

**Faculty of Engineering**  
**Alexandria University**  
**CSED 2024**  
**Numerical**  
**CSE 211**

# **Report**

## **Numerical Project**

### **Phase 2**

Aliaa Fathy Elsayed  
19016032

Aliaa Ibrahim Ahmed  
19016031

Bassant Yasser Salah  
19017262

Nada Mohamed Ibrahim  
19016782

Toka Ashraf Abo elwafa  
19015539

Equation Solver

Screen Capture Pro

*Solve Equation*

Method: Bisection

Equations:

Variables:

Output:

Run Time:

Plot Set Attributes Solve Clear

---

## Table of contents:

1. Brief Description.
2. Flowchart or pseudo-code for each method:
  - a) Bisection Method
  - b) False Position.
  - c) Fixed point.
  - d) Newton-Raphson.
  - e) Secant Method.
3. Sample runs for each method.
4. Comparison between different methods (time complexity , convergence, best and approximate errors).
5. Data structure used.
6. Link to the demo video.

## 1. Brief Description:

It is a calculator that helps the user to solve non-linear (find their roots) equations with 5 different methods which are (Bisection Method- False position- Fixed point- Newton Raphson- Secant Method).

---

## 2. Flowchart or pseudo-code for each method:

### -pseudo-code

#### a) Bisection Method:

```
Function bisection(double xu, double xl, String equation, String condition, double stop,
    int precision) {
    p ← "#." + "#".repeat(precision);
    DecimalFormat df ← new DecimalFormat(p);
    EvaluateExpression evaluate ← new EvaluateExpression();
    xr ← 0;
    fxr ← evaluate.eval(equation, xu);
    fxl ← evaluate.eval(equation, xl);
    ea ← 10;
    i ← 0;
    if (fxr * fxl is less than 0 && xu is greater than xl) {
        if (condition is equals to ("Number of Iterations")) {
            while (i is less than stop) {
                xr ← (xu + xl) / 2;
                fxr ← evaluate.eval(equation, xr);
                fxl ← evaluate.eval(equation, xl);
                if (fxr * fxl is less than 0) {
                    xu ← Double.parseDouble(df.format(xr));
                    End if;
                } else if (fxr * fxl is greater than 0) {
                    xl ← Double.parseDouble(df.format(xr));
                    End elseif;
                }
            }
        } else if (fxl * fxr is equal to 0) {
            Break;
            End elseif;
        }
        i ← i+1;
    }
    } else {
        while (ea is greater than stop) {
```

```

    xr  $\leftarrow$  (xu + xl) / 2;
    fxr  $\leftarrow$  evaluate.eval(equation, xr);
    fxl  $\leftarrow$  evaluate.eval(equation, xl);
    if (fxr * fxl is less than 0) {
        xu  $\leftarrow$  Double.parseDouble(df.format(xr));
    } else if (fxr * fxl is greater than 0) {
        xl  $\leftarrow$  Double.parseDouble(df.format(xr));
    } else if (fxl * fxr is equal to 0) {
        break;
    }
    ea  $\leftarrow$  Math.abs((xu - xl) / xl);
    l  $\leftarrow$  i+1;
    End while;
}
}
return df.format(xr);
}
else {
    return "This function has no Roots in this interval";
    End else;
}
}

```

**b) False Position:**

```

max  $\leftarrow$  (int) Math.pow(10, 6);
mid  $\leftarrow$  0;
point  $\leftarrow$  0;
function False_position(String equation, String choice, double value, double
lower, double upper, int significant) {
    Point  $\leftarrow$  lower;
    Evaluation evaluate  $\leftarrow$  new Evaluation();
    iteration  $\leftarrow$  0;
    Eps  $\leftarrow$  0;
    Plot draw = new Plot();
    //drawing graph of the given function
    draw.table(equation, lower, upper);
    switch (choice) {
    case "Number of Iterations":
        iteration  $\leftarrow$  value;
        break;
    case "Absolute Relative Error":
        Eps  $\leftarrow$  value;
        break;
    }
    if(choice.equals("Number of Iterations")) {
        if(evaluate.eval(equation, lower)*evaluate.eval(equation, upper) is less than 0) {
            for ( i from 0 to iteration; increase by 1) {

```

```

nom ← precision(lower*evaluate.eval(equation,upper)-upper*evaluate.eval(equation,
lower),significant);
dom ← precision(evaluate.eval(equation, upper)-evaluate.eval(equation,
lower),significant);
point ← precision((nom)/(dom),significant);
if(evaluate.eval(equation, point)*evaluate.eval(equation, lower) is greater than 0) {
    Lower ← point;
    End if
}
else if(evaluate.eval(equation, point)*evaluate.eval(equation, lower) is less than 0) {
    upper ← point;
    End elseif;
}
else {
    Break;
    End else;
}
}
}
else {
    return "This function has no Roots in this interval";
    End else;
}
}

else if(choice.equals("Absolute Relative Error")) {
    if(evaluate.eval(equation, lower)*evaluate.eval(equation, upper) is less than 0) {
        while (Math.abs(evaluate.eval(equation, point)) is greater than Eps) {
            Print (evaluate.eval(equation, point));

nom ← precision(lower*evaluate.eval(equation,upper)-upper*evaluate.eval(equation,
lower),significant);
do ← precision(evaluate.eval(equation, upper)-evaluate.eval(equation,
lower),significant);
point ← precision((nom)/(dom),significant);
if(evaluate.eval(equation, point)*evaluate.eval(equation, lower) is greater than 0) {
    lower ← point;
    End if;
}
else if(evaluate.eval(equation, point)*evaluate.eval(equation, lower) is less than 0) {
    upper ← point;
    End elseif;
}
}
}

```

```

else {
    return "This function has no Roots in this interval";
}
return Double.toString(point);
}

```

#### c) Fixed Point:

```

Function FixedPoint(double x0, String input, int iterationlimit, double epsilon) {
    gxfunc ← input + "+x";
    Print (gxfunc);
    xnew ← 0;
    i ← 0;
    xnew ← EvaluateExpression.eval(gxfunc, x0);
    while (Math.abs((xnew - x0) / xnew) * 100 is greater than or equal epsilon && i is
less than or equal to iterationlimit)
    {
        x0 ← xnew;
        xnew ← EvaluateExpression.eval(gxfunc, x0);
        I increase by 1;
        if (xnew is equal to 0) {
            Break;
        End if;
    }
}
Print (xnew);
return xnew; }

```

#### d) Newton-Raphson:

```

Function newton(String input, double x0, int iteration, double epsilon) {
    Differentiator diff ← new Differentiator();
    Function function ← new Function(input);
    Function differentiation ← diff.differentiate(function, true);
    fraction ← 0;
    i ← 0;
    x1 ← x0 - EvaluateExpression.eval(input, x0)
/EvaluateExpression.eval(differentiation.getEquation(), x0);
    while (Math.abs(((x1 - x0) / x1) * 100) is greater than or equal to epsilon &&
i is less than or equal to iteration) {
        x0 ← x1;
        fraction ← EvaluateExpression.eval(input, x0) /
EvaluateExpression.eval(differentiation.getEquation(), x0);
        x1 ← x0 - fraction;
    }
}

```

```

    I increase by 1;
}
Print ("the root is = " + Math.round(x1 * 100.0) / 100.0);
return Math.round(x1 * 100.0) / 100.0;
}

```

**e) Secant Method:**

**//function To perform precision**

```

Function precision(double num, int precision) {
    BigDecimal round ← new BigDecimal(Double.toString(num));
    round ← round.setScale(precision, RoundingMode.HALF_UP);
    return round.doubleValue();
}

```

**//Function to evaluate the required root by the rule**

```

Function EvaluateRoot(double Xj , double Xi, double Fxj,double Fxi,int
precision ) {
    double X;
    X ← precision( Xi ,precision) - (precision(precision(Fxi*(Xi-Xj),precision) /
precision(Fxi - Fxj ,precision),precision));
    return precision( X ,precision);
}

```

**//Function to Evaluate the absolute relative error after each step**

```

Function EvaluateRelativeError(double X , double Xi,int precision) throws
NumberFormatException {
    double Ea;
    Ea ← Math.abs(( precision(X,precision) - precision(Xi,precision)) /
precision(X,precision) )*100;
    return precision(Ea,precision);}

```

```

Function SecantMethod(int precision ,String choice, double value , double
Xj , double Xi,String function ) {
    //value : number of a maximum iterations or Eps based on the choice of the user.
    // fun: the function that needs to get roots of it
    //Xj : X(i-1)
    //Xi : Xi
    //X : X(i+1) >> the required root
    X ← 0;
    Fxj,Fxi;
    MaxIterations ← 0;
    Eps ← 0 ,E ← 0;
}

```

```

switch (choice) {
  case "Number of Iterations":
    MaxIterations  $\leftarrow$  value;
    break;
  case "Absolute Relative Error":
    Eps  $\leftarrow$  value;
    Break;
End switch
if(MaxIterations equal to value) {
  for( i  $\leftarrow$  0 to MaxIterations increased by 1) {
    Fxj  $\leftarrow$  ev.eval(function,precision(Xj,precision));
    Fxi  $\leftarrow$  ev.eval(function,precision(Xi,precision));
    X  $\leftarrow$  EvaluateRoot( Xj , Xi, Fxj, Fxi, precision );
    E  $\leftarrow$  EvaluateRelativeError(X , Xi , precision);
    Xj  $\leftarrow$  precision(Xi,precision);
    Xi  $\leftarrow$  X;
  End for
else if(Eps equal to value) {
  Fxj  $\leftarrow$  ev.eval(function,precision(Xj,precision));
  Fxi  $\leftarrow$  ev.eval(function,precision(Xi,precision));
  X  $\leftarrow$  EvaluateRoot( Xj , Xi, Fxj, Fxi, precision );
  E  $\leftarrow$  EvaluateRelativeError(X , Xi , precision);
  while (E greater than Eps) {
    Fxj  $\leftarrow$  ev.eval(function,precision(Xj,precision));
    Fxi  $\leftarrow$  ev.eval(function,precision(Xi,precision));
    X  $\leftarrow$  EvaluateRoot( Xj , Xi, Fxj, Fxi, precision );
    E  $\leftarrow$  EvaluateRelativeError(X , Xi , precision);
    Xj  $\leftarrow$  precision(Xi,precision);
    Xi  $\leftarrow$  X;
  End while
  return precision(X,precision) ;
End if
return precision(X,precision);
}

```



### 3. Sample runs for each method:

Function:  $\sin(x)=0$

**Solve Equation**

Method: Bisection

Equations:  $\sin(x)=0$

Variables:

Output: X = This function has no Roots in this interval

Run Time: 944900

Parameters dialog:

Parameter

Precision: 9

Xu: 1, Xl: -1

Number of Iterations: 50

OK

Buttons: Plot, Set Attributes, Solve, Clear

**Solve Equation**

Method: Bisection

Equations:  $\sin(x)=0$

Variables:

Output: X = 0.0

Run Time: 1372800

plot panel:

Graph of  $y = \sin(x)$  vs  $x$ . The x-axis ranges from -4.0 to 4.0, and the y-axis ranges from -1.0 to 1.0. The root is marked at  $x = 0.0000000000000000$ .

Buttons: Plot, Set Attributes, Solve, Clear

**Solve Equation**

Method: Bisection

Equations:  $e^x(x) - 8x\sin(x) + 7 = 0$

Variables:

Output:

Run Time:

Parameters

Precision: 8

Xu: 0 Xl: -2

Absolute Relative Error: 0.001

OK

Plot Set Attributes Solve Clear

**Solve Equation**

Method: Bisection

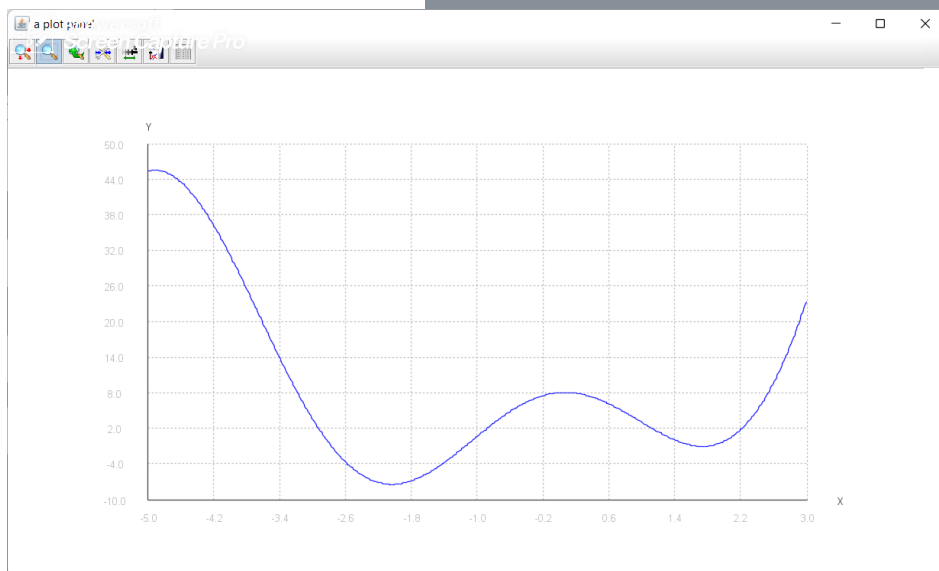
Equations:  $e^x(x) - 8x\sin(x) + 7 = 0$

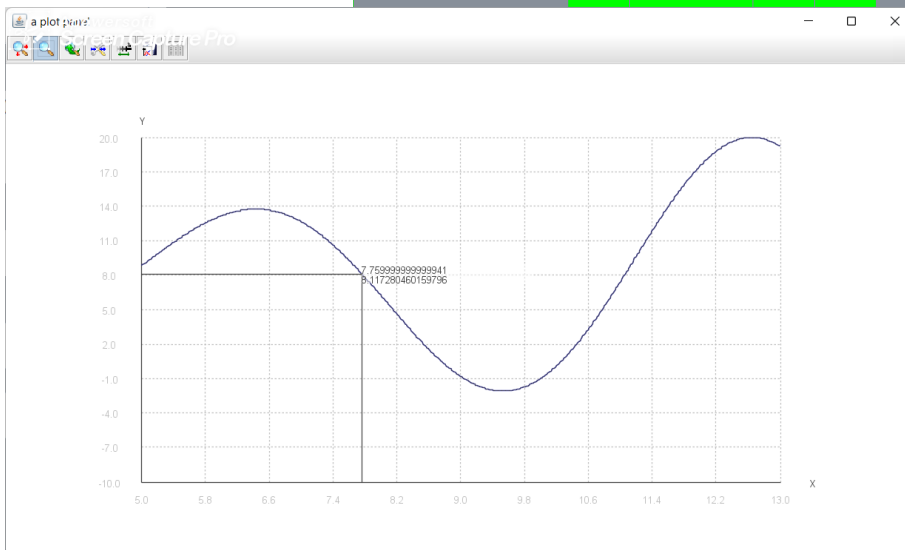
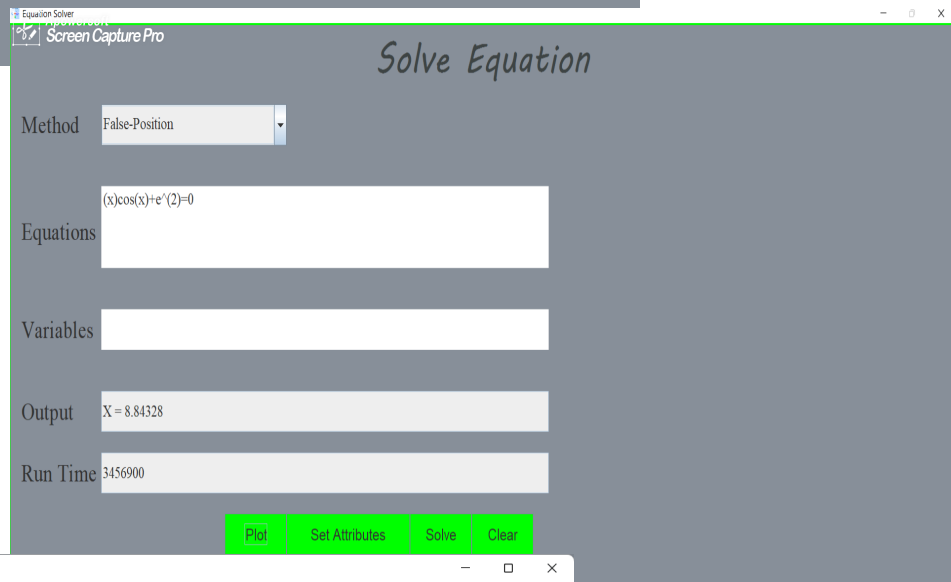
Variables:

Output:  $X = -1.05566407$

Run Time: 3483000

Plot Set Attributes Solve Clear





Equation Solver

## Solve Equation

Method: False-Position

Equations:

Variables:

Output:

Run Time:

Parameters

### Parameter

Precision:

Xu:  Xl:

Number of Iterations: 10

Apowersoft  
Screen Capture Pro

## Solve Equation

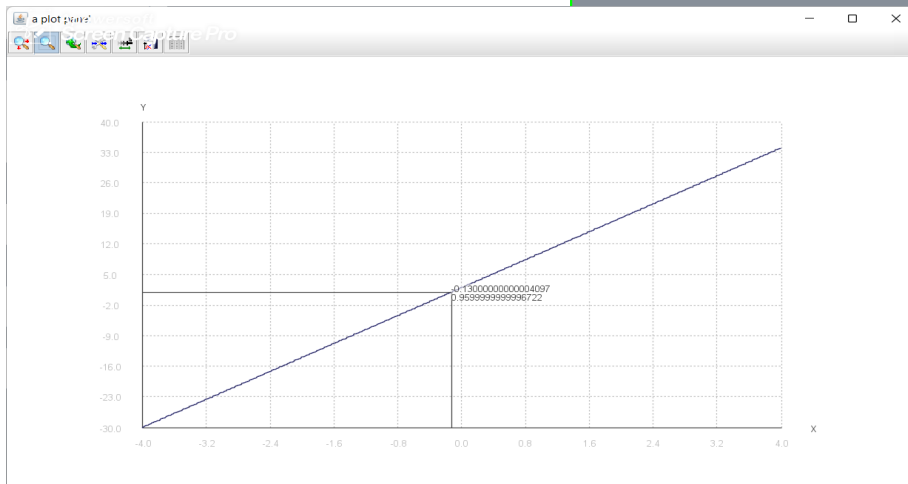
Method: False-Position

Equations:

Variables:

Output:

Run Time:



**Solve Equation**

Method:

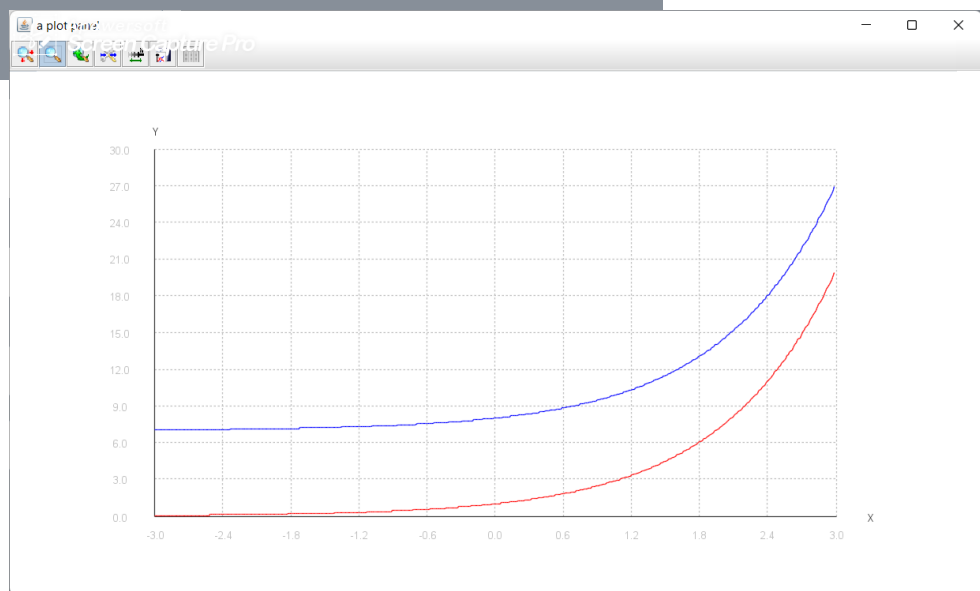
Equations:

Variables:

Output:

Run Time:

Diverges means that the function doesn't have a root.



**Solve Equation**

Method: Fixed Point

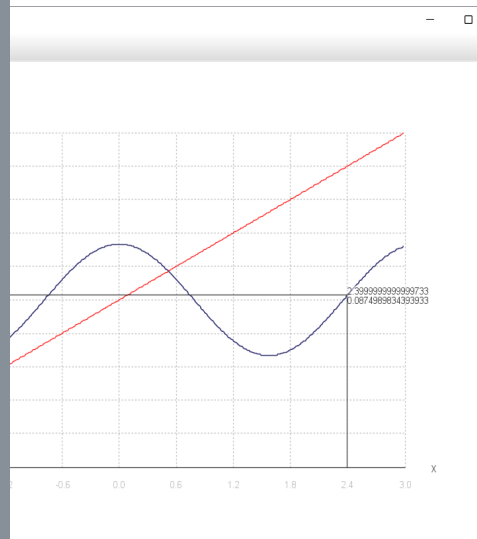
Equations:  $\cos(2x)=0$

Variables:

Output:  $X = 0.7066866281$

Run Time: 1534000

Plot Set Attributes Solve Clear



**Solve Equation**

Method: Fixed Point

Equations:  $e^{(-2x)}*(x-1)=0$

Variables:

Output:

Run Time:

Parameters

Precision: 8

Initial Guess: 0.99

Itr: 50 E: 0.00001

OK

Plot Set Attributes Solve Clear

**Solve Equation**

Method: Fixed Point

Equations:  $e^{(-2x)}*(x-1)=0$

Variables:

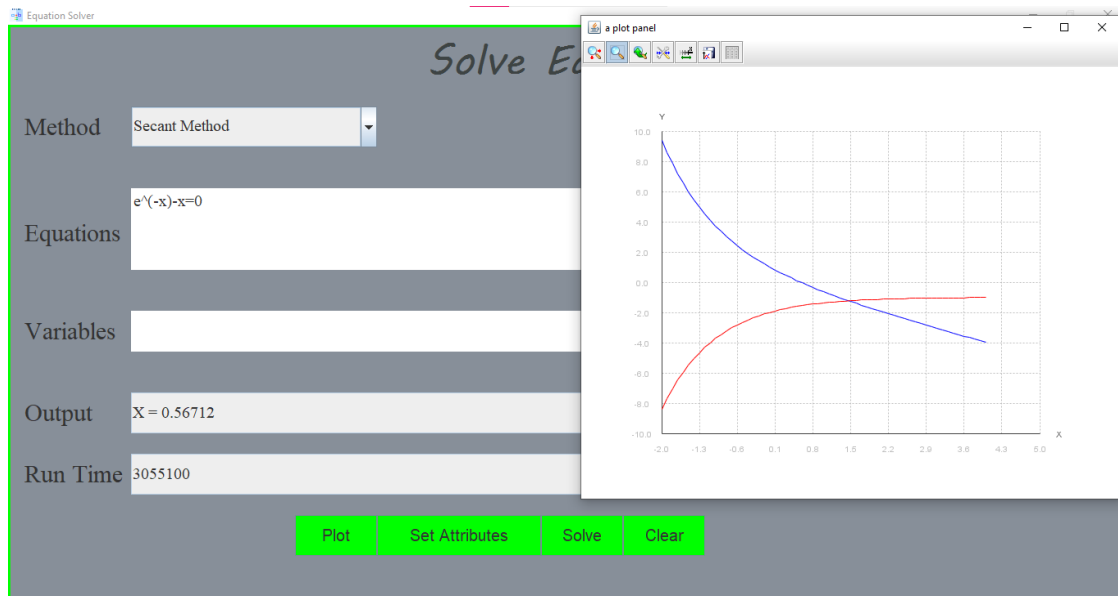
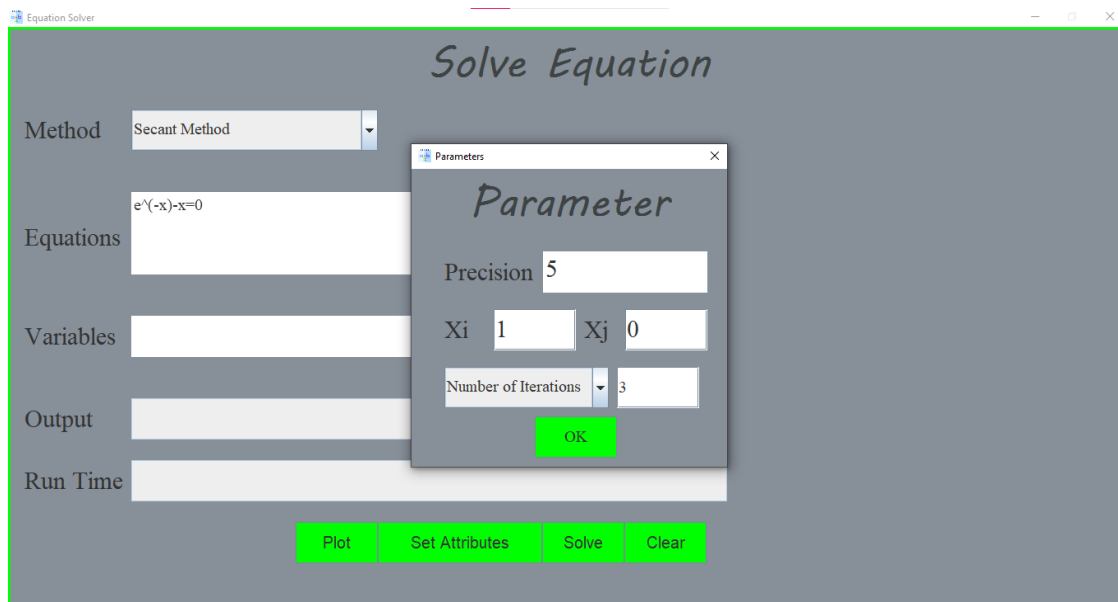
Output:  $X = \text{diverge monotonic}$

Run Time: 45354100

Plot Set Attributes Solve Clear

a plot panel

A graph showing the function  $y = e^{(-2x)}*(x-1)$  in blue and a horizontal line  $y = 0$  in red. The x-axis ranges from -3.0 to 3.0, and the y-axis ranges from -2000.0 to 1000.0. The blue curve starts at  $(-3, -117.0)$  and increases towards the red line, but does not cross it within the visible range, indicating divergence.



Equation Solver

## Solve Equation

Method: Secant Method

Equations:  $x^2 - 2 = 0$

Variables:

Output:

Run Time:

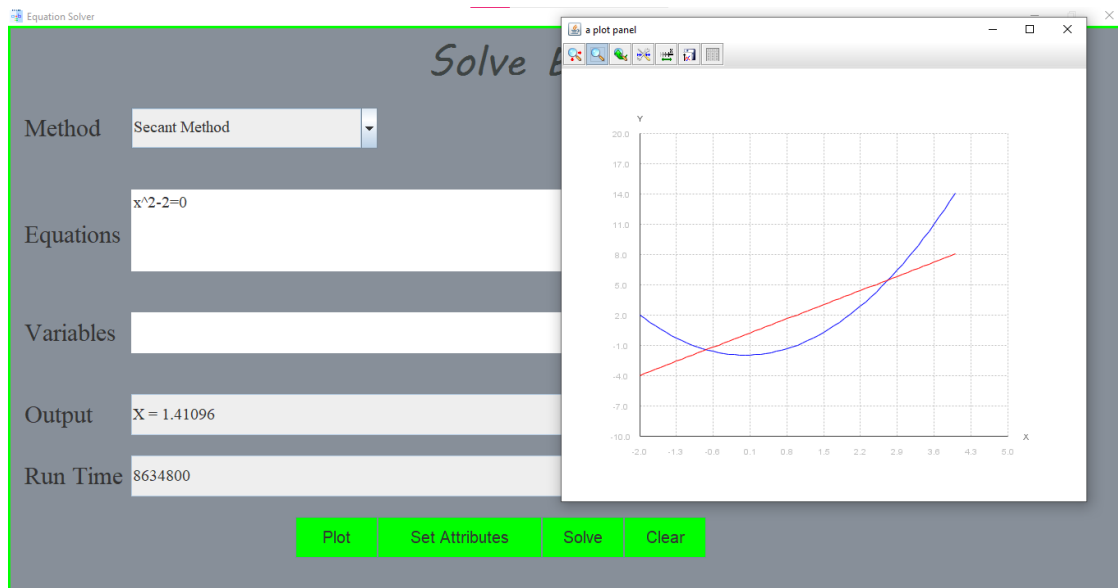
Precision: 5

Xi: 1 Xj: 0.5

Absolute Relative Error: 5

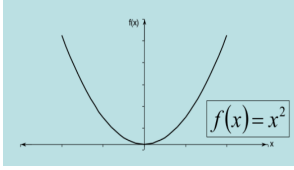
OK

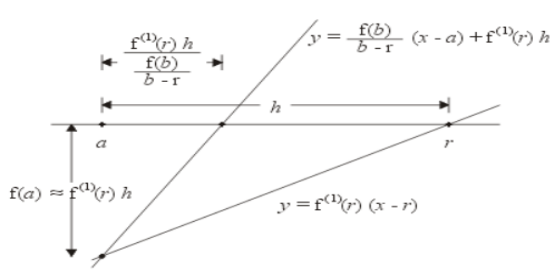
Plot Set Attributes Solve Clear





4. Comparison between different methods (time complexity , convergence, best and approximate errors):

	<u>Bisection Method</u>
Time complexity	<ul style="list-style-type: none"> <li>• <b><math>O(\log n)</math></b>.It depends on the input function and assumed values of the guesses.</li> <li>• Rate of convergence= <math>1/(O^n)</math> exponential decrease.</li> </ul>
Best case	It is guaranteed that the solution will be converged.
Worst Case	<ul style="list-style-type: none"> <li>• The system will be stopped as long as there is no root between upper and lower guesses.</li> <li>• If the root is very large ,the approximate error will fail to stop the method.</li> </ul>
Convergence	The method always converges as long as the guesses are correct (which means that there is a root between guesses.
Pitfall	<p>If the function touches the x-axis exactly, it is hard to find lower and upper guesses,so it is hard to find the root.</p>  <p>If there are two roots between guesses of upper and lower, they won't be found.</p>
Approximate Errors	<p>New estimate root:</p> $x_m = \frac{x_l + x_u}{2}$ <p>The error halved in each iteration step, as the estimated root is halved from the original one.</p> $ e_a  = \left  \frac{x_r^{new} - x_r^{old}}{x_r^{new}} \right  \times 100$

	<b><u>False Position</u></b>
Time complexity	<ul style="list-style-type: none"> <li><math>O(n)</math></li> </ul>
Best case	It always converges, even faster than the Bisection Method.
Worst Case	It is sometimes slower than the Bisection method in the worst cases.
Convergence	<ul style="list-style-type: none"> <li>- It always converges as long as there is a root between upper and lower bound.</li> <li>-Rate of convergence depends on the position of the root with respect to the bounds given.</li> <li>-Linear convergence</li> </ul>
Pitfall	<ul style="list-style-type: none"> <li>• Even if the convergence of the function is certain, it is relatively slow.</li> <li>• There are some roots that can't be found in some equations like <math>x^2</math> as there are no bracketing values.</li> <li>• Linearity of convergence.</li> <li>• If the function has some discontinuous points, false positions can't be applied.</li> <li>• Can't find two roots between upper and lower bounds.</li> </ul>
Approximate Errors	<p>Estimate root in each iteration is calculated as:</p> $x_r = \frac{x_U f(x_L) - x_L f(x_U)}{f(x_L) - f(x_U)}$ <p>Error in each iteration= h-difference between (new point and old one on x-axis). Smaller interval in the given graph.</p> 

	<u><b>Fixed point</b></u>
Pitfalls	<ul style="list-style-type: none"> <li>It will not always converge. There are infinitely many rearrangements of <math>f(x) = 0</math> into <math>x = g(x)</math>. Some rearrangements will only converge given a starting value very close to the root, and some will not converge at all.</li> </ul>
convergence	<p>the sequence converges linearly to the fixed point.</p> <p>(a) If <math> g'(x)  &lt; 1</math>, <math>g'(x)</math> is +ve converge, monotonic.</p> <p>(b) If <math> g'(x)  &lt; 1</math>, <math>g'(x)</math> is -ve converge, oscillate.</p>
Time complexity	$O(\log n)$
approximate errors	<ul style="list-style-type: none"> <li>When the method converges, the error is roughly proportional to or less than the error of the previous step</li> </ul>

	<u><b>Newton-Raphson</b></u>
Time complexity	$O((\log n) F(n))$ where $F(n)$ is the cost of calculating $f(x)/f'(x)$
convergence	It converges quadratically( the accuracy gets doubled at each iteration).
pitfalls	<ul style="list-style-type: none"> <li>-Division by zero problem can occur.</li> <li>-In case of multiple roots, this method converges slowly.</li> <li>-It may jump from one location close to one root to a location that is several roots away.</li> </ul>
Best Case	<ul style="list-style-type: none"> <li>It converges faster than almost any other alternative iteration scheme based on other methods of converting the original <math>f(x)</math> to a function with a fixed point.</li> </ul>
approximate errors	Each error is approximately proportional to the square of the previous error Then the number of correct decimal places roughly doubles with each approximation.

	<b><u>Secant</u></b>
Time complexity	$O(h^{1.618})$ .
convergence	<ul style="list-style-type: none"> <li>It converges to the root of the function if the initial values are sufficiently close to the root.</li> <li>The order of convergence is 1.618 (the golden ratio). The convergence is superlinear, but not quite quadratic.</li> </ul>
Pitfalls	<ul style="list-style-type: none"> <li>It fails by bad initial values or rounding error computations.</li> <li>If we have the wrong starting values the method diverges (produces increasingly nonsensical results).</li> </ul>
Best Case	<ul style="list-style-type: none"> <li>It converges faster than the bisection method.</li> <li>It does not require use of the derivative of the function.</li> </ul>
approximate errors	<ul style="list-style-type: none"> <li>When the method converges, the error is roughly proportional to or less than the error of the previous step.</li> </ul>

### **Data Structure Used:**

We didn't use any specific data structure in both methods or validation. Instead we used some external libraries and built in methods that ease up the work.

### **Built-In methods Used:**

#### **BigDecimalsetScale():**

- `round = round.setScale(precision, RoundingMode.HALF_UP);`
- It is used to rescale numbers entered according to determined significant figures.
- It used external library which is import  
**1. java.math.BigDecimal.**

#### **Rounding up mode:**

- It is the rounding mode that determines that the number entered in setScale will be rounded up.
- It used external library which is import:  
**1. java.math.BigDecimal.**

#### **Expression Builder():**

- We used it to build an expression from the input string that the user will enter. it also helped us to evaluate the expression easily, just entering the string equation and the value of the variable.

```

net.objecthunter.exp4j.Expression expression = new ExpressionBuilder(equation)
    .function(ln)
    .variables("x")
    .build()
    .setVariable("x", x);

double result = expression.evaluate();

```

- It used external library which is import:  
**1. net.objecthunter.exp4j.ExpressionBuilder.**

#### **Plot2SPanel():**

- Class used to show data given in a graph (x-axis and y- axis), we used it in all methods to represent the given function.
- It used external library which is import:  
**1. org.math.plot.\*.**

#### **Differentiation used in Newton Raphson:**

```
Differentiator diff = new Differentiator();
Function function = new Function(input);
Function differentiation = diff.differentiate(function, true);
```

- Import external library:  
**1-MathEngine**  
<https://github.com/raharrison/MathEngine>

#### **Assumptions:**

- When entering any function inside sin or cosine we must enter it between two brackets like **sin(x)**.
- When entering beside function inside sin or cosine we must enter it between two brackets like **(3x)sin(x)**.
- The power of exponential function must be entered between two brackets **e^(-4x)**.
- The format of the function must be in following format **4x+sin(3x)-4=0**, if we entered it in the following shape **x= 4x+sin(3x)-4** or **4x+sin(3x)-4=x** then x will be calculated in the other side of the equation and the equation will be calculated in the following format **3x+sin(3x)-4=0**

[Link to the demo video.](#)