CS213: Numerical Analysis and Computer Applications

Final Project Report

Techniques for Root Finding, Interpolation & Systems of Linear Equations

Team:

Ayman Ahmed Samy	15
Khaled Barie Mostafa	21
Khaled Abdelfattah	22
Abdelhakeem Osama	34
Mohammed Kamal Abd Elrahman	59

Index

Tools

We used python 3.6 to develop the project. We also used the following libraries:

- PyQt5: Used in creating the GUI.
- Matplotlib: For plotting all charts and graphs used throughout the application
- Sympy: For parsing input equations and differentiation.

Design

We used some simple design patterns in order to facilitate the process of development and keep the project maintainable for more features. We used an MVC architecture where separated the main logic of the methods implemented from the GUI the user interacts with. We also used the Factory design pattern in the root finding part of the program where the user selects whichever method they want to use and the results are generated through the root finder factory.

What we added

- Implementation for the modified Newton methods in the lectures:
  - First modified Newton method which requires multiplicity to be given
  - Second modified Newton method
- A module to solve systems of linear equations (Max: 10 equations) using Gauss-Jordan method

Assumptions

- In interpolation
  - we use all points supplied by the user to interpolate given query points regardless of the polynomial order.
  - Query points have to be within the range of the sample points given, otherwise an error message is shown.

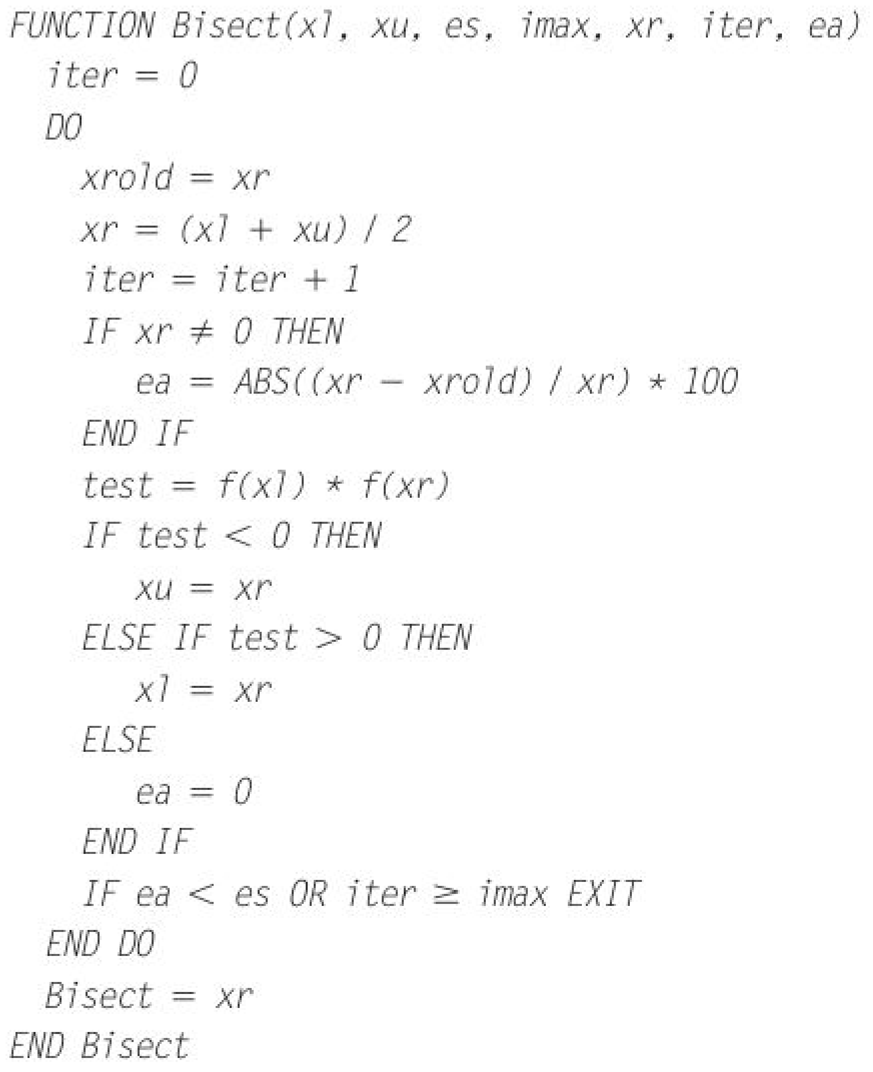
Part 1: Root-Finding

Objective

Implementing the following six root-finding methods as well as a general algorithm for finding nearly all the roots of a given function.

1.1. Implemented Algorithms

1.1.1. Bisection



```
FUNCTION Bisect(xl, xu, es, imax, xr, iter, ea)
  iter = 0
  DO
    xrold = xr
    xr = (xl + xu) / 2
    iter = iter + 1
    IF xr ≠ 0 THEN
       ea = ABS((xr − xrold) / xr) * 100
    END IF
    test = f(xl) * f(xr)
    IF test < 0 THEN
       xu = xr
    ELSE IF test > 0 THEN
       xl = xr
    ELSE
       ea = 0
    END IF
    IF ea < es OR iter ≥ imax EXIT
  END DO
  Bisect = xr
END Bisect
```

1.1.2. False-Position

```
FUNCTION ModFalsePos(xl, xu, es, imax, xr, iter, ea)
  iter = 0
  fl = f(xl)
  fu = f(xu)
  DO
    xrold = xr
    xr = xu − fu * (xl − xu) / (fl − fu)
    fr = f(xr)
    iter = iter + 1
    IF xr <> 0 THEN
      ea = Abs((xr − xrold) / xr) * 100
    END IF
    test = fl * fr
    IF test < 0 THEN
      xu = xr
      fu = f(xu)
      iu = 0
      il = il + 1
      If il ≥ 2 THEN fl = fl / 2
    ELSE IF test > 0 THEN
      xl = xr
      fl = f(xl)
      il = 0
      iu = iu + 1
      IF iu ≥ 2 THEN fu = fu / 2
    ELSE
      ea = 0
    END IF
    IF ea < es OR iter ≥ imax THEN EXIT
  END DO
  ModFalsePos = xr
END ModFalsePos
```

1.1.3. Fixed-Point

```
FUNCTION Fixpt(x0, es, imax, iter, ea)
  xr = x0
  iter = 0
  DO
    xrold = xr
    xr = g(xrold)
    iter = iter + 1
    IF xr ≠ 0 THEN
      $ea = \left| \frac{xr - xrold}{xr} \right| \cdot 100$
    END IF
    IF ea < es OR iter ≥ imax EXIT
  END DO
  Fixpt = xr
END Fixpt
```

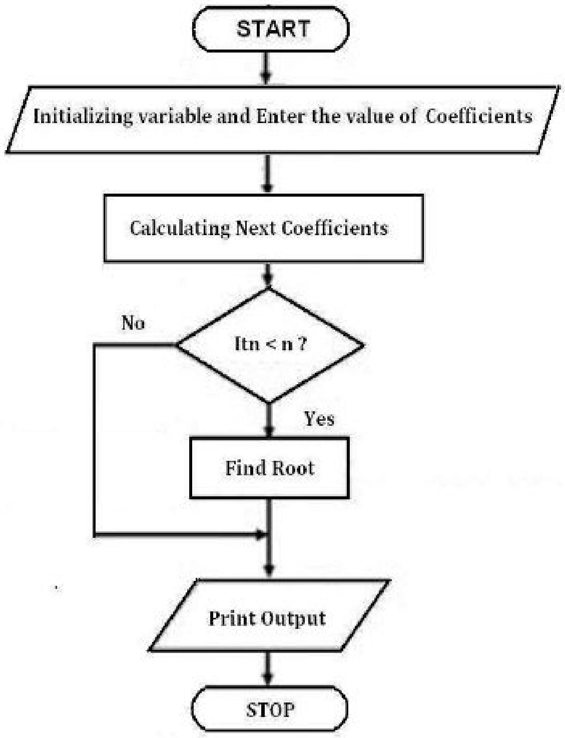
1.4. Newton-Raphson

```
1. Choose $\in > 0$ (function tolerance $|f(x)| < \in$)
          $m > 0$ (Maximum number of iterations)
          $x_0$ - initial approximation
          k - iteration count
   Compute $f(x_0)$
2. Do { $q = f'(x_0)$      (evaluate derivative at $x_0$)
        $x_1 = x_0 - f_0/q$
        $x_0 = x_1$
        $f_0 = f(x_0)$
        $k = k+1$
      }
3. While ($|f_0| \geq \in$) and ($k \leq m$)
4. $x = x_1$ the root.
```

1.1.5. Secant

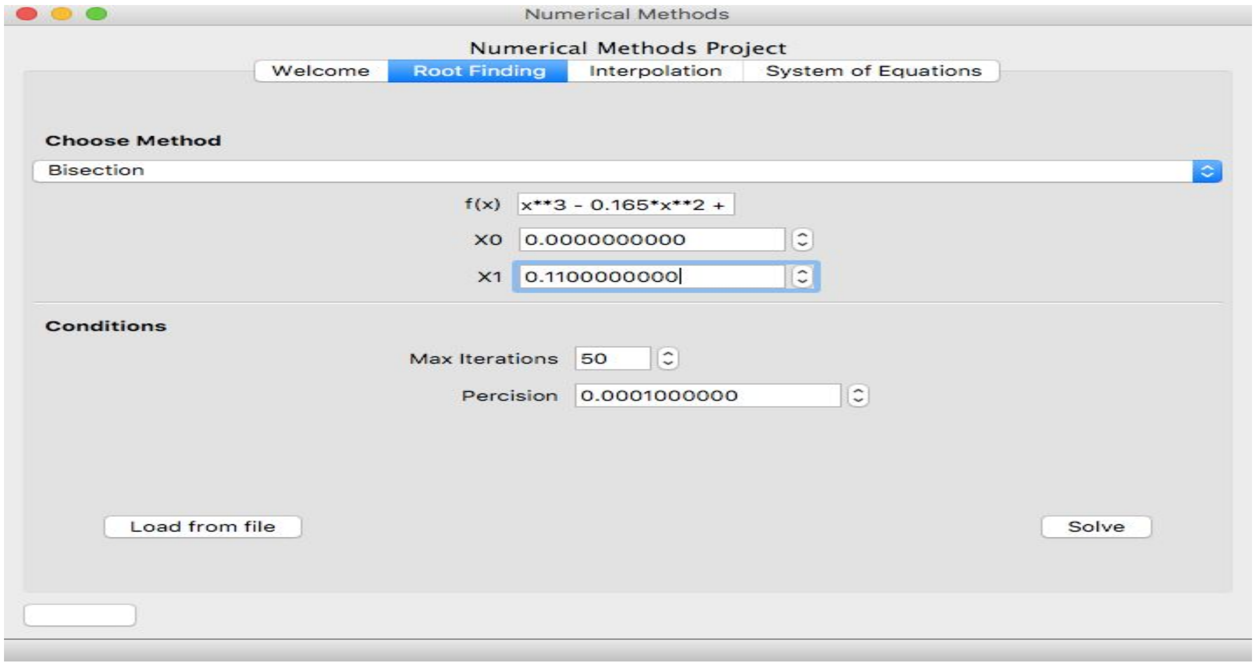
```
1.  Choose $\in > 0$ (function tolerance $|f(x)| \leq \in$ )
           $m > 0$ (Maximum number of iterations)
           $x_0$ , $x_1$ (Two initial points near the root )
           $f_0 = f(x_0)$
           $f_1 = f(x_1)$
           $k = 1$   (iteration count)
2.  Do  {
           $x_2 = x_1 - \left(\frac{x_1 - x_0}{f_1 - f_0}\right) f_1$
           $x_0 = x_1$
           $f_0 = f_1$
           $x_1 = x_2$
           $f_1 = f(x_2)$
           $k = k+1$          }
3.  While ($|f_1| \geq \in$) and ($m \leq k$)
```

1.1.6. Birge-Vieta


START
Initializing variable and Enter the value of Coefficients
Calculating Next Coefficients
No
Itn < n ?
Yes
Find Root
Print Output
STOP

Sample runs:

1.1. Bisection


Numerical Methods

Numerical Methods Project

Choose Method

Bisection

f(x) x**3 - 0.165*x**2 +

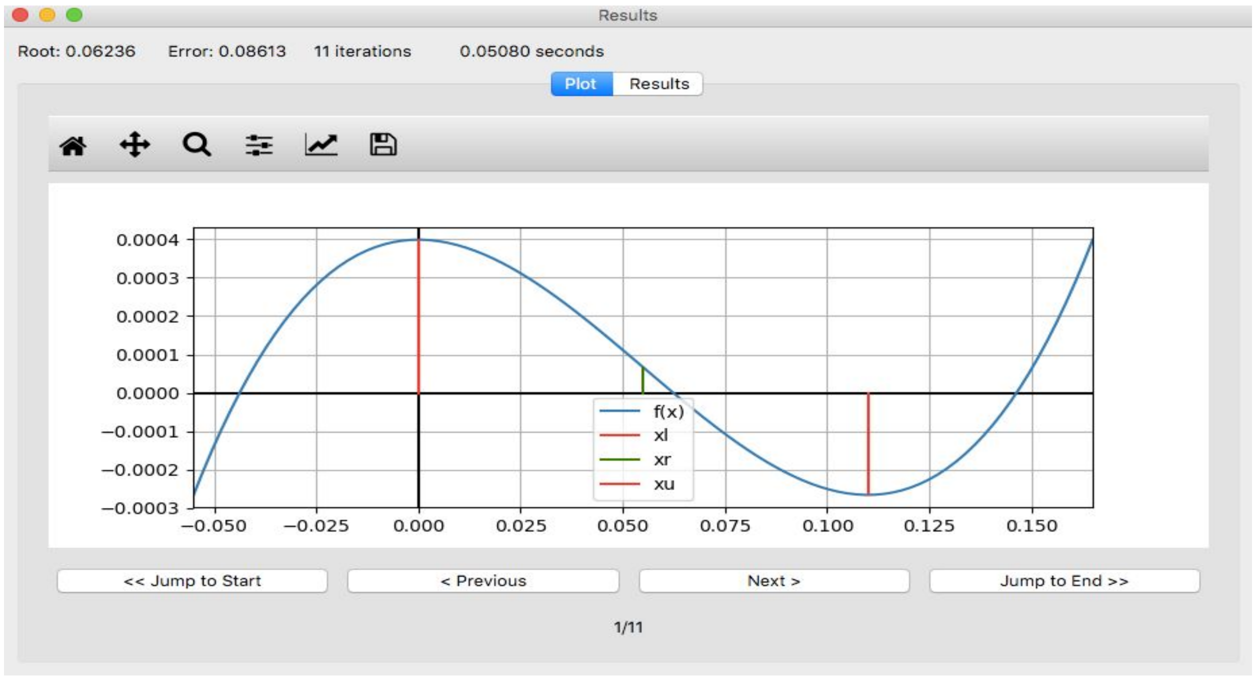
X0 0.0000000000

X1 0.1100000000

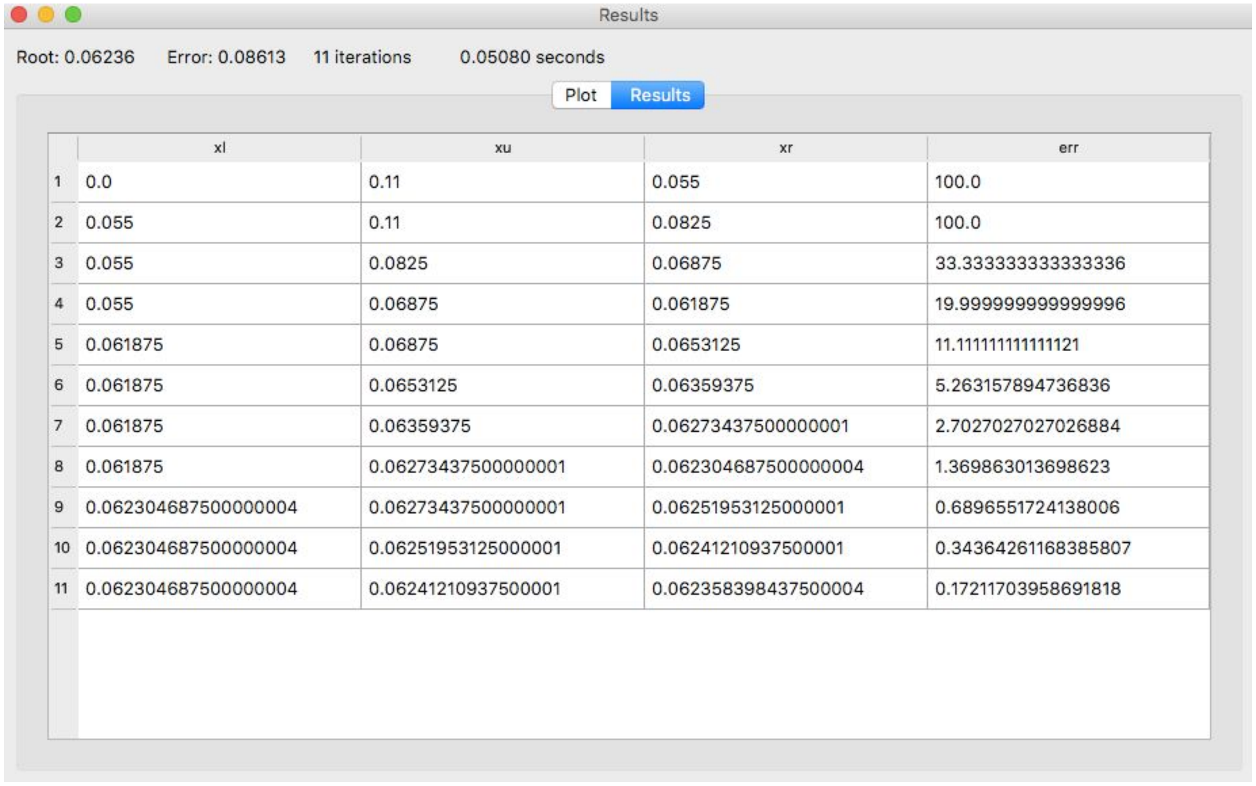
Conditions

Max Iterations 50

Percision 0.0001000000


Results

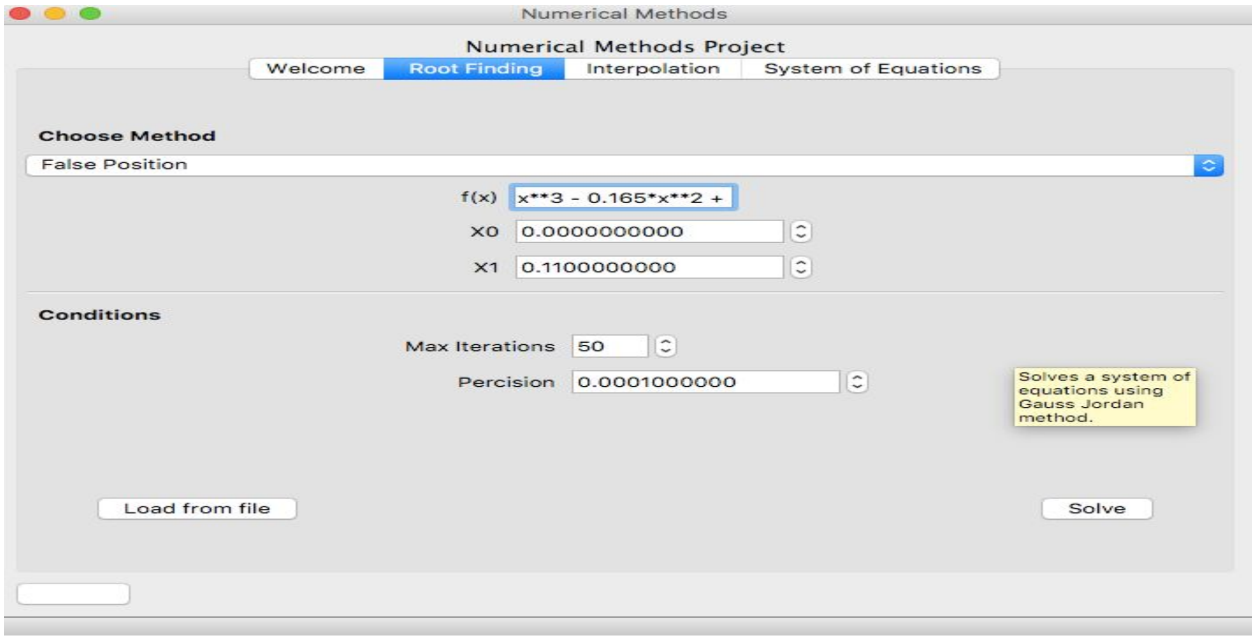
Root: 0.06236 Error: 0.08613 11 iterations 0.05080 seconds


Results

Root: 0.06236 Error: 0.08613 11 iterations 0.05080 seconds

	xl	xu	xr	err
1	0.0	0.11	0.055	100.0
2	0.055	0.11	0.0825	100.0
3	0.055	0.0825	0.06875	33.333333333333336
4	0.055	0.06875	0.061875	19.999999999999996
5	0.061875	0.06875	0.0653125	11.111111111111121
6	0.061875	0.0653125	0.06359375	5.263157894736836
7	0.061875	0.06359375	0.06273437500000001	2.7027027027026884
8	0.061875	0.06273437500000001	0.062304687500000004	1.369863013698623
9	0.062304687500000004	0.06273437500000001	0.06251953125000001	0.6896551724138006
10	0.062304687500000004	0.06251953125000001	0.06241210937500001	0.34364261168385807
11	0.062304687500000004	0.06241210937500001	0.062358398437500004	0.17211703958691818

1.2. False position



Choose Method

False Position

f(x) x**3 - 0.165*x**2 +

X0 0.0000000000

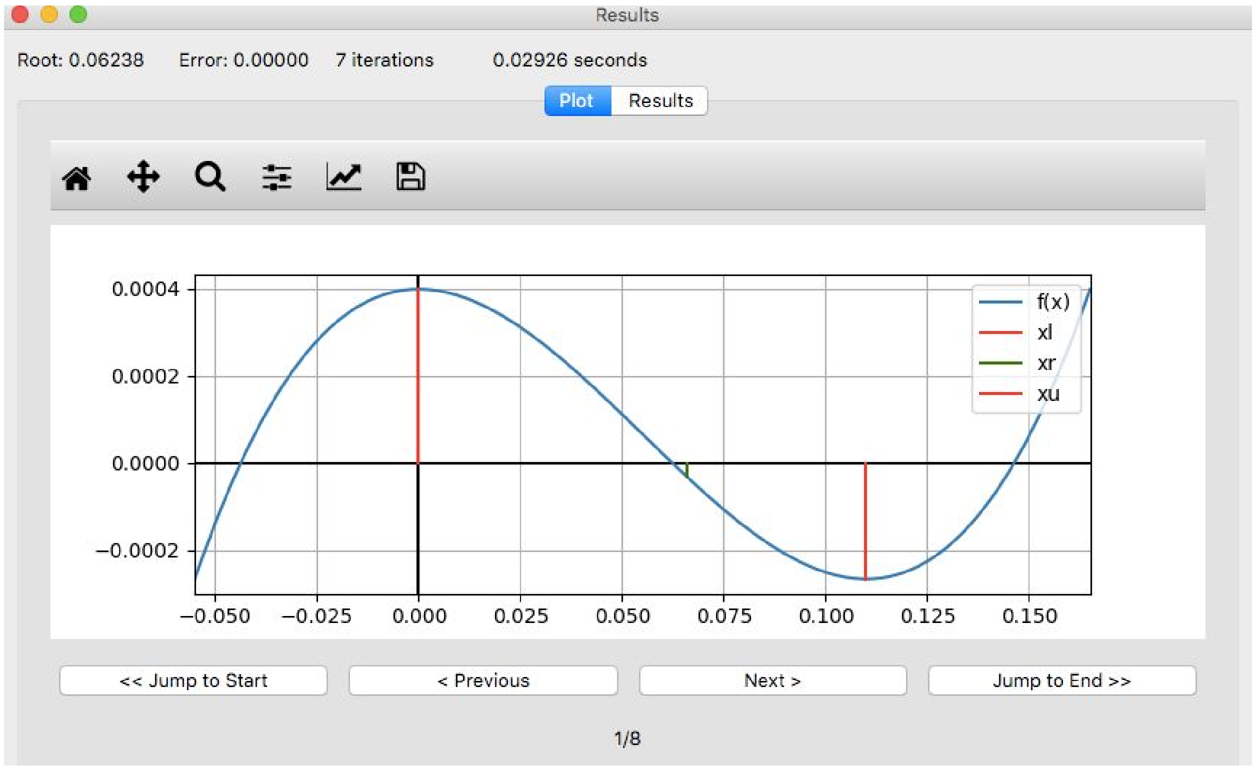
X1 0.1100000000

Conditions

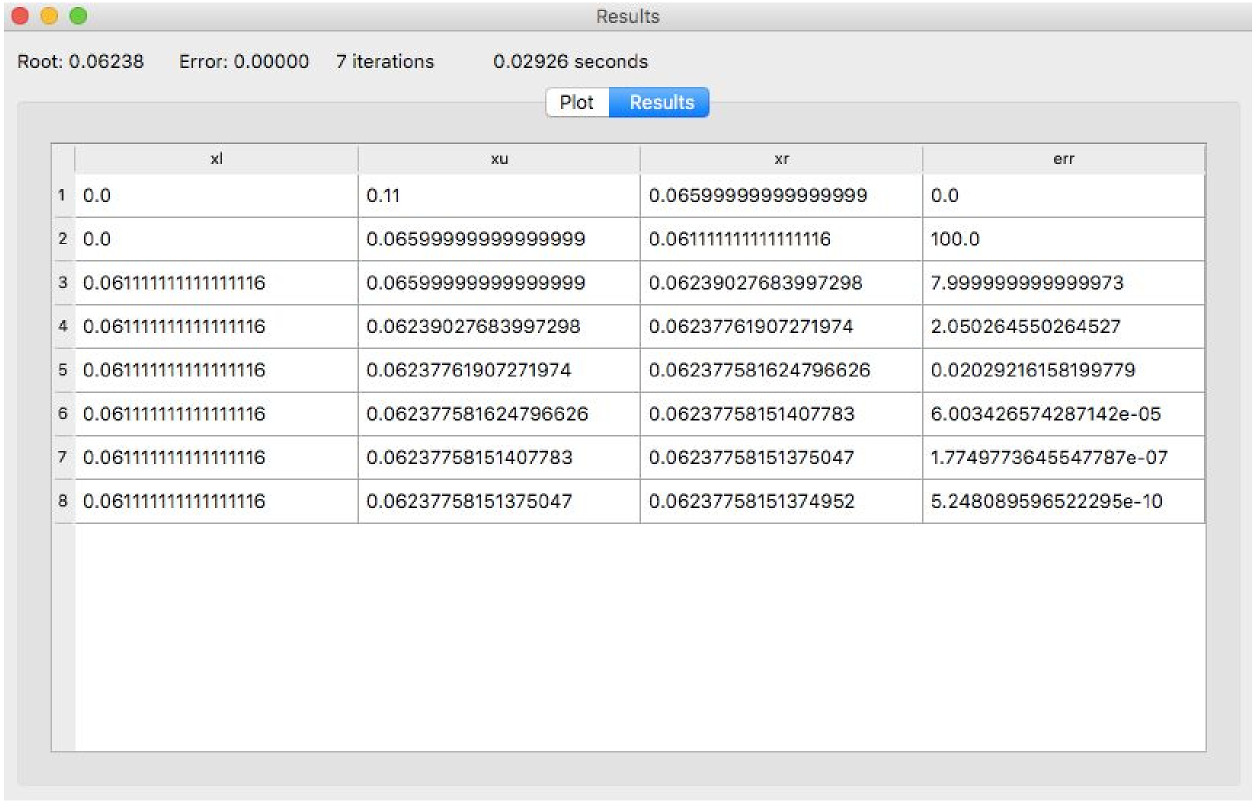
Max Iterations 50

Percision 0.0001000000

Solves a system of equations using Gauss Jordan method.


Results

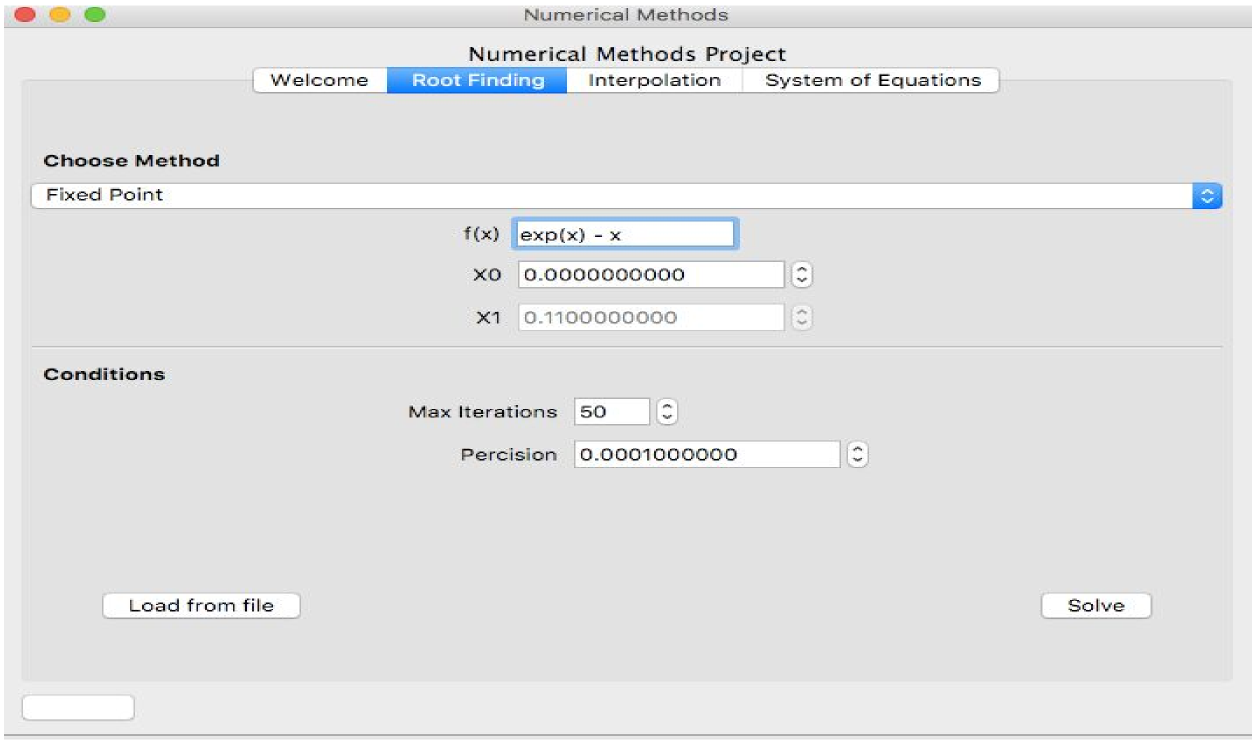
Root: 0.06238 Error: 0.00000 7 iterations 0.02926 seconds


Results

Root: 0.06238 Error: 0.00000 7 iterations 0.02926 seconds

	xl	xu	xr	err
1	0.0	0.11	0.06599999999999999	0.0
2	0.0	0.06599999999999999	0.061111111111111116	100.0
3	0.061111111111111116	0.06599999999999999	0.06239027683997298	7.999999999999973
4	0.061111111111111116	0.06239027683997298	0.06237761907271974	2.050264550264527
5	0.061111111111111116	0.06237761907271974	0.062377581624796626	0.020292161581997779
6	0.061111111111111116	0.062377581624796626	0.06237758151407783	6.003426574287142e-05
7	0.061111111111111116	0.06237758151407783	0.06237758151375047	1.7749773645547787e-07
8	0.061111111111111116	0.06237758151375047	0.06237758151374952	5.248089596522295e-10

1.3. Fixed point



Choose Method

Fixed Point

f(x) exp(x) - x

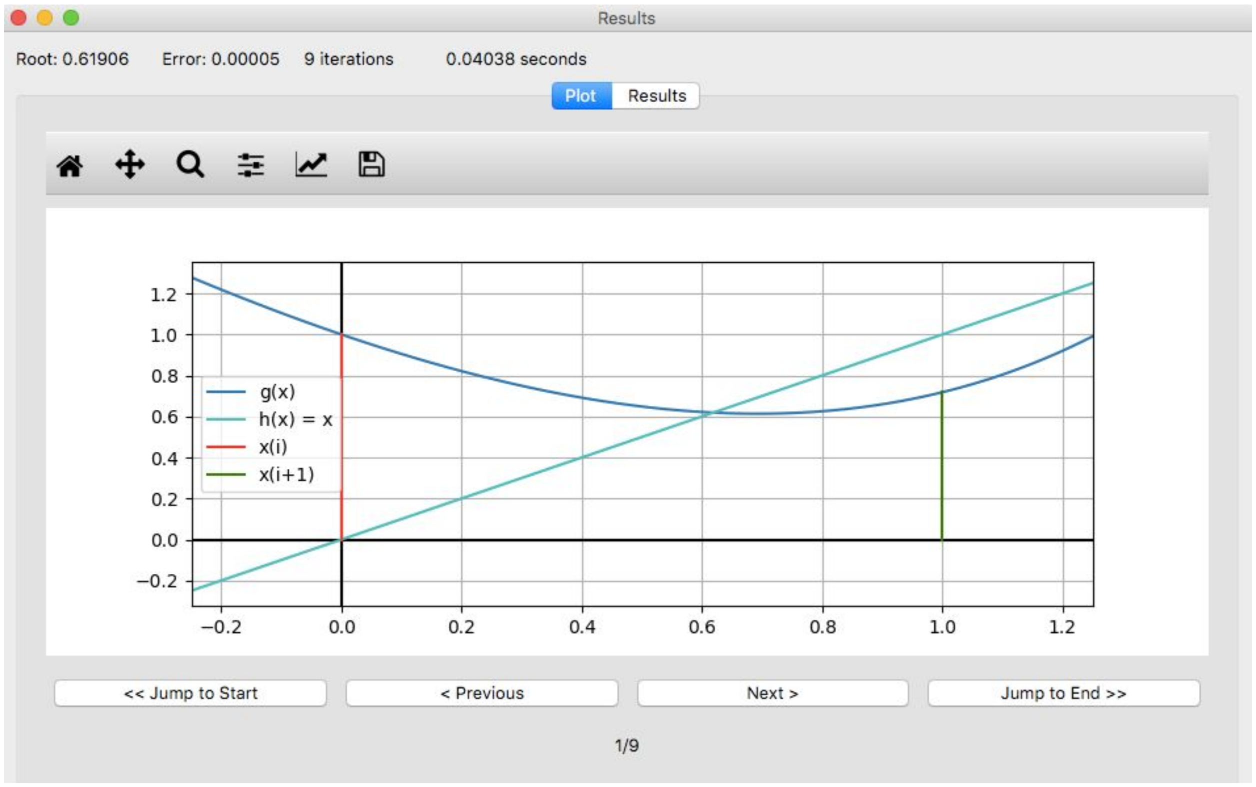
X0 0.0000000000

X1 0.1100000000

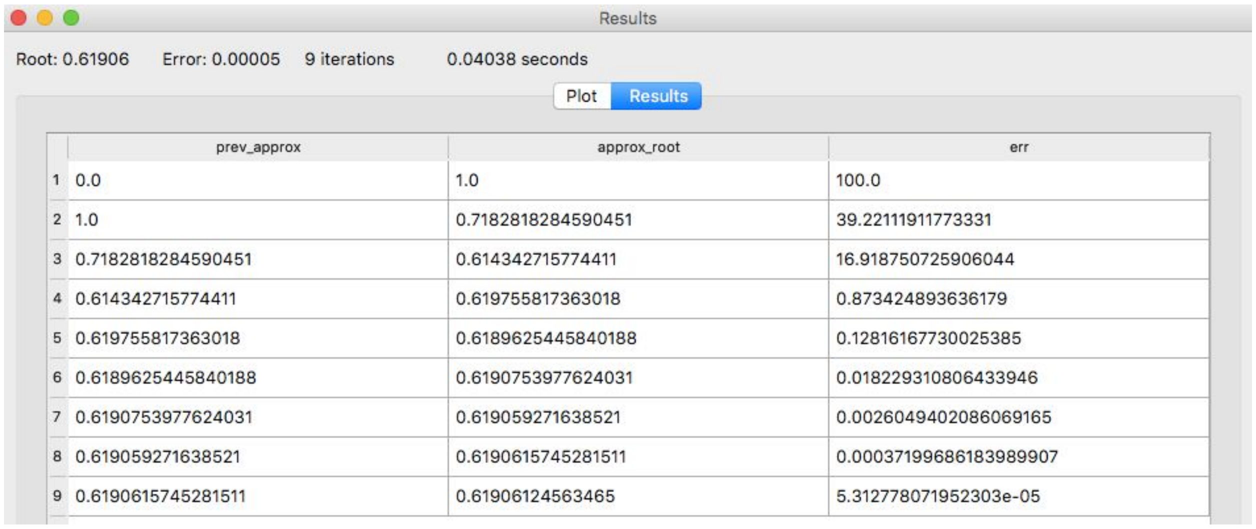
Conditions

Max Iterations 50

Percision 0.0001000000



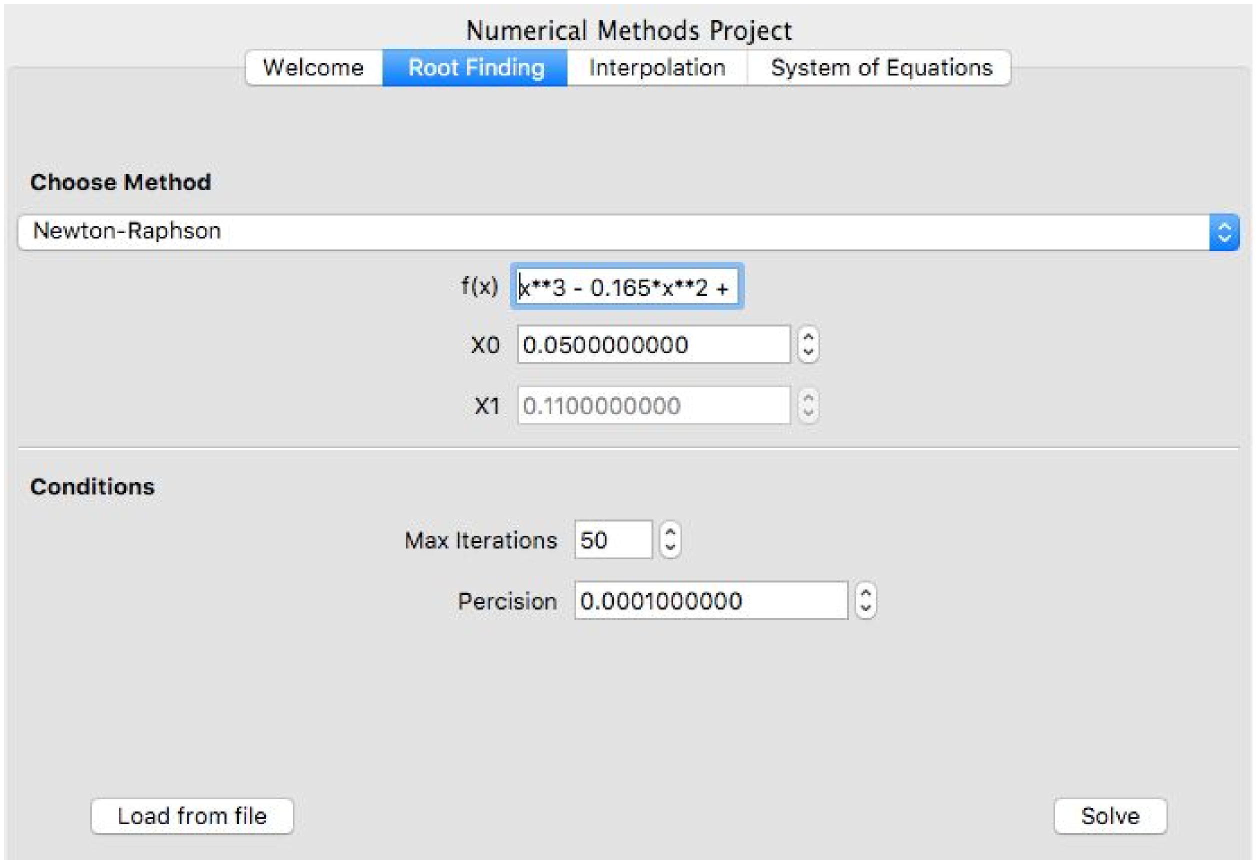
Root: 0.61906 Error: 0.00005 9 iterations 0.04038 seconds



Root: 0.61906 Error: 0.00005 9 iterations 0.04038 seconds

	prev_approx	approx_root	err
1	0.0	1.0	100.0
2	1.0	0.7182818284590451	39.22111911773331
3	0.7182818284590451	0.614342715774411	16.918750725906044
4	0.614342715774411	0.619755817363018	0.873424893636179
5	0.619755817363018	0.6189625445840188	0.12816167730025385
6	0.6189625445840188	0.6190753977624031	0.018229310806433946
7	0.6190753977624031	0.619059271638521	0.0026049402086069165
8	0.619059271638521	0.6190615745281511	0.00037199686183989907
9	0.6190615745281511	0.61906124563465	5.312778071952303e-05

1.4.1. Newton Raphson



Choose Method

Newton-Raphson

f(x) x**3 - 0.165*x**2 +

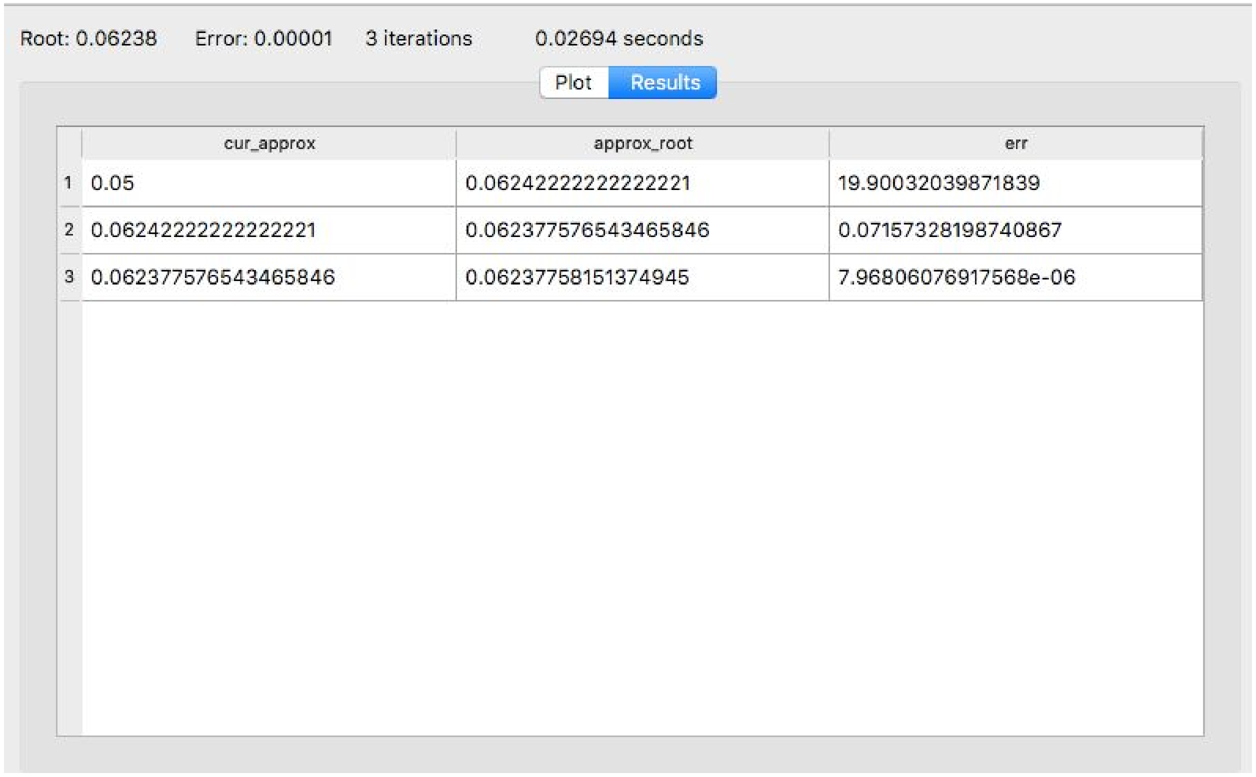
X0 0.0500000000

X1 0.1100000000

Conditions

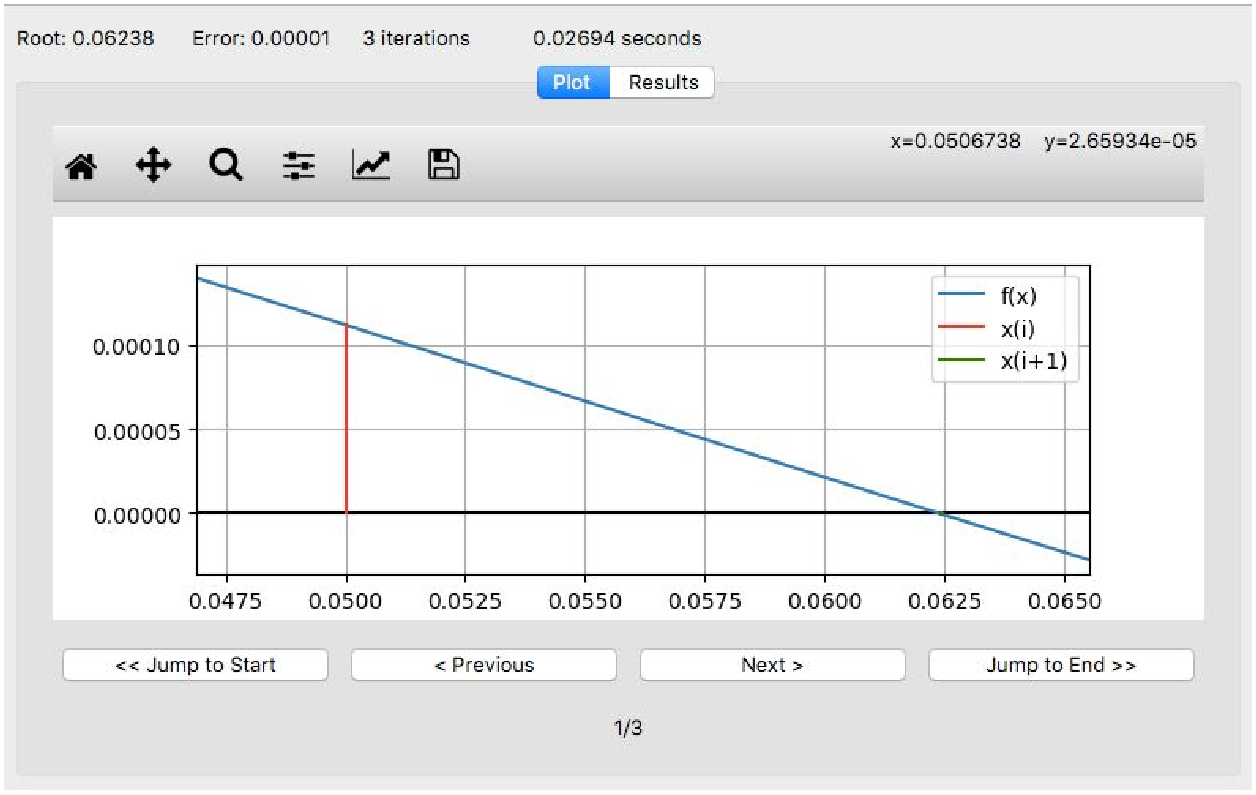
Max Iterations 50

Percision 0.0001000000



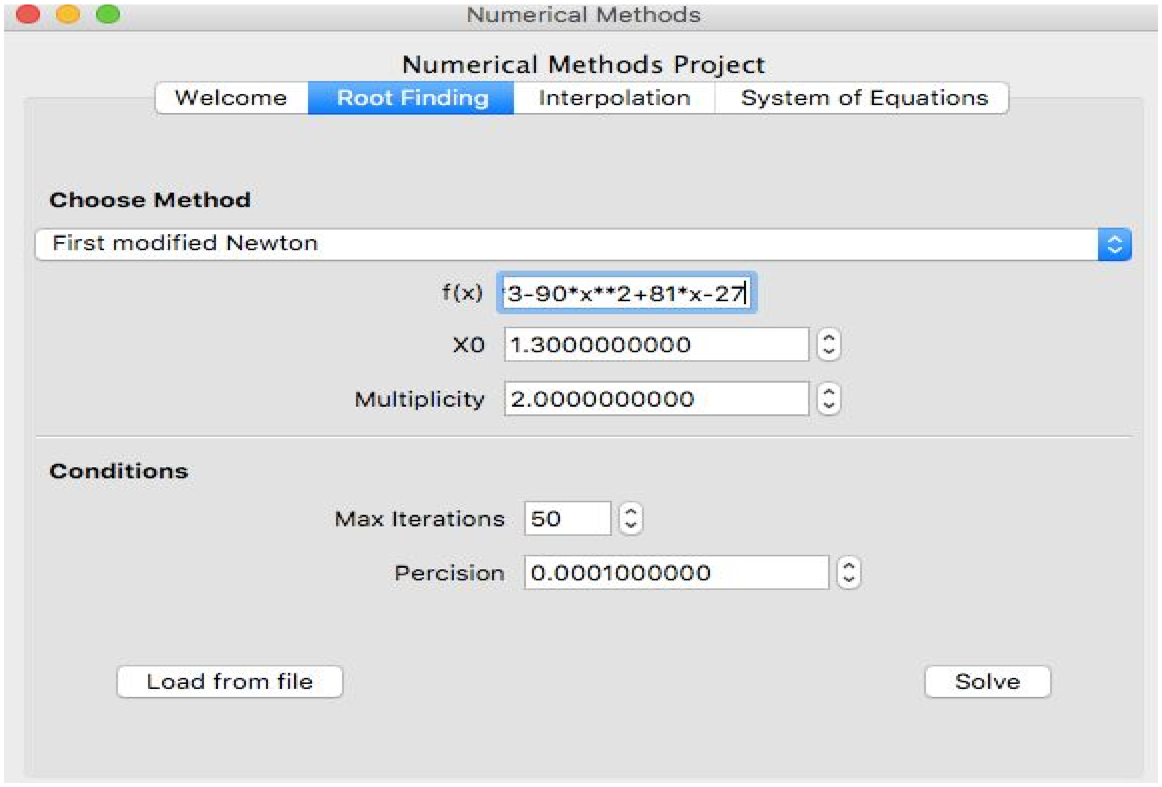
Root: 0.06238 Error: 0.00001 3 iterations 0.02694 seconds

	cur_approx	approx_root	err
1	0.05	0.06242222222222221	19.90032039871839
2	0.06242222222222221	0.062377576543465846	0.07157328198740867
3	0.062377576543465846	0.06237758151374945	7.96806076917568e-06



Root: 0.06238 Error: 0.00001 3 iterations 0.02694 seconds

1.4.2. First modified Newton Raphson


Numerical Methods

Numerical Methods Project

Choose Method

First modified Newton

f(x) 3-90*x**2+81*x-27

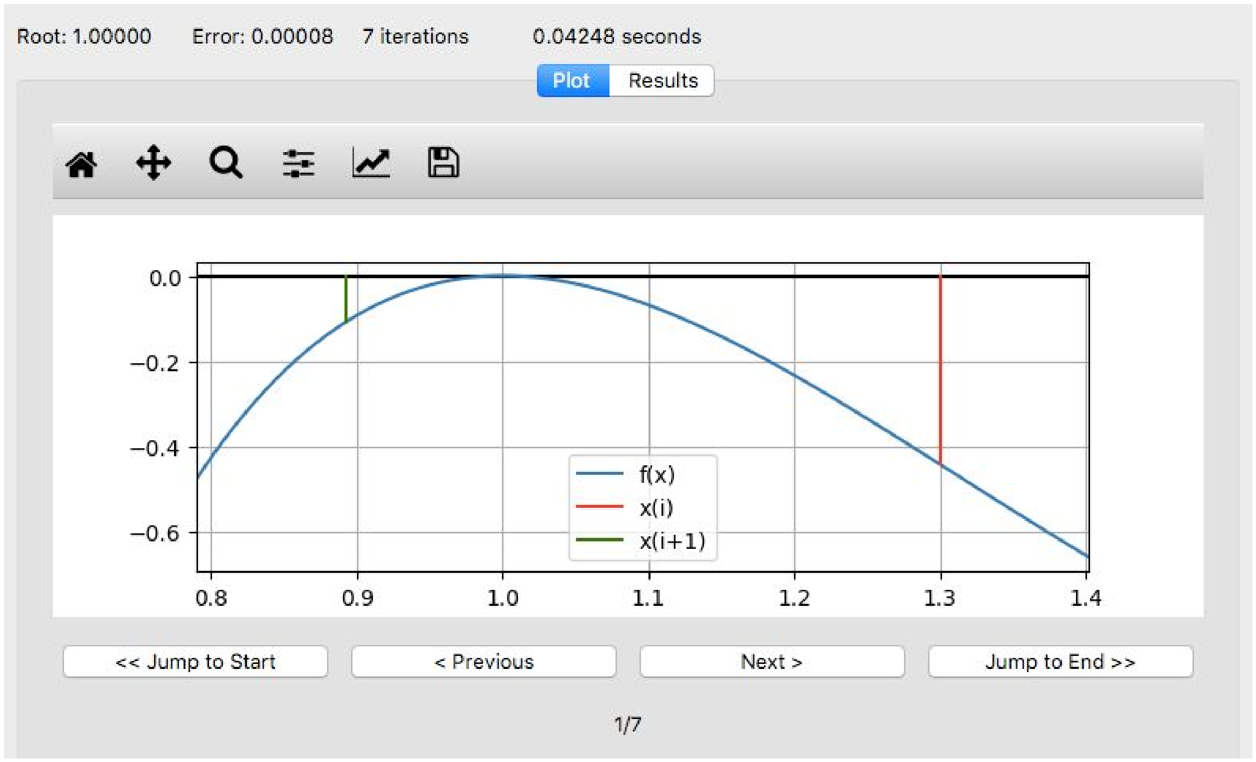
X0 1.3000000000

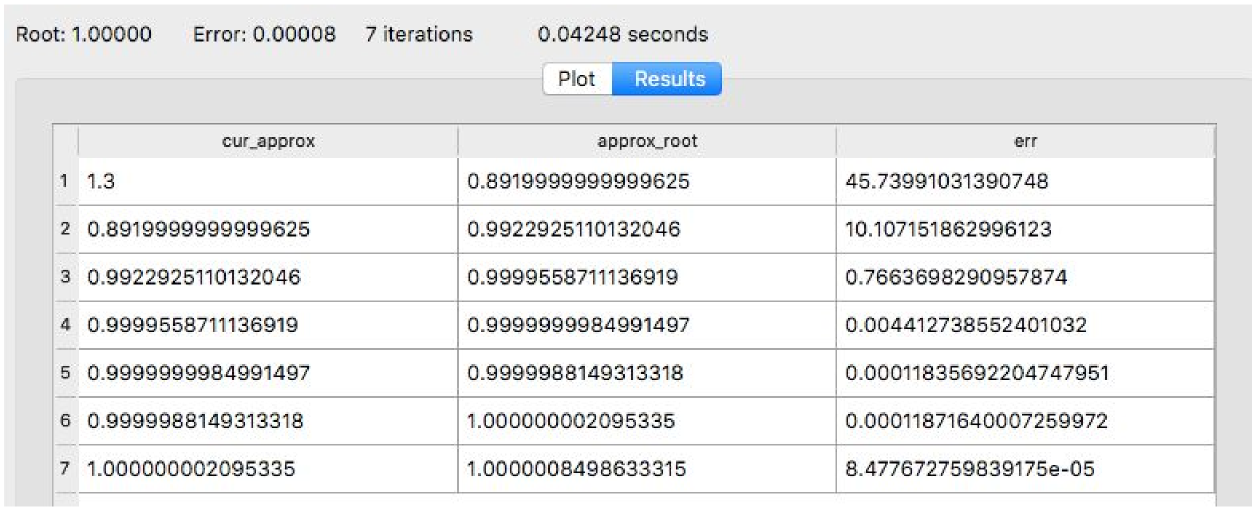
Multiplicity 2.0000000000

Conditions

Max Iterations 50

Percision 0.0001000000

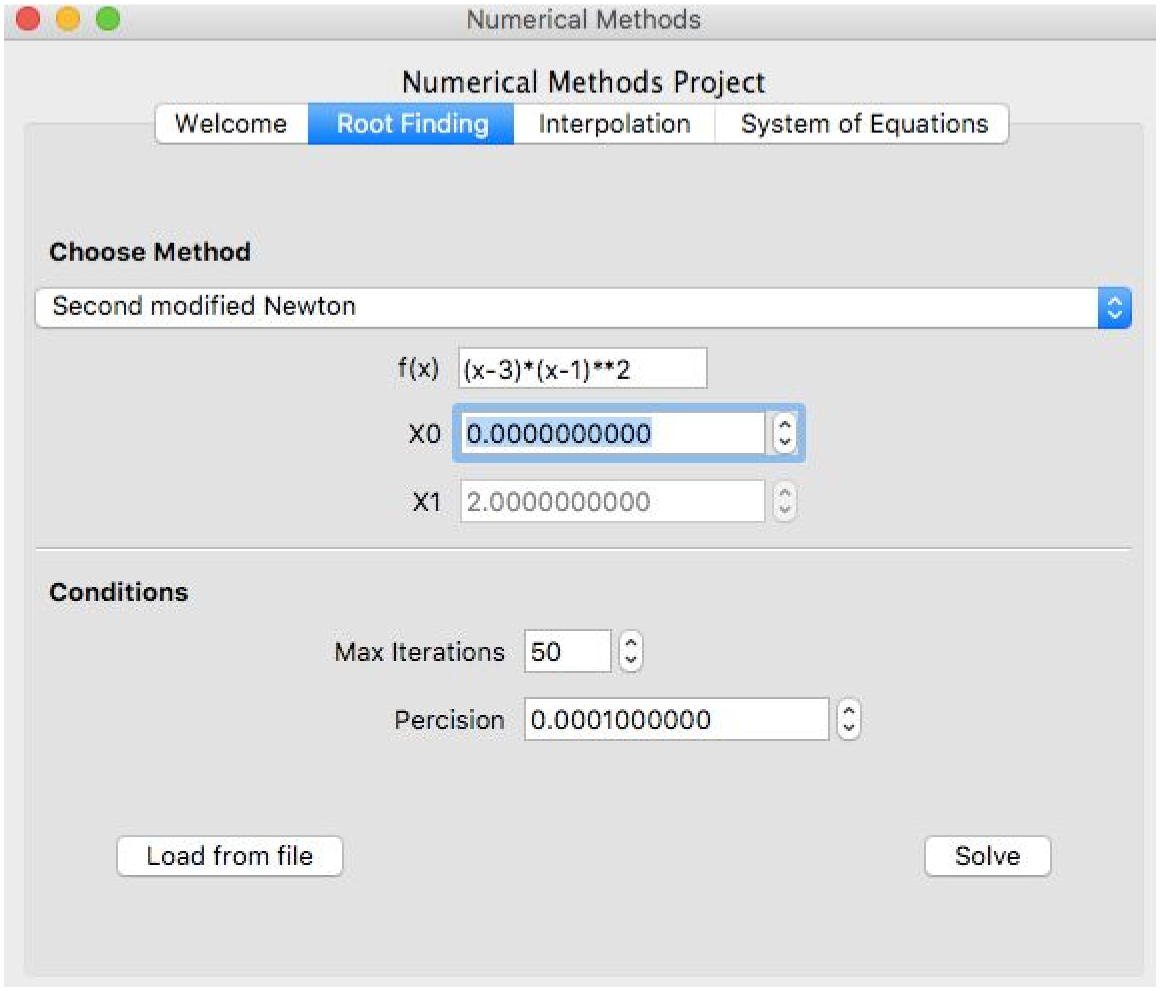

Root: 1.00000 Error: 0.00008 7 iterations 0.04248 seconds



Root: 1.00000 Error: 0.00008 7 iterations 0.04248 seconds

	cur_approx	approx_root	err
1	1.3	0.8919999999999625	45.73991031390748
2	0.8919999999999625	0.9922925110132046	10.107151862996123
3	0.9922925110132046	0.9999558711136919	0.7663698290957874
4	0.9999558711136919	0.9999999984991497	0.004412738552401032
5	0.9999999984991497	0.9999988149313318	0.00011835692204747951
6	0.9999988149313318	1.000000002095335	0.00011871640007259972
7	1.000000002095335	1.0000008498633315	8.477672759839175e-05

1.4.3. Second modified Newton Raphson



Choose Method

Second modified Newton

f(x) (x-3)*(x-1)**2

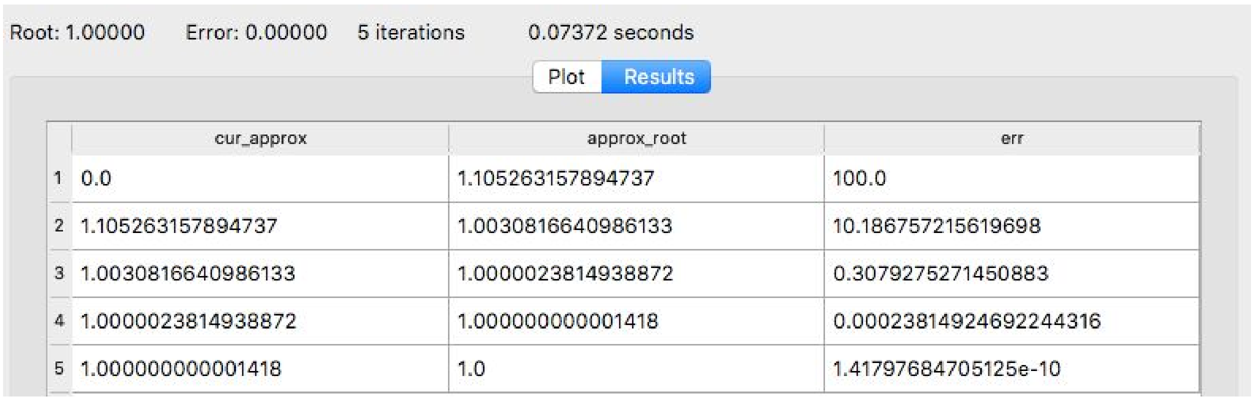
X0 0.0000000000

X1 2.0000000000

Conditions

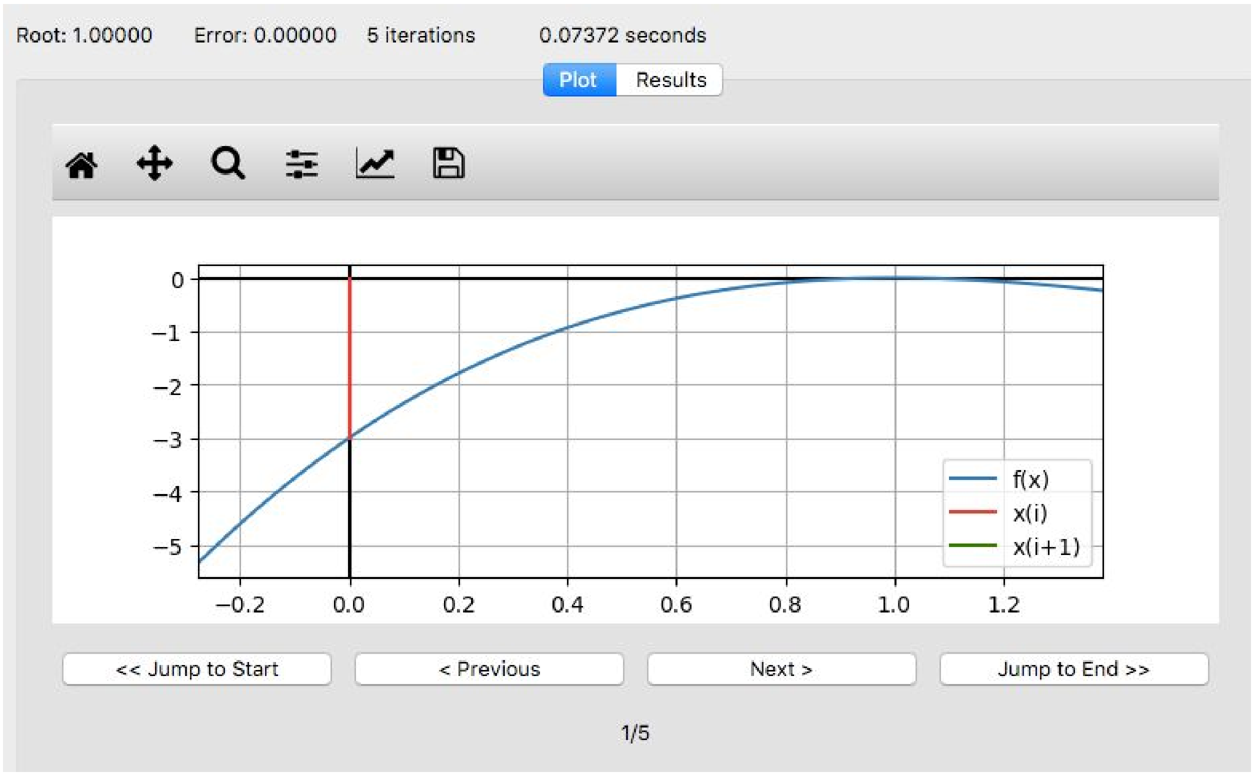
Max Iterations 50

Percision 0.0001000000



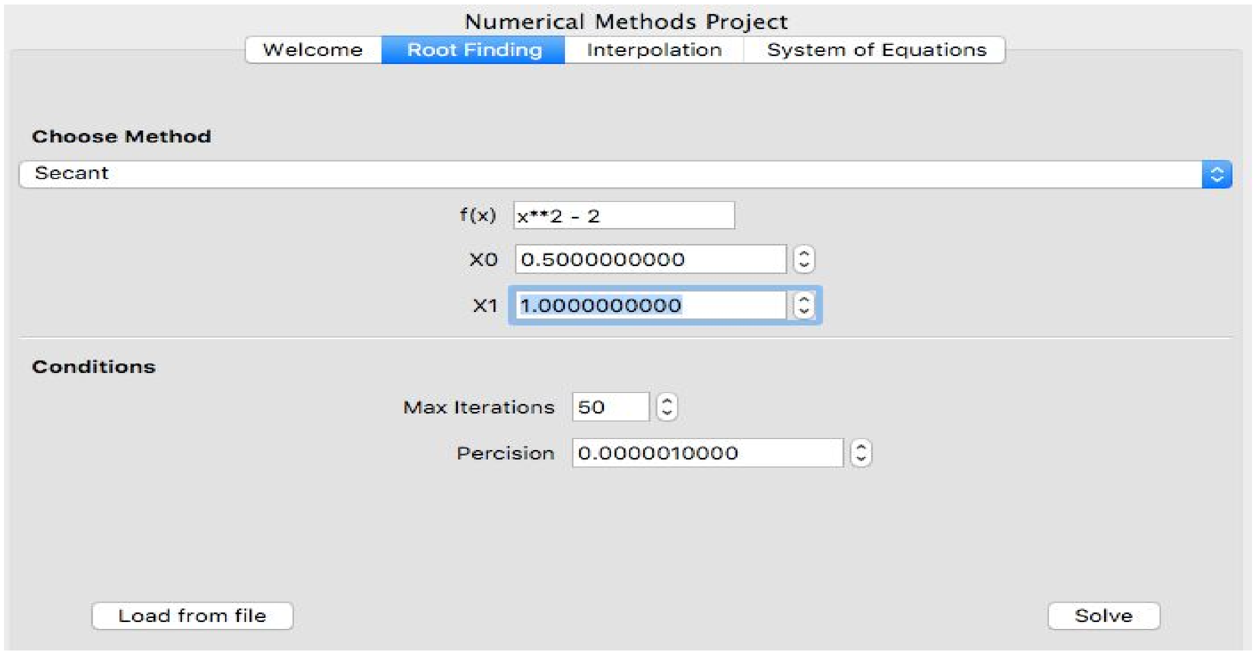
Root: 1.00000 Error: 0.00000 5 iterations 0.07372 seconds

	cur_approx	approx_root	err
1	0.0	1.105263157894737	100.0
2	1.105263157894737	1.0030816640986133	10.186757215619698
3	1.0030816640986133	1.0000023814938872	0.3079275271450883
4	1.0000023814938872	1.000000000001418	0.00023814924692244316
5	1.000000000001418	1.0	1.41797684705125e-10



Root: 1.00000 Error: 0.00000 5 iterations 0.07372 seconds

1.5. Secant method



Choose Method

Secant

f(x) x**2 - 2

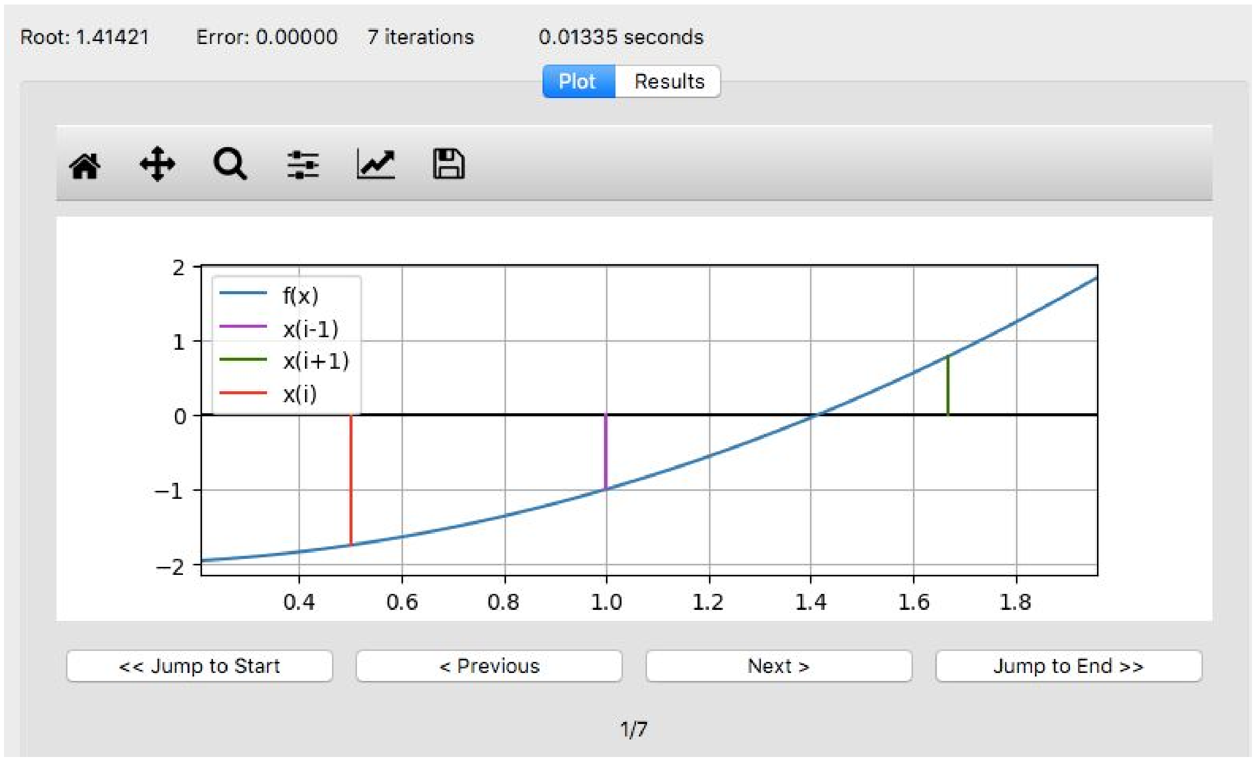
X0 0.5000000000

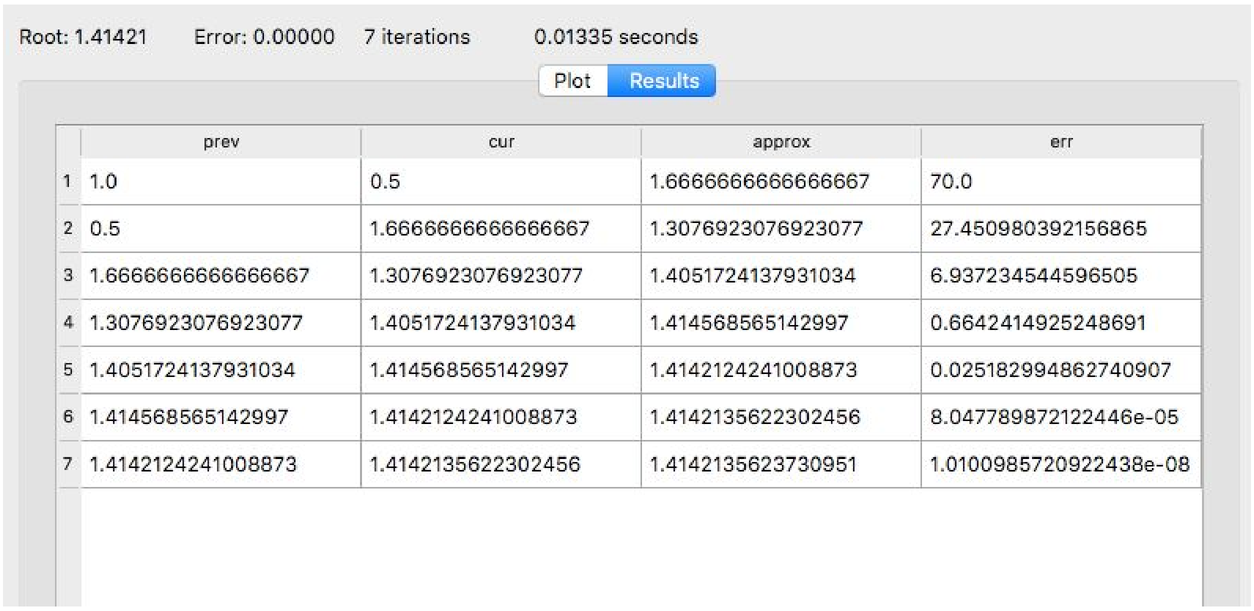
X1 1.0000000000

Conditions

Max Iterations 50

Percision 0.0000010000

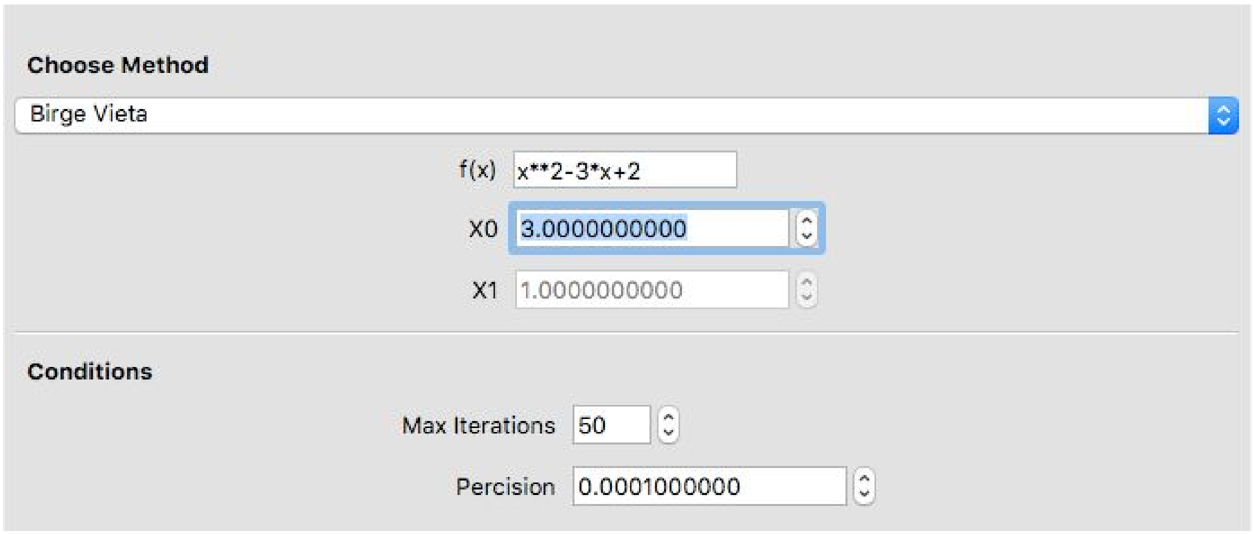

Root: 1.41421 Error: 0.00000 7 iterations 0.01335 seconds



Root: 1.41421 Error: 0.00000 7 iterations 0.01335 seconds

	prev	cur	approx	err
1	1.0	0.5	1.6666666666666667	70.0
2	0.5	1.6666666666666667	1.3076923076923077	27.450980392156865
3	1.6666666666666667	1.3076923076923077	1.4051724137931034	6.937234544596505
4	1.3076923076923077	1.4051724137931034	1.414568565142997	0.6642414925248691
5	1.4051724137931034	1.414568565142997	1.4142124241008873	0.025182994862740907
6	1.414568565142997	1.4142124241008873	1.4142135622302456	8.047789872122446e-05
7	1.4142124241008873	1.4142135622302456	1.4142135623730951	1.0100985720922438e-08

1.6. Birge Vieta



Choose Method

Birge Vieta

f(x) x**2-3*x+2

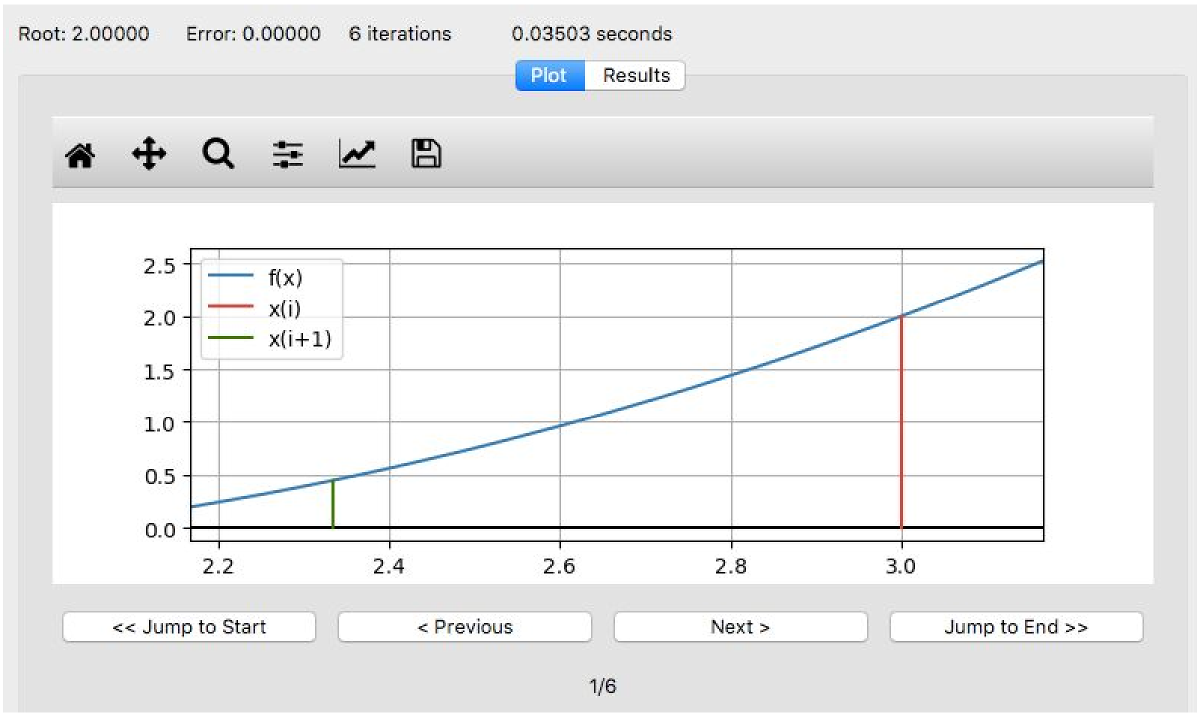
X0 3.0000000000

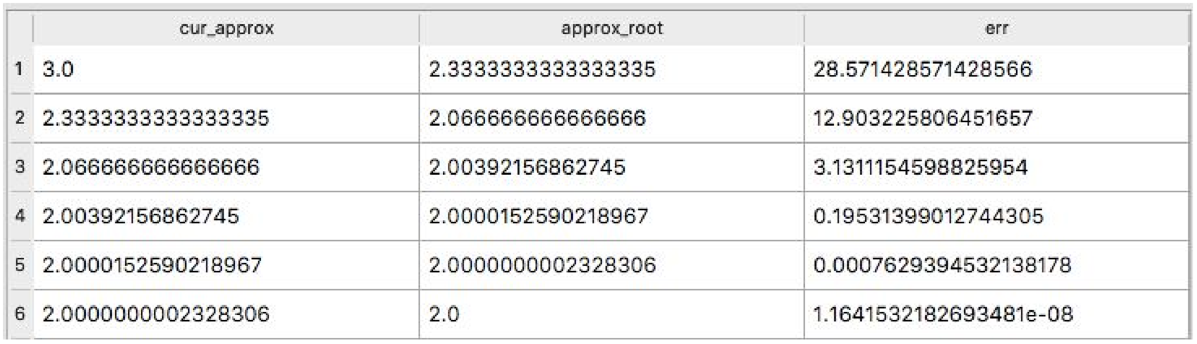
X1 1.0000000000

Conditions

Max Iterations 50

Percision 0.0001000000


Root: 2.00000 Error: 0.00000 6 iterations 0.03503 seconds



	cur_approx	approx_root	err
1	3.0	2.3333333333333335	28.571428571428566
2	2.3333333333333335	2.0666666666666666	12.903225806451657
3	2.0666666666666666	2.00392156862745	3.1311154598825954
4	2.00392156862745	2.0000152590218967	0.19531399012744305
5	2.0000152590218967	2.0000000002328306	0.0007629394532138178
6	2.0000000002328306	2.0	1.1641532182693481e-08

Problematic functions

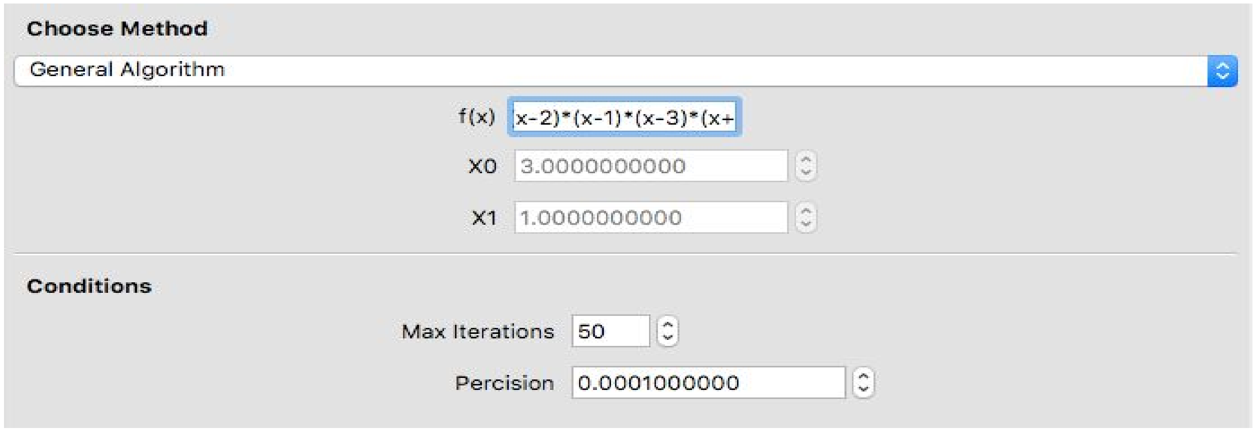
- In Newton’s method the function with first derivative equal to zero case divide by the zero problem, we try to move the current approximation using epsilon to escape the the first derivative.
- In Newton's method the existence of deflection point cause the method to jump big interval.
- Open method that not guaranteed to convert set to have maximum iteration bound.

1.2. General Algorithm:

In this part we aimed to evaluate the most number of roots from any given equation. We used a technique of approximating the function given into a tenth degree Maclaurin series and then iteratively using Newton-Raphson method to find up to ten roots for the equation.

Sample runs for general algorithm:

f(x) = (x - 1)*(x - 2)*(x - 3)*(x + 4)



Choose Method

General Algorithm

f(x) x-2)*(x-1)*(x-3)*(x+

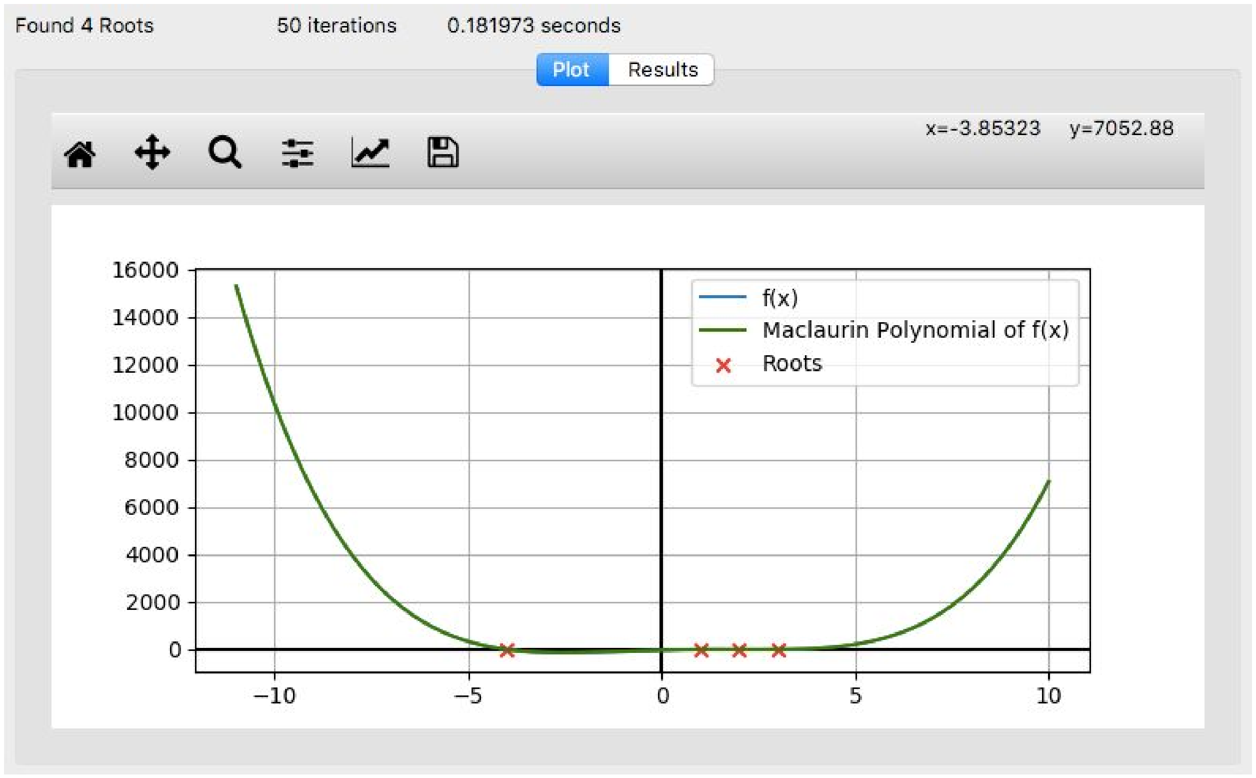
X0 3.0000000000

X1 1.0000000000

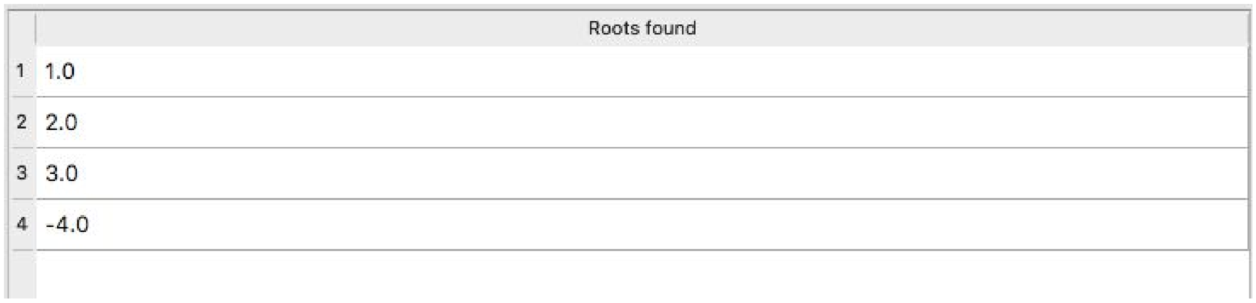
Conditions

Max Iterations 50

Percision 0.0001000000



Found 4 Roots 50 iterations 0.181973 seconds



	Roots found
1	1.0
2	2.0
3	3.0
4	-4.0

Part 2: Interpolation:

Objective:

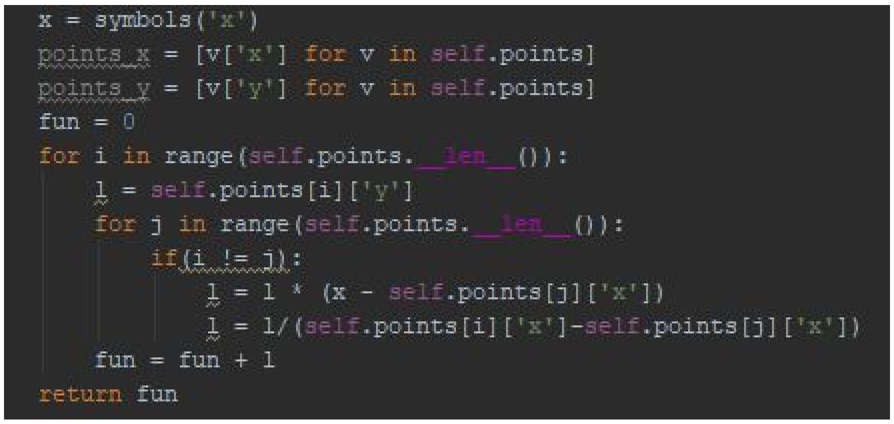
Implementing the 2 interpolation method Lagrange & Newton divided difference.

Specifications:

- Generate a function passes through all given points.
- Reading points from file or entering them manually.
- Estimate the value for a given queries points.

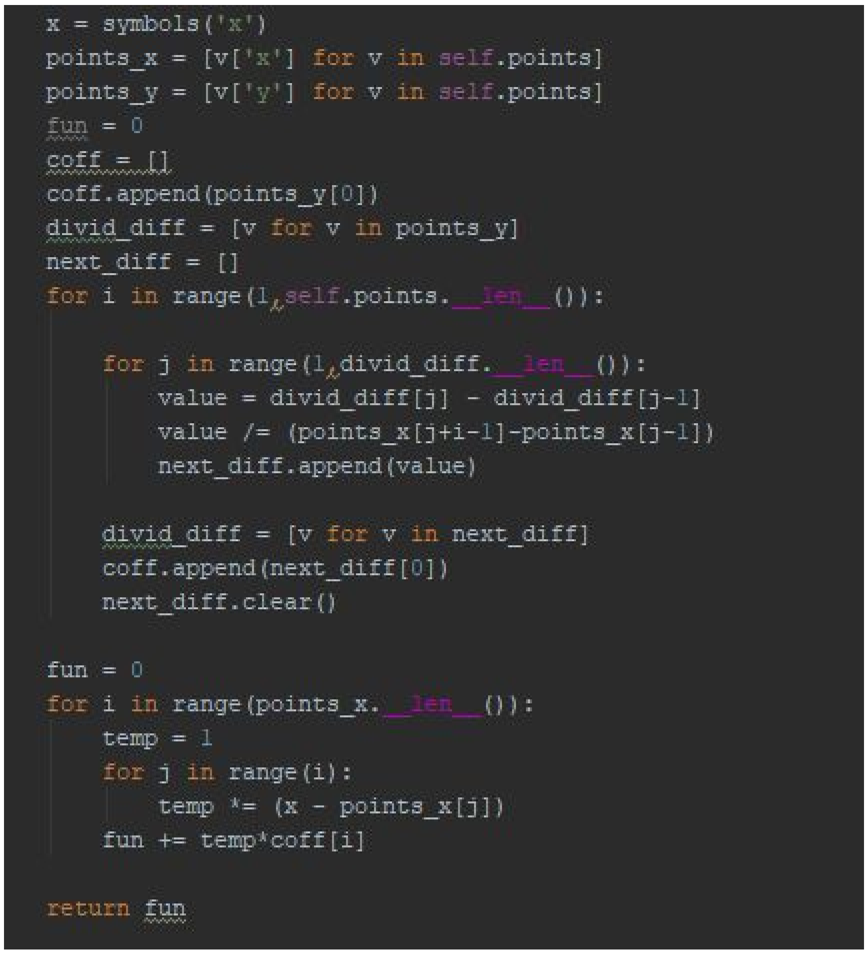
Algorithms:

Lagrange:



```
x = symbols('x')
points_x = [v['x'] for v in self.points]
points_y = [v['y'] for v in self.points]
fun = 0
for i in range(self.points.__len__()):
    l = self.points[i]['y']
    for j in range(self.points.__len__()):
        if(i != j):
            l = l * (x - self.points[j]['x'])
            l = l/(self.points[i]['x']-self.points[j]['x'])
    fun = fun + l
return fun
```

Newton divided differences:



```
x = symbols('x')
points_x = [v['x'] for v in self.points]
points_y = [v['y'] for v in self.points]
fun = 0
coff = []
coff.append(points_y[0])
divid_diff = [v for v in points_y]
next_diff = []
for i in range(1,self.points.__len__()):

    for j in range(1,divid_diff.__len__()):
        value = divid_diff[j] - divid_diff[j-1]
        value /= (points_x[j+i-1]-points_x[j-1])
        next_diff.append(value)

    divid_diff = [v for v in next_diff]
    coff.append(next_diff[0])
    next_diff.clear()

fun = 0
for i in range(points_x.__len__()):
    temp = 1
    for j in range(i):
        temp *= (x - points_x[j])
    fun += temp*coff[i]

return fun
```

- Implementing the 2 algorithms in python

  Used built in Library:
    - SymPy for manipulating symbols operations.

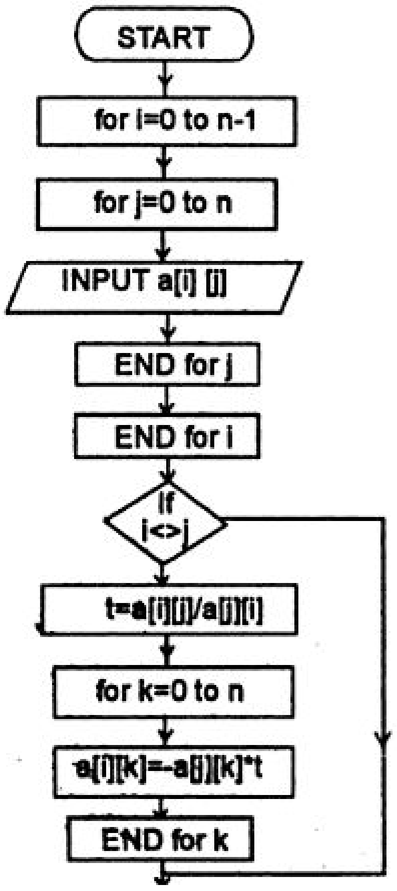
Analysis:

Both Lagrange & Newton divided difference require O(n^2)

Part 3: Systems of linear equations:

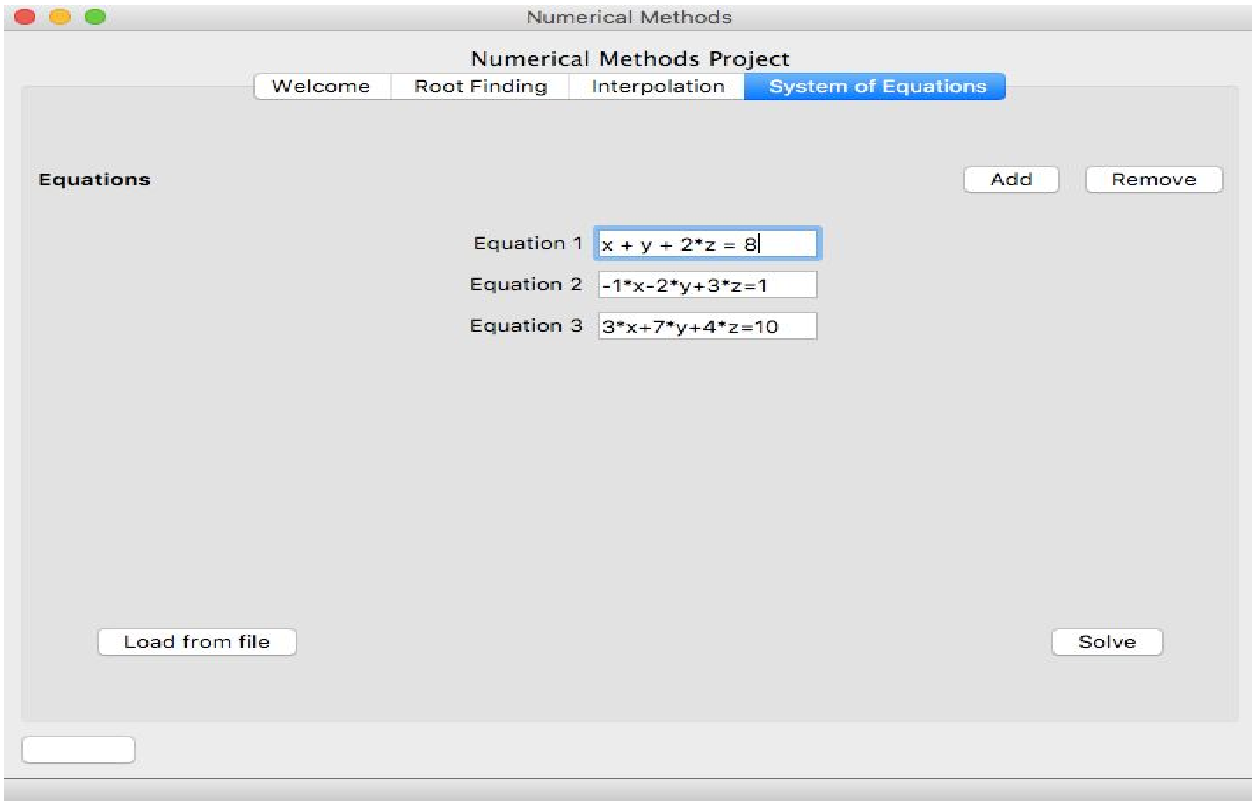
A solver for a system of equations using Gauss-Jordan algorithm

- **Gauss-Jordan Algorithm**

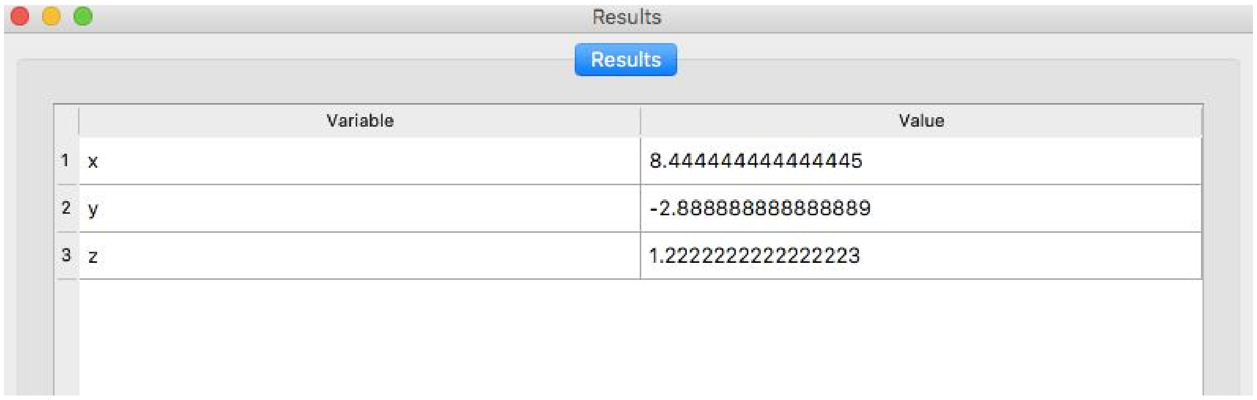


- **Sample runs:**

Run #1

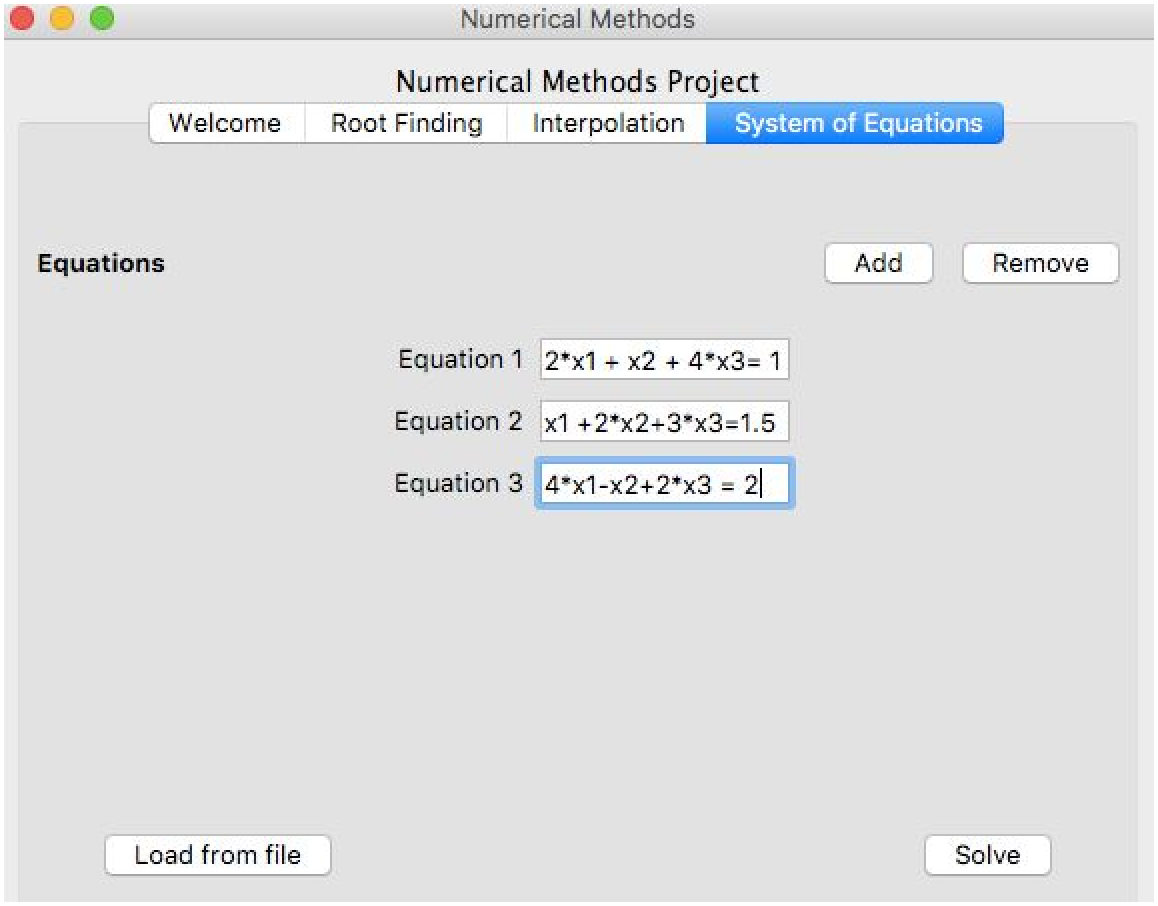


Equation 1 x + y + 2*z = 8
Equation 2 -1*x-2*y+3*z=1
Equation 3 3*x+7*y+4*z=10



	Variable	Value
1	x	8.444444444444445
2	y	-2.888888888888889
3	z	1.2222222222222223

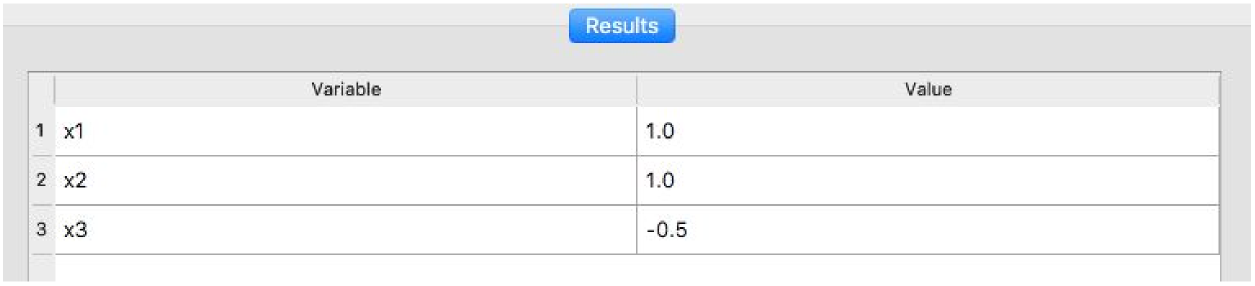
Run #2



Equation 1 2*x1 + x2 + 4*x3= 1

Equation 2 x1 +2*x2+3*x3=1.5

Equation 3 4*x1-x2+2*x3 = 2



	Variable	Value
1	x1	1.0
2	x2	1.0
3	x3	-0.5