

Numerical Analysis and Computer Applications **Semester Project**

Team Members

| | |
|---------------------|------|
| Bassam Mattar | (16) |
| Sajed Hassan | (28) |
| Abdelrahman Omran | (37) |
| Mohamed El-Maghraby | (55) |
| Mahmoud Tarek Samir | (62) |
| Hesham Medhat | (70) |



Index

| | |
|---|----|
| Cover and Team Members | 01 |
| Index | 02 |
| Introduction | 03 |
| Part I: Root Finding | 04 |
| • Bracketing Methods | |
| ◦ Bisection | 06 |
| ◦ False-position | 09 |
| • Open Methods | |
| ◦ Fixed point | 13 |
| ◦ Newton-Raphson | 17 |
| ◦ Secant | 22 |
| • Bierge Vieta | 27 |
| • General Algorithm | 33 |
| Part II: Numerical Interpolation | |
| • Lagrange Interpolation | 37 |
| • Newton Interpolation | 43 |



Introduction

This is the semester project for the course of '*Numerical Analysis and Computer Applications*' - CS213 for the 2nd year of Computer and Systems Engineering Department to be delivered to *Dr. Wafaa ElHaweet* and the teaching assistant *Eng. Omar Salaheldine*.

Our project is developed in Python programming language. We used [appJar](#) library to create the interactive GUI.

The project is composed of two main segments which are:

1. **Part I : Root Finding**
 - a. Bracketing Methods.
 - b. Open Methods.
 - c. Bierge Vieta
 - d. General Algorithm.
2. **Part II : Numerical Interpolation**
 - a. Newton Interpolation.
 - b. Lagrange Interpolation.

It is advised to refer to the index to find each section easily.

Part I: Root Finding




Specialised in finding the roots of equations supporting polynomials and different types of functions.

Included sections:

- a) Bracketing Methods.
- b) Open Methods.
- c) Bierge Vieta
- d) General Algorithm.

Here are the implemented methods each in which we will discuss in detail:

- 
- 1- Bisection (*bracketing method*).
 - 2- False-position (*bracketing method*).
 - 3- Fixed point (*open method*).
 - 4- Newton-Raphson (*open method*).
 - 5- Secant (*open method*).
 - 6- Bierge Vieta. (*open method*).
 - 7- General method. (*To be explained in detail*).

a) Bracketing Methods

Introduction

The main idea of bracketing methods is to include the root(s) in an interval such that the function is considered to have moved across the horizontal axis (abscissa) as it evaluates to values across each side (teller being that the evaluations at the interval's sides are of negative multiplication due to difference in sign). Which was proven to mean that in the ordinary cases, there is an odd number of roots included in the bracketed interval.

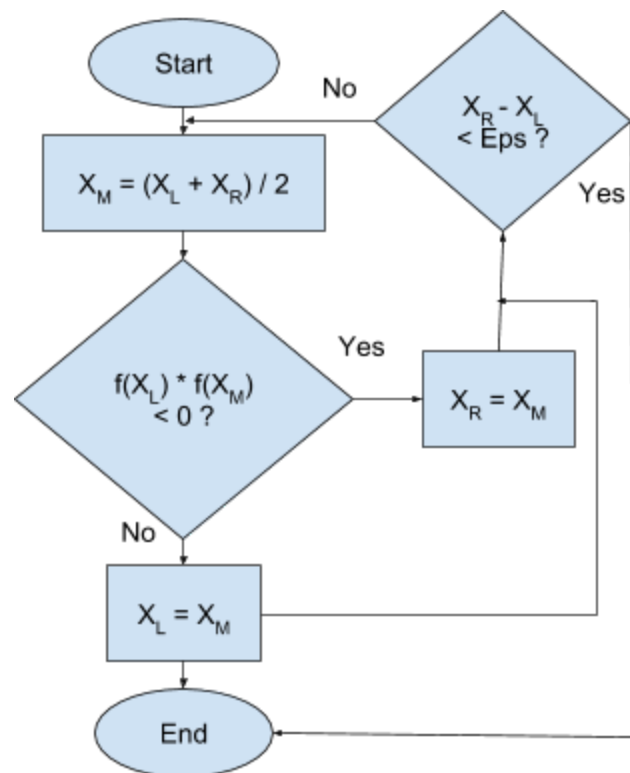
Thus, the bracketing methods aim to iteratively shorten the interval in order to find the root within a certain level of significance (epsilon).

Bisection Method

Introduction

The bracketing mechanism is done by finding the function's value in the middle of the bracketed interval. Then moving to a subproblem of the same type in either the left-halved interval or right-halved interval by checking the bracketing condition's validation for each.

Flow chart

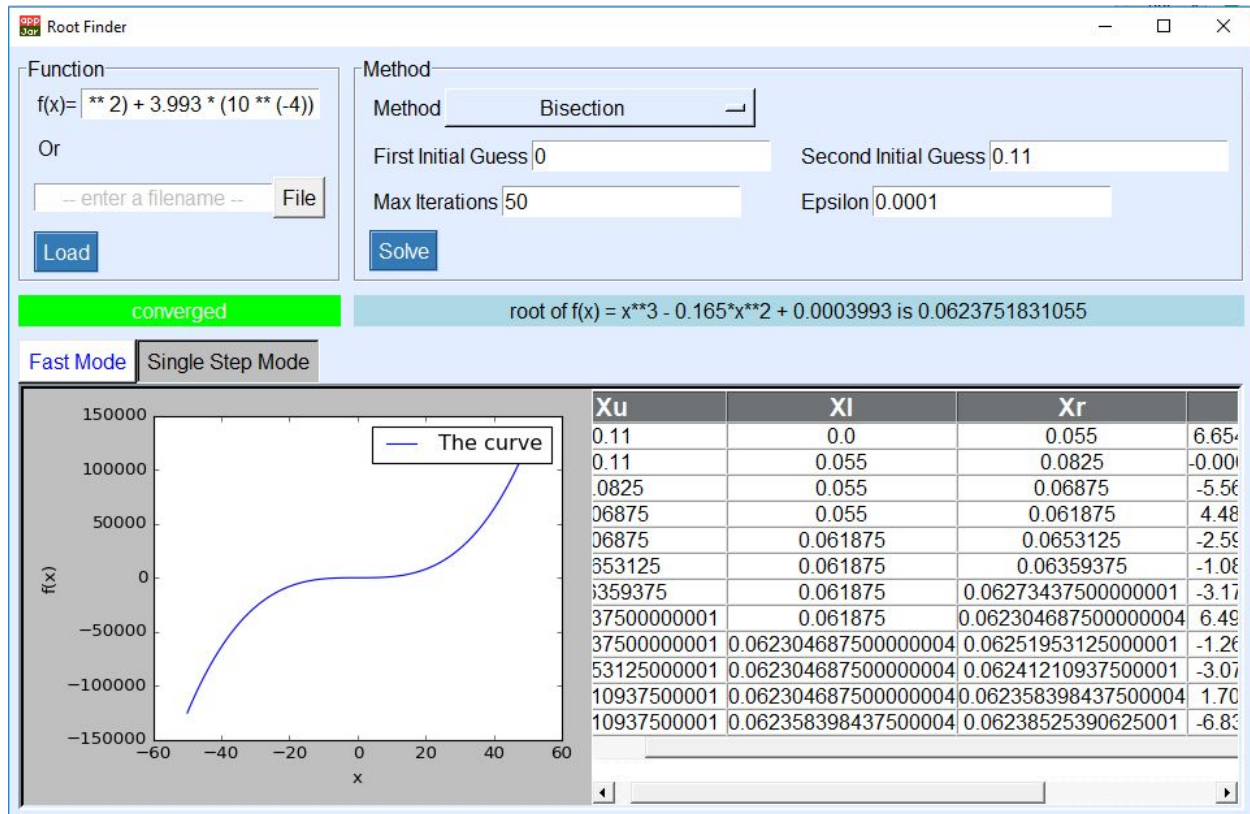


Time complexity: $O(\log_2 n)$ - logarithmic.

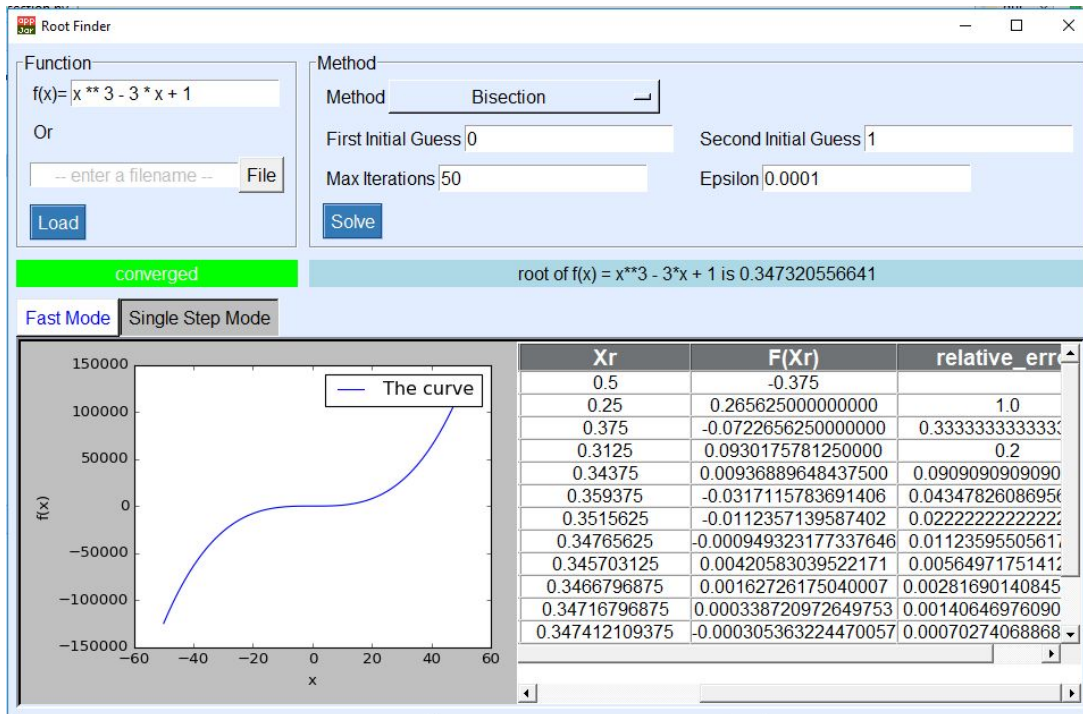
Space complexity: $O(1)$ - constant.

Sample Runs

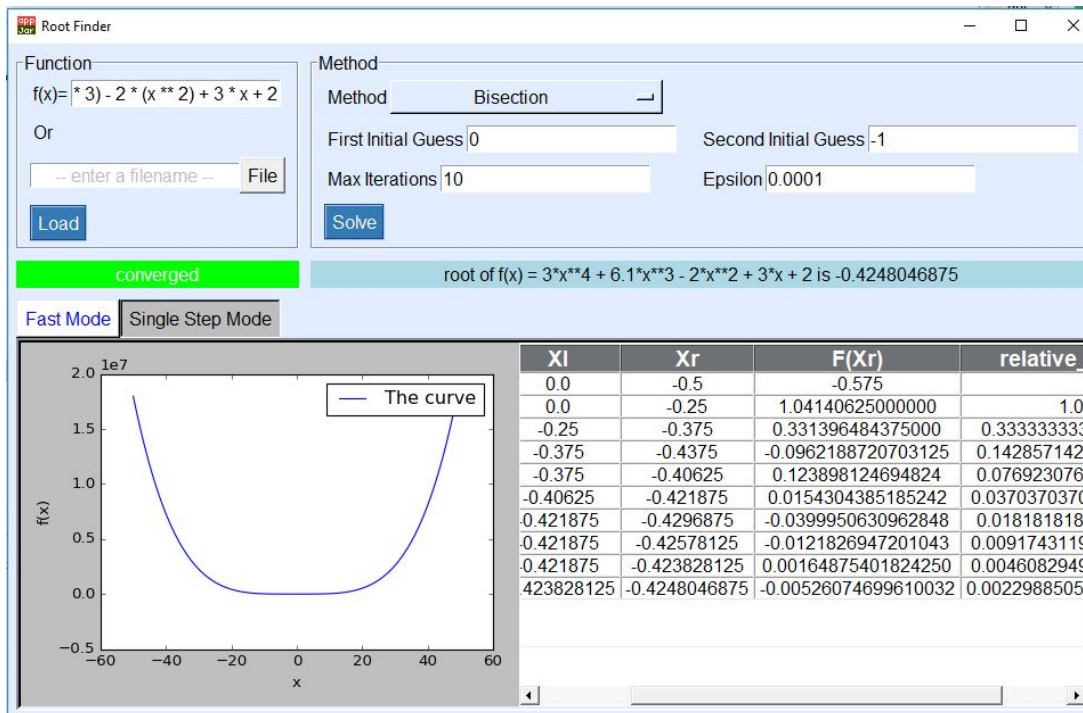
Run #1:



Run #2:



Run #3:



Comments

The greatest advantages of this algorithm are that:

- It will definitely converge and reach the root.
- Its number of iterations can be calculated beforehand as the interval is halved each time.

The disadvantages are that:

- It is slow
- Like all bracketing methods, two points are needed initially.

False Position (Regula Falsi) Method

Introduction

The bracketing mechanism is by connecting a line between the two function's values at the terminals of the interval. The root estimate is the intersection of that line with the abscissa. To get a better estimate, we evaluate the function's value at the x-coordinate intersection found at that iteration, and run the same bracketing test with the terminals and use the valid interval to find a better estimate in the next iteration.

Regula Falsi, like Bisection, always converges, usually considerably faster than Bisection—but sometimes much slower than Bisection.

Pseudocode

```
loop
     $x_2 = x_1 - (f(x_1) * (x_1 - x_0) / (f(x_1) - f(x_0)))$ ;
     $c = f(x_2)$ ;
    if absolute(c) < tolerance
        break
    end if
```

```

if f(x0)*c < 0
    x1=x2;
    continue
else
    x0=x2;
    continue
end if
end

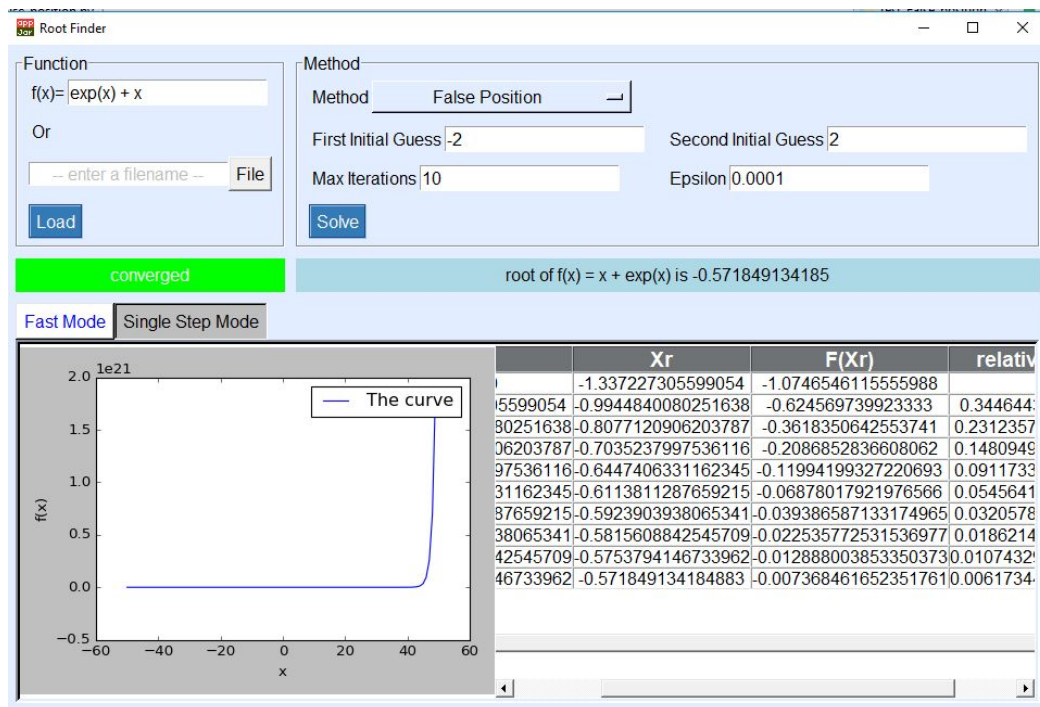
```

Time complexity: Nondeterministic. Expected to be $O(\log_2 n)$. Details in the introduction.

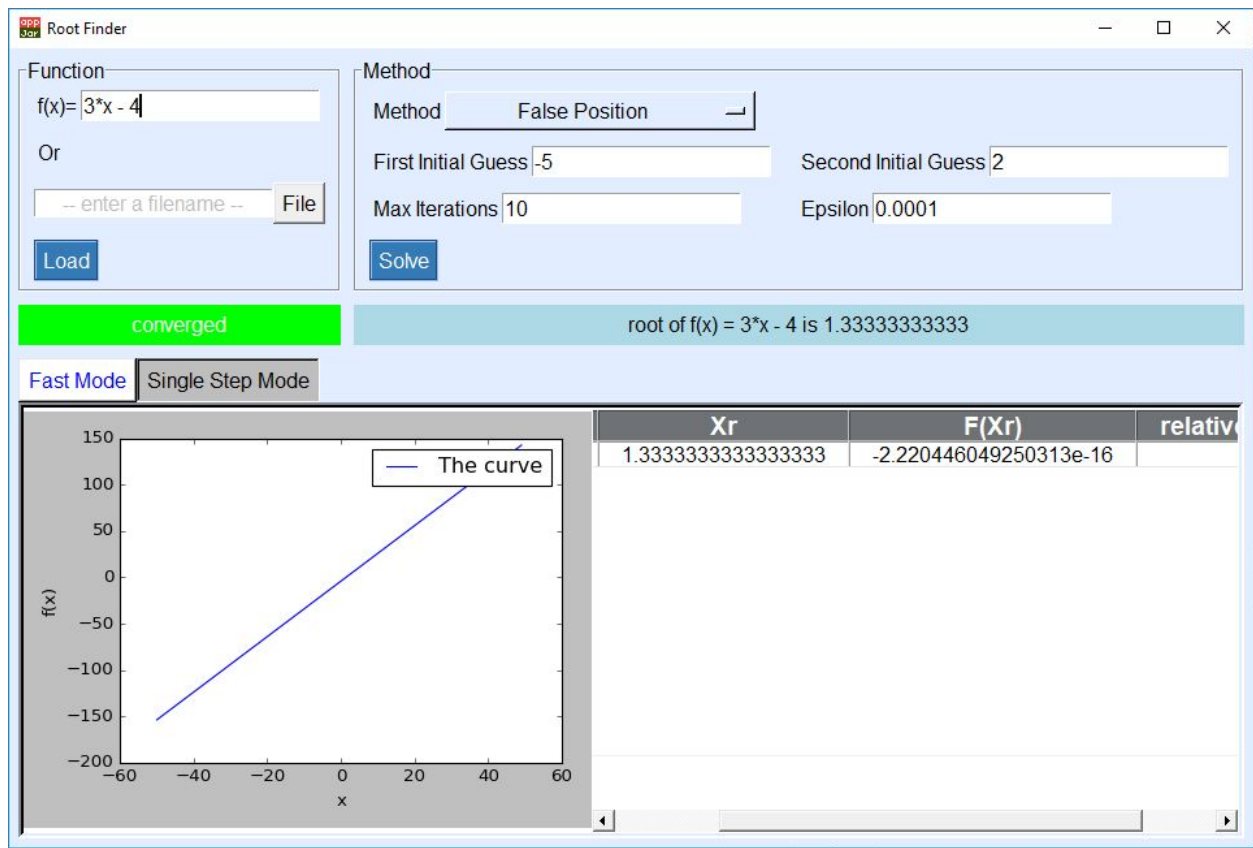
Space complexity: $O(1)$ - constant.

Sample Runs

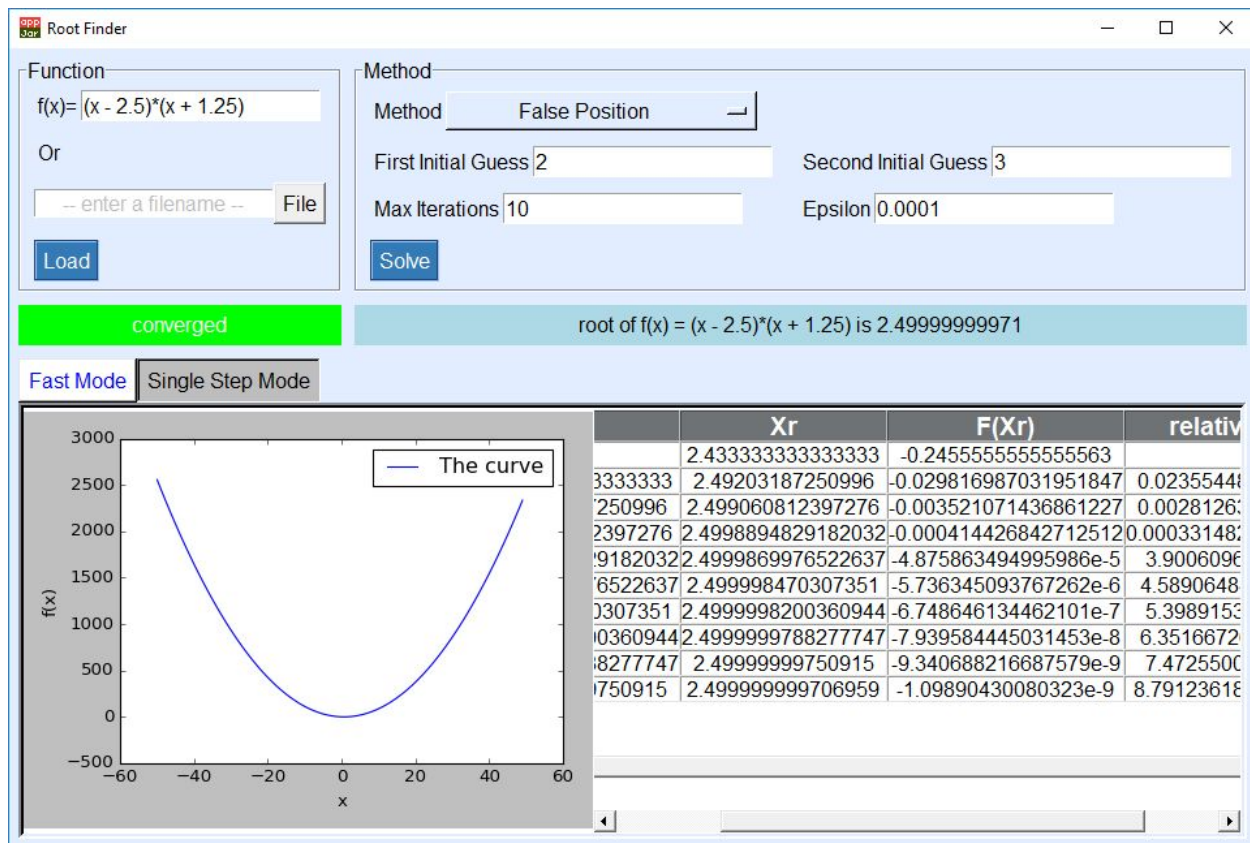
Run #1:



Run #2:



Run #3:



Comments

The greatest advantages of this algorithm are that:

- It will definitely converge and reach the root.
- It is usually better in practice than the Bisection method.

The disadvantages are that:

- For some steep functions, it could be extremely slow.
- Like all bracketing methods, two points are needed initially.

It is advised to use a smart mechanism that, perhaps uses Bisection or another method whenever the progress in shortening the interval gets to be very slow to escape the steep of these corner functions.

b) Open Methods

Introduction

The open points try to find the root in a way that doesn't require as much initial knowledge as the bracketing methods which rely on having a certain interval where a root is guaranteed to exist within.

The open methods for root finding rely mostly on trying to adapt with the function's behaviour, mostly through its first derivative or its estimate, to guide it to the roots of the equation.

Fixed-Point Method

Introduction

This method relies on finding the intersection between two functions; $y = x$ and $y = g(x)$. This is why it iterates through this formula of equating them to get $g(x) = x$ and iteratively closer to the solution.

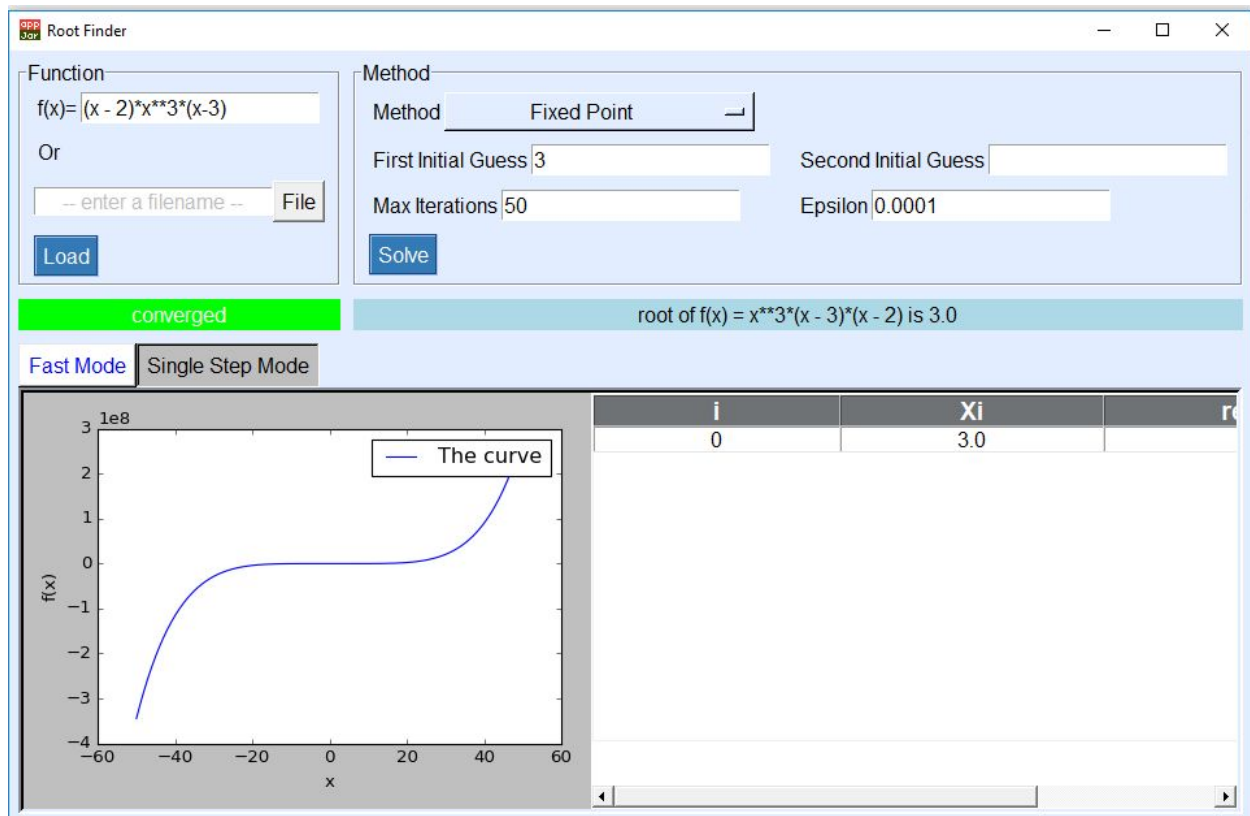
We get the function $g(x)$ by separating an x from the function in question $f(x)$. The intersection x -coordinate shall be found to be the x -coordinate of the root.

Pseudocode

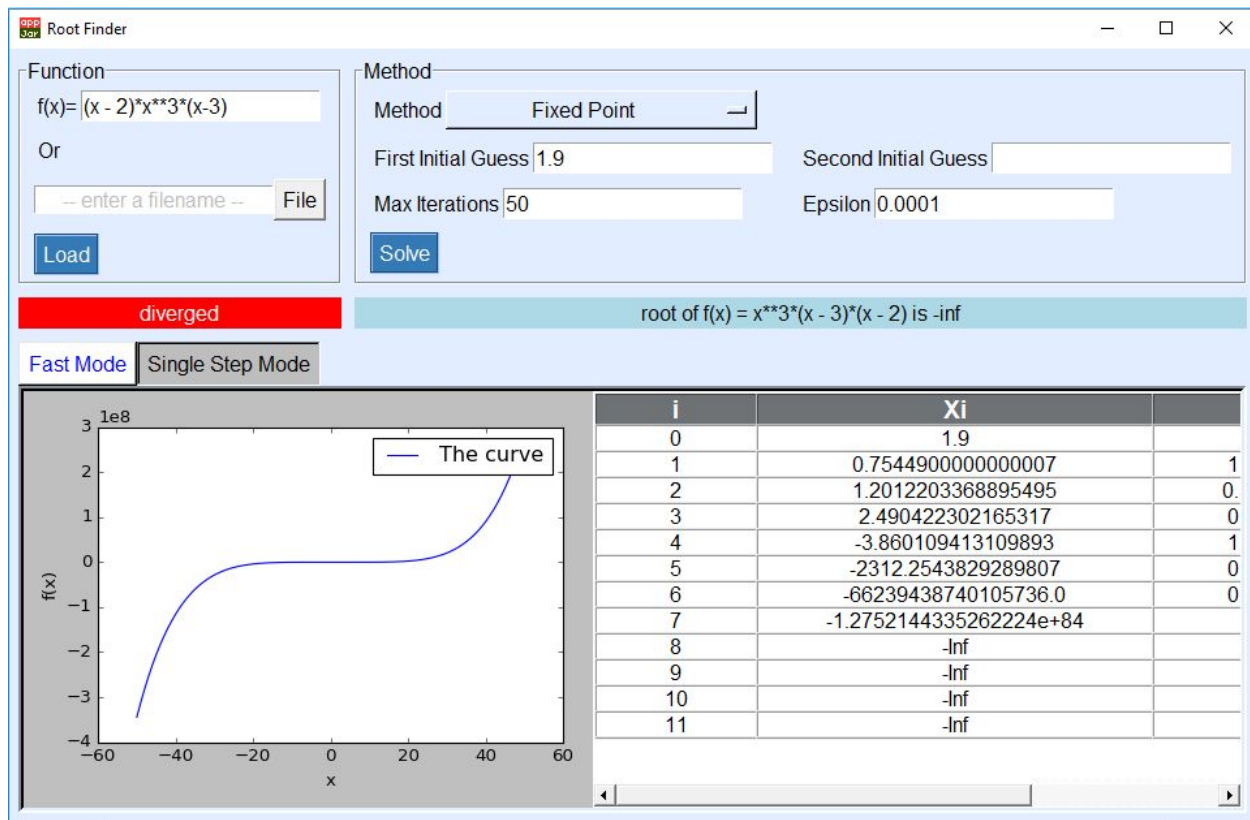
1. It's assumed that all given functions contains at least on x as the last term
2. $g(x)$ is extracted
3. Substitute x_i in $g(x)$ to calculate x_{i+1}
4. Repeat 3 until the error becomes acceptable or an error occurs.

Sample Runs

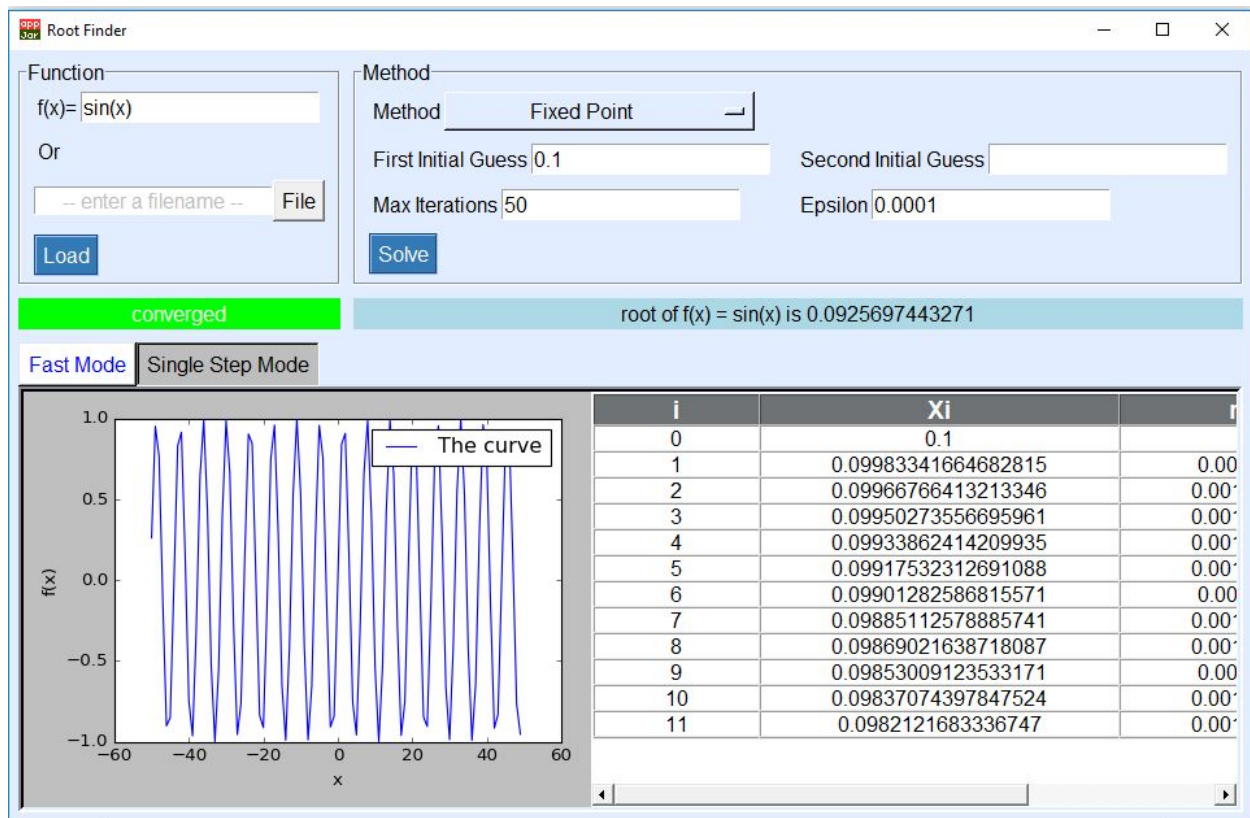
Run #1:



Run #2:



Run #3:



Comments

The greatest advantages of this algorithm are that:

- It has linear convergence
- It is usually better in practice than the Bisection method.

The disadvantages are that:

- Aside from the fast “rate” of convergence, the convergence occur only when the absolute value of the slope of $y_2 = g(x)$ is less than the slope of $y_1 = x$.

Newton-Raphson Method

Introduction

The idea of the method is as follows: one starts with an initial guess which is reasonably close to the true root, then the function is approximated by its tangent line (which can be computed using the tools of calculus), and one computes the x-intercept of this tangent line (which is easily done with elementary algebra). This x-intercept will typically be a better approximation to the function's root than the original guess, and the method can be iterated.

Pseudocode

Begin Iterations:

For i = 1 to Max

$DY = FP(Answer)$

$Change = Y / DY$

$Answer = Answer - Change$

$Y = Func(Answer)$

 If ($|Y| < Epsilon$ OR $|Change| < Epsilon$)

 Return

 END

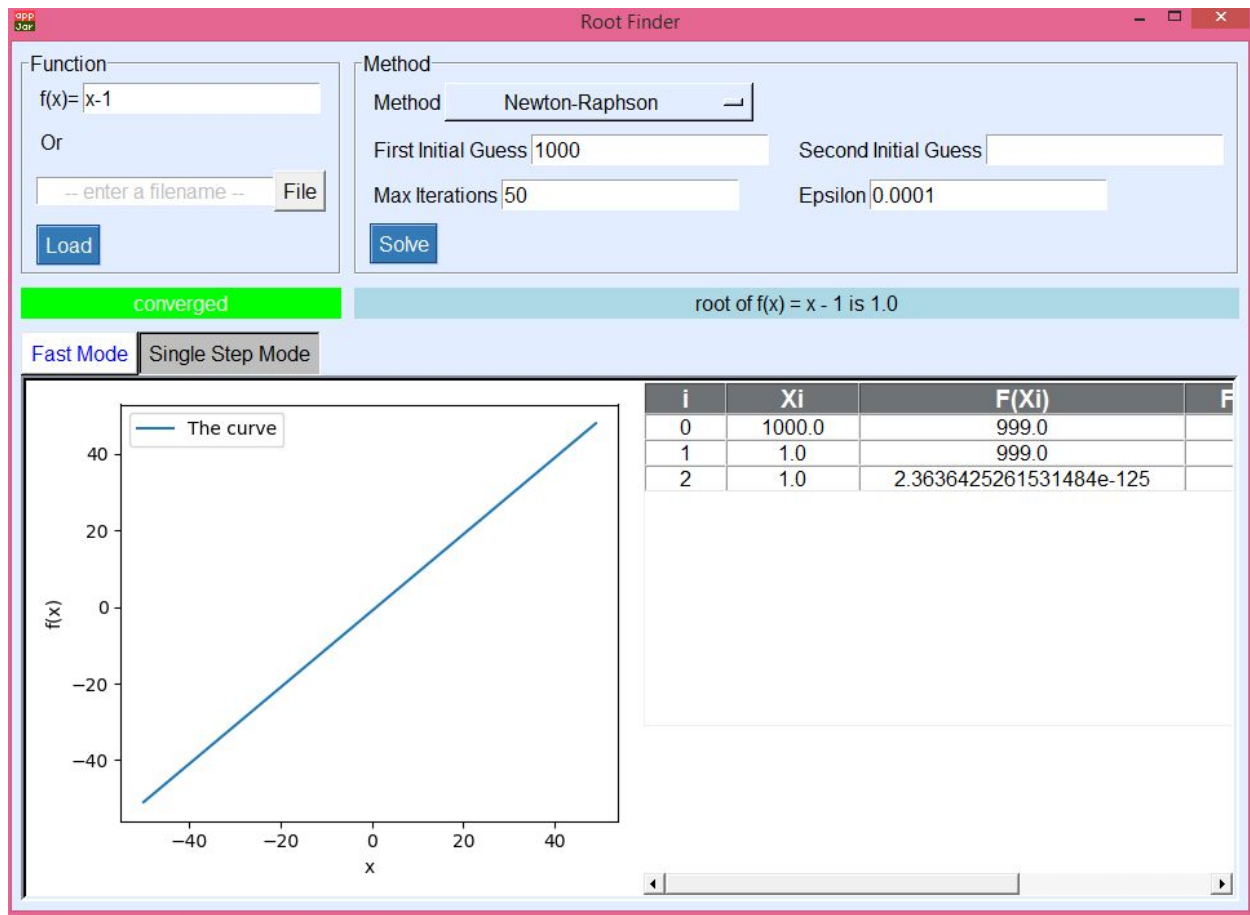
END

Sample Runs

Run #1:



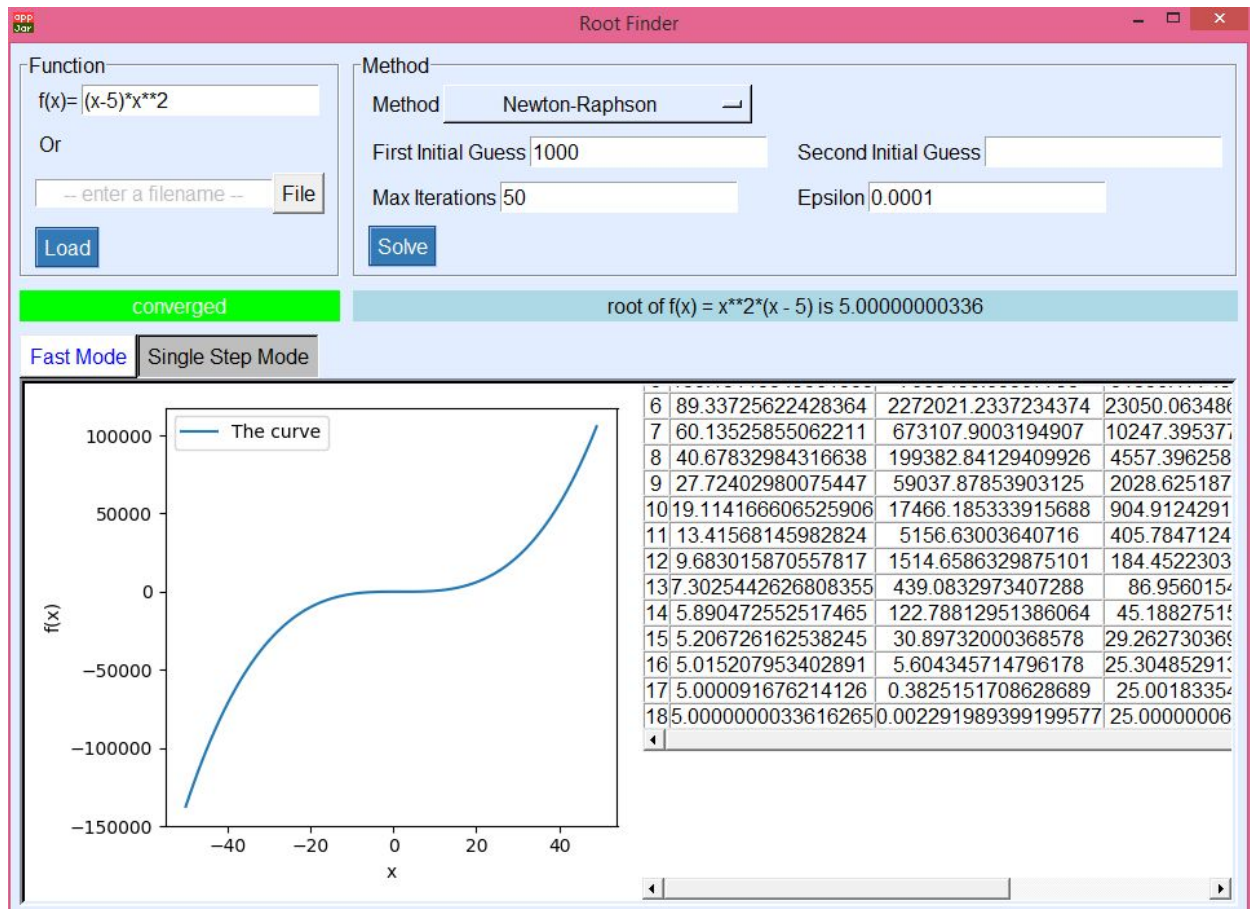
Run #2:



Run #3:



Run #4:



Comments

One of the main advantages of this algorithm is that it can deal with complex functions and it can be improved to similar versions that handle multiplicity of roots.

Secant Method

Introduction

The secant method is in theory, the same as the Newton-Raphson method explained earlier. However, instead of finding the first derivative through calculus, we use a simple $\Delta y / \Delta x$ using the points of previous two iterations. Which introduces the disadvantage of the Secant method which is the fact that it needs two points initially not one, unlike most of the open methods.

Pseudocode

```
while True:

    i = i + 1

    numerator =
float(self.function_formula.evalf(subs={self.X:
self.initial_xi})) * (self.initial_xi - self.initial_xi_1))
    denominator =
float(self.function_formula.evalf(subs={self.X:
self.initial_xi})) - (self.initial_xi - self.initial_xi_1))

    iterative_x = self.initial_xi - (numerator /
denominator)
    relative_error = (iterative_x - self.initial_xi) /
iterative_x

    fxi =
float(self.function_formula.evalf(subs={self.X: iterative_x}))
    fxi1 =
float(self.function_formula.evalf(subs={self.X: self.initial_xi

    # break when reach max iteration or precision
```

```

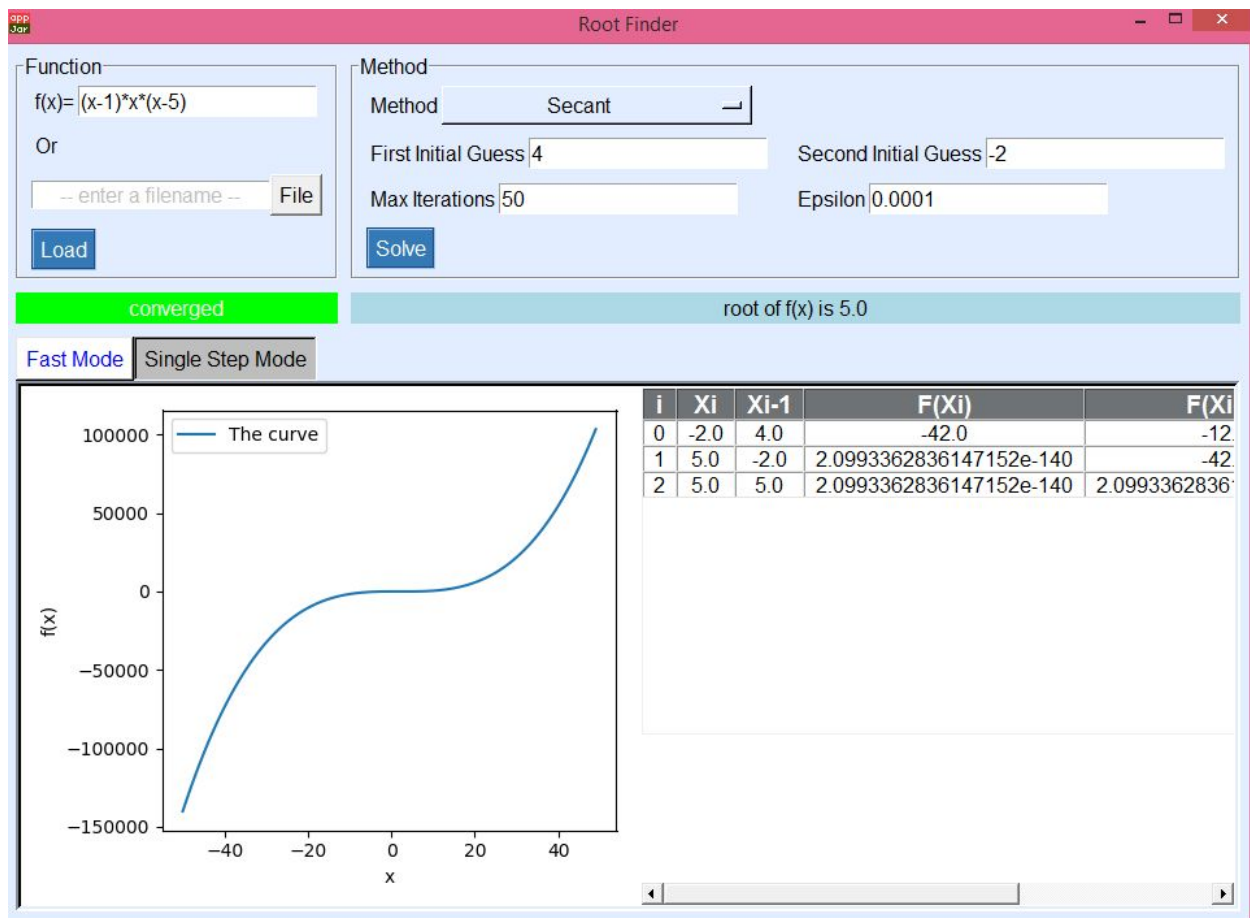
        if (math.fabs(relative_error) <= self.precision) |
(i >= self.max_iterations):
            break

self.initial_xi_1 = self.initial_xi
self.initial_xi = iterative_x

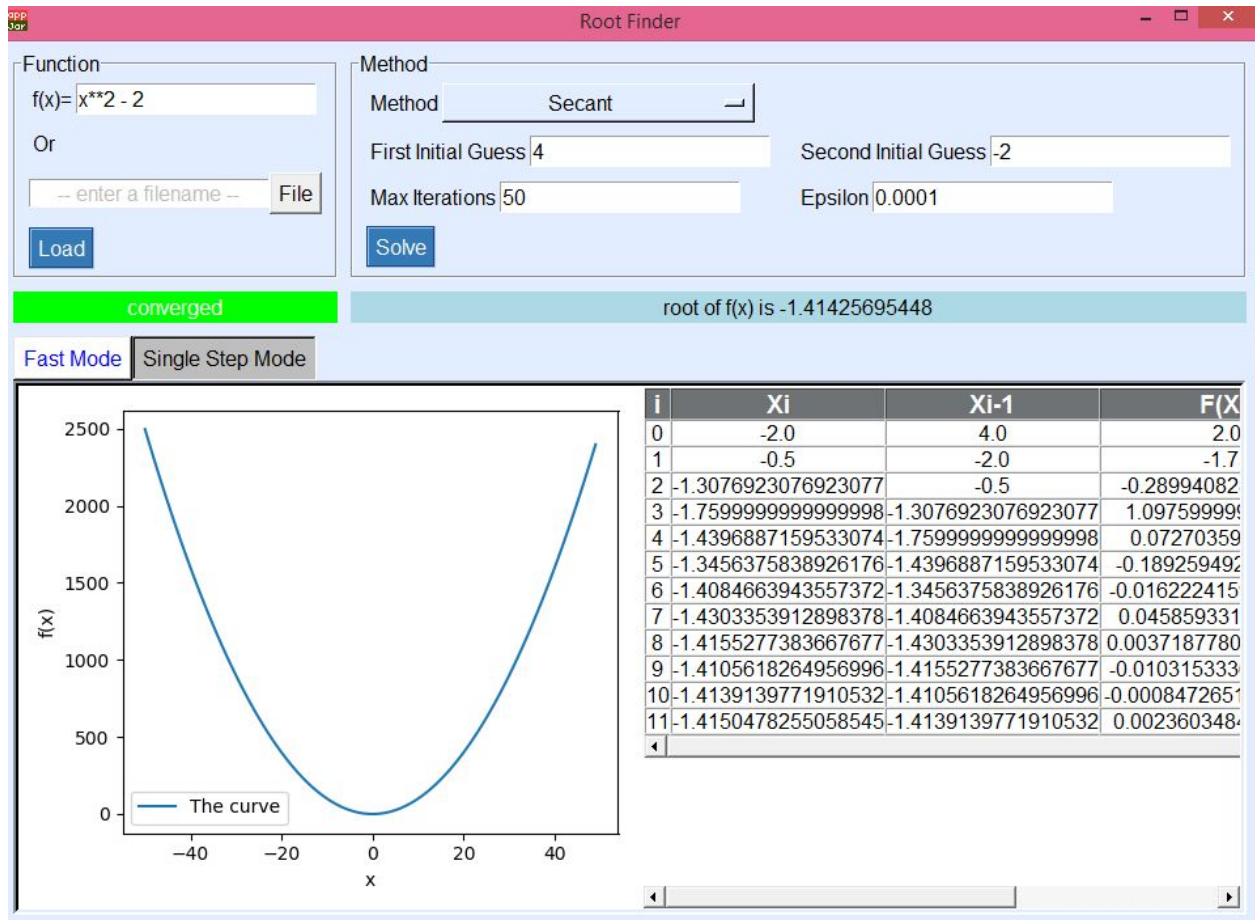
```

Sample Runs

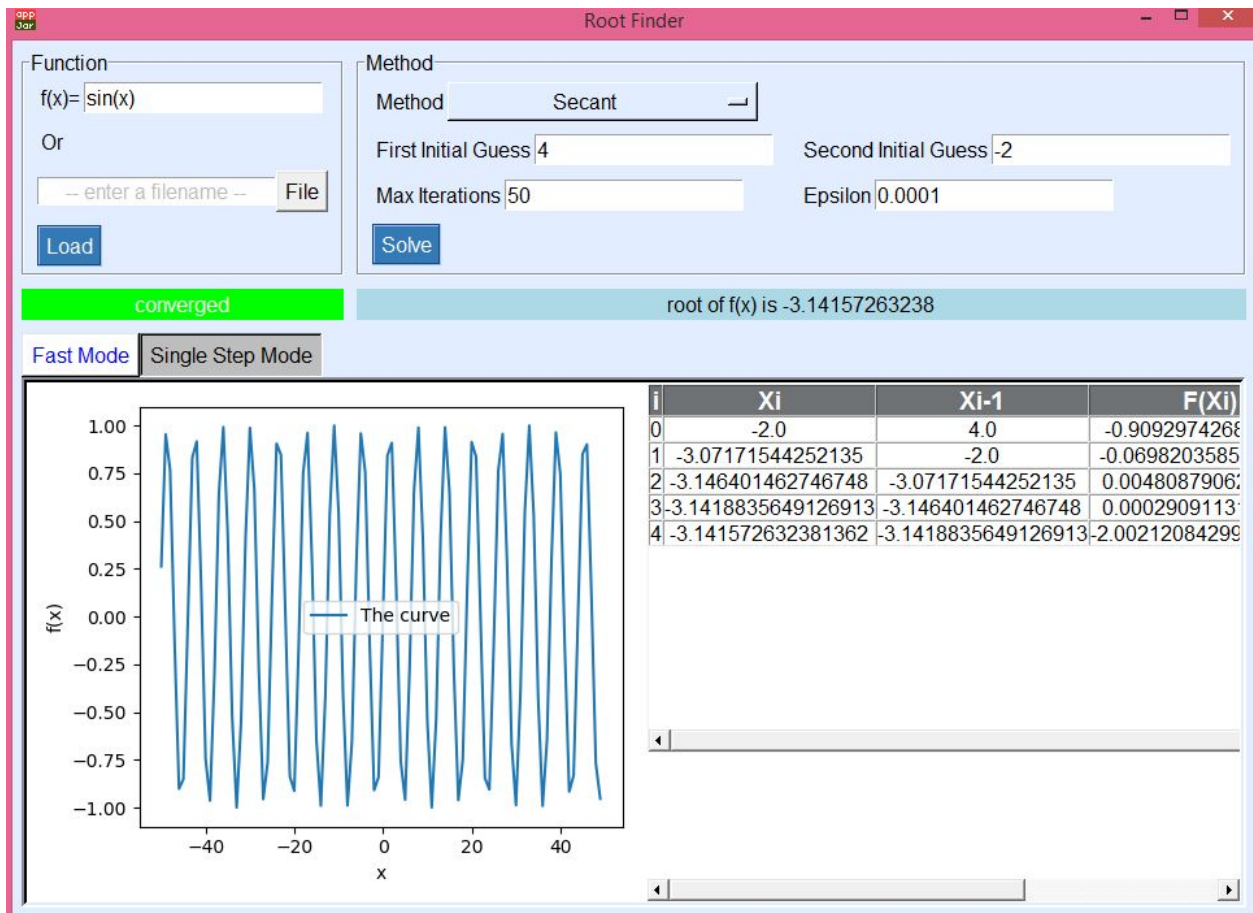
Run #1:



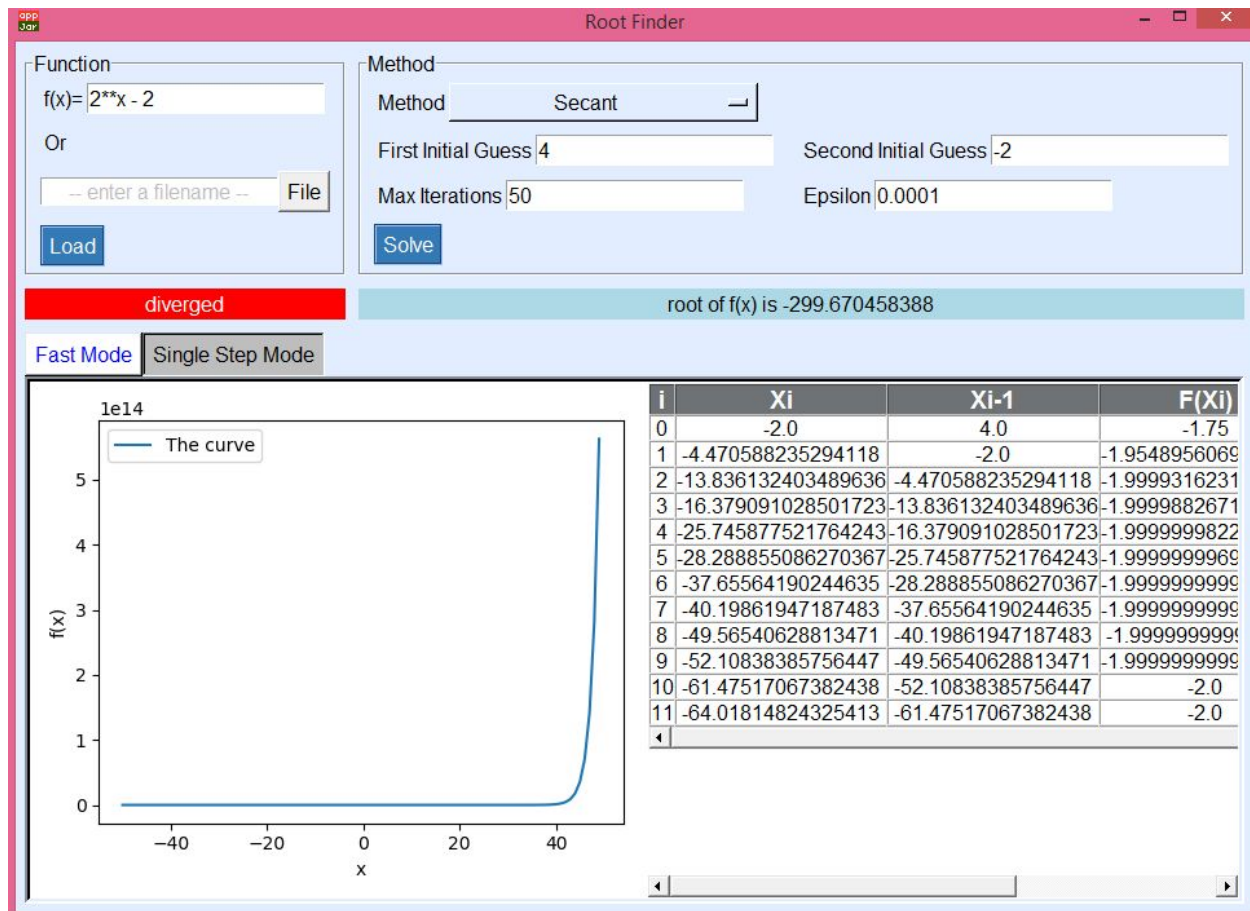
Run #2:



Run #3:



Run #4:



Comments

The same improvements to Newton-Raphson for handling multiplicity can be used to improve this method as well.

The main disadvantage as explained in the introduction is that this method requires two points initially not one, unlike most of the open methods.

c) Bierge Vieta Method

Introduction and Explanation

This method is quite complicated so we are obliged to get more into detail in this one to comprehend how it works.

This is an iterative method to find a real root of the nth degree polynomial equation $f(x) = P_n(x) = 0$ of the form

$$a_n x^n + a_{n-1} x^{n-1} + \dots + a_1 x + a_0 = 0$$

The theory can be understood better if we consider the above nth degree polynomial in the form

$$x^n + a_1 x^{n-1} + a_2 x^{n-2} + \dots + a_{n-1} x + a_n = 0$$

If s is a real root of $P_n(x) = 0$ then $P_n(x) = (x-s)Q_{n-1}(x)$ where $Q_{n-1}(x)$ is an $(n-1)$ th degree polynomial of the form

$$Q_{n-1}(x) = x^{n-1} + b_1 x^{n-2} + \dots + b_{n-2} x + b_{n-1}.$$

If p is any approximation to s then $P_n(x) = (x-p)Q_{n-1}(x) + R$ where R is the residue which depends on p .

Now starting with p , we can use some iterative method to improve the value of p such that

$$R(p) = 0.$$

If we apply the Newton-Raphson method with a starting value p_0 , the iterative scheme can be written as

$$p_{i+1} = p_i - \frac{P_n(p_i)}{P'_n(p_i)}$$
$$i = 0, 1, 2, \dots$$

$$P'_n(p_i)$$

Now by comparing the coefficients of P_n and $(x-p)Q_{n-1}(x) + R$ we get

$$\begin{aligned}
 a_1 &= b_1 - p & b_1 &= a_1 + p \\
 a_2 &= b_2 - pb_1 & b_2 &= a_2 + pb_1 \\
 &\dots & & \\
 a_i &= b_i - pb_{i-1} & b_i &= a_i + pb_{i-1} \\
 &\dots & & \\
 a_n &= R - pb_{n-1} & R = b_n &= a_n + pb_{n-1}
 \end{aligned}$$

(or)

$$b_i = a_i + pb_{i-1} \quad i=1,2,\dots,n$$

with $b_0=1$ and $R = b_n = P_n(p)$

To find $P'_n(p)$, let us differentiate the equation

$$b_i = a_i + pb_{i-1}$$

with respect to p

$$p \quad db_i / dp = b_{i-1} + p (db_{i-1} / dp)$$

if we substitute $(db_i / dp) = c_{i-1}$ then

$$p \quad c_{i-1} = b_{i-1} + pc_{i-2} \quad (\text{or})$$

$$c_i = b_i + pc_{i-1} \quad i=1, 2, \dots, n-1$$

Then the c_{n-1} obtained from the last equation is nothing but

$$c_{i-1} = db_n / dp = dR / dp = P'_n(p)$$

That is the Newton's method now can be written as

$$p_{i+1} = p_i - b_n / c_{n-1}$$

On convergence this iterative process will give one root p of the polynomial equation $P_n(x) = 0$. Now the deflated polynomial equation $Q_{n-1}(x) = 0$ can be used to find the other real roots.

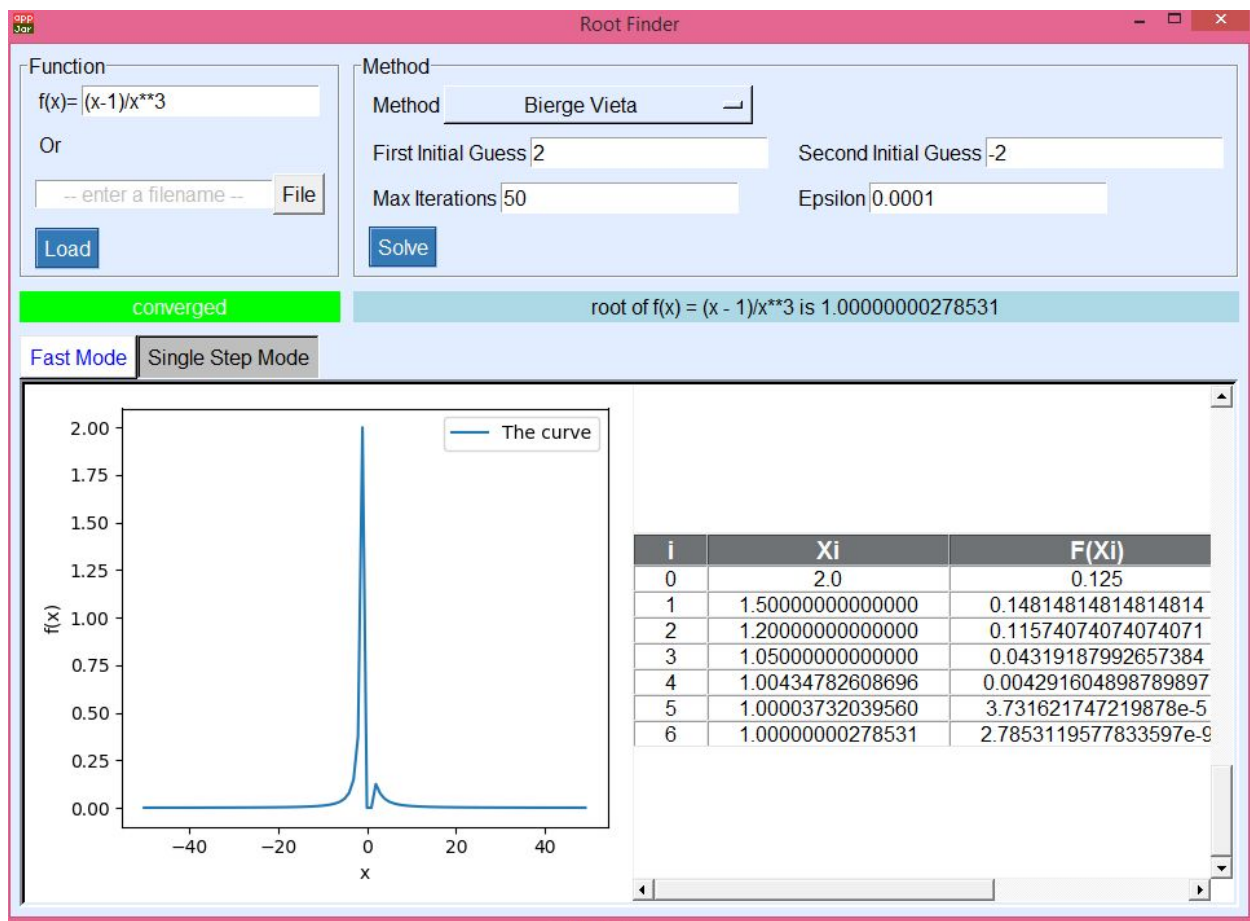
This method is often called as Birge-Vieta method.

Pseudocode

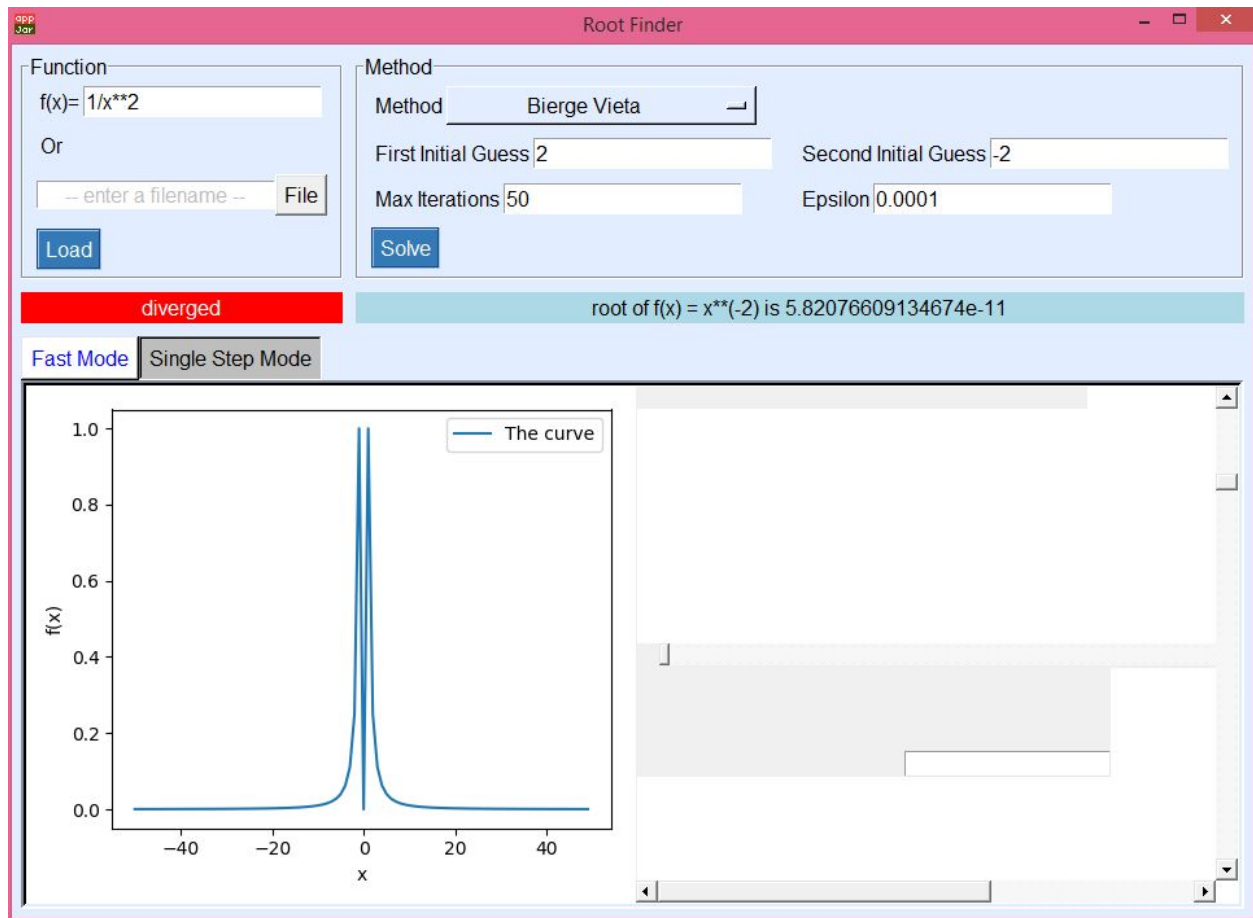
- Make sure the function is not a multi polynomial
- Extract coefficients from it.
- Copy the higher coff into the beging of b and c arrays
- Iterate over all coffs and calculate the b's and c's as explained above.
- Calculate the new value of x to be $x_{\text{new}} = x_{\text{old}} - (b_0 / c_1)$.

Sample Runs

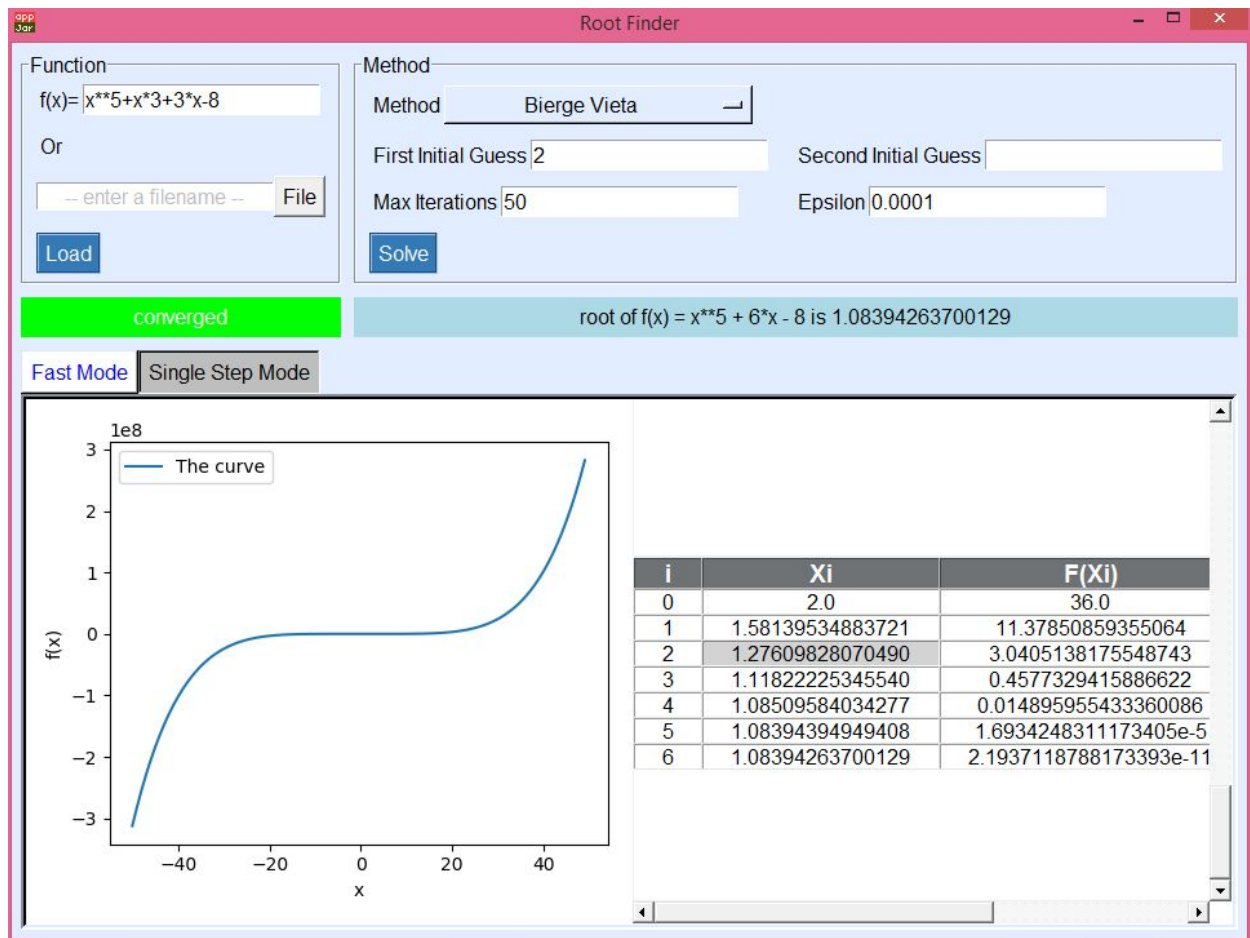
Run #1:



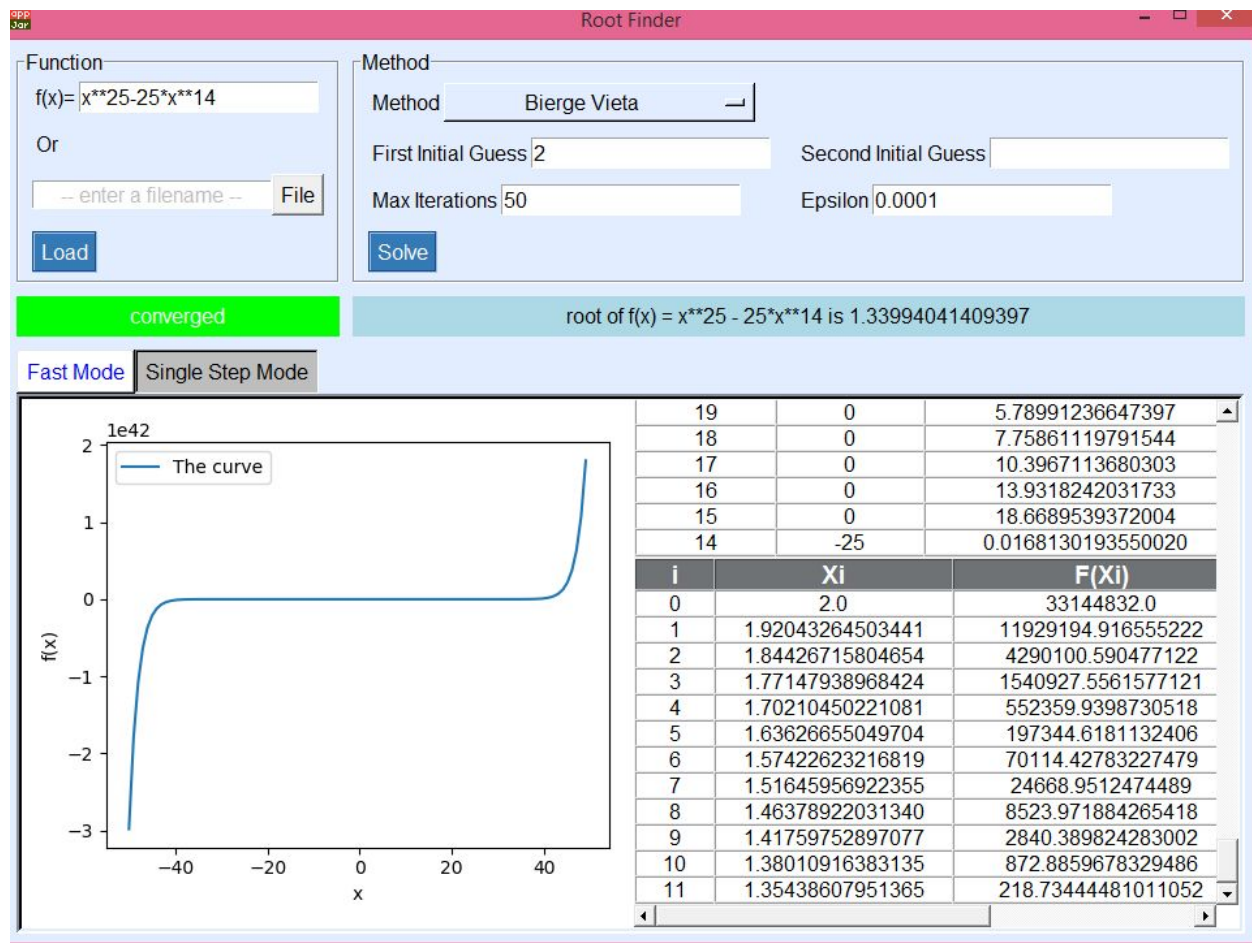
Run #2:



Run #3:



Run #4:



Comments

Used built-in functions for all methods:

Sympy.* :


It is used for using variable x in calculations and have all arithmetic operations upon it.

Math:

It is used for having mathematical modules such as $\text{ceil}()$, $\text{abs}()$, etc...

Numpy.arange:

It is used for having evenly spaced values within a given interval.



We stored coefficients of b but didn't need to store coefficients of c as they are not used for the next step. We constructed a table to store a and b.

d) General Algorithm

Introduction

This mode runs by being given a function to compute as much as possible of its roots, it's a combine of two methods introduced in this report in such a way to come over the disadvantage of each method.

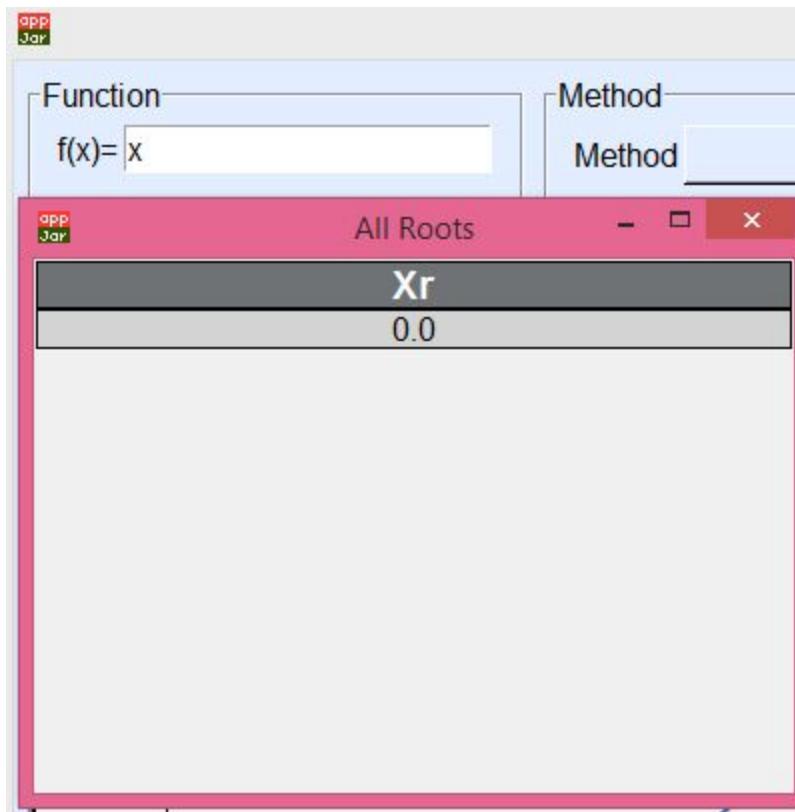
Newton and bisection methods are used (An iterative method along with a bracketing one)

Pseudocode

- generate a random initial guess and use newton method to compute a root.
- Divide the function by the found root factor
- Generate another one until all roots are found or a max of 100 trial reached or number of found roots exceeds the difference between the numerator and denominator order.
- Exclude any non valid root
- Perform a by section 2000 times to find the critical roots by stepping into all integers from -1000 to 1000 with a lower bound of step - uniformly random generated number and an upper bound of step + uniformly random generated number
- Combine the result of one method to other dominant method depending on the nature of the function.

Sample Runs

Run #1:



Run #2:

The screenshot shows a Java application window titled "All Roots". The main window has a pink title bar and a light blue background. It contains two input fields: "Function" and "Method". The "Function" field contains the text $f(x) = (x-1)*(x-5)$. The "Method" field is empty. Below the input fields, there is a table with the following data:

| Xr |
|-----|
| 1.0 |
| 5.0 |

Run #3:

The screenshot shows a Java application window titled "All Roots". The main window has a pink title bar and a light blue background. It contains two input fields: "Function" and "Method". The "Function" field contains the text $f(x) = (x-1)*(x-5)*x$. The "Method" field is empty. Below the input fields, there is a table with the following data:

| Xr |
|-----|
| 5.0 |
| 1.0 |
| 0.0 |

Run #4:

app.jar

Function
 $f(x) = (x-3) \cdot (x-1.92)^2$

Or

Method
Method
First Initial Guess

app.jar All Roots

| Xr |
|--------------------|
| 3.0 |
| 1.9199999999988384 |
| 1.919999999996484 |

Part II: Numerical Interpolation

Introduction

The problem of interpolation states that given a set of points about a certain function, we are to anticipate the location of a point in-between using numerical methods.

a) Lagrange Interpolation Method

Introduction

$$L_n(x) = \sum_{i=0}^n y_i \prod_{k=0, k \neq i}^n \frac{(x - x_k)}{(x_i - x_k)} = \sum_{i=0}^n y_i \frac{(x - x_0) \cdots (x - x_{i-1})(x - x_{i+1}) \cdots (x - x_n)}{(x_i - x_0) \cdots (x_i - x_{i-1})(x_i - x_{i+1}) \cdots (x_i - x_n)}.$$

- 1- The formula depend on (n+1) points where x_i are all different.
- 2- The purpose of this formula to determine a unique polynomial of degree n (or lower if the interpolating points lie on specific curve).
- 3- The function came from this formula is linear combination of distinct functions.



Pseudocode

- 1- Get all points.
- 2- Get all queries.
- 3- Get all Lagrange functions as shown in formula.
- 4- Get polynomial from Lagrange functions.
- 5- Plot the function through X_0 to X_n by interpolating many points with suitable step.
- 6- Plot all points (X_0, Y_0) to (X_n, Y_n) with special symbol to indicate the original points.
- 7- Plot all queries by interpolate them to get the corresponding (Y) value and identify them with special symbol.
- 8- Show plot window.

Sample Run

Example

A robot arm with a rapid laser scanner is doing a quick quality check on holes drilled in a rectangular plate. The hole centers in the plate that describe the path the arm needs to take are given below.

If the laser is traversing from $x = 2$ to $x = 4.25$ in a linear path, find the value of y at $x = 4$ using the Newton's Divided Difference method for a fifth order polynomial.

| x (m) | y (m) |
|---------|---------|
| 2 | 7.2 |
| 4.25 | 7.1 |
| 5.25 | 6.0 |
| 7.81 | 5.0 |
| 9.2 | 3.5 |
| 10.6 | 5.0 |

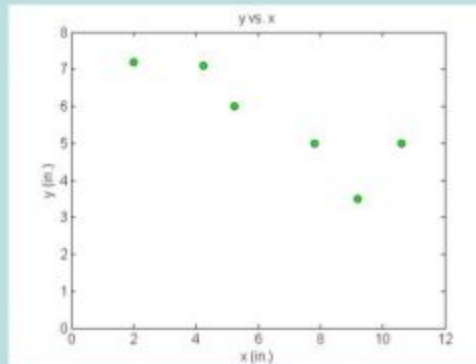


Figure 2 Location of holes on the rectangular plate.

27

Interpolation Calculator

Sample Points

x = 10.6

y = 5

add point

| x | y | Action |
|------|-----|--------|
| 2.0 | 7.2 | delete |
| 4.25 | 7.1 | delete |
| 5.25 | 6.0 | delete |
| 7.81 | 5.0 | delete |
| 9.2 | 3.5 | delete |
| 10.6 | 5.0 | delete |

Queries

x (query) = 10.6

add query

| x | Action |
|------|--------------|
| 2.0 | delete query |
| 4.25 | delete query |
| 5.25 | delete query |
| 7.81 | delete query |
| 9.2 | delete query |
| 10.6 | delete query |

Interpolation Method

Polynomial Order 5

Method Lagrange

Or

-- enter a filename --

File

load

Interpolate

generate plot

Interpolation Result

$$P(x) = 0.007x^5 - 0.2x^4 + 2.79x^3 - 15.85x^2 + 41.34x - 30.9$$

execution time = 1.410329818725586 s

$$\{L_0(x) = -0.0004x^5 + 0.01x^4 - 0.2x^3 + 1.43x^2 - 4.89x + 6.46\}$$

$$\{L_1(x) = 0.004x^5 - 0.1x^4 + 1.84x^3 - 11.44x^2 + 32.45x - 31.76\}$$

$$\{L_2(x) = -0.006x^5 + 0.2x^4 - 2.46x^3 + 14.63x^2 - 39.26x + 36.82\}$$

$$\{L_3(x) = 0.005x^5 - 0.2x^4 + 1.78x^3 - 9.66x^2 + 23.92x - 21.19\}$$

$$\{L_4(x) = -0.004x^5 + 0.1x^4 - 1.23x^3 + 6.41x^2 - 15.48x + 13.49\}$$

$$\{L_5(x) = 0.0009x^5 - 0.02x^4 + 0.3x^3 - 1.38x^2 + 3.27x - 2.81\}$$

Interpolation Calculator

Sample Points

x = 10.6

y = 5

add point

| x | y | Action |
|------|-----|--------|
| 2.0 | 7.2 | delete |
| 4.25 | 7.1 | delete |
| 5.25 | 6.0 | delete |
| 7.81 | 5.0 | delete |
| 9.2 | 3.5 | delete |
| 10.6 | 5.0 | delete |

Queries

x (query) = 10.6

add query

| x | Action |
|------|--------------|
| 2.0 | delete query |
| 4.25 | delete query |
| 5.25 | delete query |
| 7.81 | delete query |
| 9.2 | delete query |
| 10.6 | delete query |

Interpolation Method

Polynomial Order 5

Method Lagrange

Or

-- enter a filename --

File

load

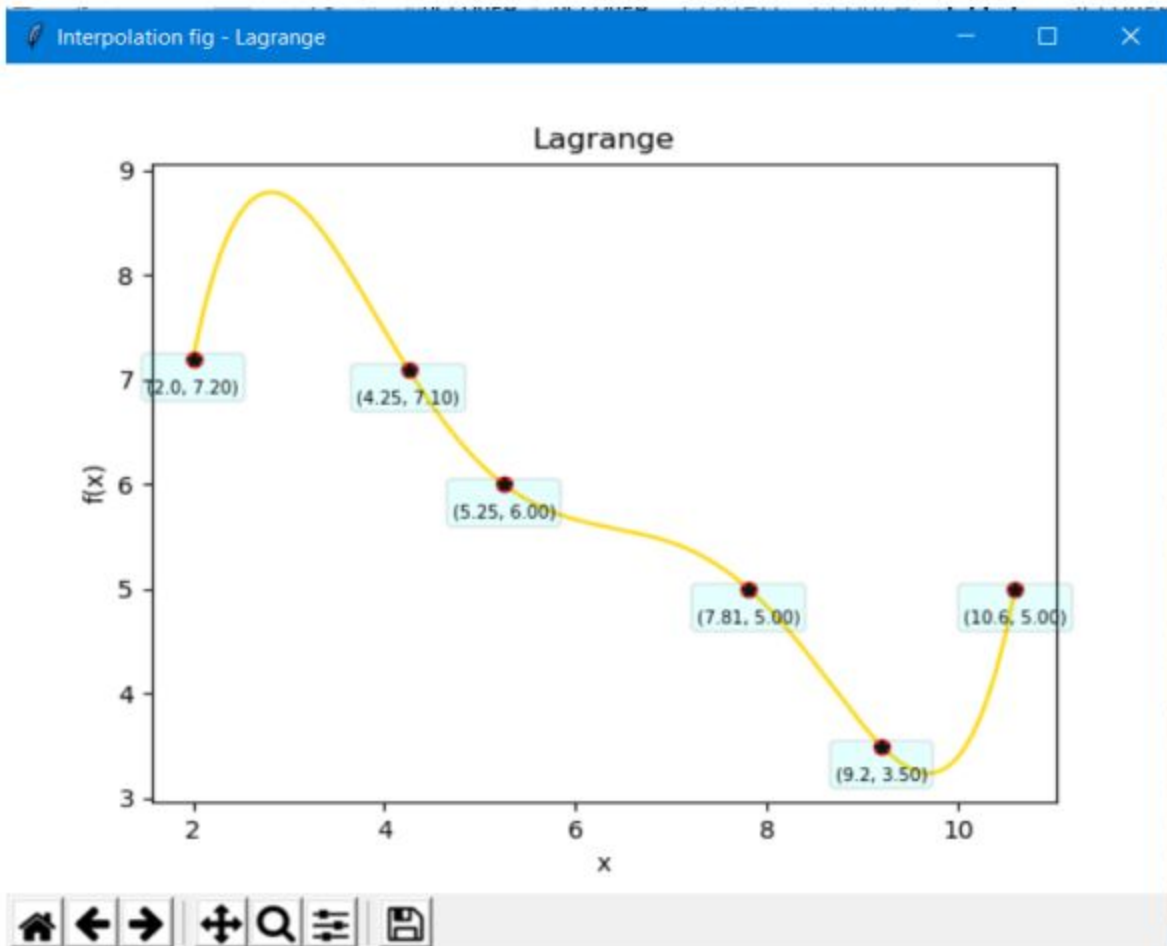
Interpolate

generate plot

Interpolation Result

| x (query) | y |
|-----------|-----|
| 2.0 | 7.2 |
| 4.25 | 7.1 |
| 5.25 | 6.0 |
| 7.81 | 5.0 |
| 9.2 | 3.5 |
| 10.6 | 5.0 |

40



Example:

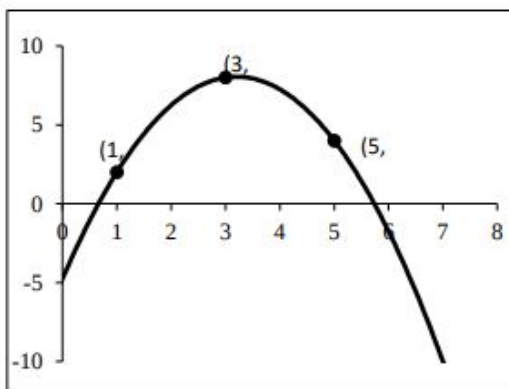
Using interpolating points: $\{(1,2), (3,8), (5,4)\}$.

From Lagrange Interpolating Method:

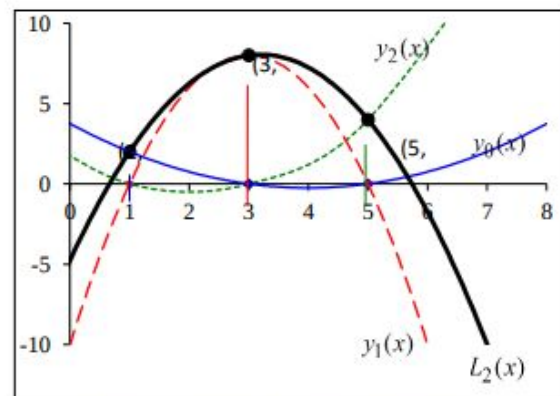
$$L_2(x) = 2 \frac{(x-3)(x-5)}{(1-3)(1-5)} + 8 \frac{(x-1)(x-5)}{(3-1)(3-5)} + 4 \frac{(x-1)(x-3)}{(5-1)(5-3)} \quad (1)$$

$$= \frac{1}{4}(x-3)(x-5) - 2(x-1)(x-5) + \frac{1}{2}(x-1)(x-3). \quad (2)$$

This function is a linear combination of three distinct quadratic functions



The quadratic interpolating polynomial



The three component quadratic functions

b) Newton Interpolation

Introduction

Also called the divided difference method.

$$f_n(x) = b_0 + b_1(x - x_0) + b_2(x - x_0)(x - x_1) + \cdots + b_n(x - x_0)(x - x_1) \cdots (x - x_{n-1})$$

$$b_0 = f(x_0) \quad b_1 = f[x_1, x_0] \quad b_2 = f[x_2, x_1, x_0] \quad \dots \quad b_n = f[x_n, x_{n-1}, \dots, x_1, x_0]$$

$$f[x_i, x_j] = \frac{f(x_i) - f(x_j)}{x_i - x_j}$$

$$f[x_i, x_j, x_k] = \frac{f[x_i, x_j] - f[x_j, x_k]}{x_i - x_k}$$

- 1- The formula depend on (n+1) points where x_i are all different.
- 2- The purpose of this formula to determine a unique polynomial of degree n (or lower if the interpolating points lie on specific curve).
- 3- The function is built upon divided differences.

Pseudocode

- 1- Get all points.
- 2- Get all queries.
- 3- Get all divided differences as shown in formula
- 4- Get polynomial from divided differences.

5- Plot the function through X_0 to X_n by interpolating many points

with suitable step.

6- Plot all points (X_0, Y_0) to (X_n, Y_n) with special symbol to indicate the original points.

7- Plot all queries by interpolate them to get the corresponding (Y) value and identify them with special symbol.

8- Show plot window.

Example:

Using interpolating points: $\{(0, 2), (2, 14), (3, 74), (4, 242), (5, 602)\}$. From divided method:

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

So we get b's values from the diagonal: $b_0 = 2, b_1 = 6, b_2 = 18, b_3 = 9, b_4 = 1$

$$P(x) = 2 + 6x + 18x(x-2) + 9x(x-2)(x-3) + 1x(x-2)(x-3)(x-4) = x^4 - x^2 + 2$$

Sample Run

Example

A robot arm with a rapid laser scanner is doing a quick quality check on holes drilled in a rectangular plate. The hole centers in the plate that describe the path the arm needs to take are given below.

If the laser is traversing from $x = 2$ to $x = 4.25$ in a linear path, find the value of y at $x = 4$ using the Newton's Divided Difference method for a fifth order polynomial.

| x (m) | y (m) |
|---------|---------|
| 2 | 7.2 |
| 4.25 | 7.1 |
| 5.25 | 6.0 |
| 7.81 | 5.0 |
| 9.2 | 3.5 |
| 10.6 | 5.0 |

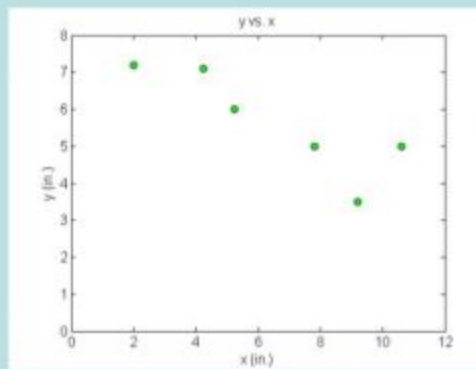


Figure 2 Location of holes on the rectangular plate.

27

Interpolation Calculator

Sample Points

x = 10.6

y = 5

add point

| x | y | Action |
|------|-----|--------|
| 2.0 | 7.2 | delete |
| 4.25 | 7.1 | delete |
| 5.25 | 6.0 | delete |
| 7.81 | 5.0 | delete |
| 9.2 | 3.5 | delete |
| 10.6 | 5.0 | delete |

Interpolation Method

Polynomial Order 5

Method Newton

Or

-- enter a filename --

File

load

Interpolate

generate plot

Queries

x (query) = 10.6

add query

| x | Action |
|------|--------------|
| 2.0 | delete query |
| 4.25 | delete query |
| 5.25 | delete query |
| 7.81 | delete query |
| 9.2 | delete query |
| 10.6 | delete query |

Interpolation Result

$$P(x) = 0.007x^5 - 0.2x^4 + 2.79x^3 - 15.85x^2 + 41.34x - 30.9$$

execution time = 0.7279634475708008 s

7.2 -0.04444444444444468

-0.3247863247863246 0.09019775649496777

-0.02300921134190442 0.007292341223225805

| x (query) | y |
|-----------|-------------------|
| 2.0 | 7.2 |
| 4.25 | 7.1 |
| 5.25 | 6.0 |
| 7.81 | 5.000000000000001 |

Interpolation Calculator

Sample Points

x = 10.6

y = 5

add point

| x | y | Action |
|------|-----|--------|
| 2.0 | 7.2 | delete |
| 4.25 | 7.1 | delete |
| 5.25 | 6.0 | delete |
| 7.81 | 5.0 | delete |
| 9.2 | 3.5 | delete |
| 10.6 | 5.0 | delete |

Interpolation Method

Polynomial Order 5

Method Newton

Or

-- enter a filename --

File

load

Interpolate

generate plot

Queries

x (query) = 10.6

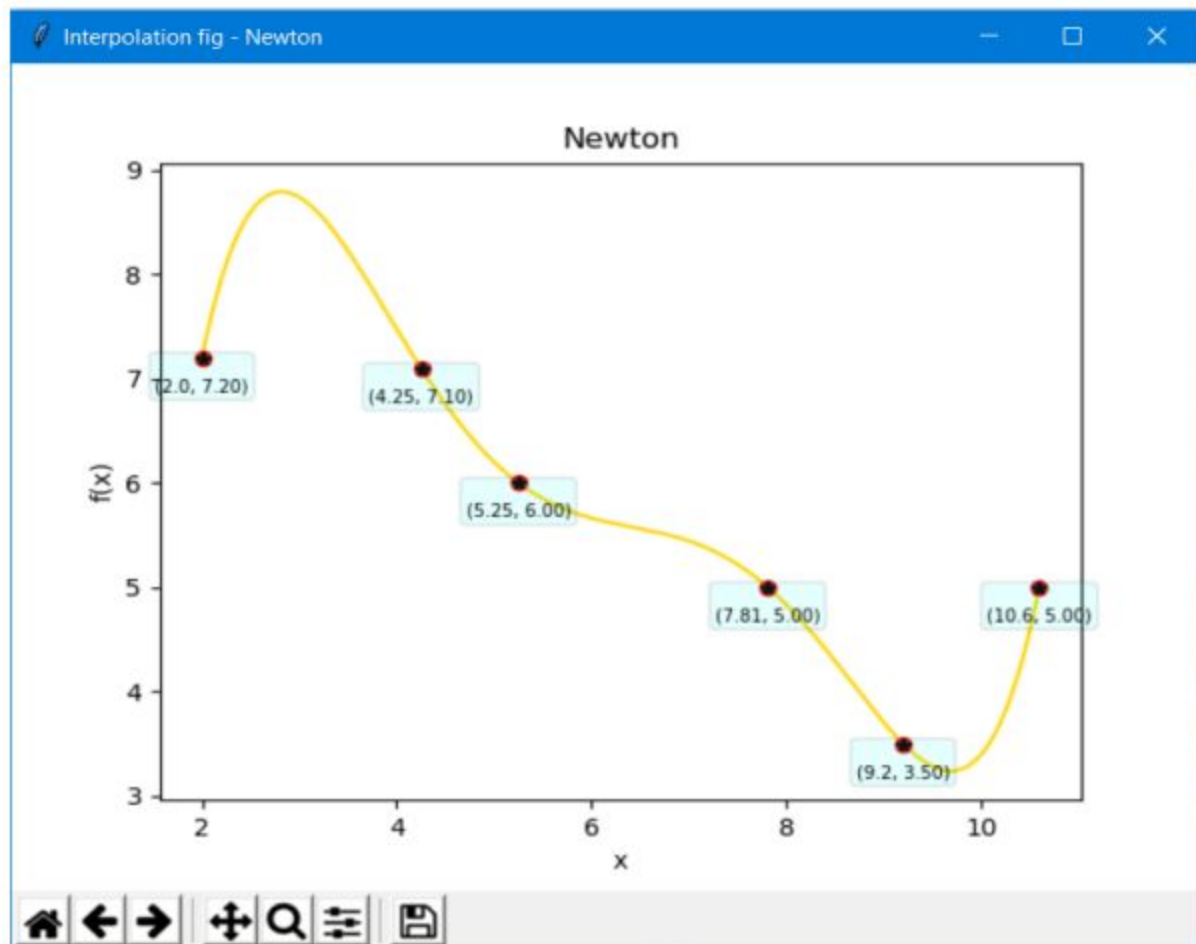
add query

| x | Action |
|------|--------------|
| 2.0 | delete query |
| 4.25 | delete query |
| 5.25 | delete query |
| 7.81 | delete query |
| 9.2 | delete query |
| 10.6 | delete query |

Interpolation Result

| x (query) | y |
|-----------|---------------------|
| 2.0 | 7.2 |
| 4.25 | 7.1 |
| 5.25 | 6.0 |
| 7.81 | 5.000000000000001 |
| 9.2 | 3.50000000000000027 |
| 10.6 | 5.0000000000000005 |

46



$$y(x) = -30.898 + 41.344x - 15.855x^2 + 2.7862x^3 - 0.23091x^4 + 0.0072923x^5, \quad 2 \leq x \leq 10.6$$



Used Built-in Functions:

1- `sympy.Symbol`:

It is used for using variable x in calculations and have all arithmetic operations upon it.

2- `sympy.Poly`:

It is used for store the result of operations that came from `sympy.Symbol` in polynomial form.

3- `matplotlib.pyplot`:

It is used for having a plotting for all points that came from the calculations of (Lagrange or Newton).

