Root Finder and Systems of Equations Solver

Developed by :
Haya Mohamed Swilam
Mohab Mosaad
Mohamed Ahmed DeifAllah
Rowan Mohamed Swilam
Youssef Zook

Reference

Overview

The Assignment is of two parts: Root Finder program where the user enters an equation to find out its solution by various methods, he can choose one of the methods or select them all ; such methods are : Bisection, False-Position, Fixed-Point, Newton-Raphson, Secant, Bierge-Vieta and General algorithm developed by the team. The program gets the nearest root to the initial point entered by the user.

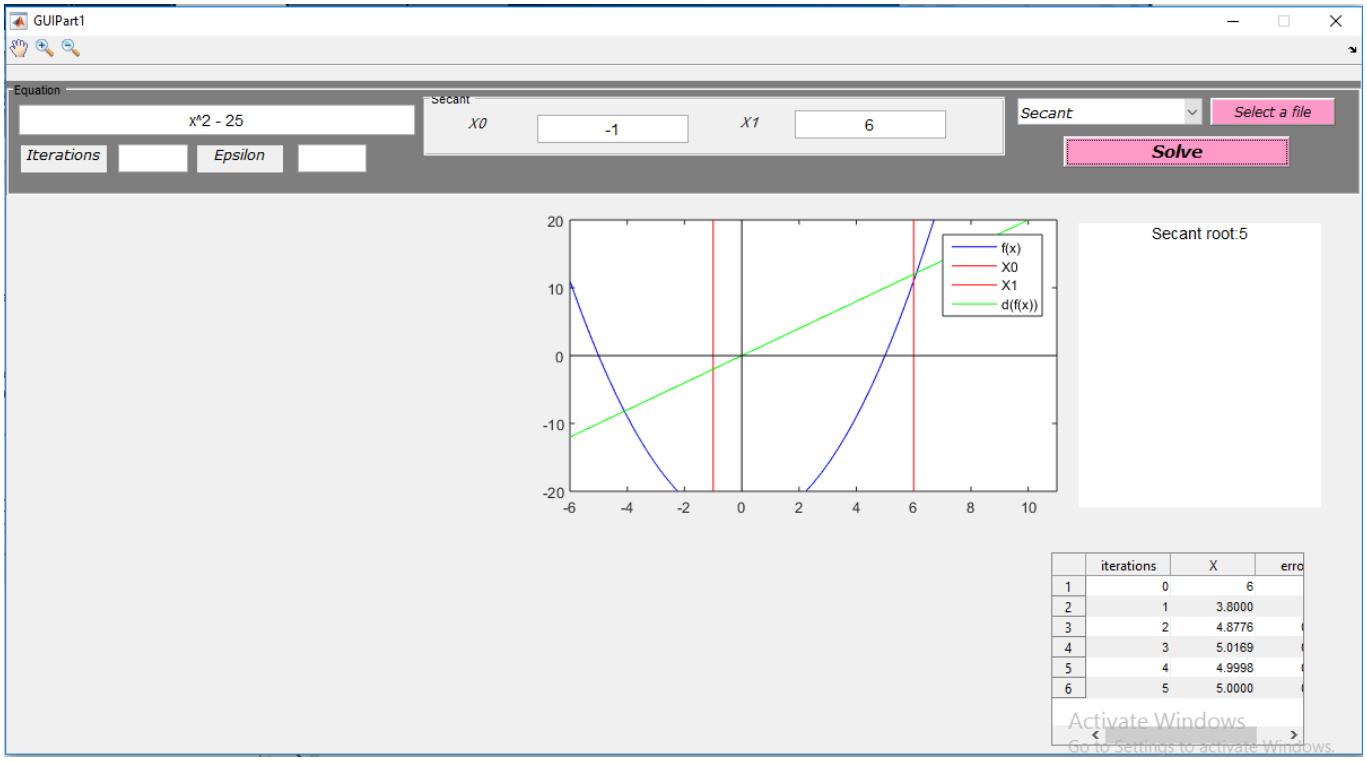
Solving Systems of Linear Equations where the user enters n numbers of equations in n unknowns and choose the method or select them all. These methods are : Gaussian-Elimination, LU-Decomposition, Gaussian-Jordan and Gauss-Seidel.

The aim is to compare and analyse the behaviour of the different numerical methods

The GUI is divided into two parts one for each part

GUI Part 1

Designed for the Root Finder Program. The user can enter input either through the GUI or by reading from a file. A dropdown list to choose the method, text boxes for the initial points, the max number of iterations and the tolerance. Where the max number of iterations and the tolerance are not necessarily added by the user; they have initial values 50 and 0.00001 respectively. A button to choose the desired file implemented with a file chooser and another to solve the equation. The Iterations, Roots and Errors are shown in a UITable and a graph to show the roots along with the number of iterations. The General Algorithm needs only the equation and it generates the initial points on its own.



A Detailed Analysis of the Methods Used

BISECTION METHOD:

- **PSEUDO CODE:**

```
function [xr] = bisection(l, u, f, maxit, es)
        if(f(l) * f(u) > 0)
                //There's no solution
        else if(f(l) == 0)
                xr = l; return;
        else if(f(u) == 0)
                xr = u; return;
        else
                xr = (l + u) / 2;
                if(f(l) * f(xr) < 0)
                        u = xr;
                else
                        l = xr;
                //end if
                for i = 1 to maxit
                        x = xr;
                        xr = (l + u) / 2;
                        if(abs((x - xr) / xr) < es) break;//end if
                //end for
        //end if
//end function
```

- **BEHAVIOR:**
  1) Choose two initial points, where it's guaranteed they have a root within. This can be checked by f(l) * f(u) < 0
  2) An estimate of the root can be given by:

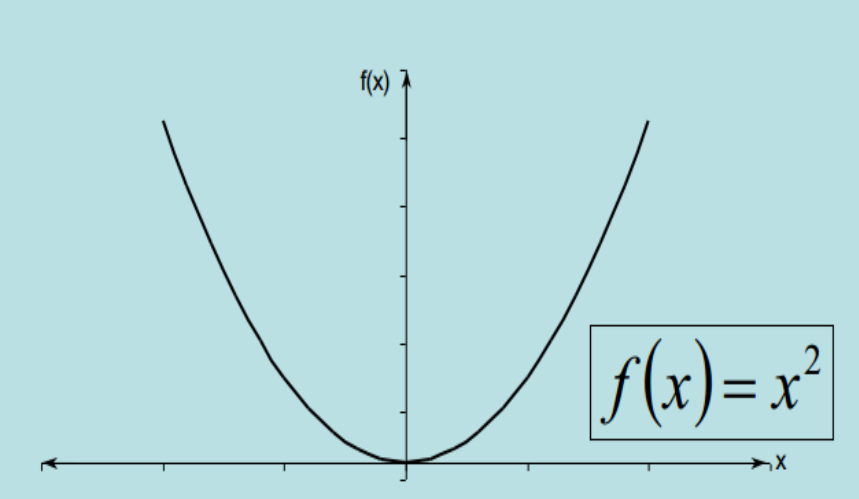
$$x_r = \frac{x_l + x_u}{2}$$

  3) Iterate over points, keeping the root inside the interval until he maximum number of iterations or reaching the error tolerance

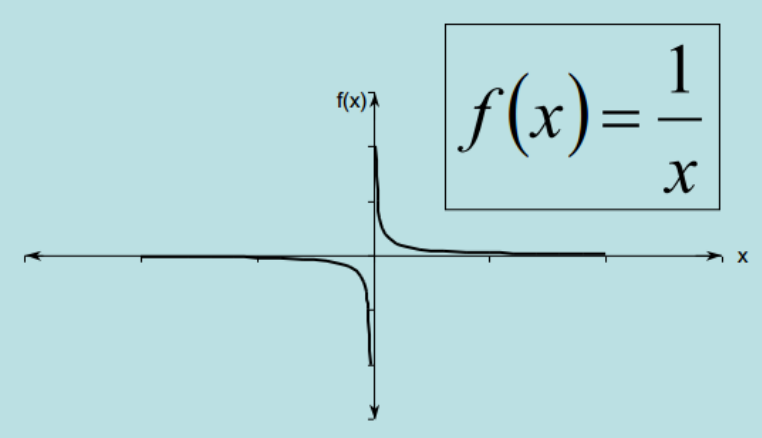
- **CONCLUSION:**
  1) It's possible to estimate the number of iterations needed to reach a specified error tolerance via that formula:
  2) Such a method always finds a root, and finds it fast.

$$k \geq \log_2\left(\left|\frac{L_0}{E_a}\right|\right)$$

- **MISBEHAVIOR:**
  1) "Bisection" method can't solve such case, where the curve tangents the x-axis only in a peak or bottom point, as shown:



2) Function changes sign but root does not exist.



FALSE-POSITION

- **PSEUDO CODE:**

```
function [root] = regulaFalsi(f, l, u, eps, iter)
        if(f(l) * f(u) > 0)
                //No solution
        //end if
        else
                root = (l * f(u) - u * f(l)) / (f(u) - f(l))
                if(f(u) * f(root) < 0)
                        l = root
                        //end if
                else
                        u = root
                i = 0;
                while(relative error > eps and i < iter)
                        i++
                        calculate the next root
                //end while
//end function
```

- **BEHAVIOR:**

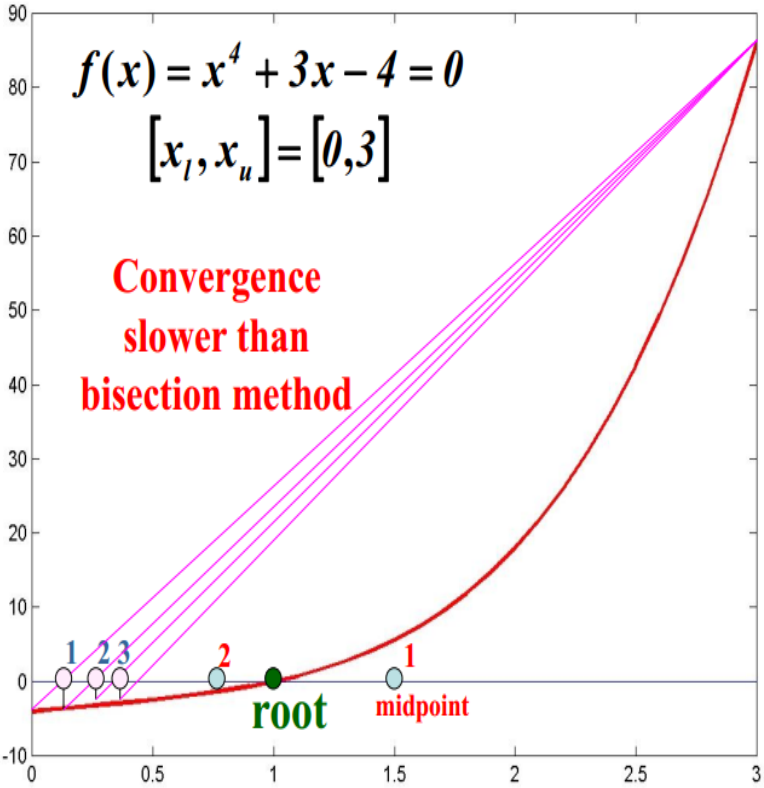
a) Find two initial points, such that definitely have a root within them.
b) Estimate the value of the root by the method formula.
c) You can exchange one of the two boundaries with the new point, in order to keep the root of the equation within the interval.

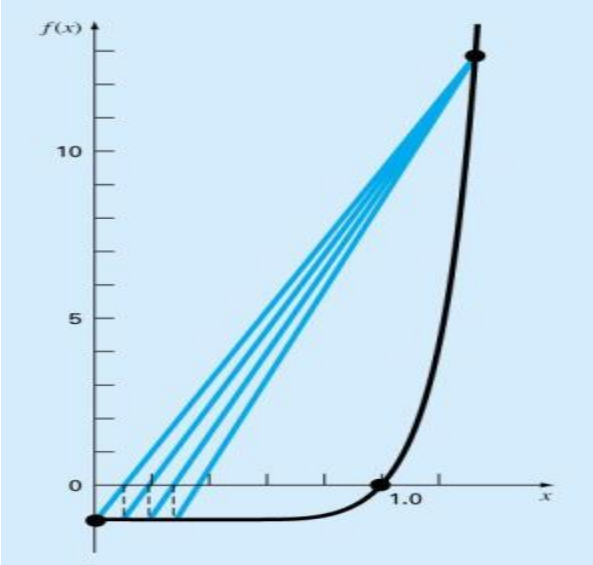
- **CONCLUSION:**

a) That method always guarantees to converge for the root in the interval.
b) In general, it can be faster than "Bisection method".

- **MISBEHAVIOR**

1) It has a very low rate of convergence if the function is very steep like in these two cases:





- **SUGGESTION:**

We can use the original formula, which is the formula of "Bisection method" in case we found that one of the bounds is stuck, or even applying "Bisection method" only once at first.

FIXED-POINT

- **PSEUDO CODE:**

```
starting with root x
  and function f(x)=0
  if(|f'(x)+1|>1)
      end (wont converge)
  else
  for(iltrations needed and error < tolerance)
      xOld=x
      x=f(x)+x
     error=|x-xOld|/|x|
```

- **BEHAVIOR**

a) starting w initial point
b) check convergence condition
c) Iterate till it's close to root as needed

- **MISBEHAVIOR**

a) After searching it appears that even if it passes the convergence condition it may not converge
b) In some function it takes a lot of iterations to converge

- **SUGGESTIONS**

a) you may change the function or use one of its applications such as Regula-Falsi

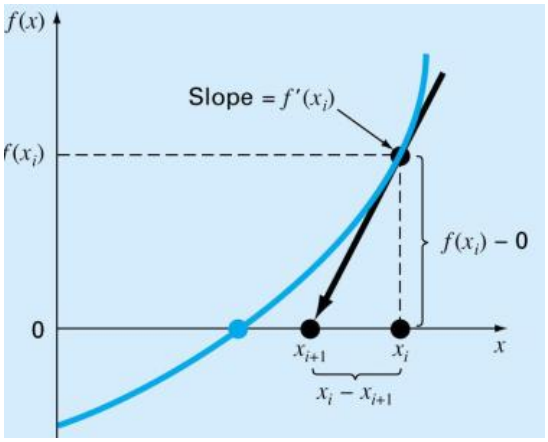
NEWTON-RAPHSON

- ## PSEUDO CODE

```
function [root] = newtonRaphson(f, x0, eps, iter)
        //calculate the derivative
        derivative = differentiate(f)
        if(derivative(x0) == 0)
                return initial point as the root
        //end if
        else
                root = x0 - f(x0) / derivative(x0)
                i = 0;
                while(derivative(root) != 0 and i < iter and relative error > eps)
                        i++
                        calculate the next root
                //end while
//end function
```

- ## BEHAVIOR

a) It make use of the slope of the function to predict the location of the root, as shown below:



b) Evaluate the derivative of the function.

c) Use an initial guess of the root to estimate the new value of the root as

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

d) Iterate until the absolute relative error is acceptable or the number of iterations has exceeded the maximum number of iterations allowed.

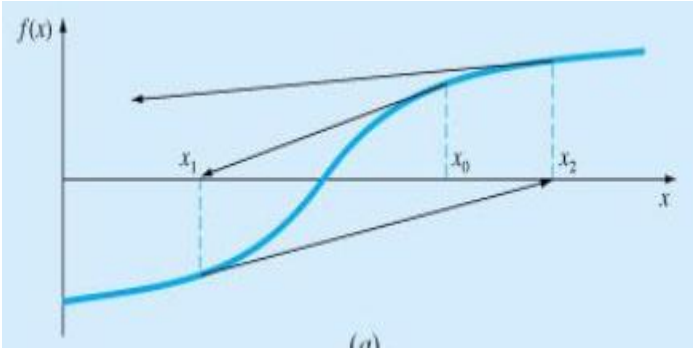
- **CONCLUSION**

e) "Newton-Rahpson method" converges quadratically (when it converges), except when the root has a multiplicity.

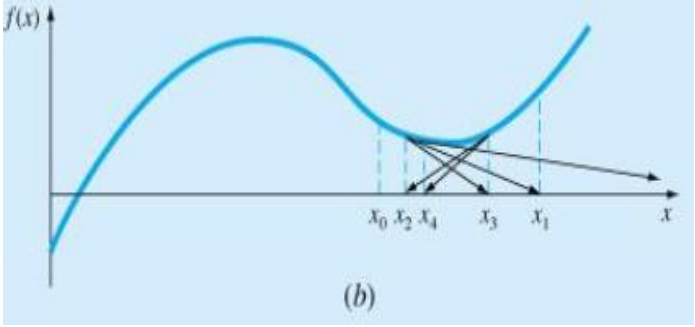
f) When the initial point is near to the root, it usually converges.

- **MISBEHAVIOR**

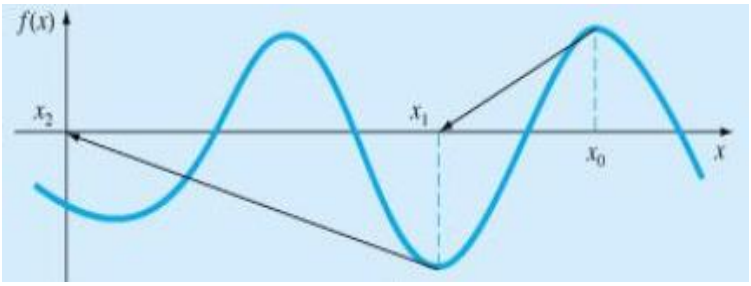
g) It diverges if there's an inflection point at the vicinity of the root.



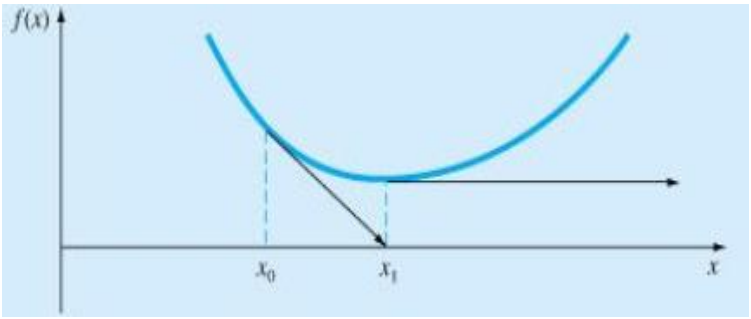
h) A local maximum or minimum causes oscillations.



i) It may jump from one location close to one root to a location that's several roots away.



j) Division by zero



- **SUGGESTIONS:**

a) Although there's no general convergence criteria for that method, but it can deal well according to good initial points, good knowledge of functions and graphical analysis, and good software that detects slow convergence or divergence.

b) In addition, you can use any of the modified methods, which behave with multiplicity of roots.

SECANT

- **PSEUDO CODE**

```
function root = secant(f, x0, x1, eps, maxit)
        for i = 1 to maxit
                root = x1 - f(x1) * (x0 - x1) / (f(x0) - f(x1))
                if(abs((root - x1) / x1) < eps)
                        break;
                //end if
                x0 = x1
                x1 = root
        //end for
//end function
```

- **BEHAVIOR**

Although "Newton-Raphson" method needs to compute the derivatives of the input function, this method approximate the derivatives by finite divided difference.

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

- **CONCLUSION**

a) That method can be considered a special case of "Regula-Falsi" method. In other words, They correspond to each other as described:

Both methods use the same expression to compute x_r.

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

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

b) Note that there's definitely a difference between the two methods, where the initial values are replaced by the new estimate.

c) It converges at faster than a linear rate, so that it is more rapidly.

- **MISBEHAVIOR**

It may not converge

- **SUGGESTIONS**
  - That method can be modified in such a way, in which the function requires only one initial point, and then the formula turns into:

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

  - where "delta" is a float number, that is a crucial factor in the performance of that method.

a) If it's too small, a subtractive cancellation can happen in the denominator.
b) Also, if it's too big, that method often becomes divergent.
c) If δ is selected properly, this method provides a good alternative for cases when developing two initial guess is inconvenient.

BIERGE-VIETA

- **PSEUDO CODE**

```
vector a contains coeff. Of the poly.
for(iltrations needed and error < tolerance)
    b[0]=c[0]=a[0]
    for i=1 to i = a size-1
        b=a+x * b[i-1]
    for i=1 to i = a size-2
        c=b+x * c[i-1]
```

if(b = 0)

break

else

x=x-b[b size]/c[c size]

- **BEHAVIOR**

a) choose initial point

b) iterate till reaching the desired tolerance or the root itself with Newton-Raphson equation

x=x-b[b size]/c[c size]

- **MISBEHAVIOR**

a. As Newton-Raphson it can take a lots of iterations to converge

b. If f'(x)=0 it won't converge

GENERAL-ALGORITHIM

- **PSEUDO CODE**

```
function [root, m] = generalAlgo(f, eps)
        root = modifiedNewton(f, eps);
        //check if it's really the root of the function
        //otherwise, there's a divergence happen, and then use modified secant one
        if(abs(f(root)) > eps)
                root = modifiedSecant(f, root, eps)
        //end if
        //get the multiplicity of the root
        m = multiplicity(f, root, eps)
//end function
```

```
function root = modifiedNewton(f, eps)
        generate a random initial point x
        //make sure it doesn't make the denominator equal 0
        get the first and second derivatives of f(x)
        root = x - f(x) * f'(x) / ([f'(x)]^2 - f(x) * f''(x))
        while(denominator != 0 && absolute relative error > eps)
                x = root
                root = x - f(x) * f'(x) / ([f'(x)]^2 - f(x) * f''(x))
        //end while
//end function

function root = modifiedSecant(f, x, eps)
        generate a random delta del
        //make sure it doesn't make the denominator equal 0
        root = x - del * x * f(x) / (f(x + del * x) - f(x))
        while(denominator != 0 && absolute relative error > eps)
                x = root
                root = x - f(x) * f'(x) / ([f'(x)]^2 - f(x) * f''(x))
        //end while
//end function

function m = multiplicity(f, root, eps)
         initialize m = 1
         //differentiate f
         f = differentiate(f)
         while(abs(f(root)) < eps)
                  m = m + 1
                  f = differentiate(f)
         //end while
//end function
```

- **BEHAVIOR**

a) First, that algorithm makes use of "modified Newton-Raphson 2" method, as it detects roots with multiplicity.

b) In such a case, where that method can't get the root with accepted tolerance, the algorithm turns its behavior to be "modified Secant" method, where its initial guess point is the last point from the previous method.

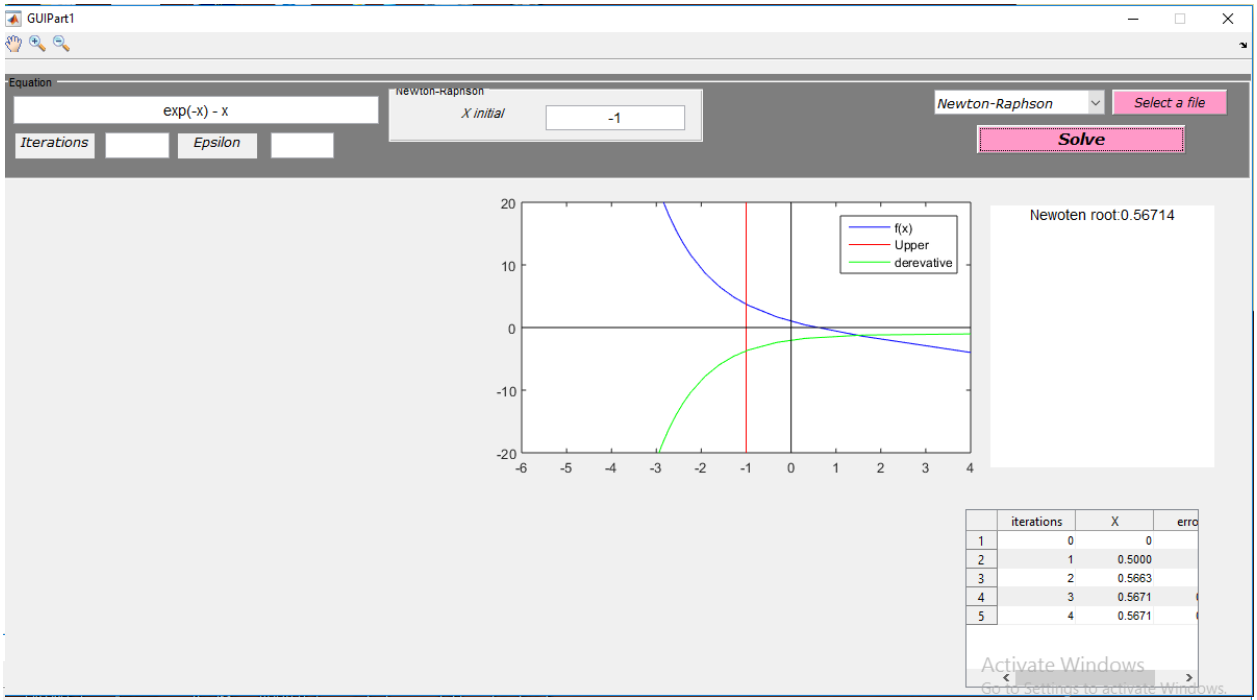
c) Now, it's the time to get the multiplicity of the result root, and that fact shown below can be used in the implementation:

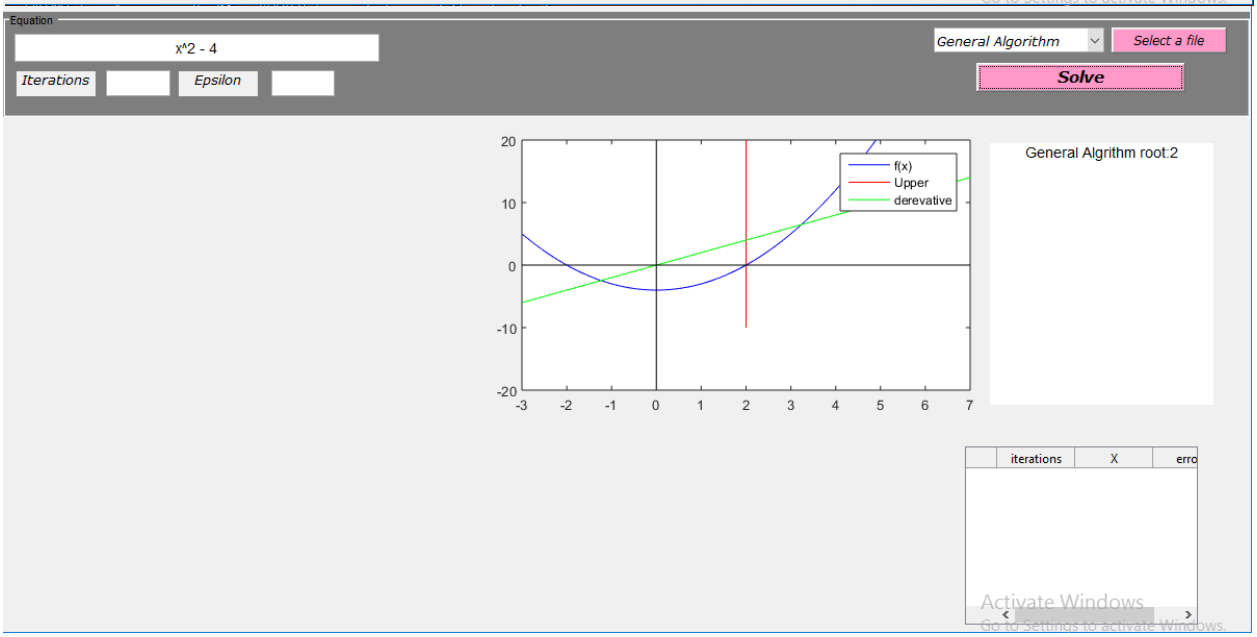
If a function has a zero at ***x=r*** with multiplicity ***m*** then the function and its first ***(m-1)*** derivatives are zero at ***x=r*** *and the* m^{th} *derivative at* ***r*** *is not zero.*

- **CONCLUSION**

We can't say that this method is the best of the best, but at least it's a combination of two very efficient methods. That can indicate to a little number of pitfalls if exist.

SAMPLE-RUNS:





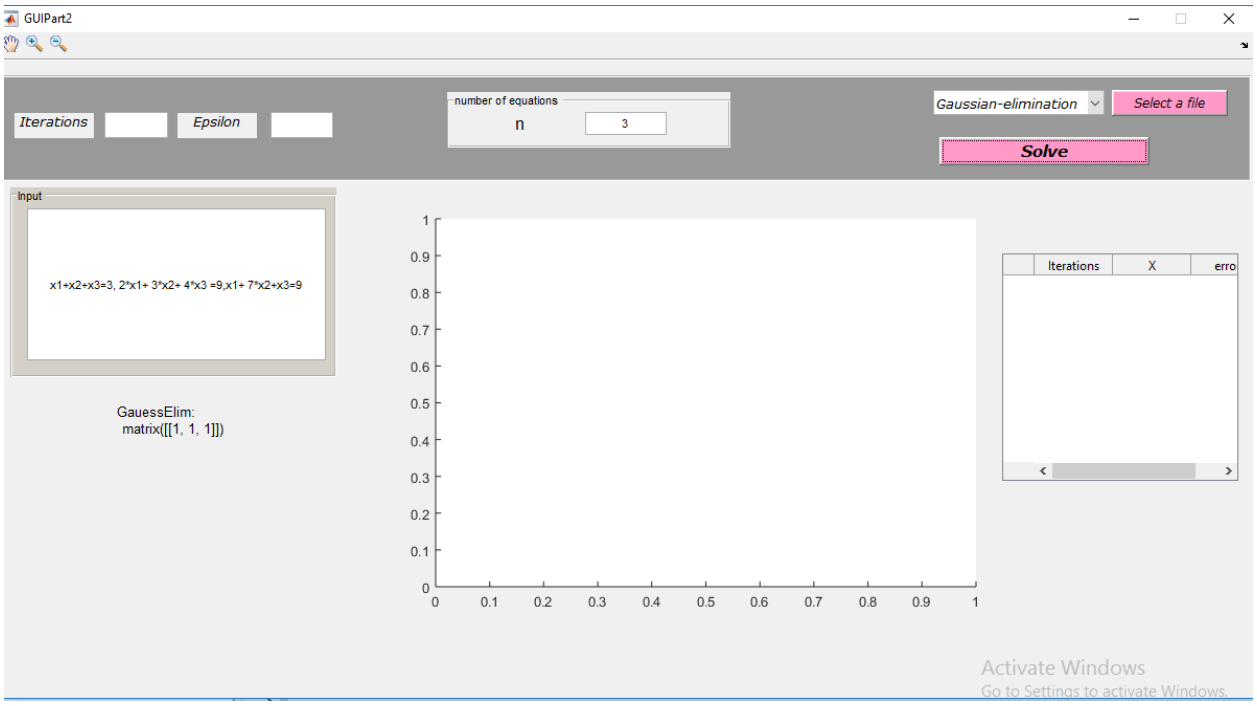
GUI Part 2

Designed for the Solving Systems of Linear Equations Program. The user can also enter input either through the GUI or by reading from a file. A dropdown list to choose the method, text boxes for the parameters, the max number of iterations and the tolerance. Where the max number of iterations and the tolerance are not necessarily added by the user; they have initial values 50 and 0.00001 respectively.

A button to choose the desired file implemented with a file chooser and another to solve the equation.

The Iterations, Roots and Errors are shown in a UITable.

A graph to compare the methods showing the roots with the number of Iterations.



A Detailed Analysis of the Methods Used

GAUSSIAN-ELEMINATION

- ### PSEUDO CODE

```
function [x] = GaussElimination(a, b)
        //Forward elimination
        for k = 1 to n-1
                for i = k+1 to n
                        factor = a[i][k] / a[k][k]
                        for j = k+1 to n
                                a[i][j] = a[i][j] - factor * a[k][j]
                        //end for
                        b[i] = b[i] - factor * b[k]
                //end for
        //end for
        //Back substitution
        x[n] = b[n] / a[n][n]
        for i = n-1 downto 1
                sum = 0
                for j = i+1 to n
                        sum = sum + a[i][j] * x[j]
                //end for
                x[i] = (b[i] - sum) / a[i][i]
        //end for
//end function
```

- ### BEHAVIOR

  The main idea of the method is to reduce the system of equations to an upper triangular matrix as follows:

$$\begin{aligned} a_{11}x_1 + a_{12}x_2 + a_{13}x_3 &= b_1 \\ a_{21}x_1 + a_{22}x_2 + a_{23}x_3 &= b_2 \\ a_{31}x_1 + a_{32}x_2 + a_{33}x_3 &= b_3 \end{aligned} \quad \Rightarrow \quad \begin{aligned} a_{11}x_1 + a_{12}x_2 + a_{13}x_3 &= b_1 \\ a'_{22}x_2 + a'_{23}x_3 &= b'_2 \\ a''_{33}x_3 &= b''_3 \end{aligned}$$

  a) Forward elimination: To remove X_K from all subsequent equations in the K_{TH} iteration.

$$\left[\begin{array}{ccc|c} a_{11} & a_{12} & a_{13} & b_1 \\ a_{21} & a_{22} & a_{23} & b_2 \\ a_{31} & a_{32} & a_{33} & b_3 \end{array}\right]$$

$$\Downarrow$$

$$\left[\begin{array}{ccc|c} a_{11} & a_{12} & a_{13} & b_1 \\ & a'_{22} & a'_{23} & b'_2 \\ & & a''_{33} & b''_3 \end{array}\right]$$

b) Back substitution, which exactly means the process, described below:

$$x_3 = b''_3 / a''_{33}$$

$$x_2 = (b'_2 - a'_{23} x_3) / a'_{22}$$

$$x_1 = (b_1 - a_{12} x_2 - a_{13} x_3) / a_{11}$$

- **CONCLUSION**

a) It's obvious from the pseudo code that it costs O (n^3) in elimination step, but O (n^2) in back substitution step.

- **MISBEHAVIOR**

a) Division by zero in both main steps.

b) Because computers carry only a limited number of significant figures, round-off errors will occur and they will propagate from one iteration to the next one. In general, that problem gets more sensitive when too many equations are to be solved together.

- **SUGGESTIONS**

a) For division by zero, pivoting strategy must be used, where rows are swapped to prevent that case.
b) Always use double-precision arithmetic, because they're more accurate.
c) Check the results if they're correct by substituting in the original system of equations.

➢ Scaling is used to reduce round-off errors and improve accuracy. Note that, scaling is to make all coefficients in a row less than or equal 1

LU-DECOMPOSITION

- **PSEUDO CODE**

```
// Assume arrays start with index 1 instead of 0.
// a: Coef. of matrix A; 2-D array. Upon successful
// completion, it contains the coefficients of
// both L and U.
// b: Coef. of vector b; 1-D array
// n: Dimension of the system of equations
// x: Coef. of vector x (to store the solution)
// tol: Tolerance; smallest possible scaled
// pivot allowed.
// er: Pass back -1 if matrix is singular.
// (Reference var.)
LUDecomp(a, b, n, x, tol, er) {
        Declare s[n] // An n-element array for storing scaling factors
        Declare o[n] // Use as indexes to pivot rows.
        // oi or o(i) stores row number of the ith pivot row.
        er = 0
        Decompose(a, n, tol, o, s, er)
        if (er != -1)
                Substitute(a, o, n, b, x)
        //end if
} //end function
```

```
Decompose(a, n, tol, o, s, er) {
	for i = 1 to n { // Find scaling factors
		o[i] = i
		s[i] = abs(a[i,1])
		for j = 2 to n
			if (abs(a[i,j]) > s[i])
				s[i] = abs(a[i,j])
			//end if
		//end inner for
	//end outer for
	}
	for k = 1 to n-1 {
		Pivot(a, o, s, n, k) // Locate the kth pivot row
		// Check for singular or near-singular cases
		if (abs(a[o[k],k]) / s[o[k]]) < tol) {
			er = -1
			return
		}
		for i = k+1 to n {
			factor = a[o[i],k] / a[o[k],k]
			// Instead of storing the factors in another matrix (2D array) L, We reuse the space in A to store the
			//coefficients of L.
			a[o[i],k] = factor
			// Eliminate the coefficients at column j in the subsequent rows
			for j = k+1 to n
				a[o[i],j] = a[o[i],j] - factor * a[o[k],j]
		}
	} // end of "for k" loop from previous page
	// Check for singular or near-singular cases
	if (abs(a[o[n],n]) / s[o[n]]) < tol)
	er = -1
}//end function
```

```
Pivot(a, o, s, n, k) {
	// Find the largest scaled coefficient in column k
	p = k // p is the index to the pivot row
	big = abs(a[o[k],k]) / s[o[k]])
	for i = k+1 to n {
		dummy = abs(a[o[i],k] / s[o(i)])
		if (dummy > big) {
			big = dummy
			p = i
		}//end if
	}//end for
	// Swap row k with the pivot row by swapping the indices. The actual rows remain unchanged
	dummy = o[p]
	o[p] = o[k]
	o[k] = dummy
}//end function
```

```
Substitute(a, o, n, b, x) {
        Declare y[n]
        y[o[1]] = b[o[1]]
        for i = 2 to n {
                sum = b[o[i]]
                for j = 1 to i-1
                        sum = sum – a[o[i],j] * y[o[j]]
                //end inner for
                y[o[i]] = sum
        }//end outer for
        x[n] = y[o[n]] / a[o[n],n]
        for i = 1 to n - 1 {
                sum = 0
                for j = n - i + 1 to n
                        sum = sum + a[o[n - i], j] * x[j]
                //end inner for
                x[n - i] = (y[o[n - i]] – sum) / a[o[n - i], n - i]
        }//end outer for
}//end function
```

- **BEHAVIOR**

  You have to solve a system of equations has the form: Ax = b

a) Decompose [A] into [L] & [U], where LU = A
b) Solve LUx = b using forward and back substitution.
c) Assume that Ux = y, and then solve Ly = b for y by forward elimination.
d) Then solve Ux = y for x by back substitution.

- **CONCLUSION**

a) It's obvious from the pseudo code that such a method costs O (n^3) to compute both [L] & [U], and costs O (n^2) to apply forward elimination & back substitution, what makes it as efficient as "Gauss elimination" method.

- **MISBEHAVIOR**

a) It may result in an undefined behavior when dividing by zero at applying the process of pivoting.
b) Swapping rows may reorder the elements in [x]

- **SUGGESTIONS:**

a) To remember how the rows were swapped, we can introduce an array O[] to store the indices of the rows
b) Concerning the space taken by the implementation, [L] & [U] can be stored in one matrix, where L takes the lower triangle, and U takes the upper one of the matrix.

GAUSSIAN-JORDAN

- **PSEUDO CODE**

```
m is the matrix
  for i=1:1:size(m,1)
        replace(m,i,max(m,i))
        if(m(i,i)=0)
            flag = 1;
        for all elements in row r and i in m
            m(i,:)=m(i,:)/m(i,i);
        for r=1:1:size(m,1)
            if(r  not equal i)
                a=m(r,i);
        for all elements in row r and i in m
            m(r)=m(r)-m(i)*a;
       if(flag==0)
        the solution is found in last column
  else
        no solution
```

- **BEHAVIOR**

a) Elimination is applied to all equations(excluding the pivot equation) instead of just the subsequent equations.
b) All rows are normalized by dividing them by their pivot elements.
c) No back substitution is required.

- **CONCLUSION**

a) Almost 50% more arithmetic operations than Gaussian elimination
b) Gauss Jordan (GJ) Elimination is preferred when the inverse of a matrix is required. [A | I]
c) Apply GJ elimination to convert A into an identity matrix. [I | A^{-1}]

- **MISBEHAVIOR**

a) Division by zero
b) Round off error
c) ill conditioned systems.

- **SUGGESTIONS**

a) To avoid division by zero use pivoting
b) To reduce round off error use pivoting and scaling

GAUSS-SEIDEL

- ## PSEUDO CODE

```
let A the matrix
and B its answers
for r=1 to maxit
    max = 0;
    for i=1 to na
       s=0;
       for j=1 to ma
           if j not equal i
                s = s + (A(i,j)* x(j));
        t=(B(i)-s)/A(i,i);
        e = t-x(i))/t;
    if e > max
    max = e;
    x(i)=t;
    if max < ea
        break;
    iter = r
    if iter > maxit
      NO root found
```

- ## BEHAVIOR & CONCLUSION

a) It may not converge or it converges very slowly.
b) If the coefficient matrix A is Diagonally Dominant Gauss-Seidel is guaranteed to converge.
    - Diagonally dominant : The coefficient on the diagonal must be at least equal to the sum of the other coefficients in that row and at least one row with a diagonal coefficient greater than the sum of

the other coefficients in that row

c) This is not a necessary condition, i.e. the system may still have a chance to converge even if A is not diagonally dominant.

- **MISBEHAVIOR**

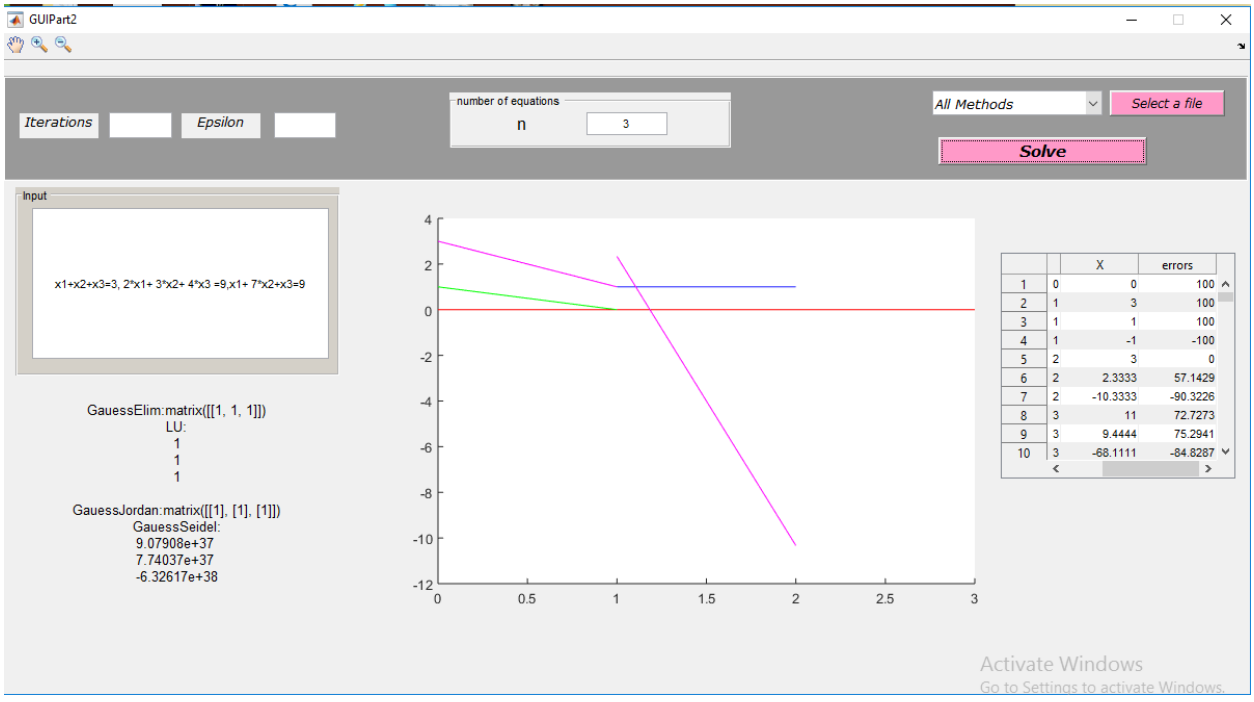
a) Even though done correctly, the answer is not converging to the correct answer

b) Not all systems of equations will converge.

- **SUGGESTIONS**

To make sure whether it converges or not we can check the above condition. If not satisfied check to see if rearranging the equations can form a diagonally dominant matrix.

SAMPLE-RUN



Problematic Functions

PART 1

a) X^2 - 4 = 0 at initial point = 0

using NEWTON-RAPHSON Method , As it is division by zero

PART 2

$$x_1 + x_2 + x_3 = 3$$

$$2x_1 + 3x_2 + 4x_3 = 9$$

$$x_1 + 7x_2 + x_3 = 9$$

GAUSS SEIDEL METHOD DIVERGES

As this system of equations can't be rearranged to have a diagonally dominant coefficient matrix.