or Maximum iterations reached;

* Stop timer.
* After function is done [iters,xnew, xold, Ees,Erel,fxold, fxnew,endtime]

are returned and displayed in GUI.

**2-Single Step Mode:**

* Similar to normal mode in executing, but differs in displaying.
* Iteration by iteration is being displayed on clicking Single Solve button, till all iteration are displayed

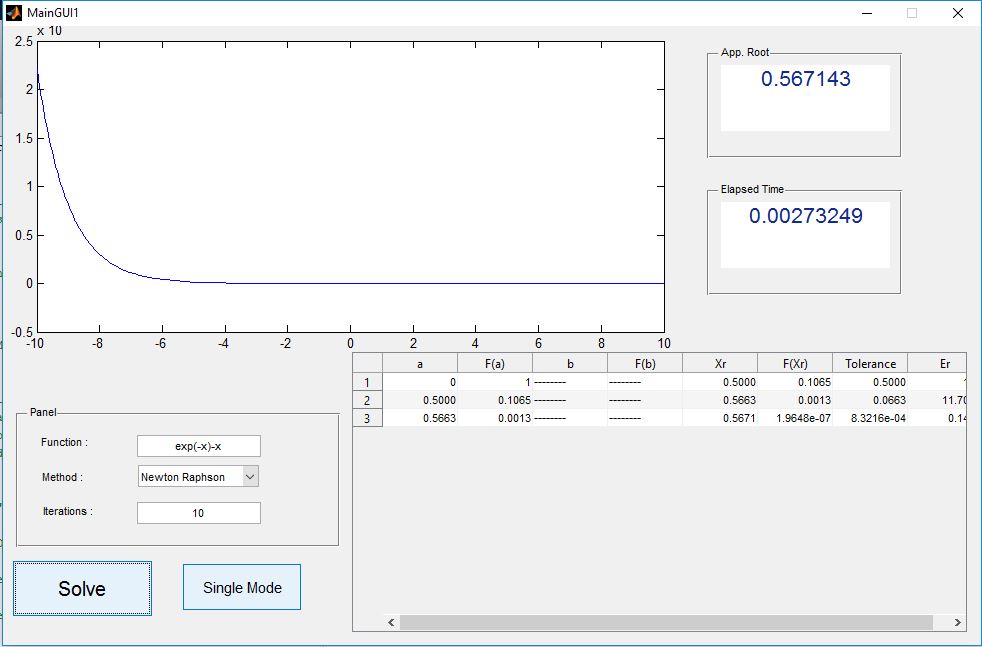
and by that a message appears to announce that tolerance reached and iterations are all displayed.

Data Structure USED:

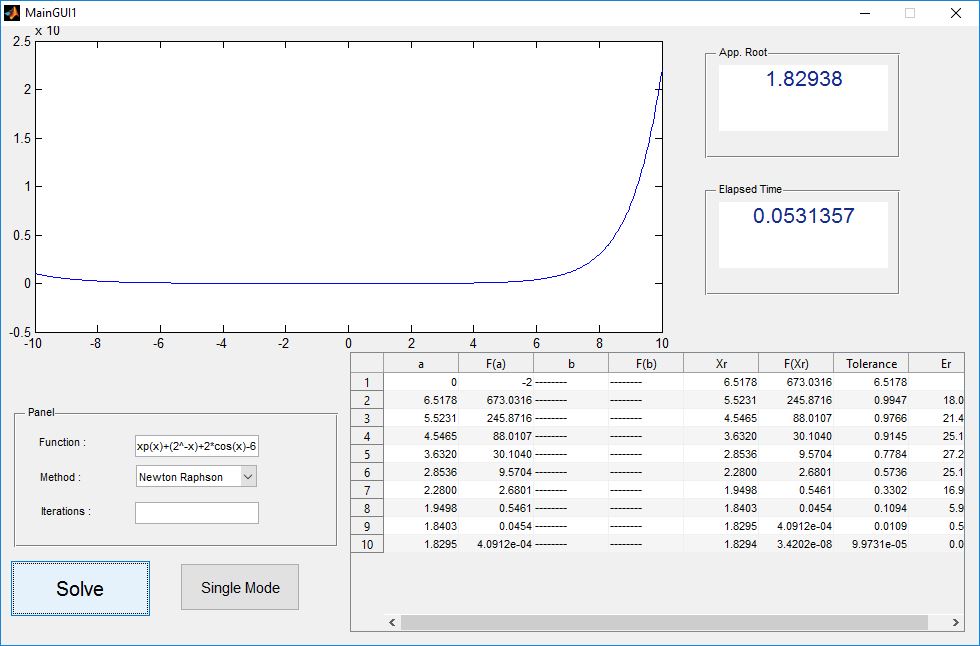
* Arrays are used to store data and to extract and deal with it, you should use cell2mat function to be able to store data.
* Arrays are easily used to extract the required data by indexing.
* Vectors are also used in storing internal data and data being passed from function to main GUI.

**Examples:**

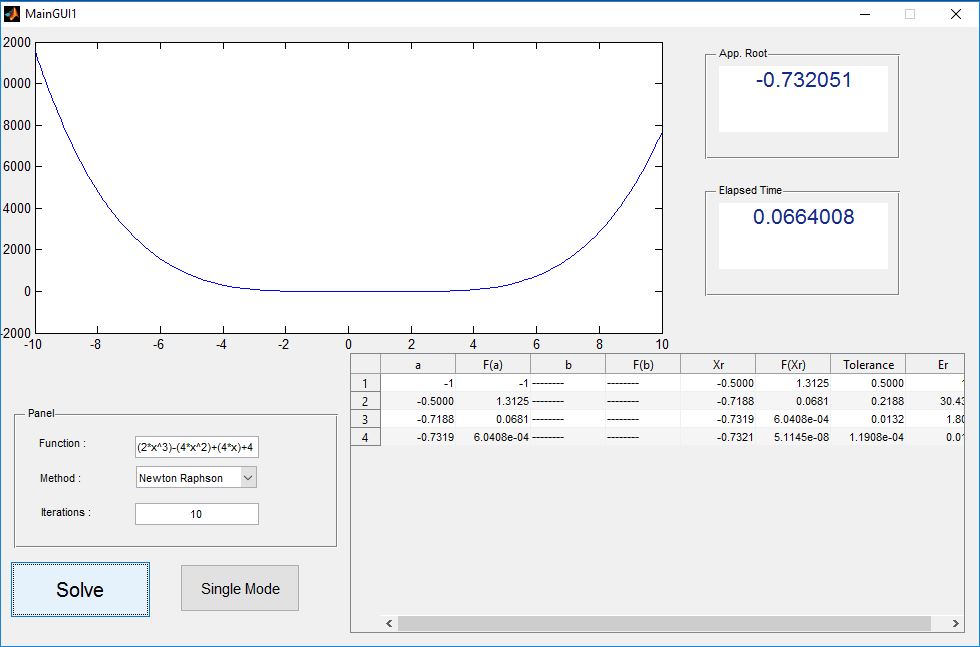
1-exp(-x)-x >>> initial point=0 , tolerance= 0.001 .



2-exp(x)+(2^-x)+2\*cos(x)-6=0 , initial point= 0 ,tolerance=0.0001.



3-(x^4)-(2\*x^3)-(4\*x^2)+(4\*x)+4 , i1= -1 , tolerance=0.01



## bisection Method

Pseudocode:

**1-Normal Mode:**

* Pass Data from GUI to Bisection Function

Bisection1(array,funct,iter)

Where array contains specified tolerance, upper bound and lower bound.

Funct is the function that we’re trying to solve for.

Iter is the number of max iterations specified by the user.

* Start timer
* if f(upper)\*f(lower) > 0 that means there is no x-axis crossing between these two points so there is no root in this interval
* If f(upper) is zero then the root is at the upper point. – stop timer and return
* If f(l) is zero then the root is at the lower point. – stop timer and return
* Otherwise we start doing iterations to determine the position of the root.

Xr= (u+l)/2

If f(xr)\*f(l) == 0 then we have reached the root .

f(xr)\*f(l) < 0 we set upper limit = xr and lower limit stays the same.

If f(xr)\*f(l) > 0 then lower limit = xr and upper limit stays the same.

* We repeat until tolerance is reached or max iterations are reached. Then we stop the timer and return.

**2-Single Step Mode:**

As normal Mode , but in displaying data,

iteration by iteration is being displayed on clicking Single Solve button, till all iteration are displayed

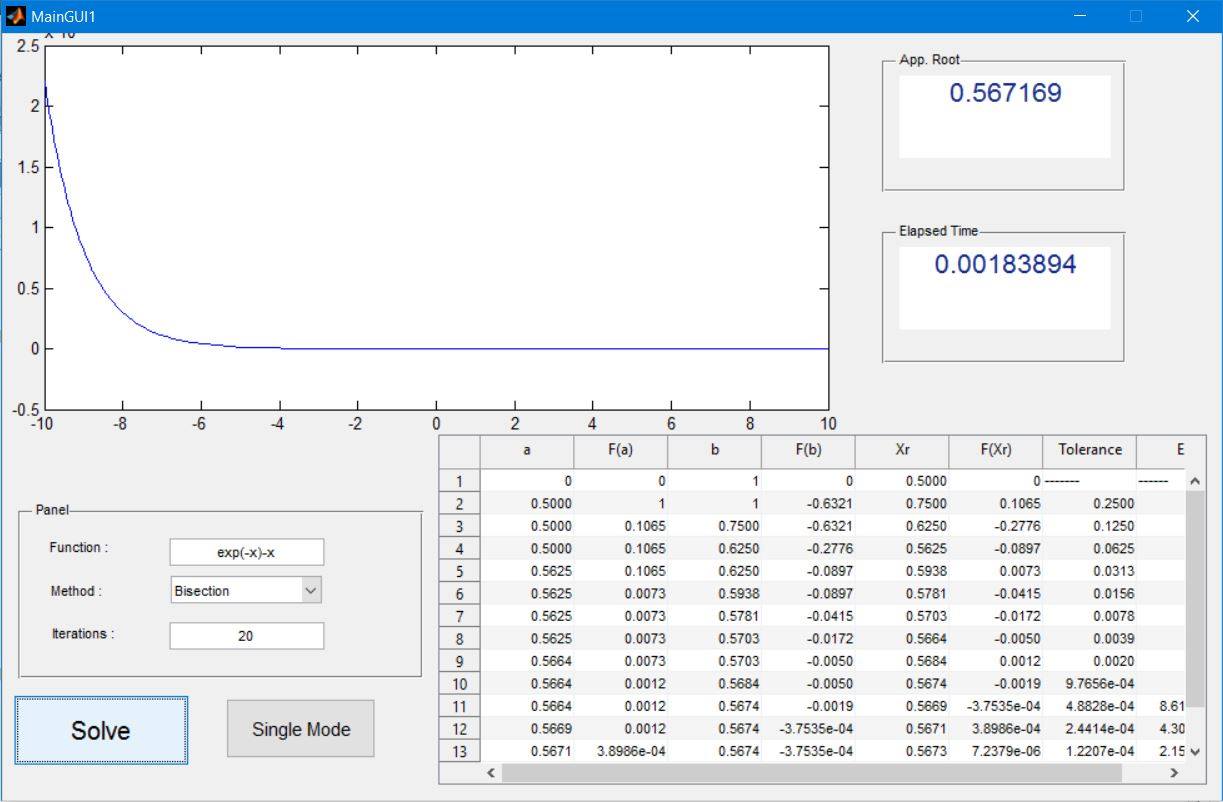
and by that a message appears to announce that tolerance reached and iterations are all displayed.

**Data Structure:**

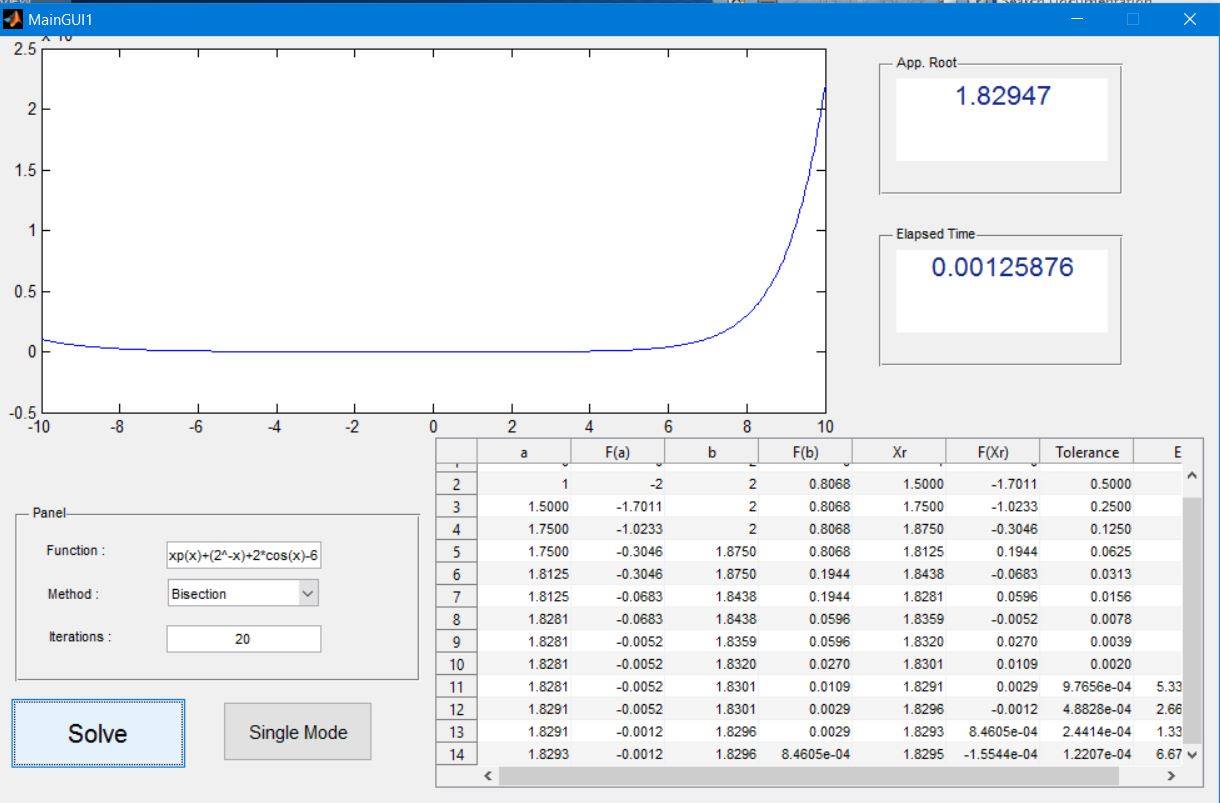
* arrays are used to store data and to extract and deal with it,you should use cell2mat function to be able to store data.
* arrays are easily used to extract the required data by indexing.
* Vectors are also used in storing internal data and data being passed from function to main GUI.

**Examples**

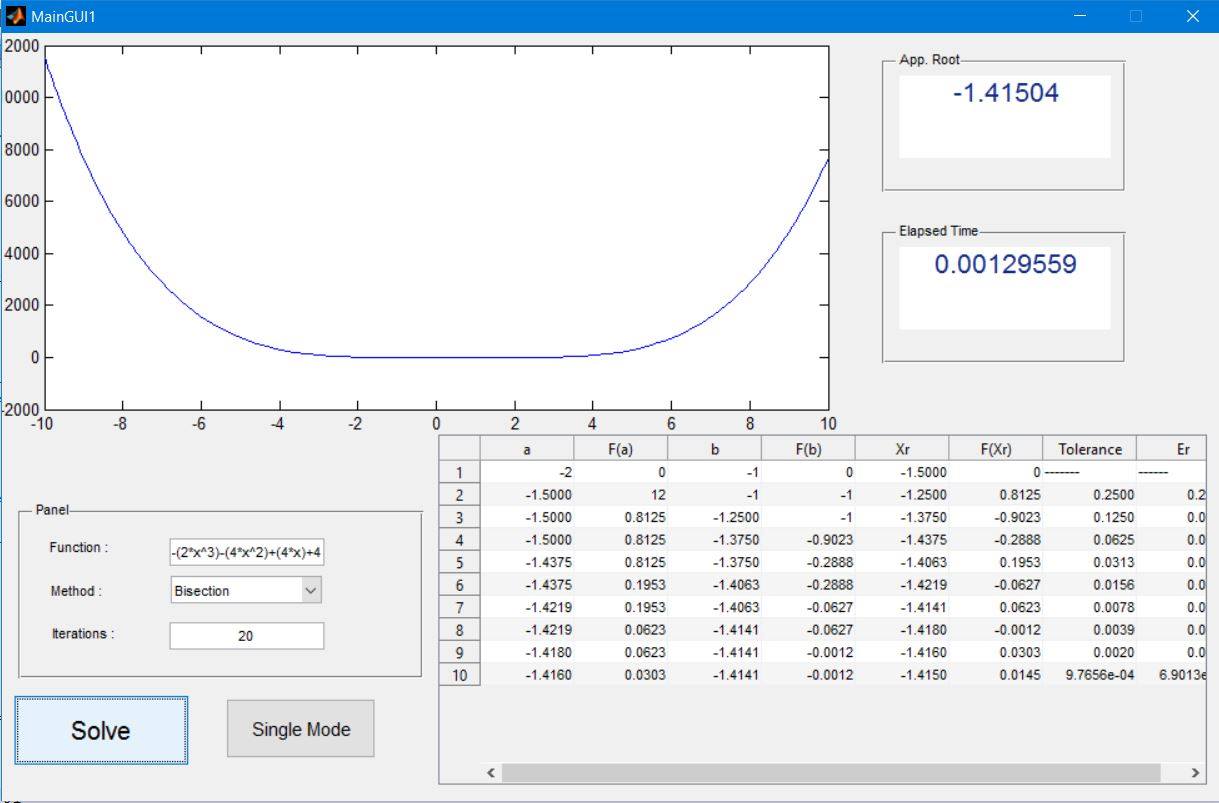
1-exp(-x)-x >>> i1=0 , i2=1 , tolerance= 0.001 .



2-exp(x)+(2^-x)+2\*cos(x)-6=0 , i1= -3 i2= 6 ,tolerance=0.0001.



3-(x^4)-(2\*x^3)-(4\*x^2)+(4\*x)+4 , i1= -2,-1 , tolerance=0.01



## false position Method

**Explanation**

The aim of the numerical analysis approximation methods is to solve equations which is not easy to be solved manually. By moving all of an equation’s terms to one side, we can get an equation that says: f(x) = 0, where f(x) is some function of the unknown variable “x”. That transforms the problem into one of finding the x-value at which f(x) = 0. That x-value is the root of this equation. The Bracketing methods start with two x-values (upper limit, lower limit), initially found by trial-and-error, at which f(x) has opposite signs. In other words: Two values (u,l) such that, f(u) is positive, and , f(l) is negative or vice versa. In that way, those two f(x) values can be said to “bracket” the root, because they’re on opposite sides of root. That's why bracketing method guarantee convergence to reach the root of the equation as it checks each iteration that the approximated root is encapsulated between a negative and a positive f(x). Although, bracketing methods execution time may be slower than other open methods but at least it guarantees the convergence unlike other methods as (Newton Raphson and Secant)

**Pseudo code**

* Function [ a,fa,b,fb,xr,fxr,tol,relerr,counter,endtime ] = False\_Position ( data\_array,funct) Inputs The user enters the function to be solved and the needed data such as in this case, the upper limit , lower limit , tolerance (to stop at if reached) and Number of maximum iterations the user wants .
* All these inputs are passed to the function in an array and the function is passed as string.

1. Convert the given string to a function by using str2func() , which is a built-in function.
2. Extract the needed data from the data\_array into variables, xu is upper limit xl is the lower limit N is number of iterations.
3. Start the timer by using tic function.
4. Check if f(xl) is equal to zero then lower limit the user chose is the real root of the function.
5. Check if f(xu) is equal to zero then upper limit the user chose is the real root of the function.
6. checks f(xl) \* f(xu) > 0 Then It returns an error because the real root will never be found between those two values.
7. checks f(xl) \* f(xu) < 0
8. Get the value of approximated root which is equal to ((xl\*f(xu))-(xu\*f(xl)))/(f(xu)-f(xl))
9. save all the values in arrays a(i) <---- saves the lower limit of each iteration. b(i) <---- saves the upper limit of each iteration. fa(i) <---- saves the value of function when x is equal to the lower limit of each iteration. fb(i) <---- saves the value of function when x is equal to the upper limit of each iteration. tol(i) <---- saves the calculated tolerance each iteration which is |current app. value - old app. value | relerr <---- saves the relative error in each iteration. xr(i) <---- saves the app. root of each iteration. fxr(i) <---- saves the value of function when x is equal to the xr(i) .
10. Checks f(xl) \* f(xr\_new) < 0 Then the coming upper limit is equal to the last app. root calculated.
11. Checks f(xu) \* f(xr\_new) < 0 Then the coming lower limit is equal to the last app. root calculated.
12. Checks abs(xr\_new - xr\_old) < tolerance Then we reached the needed approximated root and return all arrays back to the gui.
13. Repeat all steps from 8 to 12 at least N times, N is maximum number of iterations.

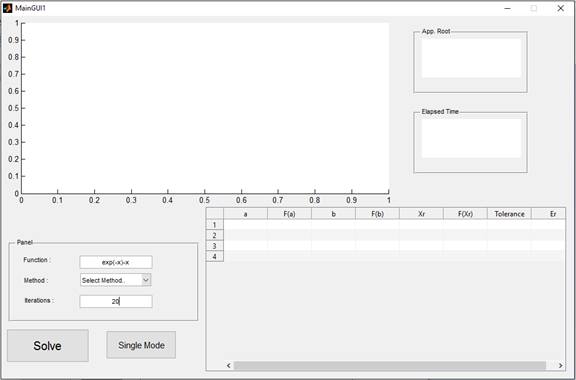
**Data Structure**:

* We used arrays to save each data we want to return all its values for each iteration. As first value at iteration number one is stored in array's element of index 1. On the other hand, In GUI we loop over the array and displays all its elements in the table.

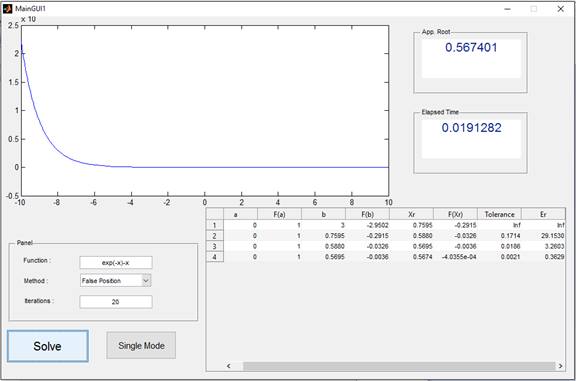
**Examples:**

Sample run to show the steps and results by using the equation , exp(-x)-x :

Enter the function and number of iterations and choose the method from the drop down menu :

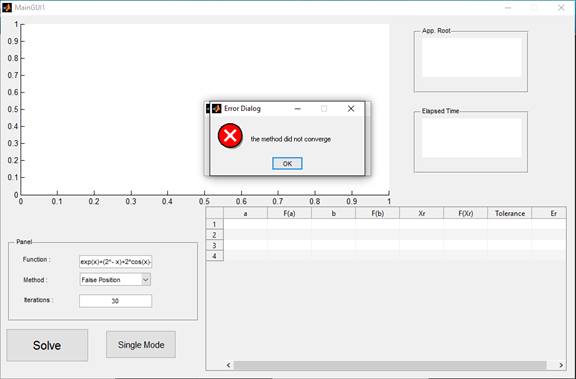


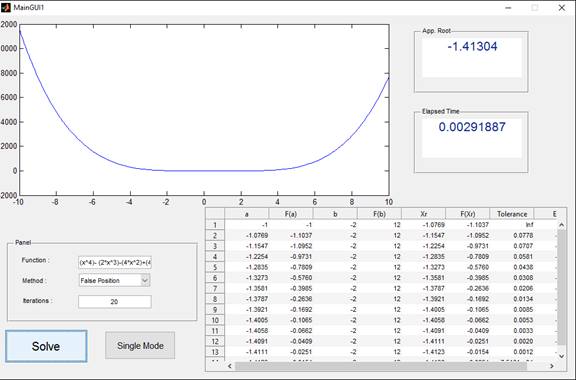
All the iterations will be displayed in the table as weel as the final elapsed time and the final approximated root :



* If you want to solve it as a single mode instead of clicking on solve , click on single mode and it will return one iteration each time single mode is clicked

Second example : exp(x)+(2^- x)+2\*cos(x)-6=0



Third example : f(x) = (x^4)- (2\*x^3)-(4\*x^2)+(4\*x)+4

## fixed point Method

**Explanation:**

* Convergence Analysis : Since we don't have two points to guarantee convergence , so we check convergence before starting the iterations by getting the differentiation of the g(x) and substitute by the initial point , if |g'(x)| < 1 then It will surely converge else it will diverge.

**Data Structure :**

We used arrays to save each data we want to return all it's values for each iteration. As first value at iteration number one is stored in array's element of index 1. On the other hand , In GUI we loop over the array and displays all it's elements in the table.

**Pseudo code:**

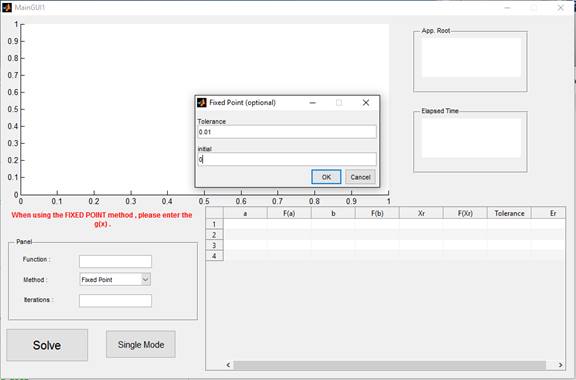
Function [ root\_xi , root\_xinew ,time\_taken ,counter ,state , ea , rel ] = fixed(array,funct) • Inputs The user enters the function to be solved and the needed data such as in this case, the initial point, tolerance (to stop at if reached) and Number of maximum iterations the user wants . All these inputs are passed to the function in an array and the function is passed as string.

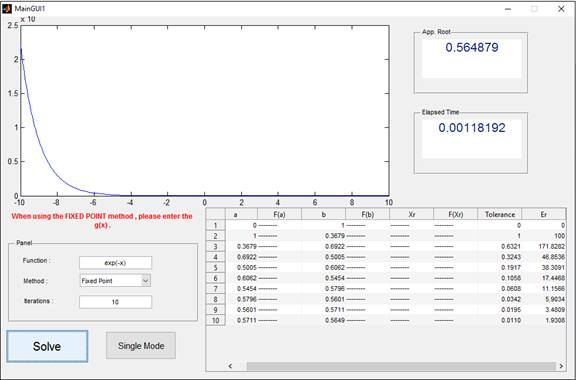
1. Convert the given string to a function by using str2func() , which is a built-in function, but the user must enter the equation in the form of g(x) = x
2. 2- Extract the needed data from the data\_array into variables, initial is the initial point. iter is number of iterations. tol is the tolerance.
3. get the differentiation of the equation and substitute by the initial given value and checks abs(G\_initial)>1 Then it will return a diverge state. but if the (G\_initial)<1 then it will complete.
4. Start the timer by using tic function.
5. Get the value of the new approximated root bu substituting in the given function by the previous app.root root\_xinew(i) = f(root\_xi(i))
6. Calculate the absolute error and the relative error and save all these data in arrays root\_xinew(i) <---- saves the new approximated root. root\_xi(i) <---- saves the previous approximated root. ea(i) <---- saves the absolute error each iteration. rel(i) <---- saves the relative error each iteration.
7. Checks abs(root\_xinew(i) - root\_xi(i)) < tol if it is true then return and get the total elapsed time.
8. Set the value of the previous app. root of next iteration equal to the current app. value.
9. Repeat all these steps from 5 to 8 at least iter times, since iter is the maximum number of iterations.

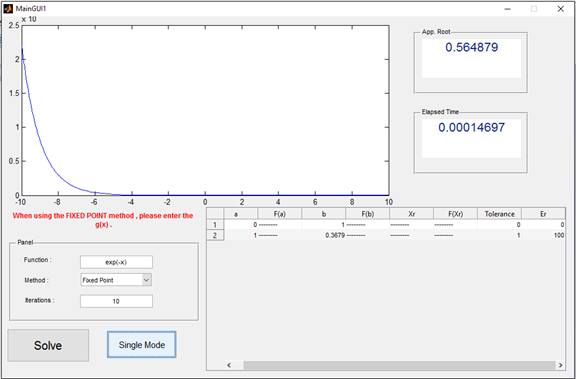
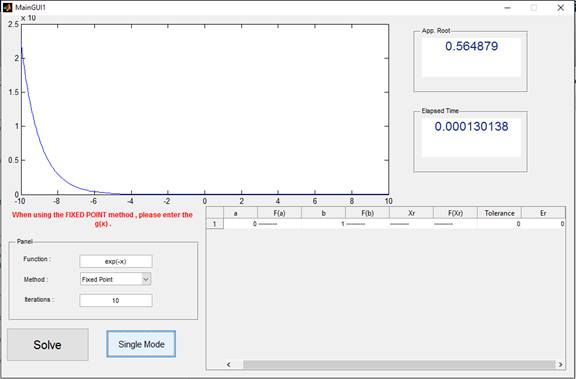
**Examples:**

f(x) = exp(-x)-x = 0 , but the user must enter it as x= g(x) where g(x) = exp(-x)

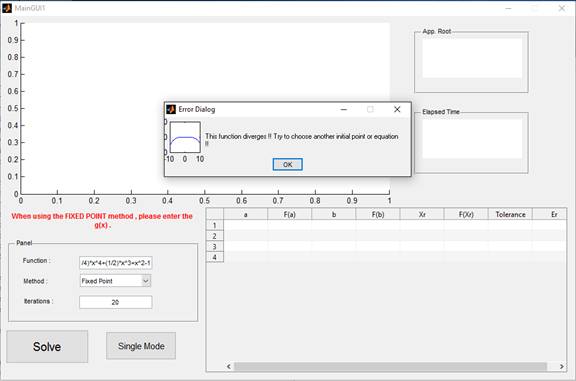
* choose the method used : when choosing fixed point method , A warning will appear reminds you to enter the g(x) form.



* enter the equation and iterations and solve. the value under a column is the value of xi and the one under b column is the xi+1
* If you want to solve it as a single mode instead of clicking on solve , click on
* Single mode and it will return one iteration each time single mode is clicked



Second example: f(x) = (x^4)- (2\*x^3)-(4\*x^2)+(4\*x)+4 , g(x) = (-1/4)\*x^4+(1/2)\*x^3+x^2-1 = x



Third example : g(x) = exp(x)+(2^- x)+2\*cos(x)-6

