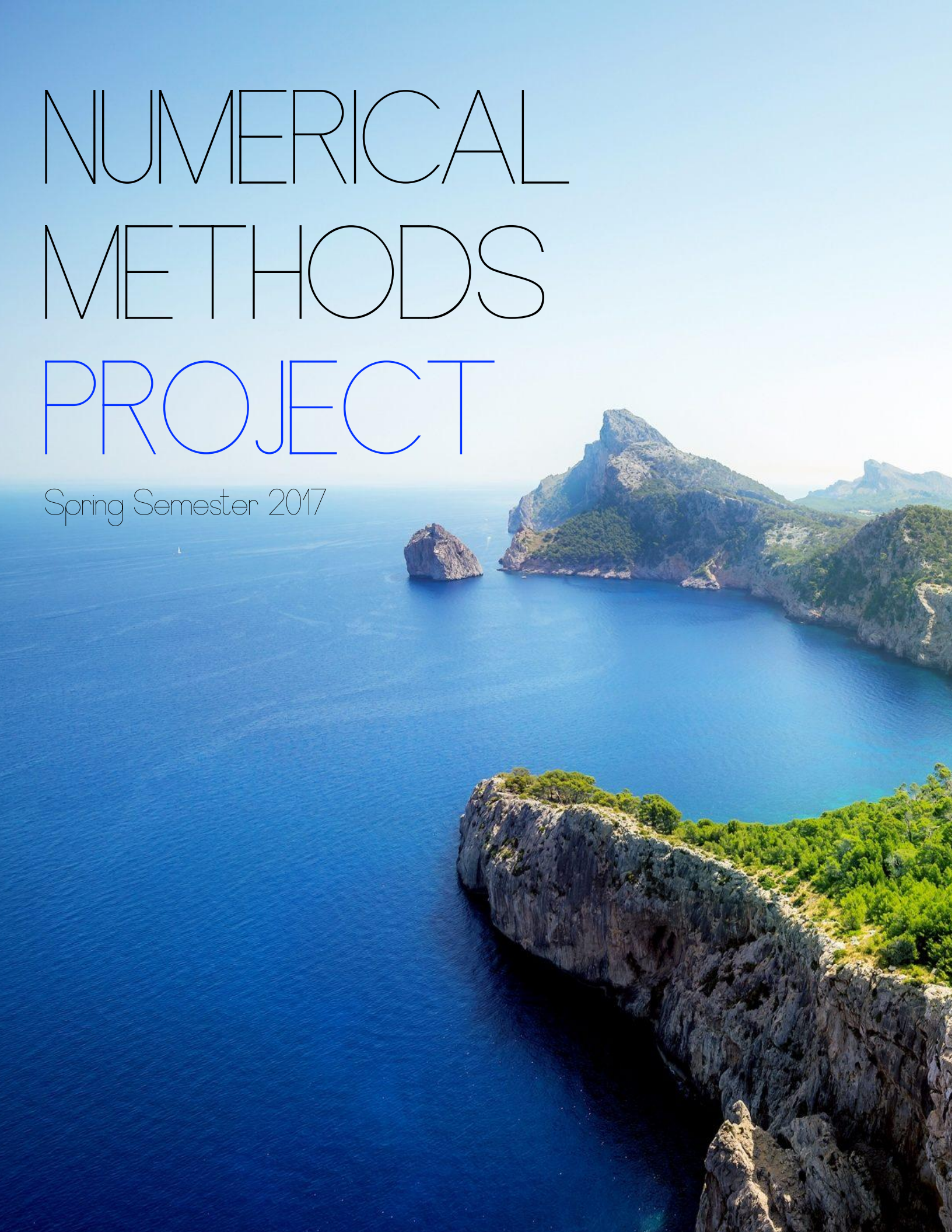
NUMERICAL METHODS PROJECT

Spring Semester 2017



Team Members[1]

Aia Lotfy
Fady Nabil
Salma Mohammed
Helana Mansour
Yassmin Barakat

[1] Names are arranged in English alphabetical order.

Part I

Comparison and Analysis of Different Numerical Methods Behavior

1. Methods' Pseudo Codes

I. Bisection Method

The function takes 5 parameters which are:

- The equation
- Initial x_L
- Initial x_U
- Max iterations number
- Allowed relative error

The function returns 3 values which are:

- Iteration Matrix
- Number of iterations
- Error message

```
bisection
 if equation(xL) * equation(xU) > 0
    return
for i=1 to Max_Iterations
     insert values in the output matrix
     xR = (xL+xU)/2
         if equation(xR) * equation(xU) < 0
              xU=xR
        x=xU
         else
              xL=xR
        x=xL
       end if
   relative_error = abs((xU-xL)/xL)
    if  relative_error <= allowed_relative_error
        break loop
    end if
end for
if not valid equation then
      error terminate operation
end if
end bisection
```

II. Regula-Falsi Method

The function takes 5 parameters which are:

- The equation
- Initial x_L
- Initial x_U
- Max iterations number
- Allowed relative error

The function returns 3 values which are:

- Iteration Matrix
- Number of iterations
- Error message

```
false_position
if equation is valid
 if equation(xL) * equation(xU) > 0
    return
for i=1 to Max_Iterations
    insert values in the output matrix
    xR= ((xL* equation(xU) )-(xU* equation(xL) ))/(equation(xL) -
equation(xU) l)
        if equation(xR) * equation(xU) < 0
        xU =xR
        x= xU
   else
        xL=xR
        x=xL
   end
   relative_error = abs((xU-xL)/xL)
    if  relative_error <= allowed_relative_error
        break loop
end for
else if not valid equation
     error message, terminate program
end false_position
```

III. Fixed Point Iteration Method

The function takes 4 parameters which are:

- The equation
- Initial x (x_o)
- Max iterations number
- Allowed relative error

The function returns 4 values which are:

- Iteration Matrix (x_R)
- Number of iterations
- g(x) equation
- Error message

```
fixed_point
x=xo
xR=x
if equation is valid then
for i=2 to Max_Iterations
    x=xR(i-1,1)
    xR(i,2)=x
    xR(i,1) =equation+xR(i,2)
    xR(i,3) =abs((xR(i,1)-xR(i,2))/xR(i,2))
    if xR < allowed_relative_error
        break
    end if
end for
else if not valid equation
     error message, terminate program
g(x) = strcat(equation,'+x')
end false_position
```

IV. Secant Method

The function takes 5 parameters which are:

- The equation
- Initial x (x_o)
- Zero x x_{zero}
- Max iterations number

- Allowed relative error

The function returns 6 values which are:
- Iteration Matrix (x_R)
- ReturnEquation
- g(x) equation
- Iterations number
- Delta
- Error message

```
secant_method
ReturnEquation(1)=equation(xo)
numerator = equation(xo)*(xo -xzero)
denominator= equation(xo)-equation(xzero)
currentValue=xo -(numerator/denominator)
relativeError =abs((currentValue-x)/currentValue)*100
error(1)=relativeError //  inserting relative error in error array
counter=1
if function is valid then
while relativeError > validError or counter < maxIterations
    if counter = 1
        x=xR(counter)
       ReturnEquation(counter+1)=equation(xo)
        numerator = equation(xo)*(currentValue - xo)
       denominator= equation(xo)-equation(xzero)
        currentValue=currentValue -(numerator/denominator)
        relativeError =abs((currentValue-x)/currentValue)*100
        counter= counter+1
        error(counter)=relativeError //  store current iteration
relative error
        xR(counter)=currentValue  //  store current iteration
calculated root
    else if  counter ≠1
        ReturnEquation(counter+1)=equation(xo)
        numerator = equation(xo)*(currentValue -xR(counter -1))
        denominator= equation(xo)-equation(xzero)
        currentValue=currentValue -(numerator/denominator)
```

```
        relativeError =abs((currentValue-x)/currentValue)*100
        counter= counter+1
        error(counter)=relativeError
        xR(counter)=currentValue
    end if
end while
else if not valid equation
     error message, terminate program
end secant_method
```

V. Newton-Raphson Method

The function takes 4 parameters which are:

- The equation
- Initial x (x_o)
- Max iterations number
- Allowed relative error

The function returns 4 values which are:

- Iteration Matrix (x_R)
- ReturnEquation
- ReturnEquationDerivative
- g(x) equation

```
newton_raphson
numerator=equation(xo)
denominator= subs(df)
xi= xo -(numerator/denominator)
xR(1)=xi
relativeError =abs((xi-x)/xi)*100
error(1)=relativeError
counter=1
if equation is valid then:
while relativeError> validError AND counter < maxIterations
    x=xR(counter)
    numerator=subs(f(x))
    denominator= subs(df)
```

```
    ReturnEquation(counter)=numerator
    ReturnEquationDerivative(counter)=denominator
    xi= x -(numerator/denominator)
    relativeError =abs((xi-x)/xi)*100
    counter=counter+1
    xR(counter)=xi
    error(counter)=relativeError
end while
x=xR(counter)
numerator=subs(f(x))
denominator= subs(df)
ReturnEquation(counter)=numerator
ReturnEquationDerivative(counter)=denominator
else if not valid equation
     error message, terminate program
end newton_raphson
```

VI. Bierge Vieta Method

function [xR,sendOrder,A,B,C ,error]
The function takes 4 parameters which are:

- The equation
- Initial x (x_o)
- Max iterations number
- Allowed relative error

%%%%%

The function returns 4 values which are:

- Iteration Matrix (x_R)
- ReturnEquation
- ReturnEquationDerivative
- g(x) equation

```
Bierge_Vieta
//  initialize variables and matrices
//  initializing matrices a,b and c
if function is valid then:
while abs(stop) >=0.00001AND iterate <= maxIterations
    iterate= iterate+1
```

```
    while(counter > 0 )
        counter = counter-1
        if counter = 1
            coffCounter= coffCounter+1
            order(coffCounter)=counter-1
            a(counter)= coff(coffCounter)
            b(counter)=a(counter)+ xIterate*b(counter+1)
            xR(iterate)=xIterate - (b(1)/c(2))
            stop= b(1)

            if iterate = 2
      // flipping columns order for matches A,B and C
                sendOrder = order'
            else
      // flipping columns order for matches A,B and C
                sendOrder =[sendOrder order']
            end if
        else if counter >1
            coffCounter= coffCounter+1//increment coeffCounter
            order(coffCounter)=counter-1
            a(counter)= coff(coffCounter)
            b(counter)=a(counter)+ xIterate*b(counter+1)
            c(counter)=b(counter)+ xIterate*c(counter+1)
        end if
    end while
    coffCounter=1
    // calculate errors and final a,b and c
    error(iterate)=abs((xR(iterate) -xR(iterate-1))/xR(iterate))
    xIterate = xR(iterate)
else if not valid equation
     error message, terminate program
end Bierge_Vieta
```

2. General Method

I. Explanation

A “brute force” based algorithm used to get *all* the real roots of the entered equation (polynomial or non-polynomial), an [-100, 100] interval is divided to sub-intervals each of length = 0.5 (i.e $x_{i+1} - x_i = 0.5$).

To guarantee convergence of the answer, bisection method (a well known bracketing method) was used to obtain the root, as we apply this method on each interval from our sub-intervals, where if there exists a root we store it and all its iterations, errors …etc, in a matrix indicating that root, if not, the next sub-interval is tested till all the sub intervals are tested, if there exists an error some where through the operations, processing terminates and an error message appears to the user.

An array of matrices is used to store the answer and be returned, where each cell of the array contains the iterations matrix of one of the obtained roots.

II. Pseudo Code

The function takes 1 parameter which is:

- The equation

The function returns 3 values which are:

- Array of roots Matrices (x_R)
- Number of roots
- Error message

```
general_method
initializing variables
for i=1 to numberOfIntervals
    lower = addInterval+0.001
    upper = addInterval+0.5
    addInterval = upper
     // call to bisection method to obtain expected roots in this
     interval
    if b ≠ 0 AND isNumber(a(b))
    store root (a) in the matrix
    index = index + 1
```

```
    end if
end for
size = index - 1
else
   error occurred terminate program
end if
end general_method
```

3. Data Structures Used

I. Matrices

Matrices where the most important data structure used through this part of the assignment, they where used to store the values and errors … etc to allow:

- The next iteration to be implemented easily
- Printing a table of iterations in the output file
- Helps in drawing the plot

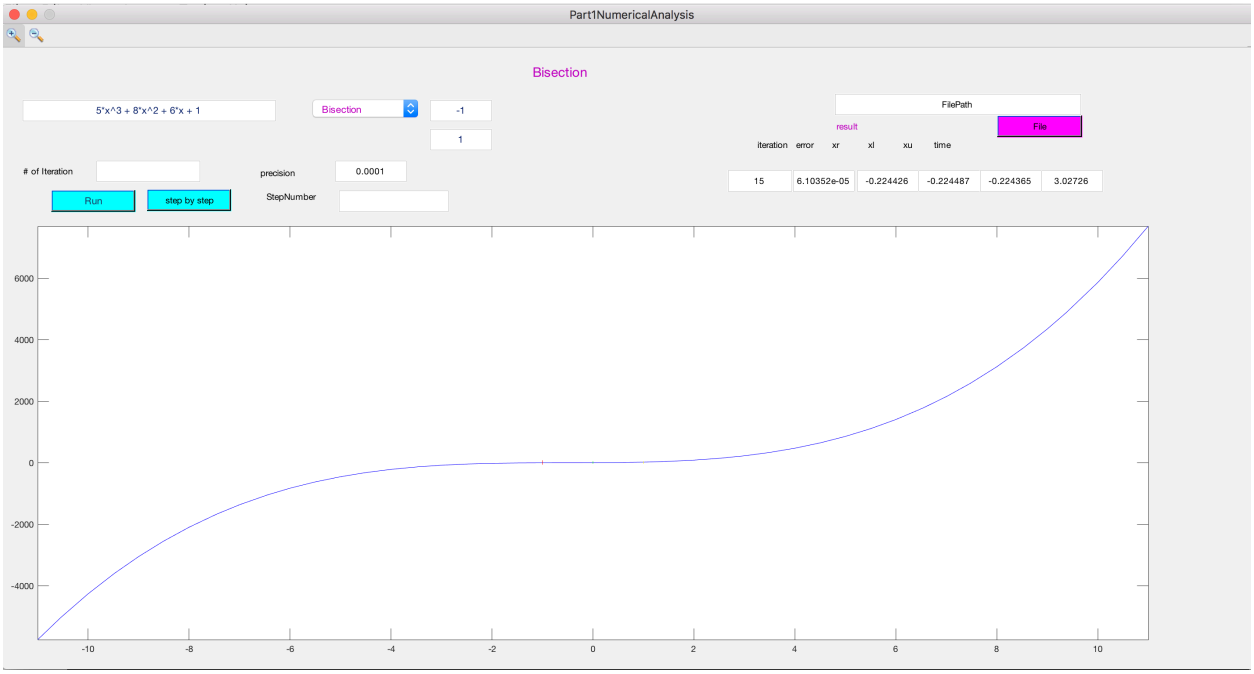
II. Structures (struct)

Since reading data from an external .txt file was one of the main requirements of this project after following the necessary instructions to read from the file, structures where used as a container for the data parsed / traversed from the .txt file containing the required data to be input.

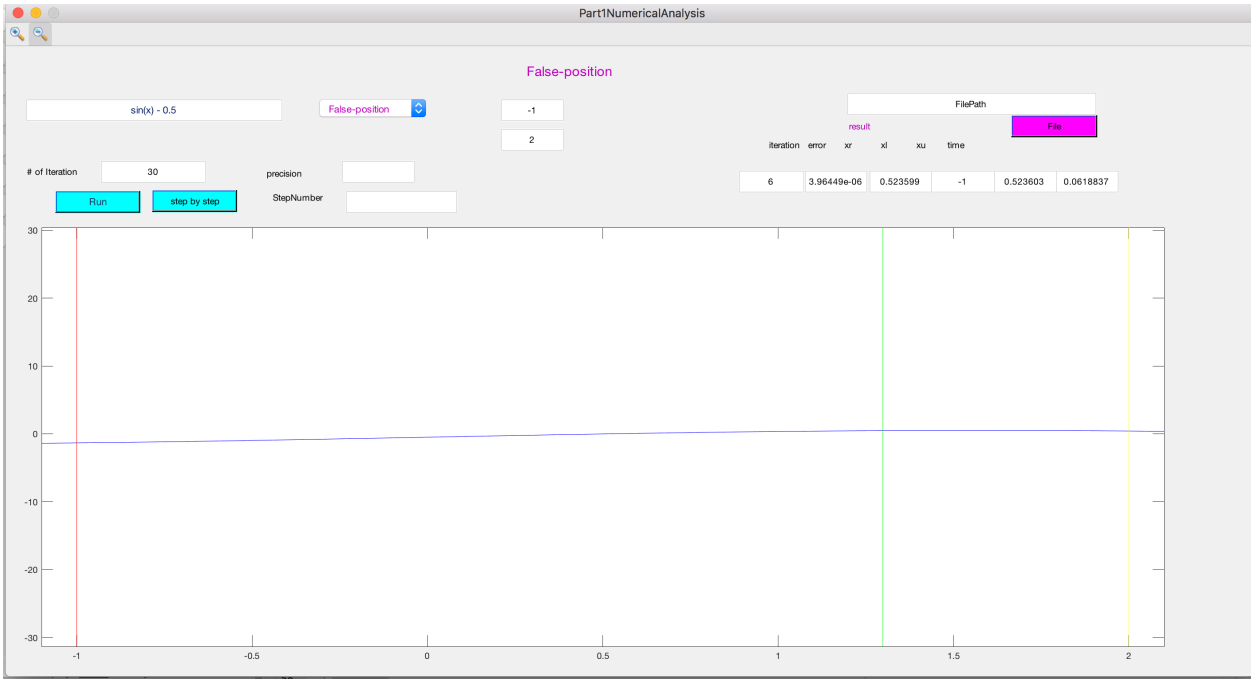
4. *Sample Tests*

I. Via GUI Operations

- **Bisection Method**[2]



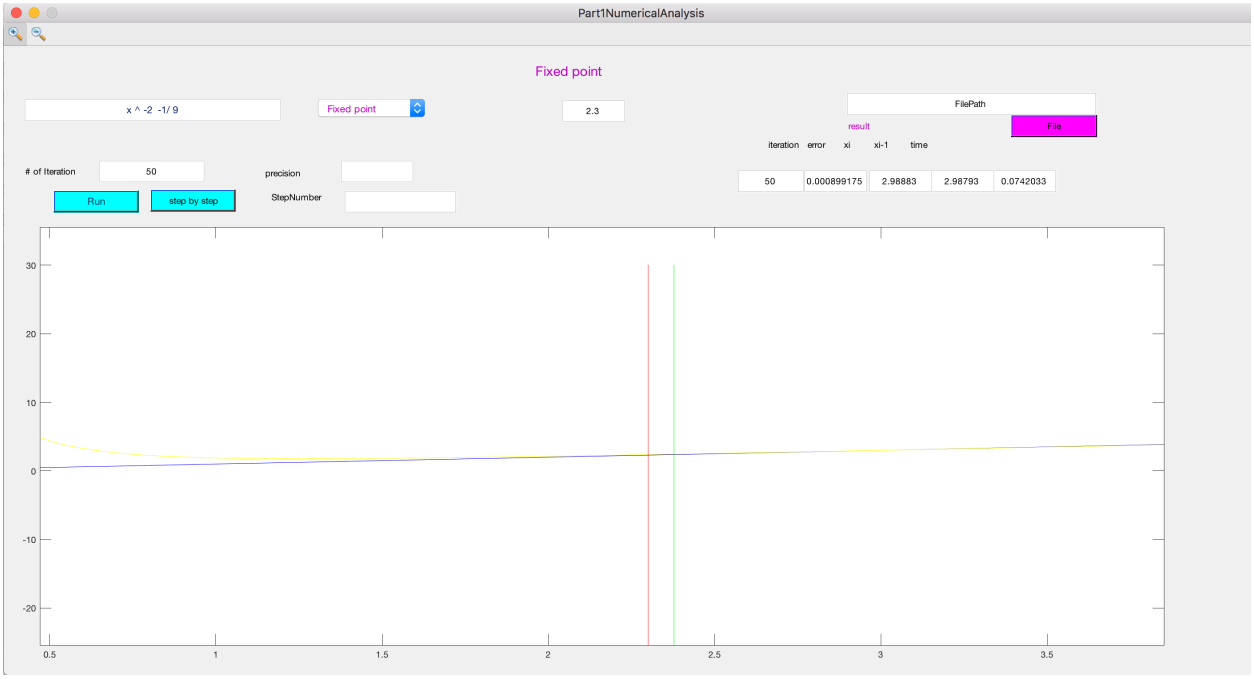
- **False Position**[3]



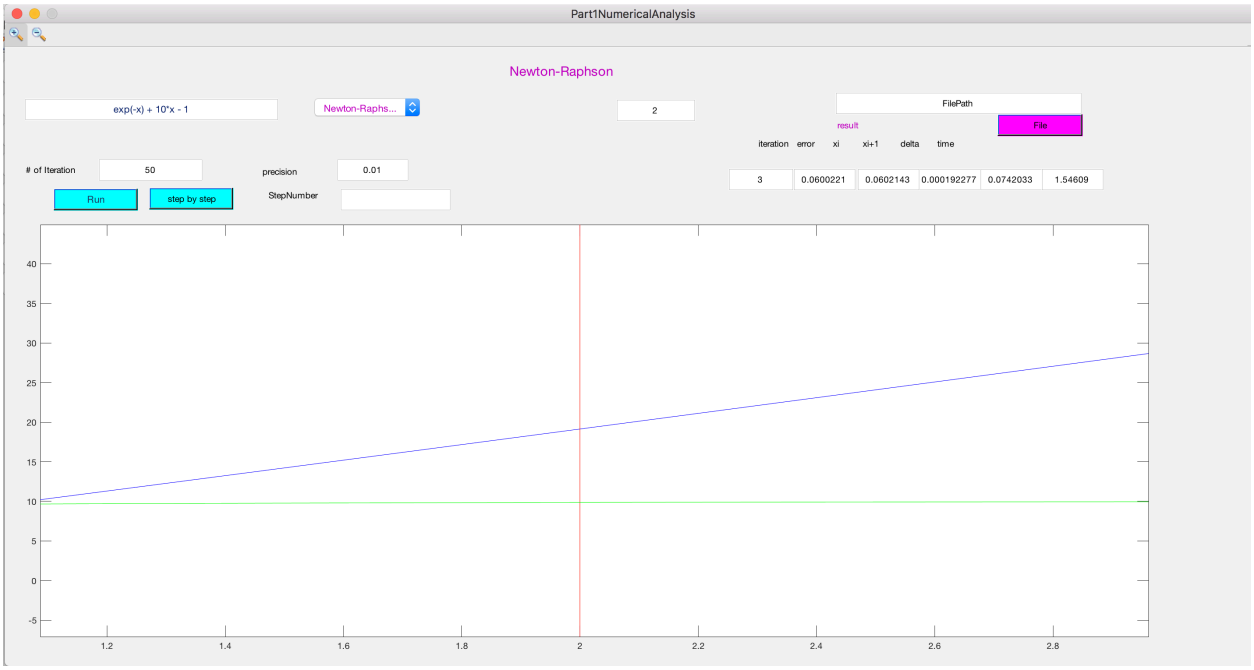
[2] Empty iterations' number box indicates default iterations value.

[3] Empty precision box indicates default allowed error value.

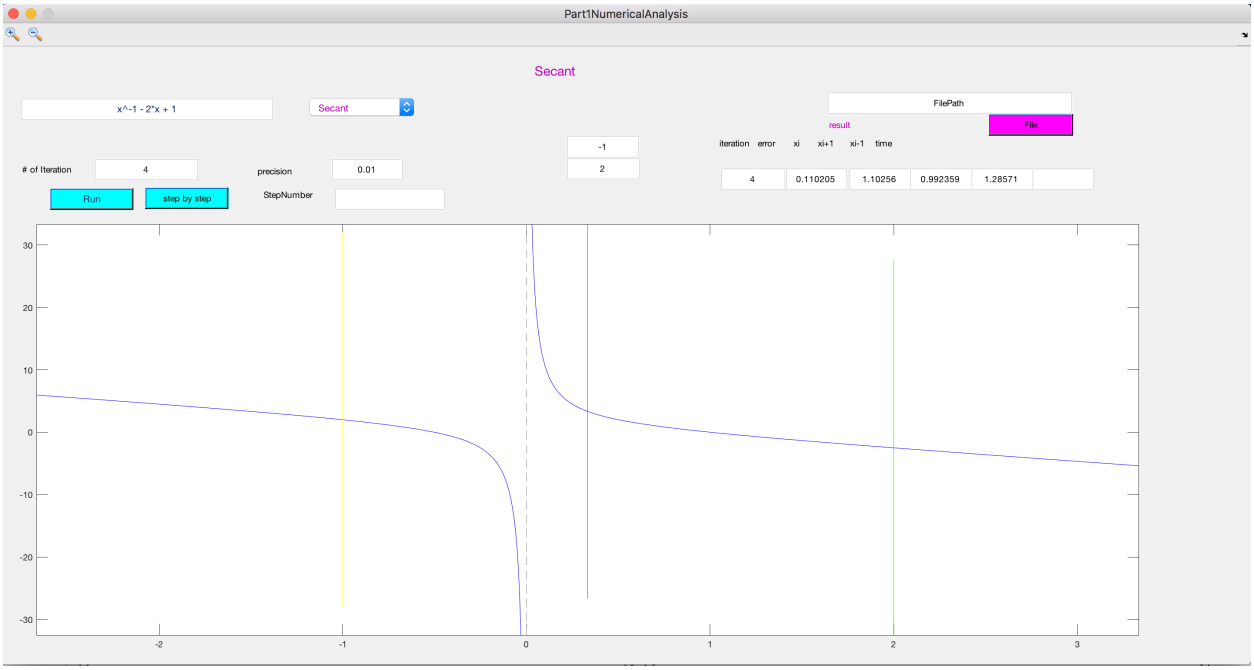
- **Fixed Point**



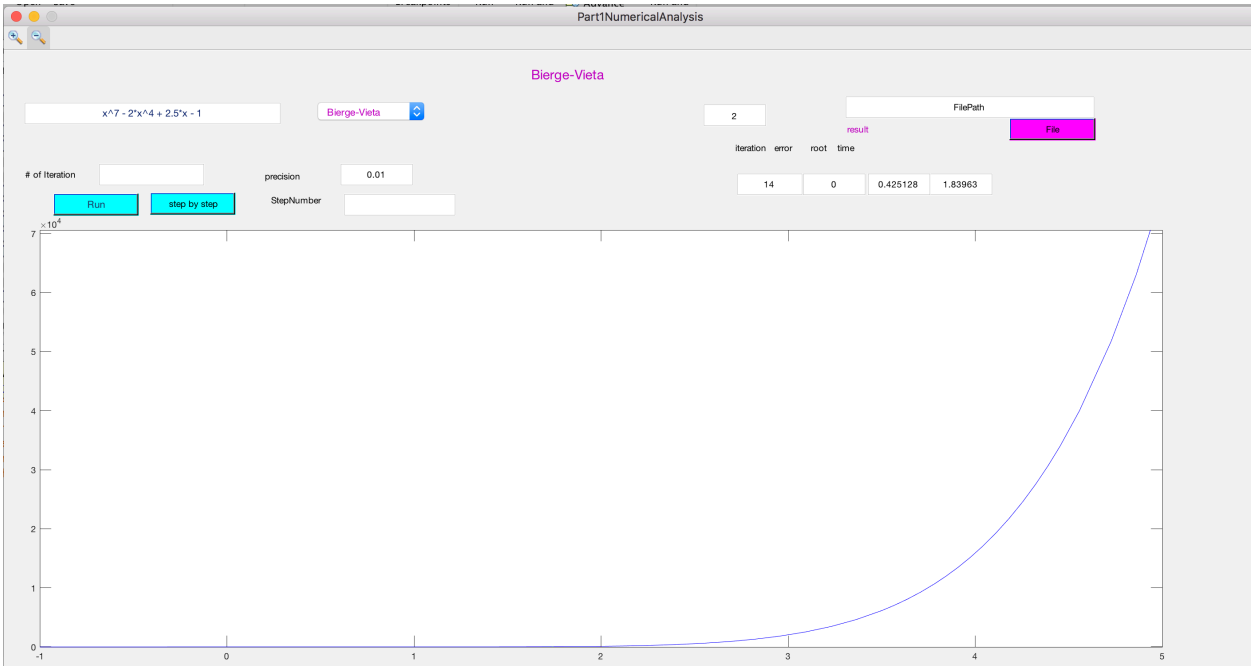
- **Newton-Raphson**



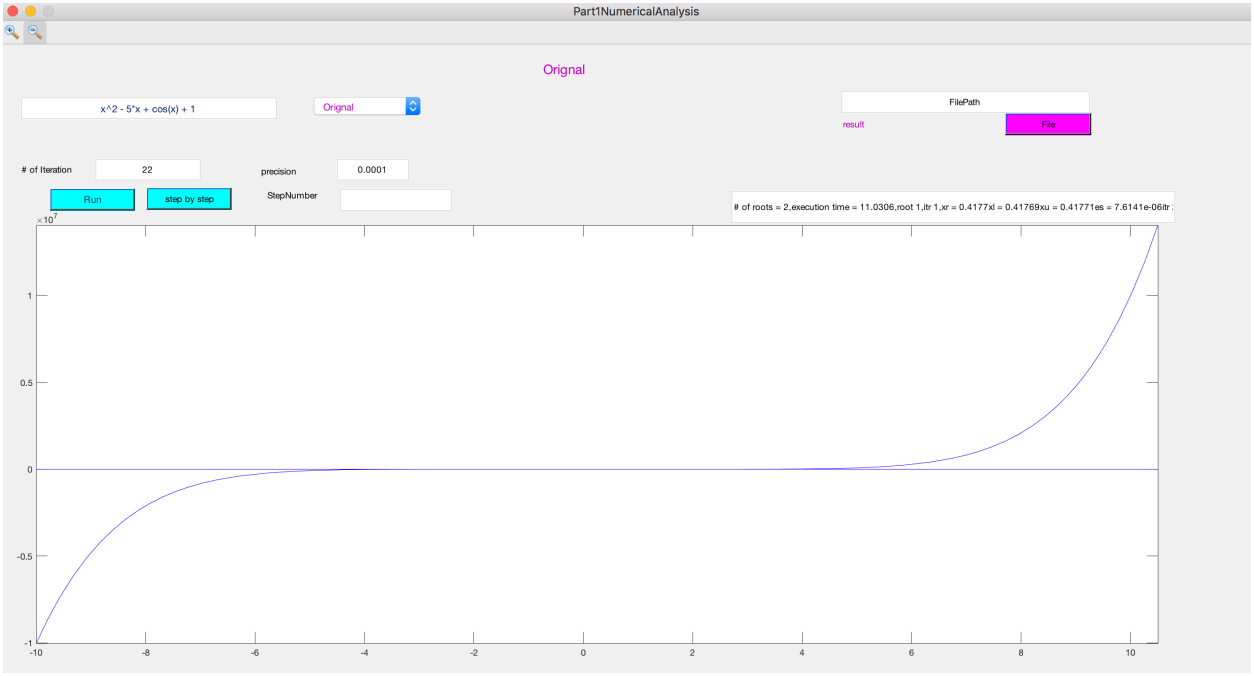
- **Secant Method**

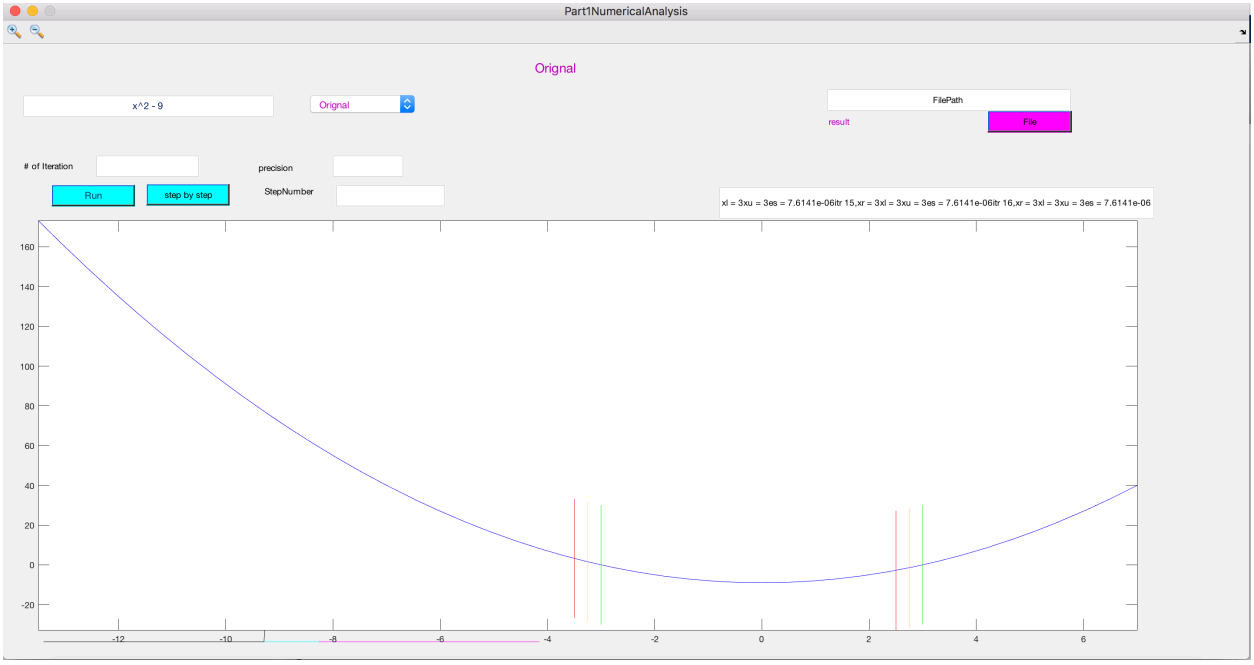


- **Bierge Vieta**

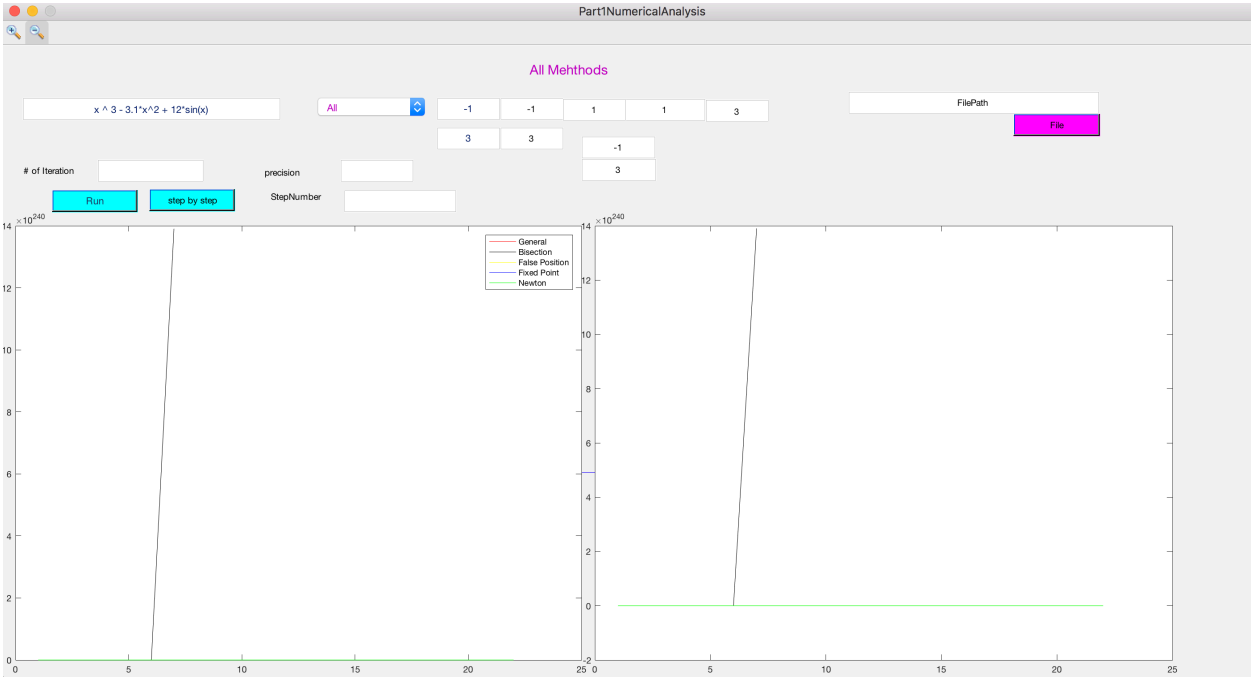


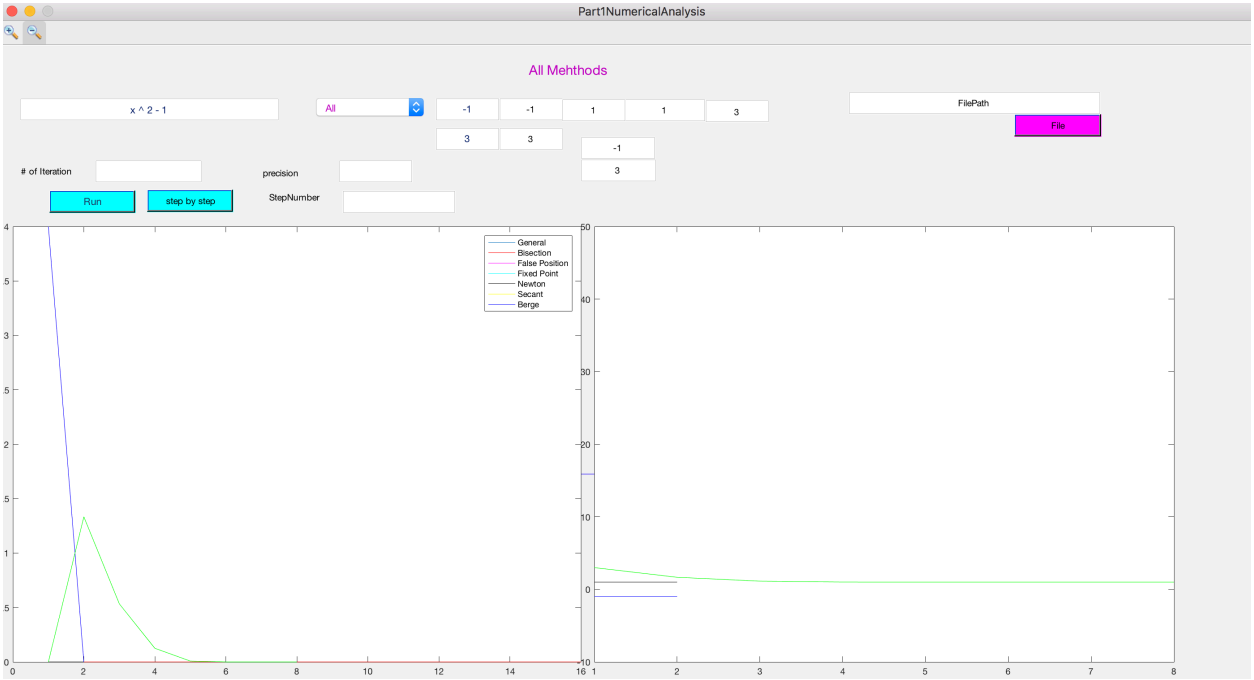
General Method





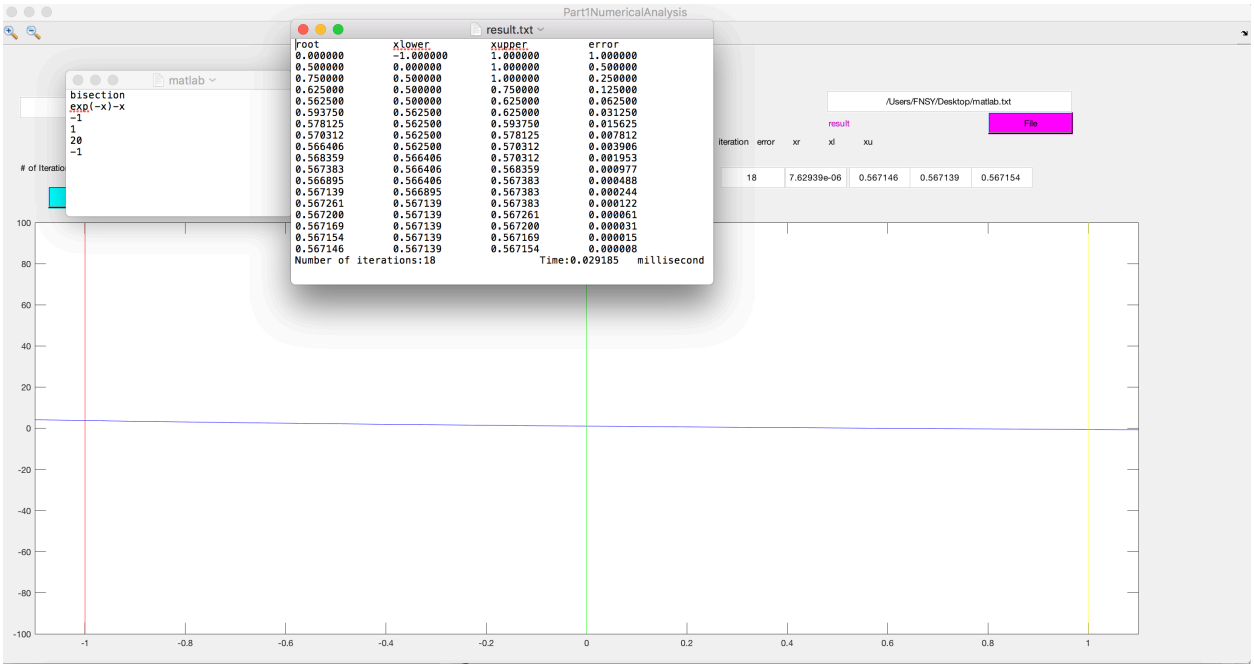
- **All methods**


Part1NumericalAnalysis
All Mehthods
x ^ 3 - 3.1*x^2 + 12*sin(x)
All
-1
-1
1
1
3
3
3
-1
3
FilePath
File
# of Iteration
precision
StepNumber
Run
step by step
General
Bisection
False Position
Fixed Point
Newton

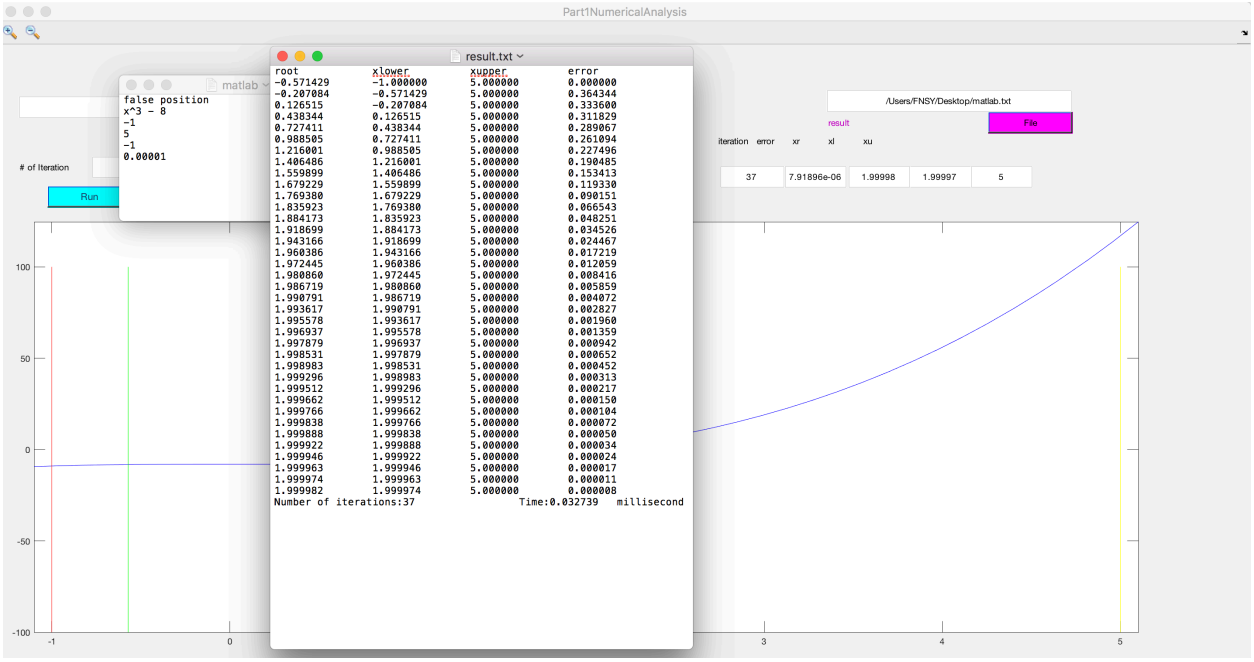

Part1NumericalAnalysis
All Mehthods
x ^ 2 - 1
All
-1
-1
1
1
3
3
3
-1
3
FilePath
File
# of Iteration
precision
StepNumber
Run
step by step
General
Bisection
False Position
Fixed Point
Newton
Secant
Berge

II. Via Files Operations

- **Bisection Method**[4]

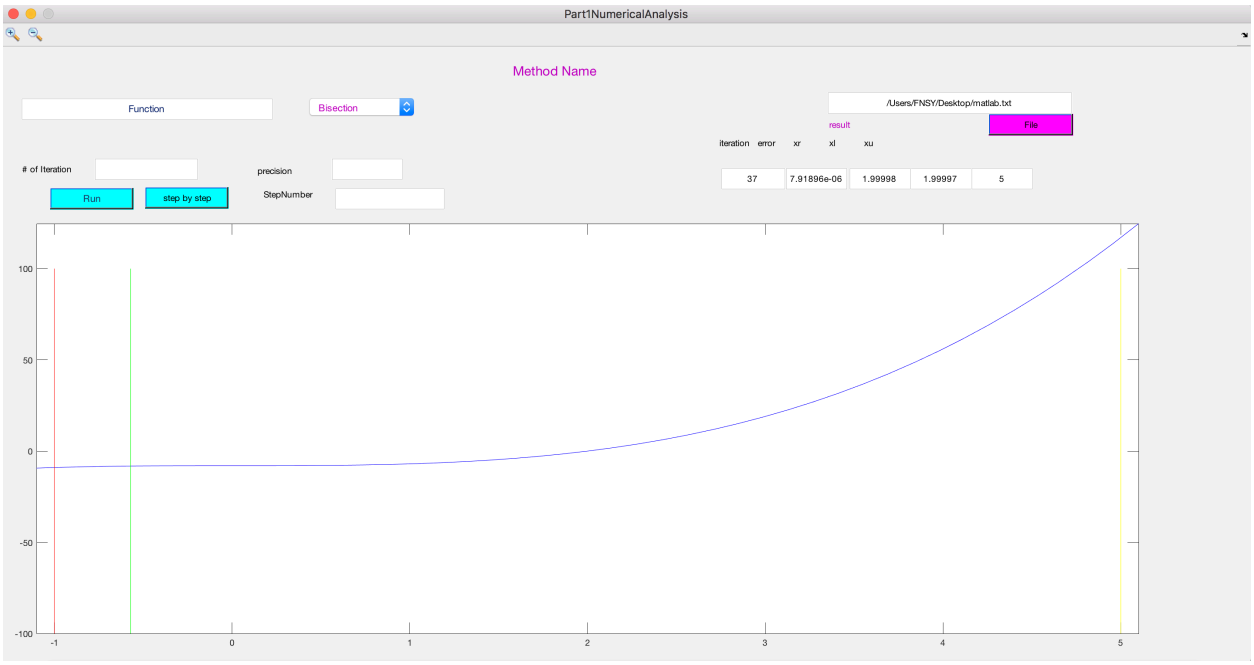


- **False Position**[5]

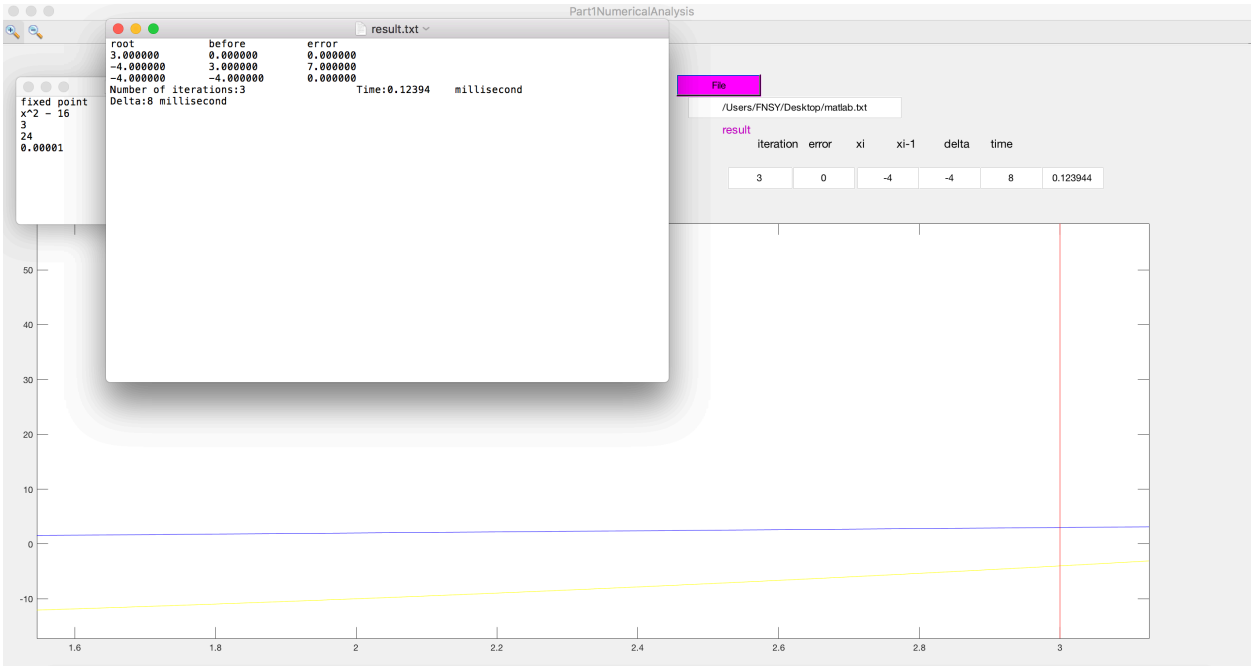


[4] Negative accepted error indicates default value.

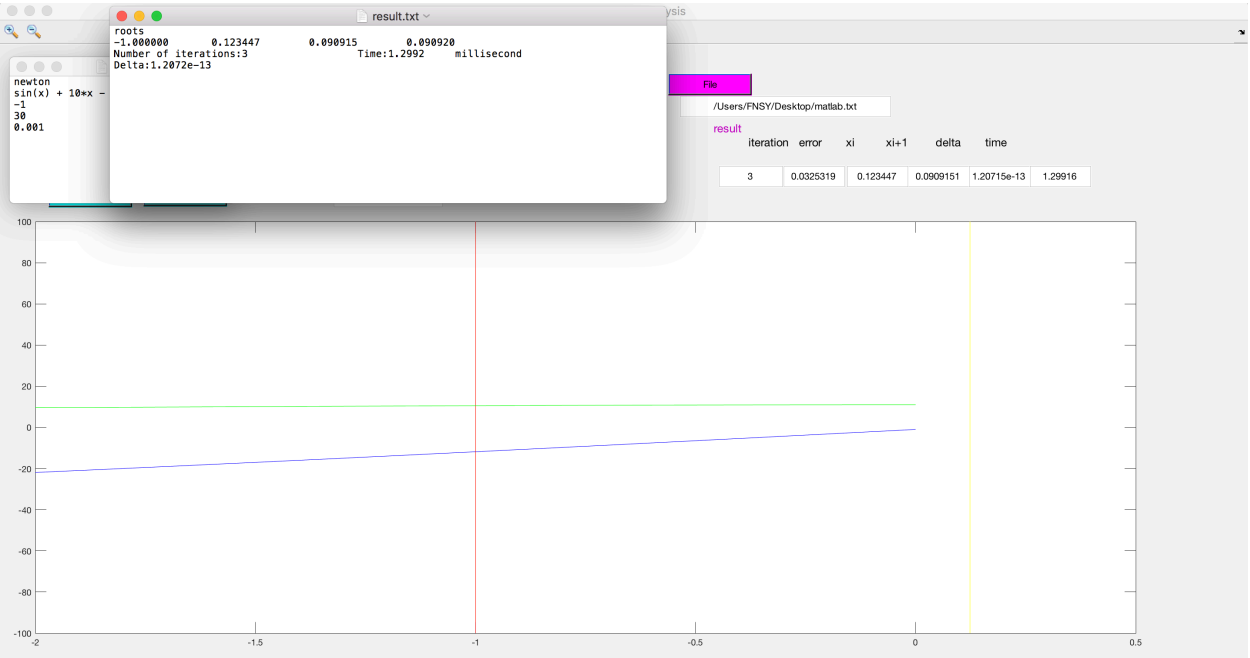
[5] Negative iterations' number indicates default value.


Part1NumericalAnalysis
Method Name
Function
Bisection
/Users/FNSY/Desktop/matlab.txt
result
File
iteration error xr xl xu
# of Iteration
precision
Run
step by step
StepNumber
37 7.91896e-06 1.99998 1.99997 5
100
50
0
-50
-100
-1 0 1 2 3 4 5

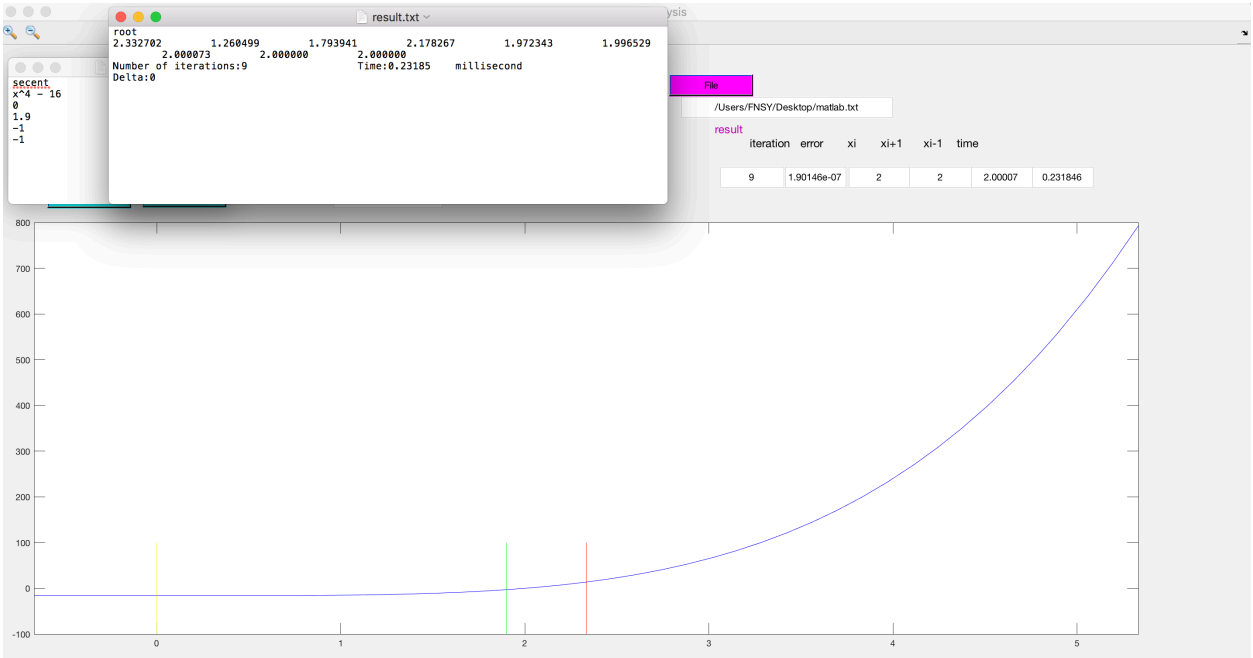
- **Fixed Point**


Part1NumericalAnalysis
result.txt
root before error
3.000000 0.000000 0.000000
-4.000000 3.000000 7.000000
-4.000000 -4.000000 0.000000
Number of iterations:3 Time:0.12394 millisecond
Delta:8 millisecond
fixed point
x^2 - 16
3
24
0.00001
File
/Users/FNSY/Desktop/matlab.txt
result
iteration error xi xi-1 delta time
3 0 -4 -4 8 0.123944
50
40
30
20
10
0
-10
1.6 1.8 2 2.2 2.4 2.6 2.8 3

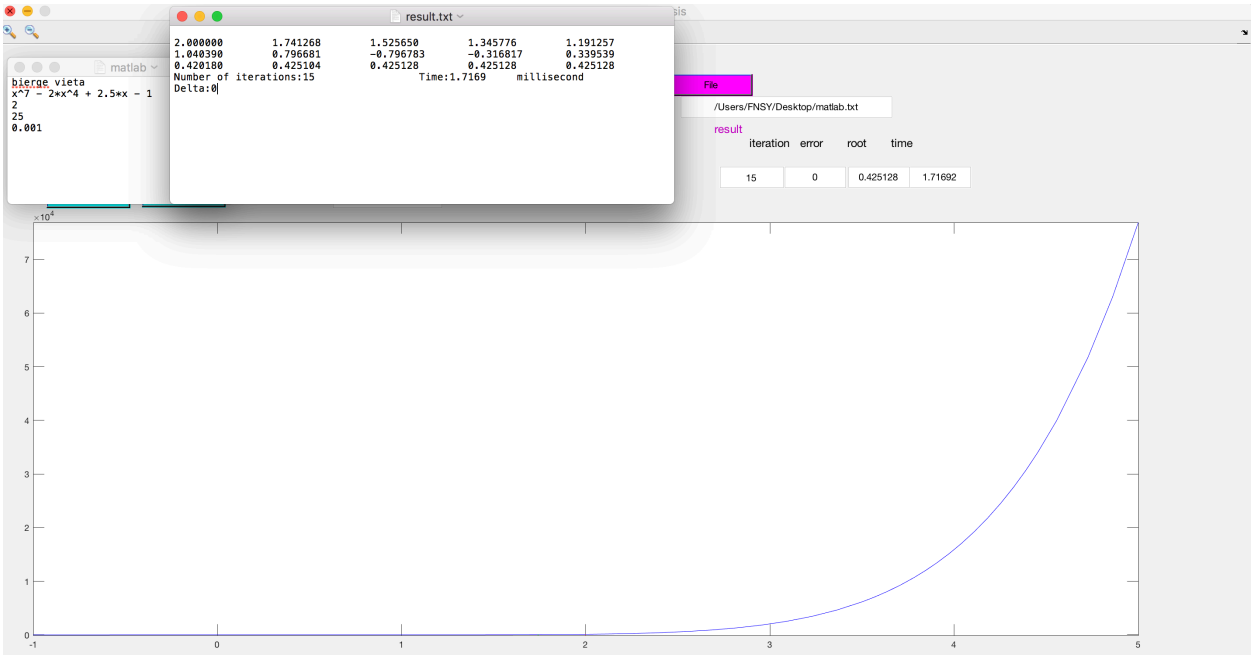
▸ Newton-Raphson



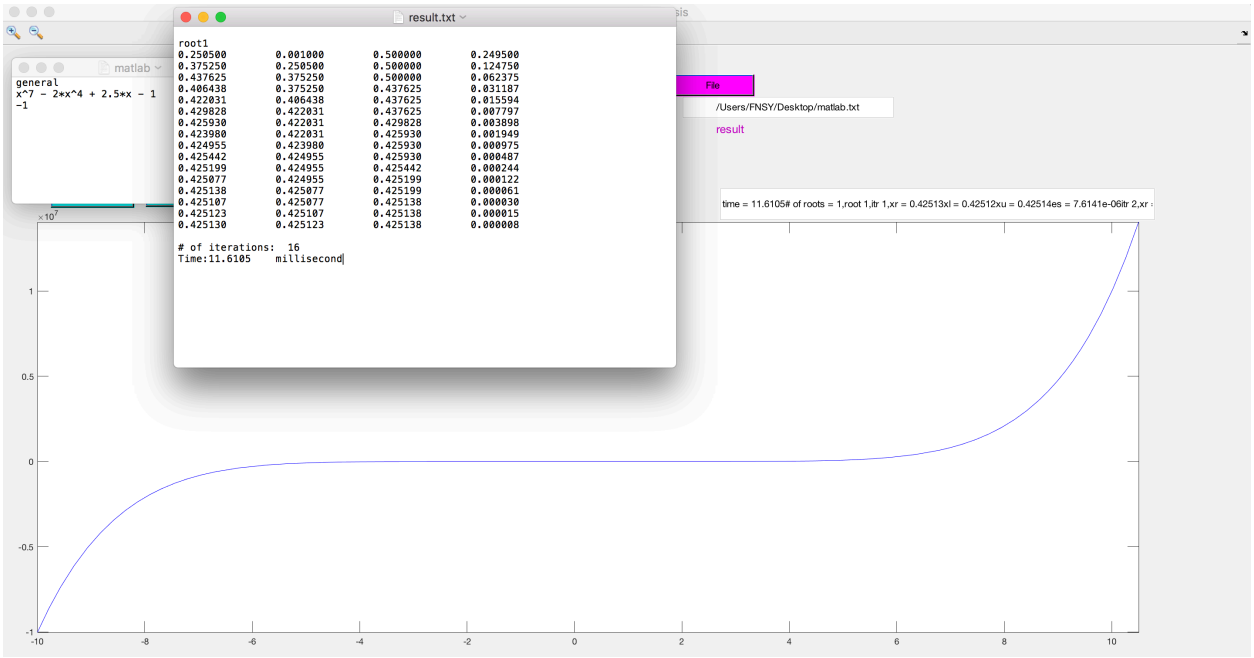
▸ Secant



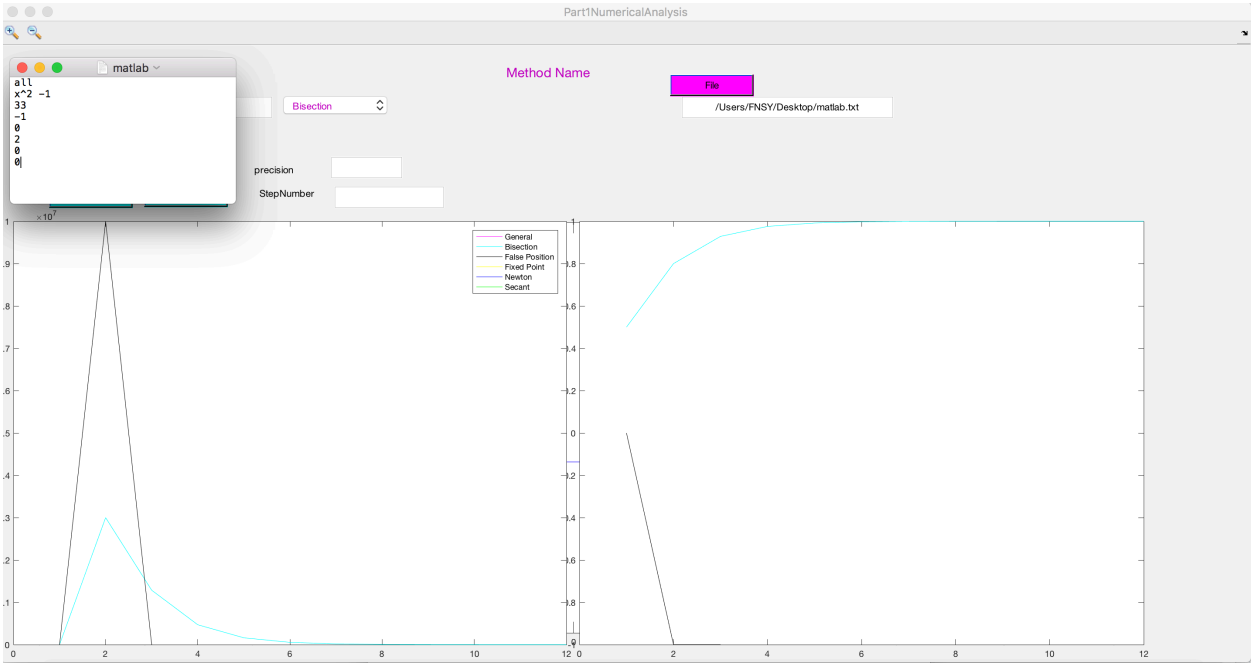
▸ Bierge Vieta



▸ General Method



All Methods



1.1 general method.txt

```
root1
-1.249500       -1.499000       -1.000000       0.249500
-1.124750       -1.249500       -1.000000       0.124750
-1.062375       -1.124750       -1.000000       0.062375
-1.031188       -1.062375       -1.000000       0.031188
-1.015594       -1.031188       -1.000000       0.015594
-1.007797       -1.015594       -1.000000       0.007797
-1.003898       -1.007797       -1.000000       0.003898
-1.001949       -1.003898       -1.000000       0.001949
-1.000975       -1.001949       -1.000000       0.000975
-1.000487       -1.000975       -1.000000       0.000487
-1.000244       -1.000487       -1.000000       0.000244
-1.000122       -1.000244       -1.000000       0.000122
-1.000061       -1.000122       -1.000000       0.000061
-1.000030       -1.000061       -1.000000       0.000030
-1.000015       -1.000030       -1.000000       0.000015
-1.000008       -1.000015       -1.000000       0.000008

# of iterations:  16
root2
0.750500        0.501000        1.000000        0.249500
0.875250        0.750500        1.000000        0.124750
0.937625        0.875250        1.000000        0.062375
0.968812        0.937625        1.000000        0.031188
0.984406        0.968812        1.000000        0.015594
0.992203        0.984406        1.000000        0.007797
0.996102        0.992203        1.000000        0.003898
0.998051        0.996102        1.000000        0.001949
0.999025        0.998051        1.000000        0.000975
0.999513        0.999025        1.000000        0.000487
0.999756        0.999513        1.000000        0.000244
0.999878        0.999756        1.000000        0.000122
0.999939        0.999878        1.000000        0.000061
0.999970        0.999939        1.000000        0.000030
0.999985        0.999970        1.000000        0.000015
0.999992        0.999985        1.000000        0.000008

# of iterations:  16
Time:10.8021    millisecond
```

1.2 bierge vieta.txt

```
0.000000        Inf     NaN     NaN
Number of iterations:1                  Time:0.20676    millisecond
Delta:0
```

1.3 Newton.txt

```
roots
0.000000        Inf     NaN
Number of iterations:1                  Time:1.031      millisecond
Delta:Inf
```

1.4 Fixed Point.txt

```
root            before          error
0.000000        0.000000        0.000000
-1.000000       0.000000        1.000000
-1.000000       -1.000000       0.000000
Number of iterations:3                  Time:0.12973    millisecond
Delta:0
```

1.5 Bisection.txt

```
root            xlower          xupper          error
1.000000        0.000000        2.000000
Number of iterations:1                  Time:0.027685   millisecond
Delta:0
```

1.6 False Position.txt

```
root            xlower          xupper          error
0.500000        0.000000        2.000000        0.000000
0.800000        0.500000        2.000000        0.300000
0.928571        0.800000        2.000000        0.128571
0.975610        0.928571        2.000000        0.047038
0.991803        0.975610        2.000000        0.016194
0.997260        0.991803        2.000000        0.005457
0.999086        0.997260        2.000000        0.001826
0.999695        0.999086        2.000000        0.000609
0.999898        0.999695        2.000000        0.000203
0.999966        0.999898        2.000000        0.000068
0.999989        0.999966        2.000000        0.000023
0.999996        0.999989        2.000000        0.000008
Number of iterations:12                 Time:0.033175   millisecond
Delta:0
```

1.7 Secent.txt

```
root
NaN
Number of iterations:1                  Time:0.089918   millisecond
Delta:0
```

5. Pitfalls

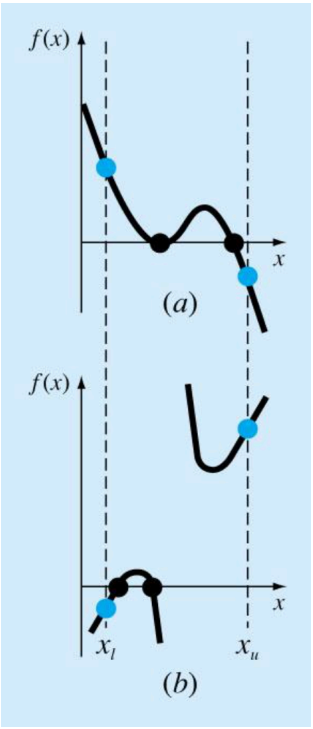
I. Bisection Method

- Can't detect multiple roots where if an interval has an even number of roots *bisection method* cannot detect any of them. Example: **Y(x) = x^2-4*x+4** on **[0,4].**

```
>>  [arr,siz,mesg] = biSection('x^2-4*x+4','0','4',50,0.00001)

arr =

     []


siz =

     0


mesg =

%%error in xl and xu%%
```

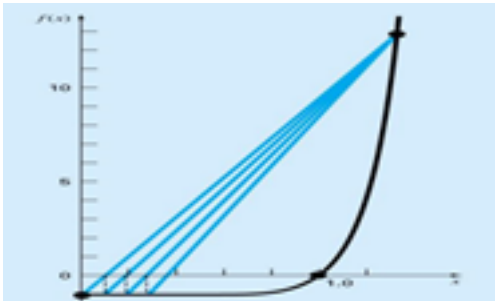
- If : function is not continuous, *bisection method* will detect a root that is not true.

```
>> [arr,siz,mesg] = biSection('x^-3','-1','1',50,0.00001)

arr =

         0   -1.0000    1.0000    1.0000
   -0.5000   -1.0000         0    0.5000
   -0.2500   -0.5000         0    0.2500
   -0.1250   -0.2500         0    0.1250
   -0.0625   -0.1250         0    0.0625
   -0.0313   -0.0625         0    0.0313
   -0.0156   -0.0313         0    0.0156
   -0.0078   -0.0156         0    0.0078
   -0.0039   -0.0078         0    0.0039
   -0.0020   -0.0039         0    0.0020
   -0.0010   -0.0020         0    0.0010
   -0.0005   -0.0010         0    0.0005
   -0.0002   -0.0005         0    0.0002
   -0.0001   -0.0002         0    0.0001
   -0.0001   -0.0001         0    0.0001
   -0.0000   -0.0001         0    0.0000
   -0.0000   -0.0000         0    0.0000
   -0.0000   -0.0000         0    0.0000


siz =

    18
```



II. False Position Method

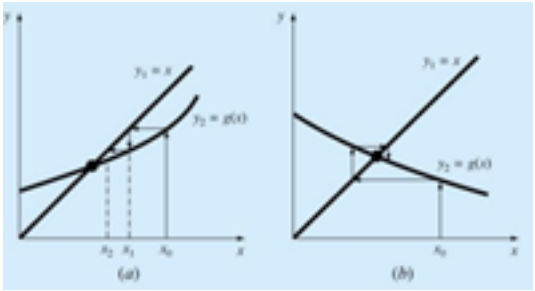
- Can't detect multiple roots (like bisection's behavior) where if an interval has an even number of roots *false position method* cannot detect any of them.



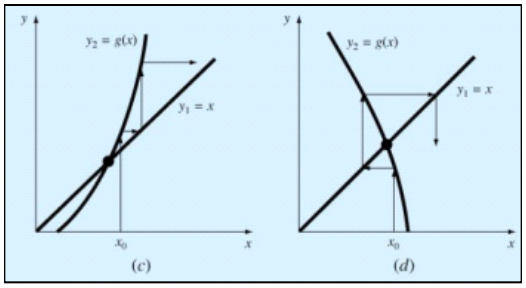
Solution: make an algorithm to detect this , then use bracketing method formula to get root.
Example : $\mathbf{y(x) = x^3 - 6x^2 + 11x^{-6}}$, on interval **[1,2].**

III. Fixed Point Method

- As we get **g(x)** from main equation and construct a magic formula , $\mathbf{X_i+1 = g\ (X_i\ ),}$ the function, **g(x)** will converge if **|g'(x)| < 1** as the error decreases with each iteration.



and will diverge if **|g'(x)| > 1** as the error increase with each iteration .



result

iteration	error	xi	xi-1	delta	time
50	NaN	Inf	Inf	-Inf	0.131043

- **The algorithm** we used to get **g(x)** , is to detect the existence of $\mathbf{ax^1}$ and remove it from equation but if it not found we add **x** then subtract **x**.
- example $\mathbf{y(x) = x^2+2x^{-3}}$

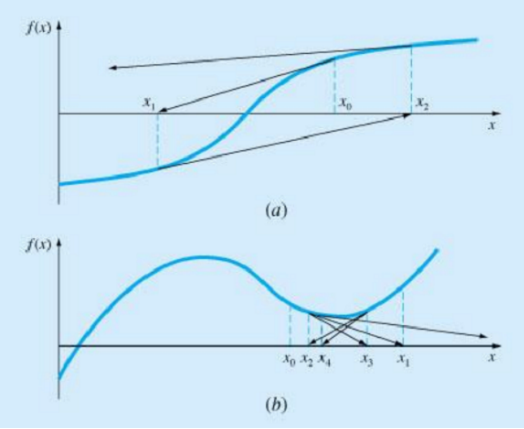
```
>> [arr, siz, mesg] = fixedPoint('x^2+2*x-3','0',50,0.00001)

arr =

         0         0         0
    1.5000         0    1.5000
    0.3750    1.5000    1.1250
    1.4297    0.3750    1.0547
    0.4780    1.4297    0.9517
    1.3858    0.4780    0.9078
    0.5398    1.3858    0.8459
    1.3543    0.5398    0.8145
    0.5830    1.3543    0.7713
    1.3301    0.5830    0.7471
    0.6154    1.3301    0.7146
    1.3106    0.6154    0.6952
    0.6411    1.3106    0.6695
    1.2945    0.6411    0.6533
    0.6622    1.2945    0.6323
    1.2808    0.6622    0.6186
    0.6798    1.2808    0.6009
    1.2689    0.6798    0.5891
    0.6949    1.2689    0.5740
    1.2585    0.6949    0.5636
    0.7080    1.2585    0.5505
    1.2493    0.7080    0.5413
```

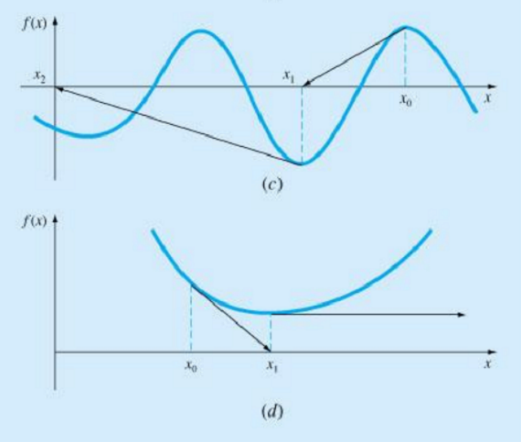
2.0000	0	0
-0.5000	2.0000	2.5000
1.3750	-0.5000	1.8750
0.5547	1.3750	0.8203
1.3462	0.5547	0.7915
0.5939	1.3462	0.7522
1.3236	0.5939	0.7297
0.6240	1.3236	0.6996
1.3053	0.6240	0.6813
0.6481	1.3053	0.6572
1.2900	0.6481	0.6419
0.6680	1.2900	0.6220
1.2769	0.6680	0.6090
0.6847	1.2769	0.5922
1.2656	0.6847	0.5808
0.6992	1.2656	0.5664
1.2556	0.6992	0.5564
0.7118	1.2556	0.5438
1.2467	0.7118	0.5349
0.7229	1.2467	0.5238

Method diverges as **g'(x) = -x** and at x = 2 , **g'(x) =-2 < 1**

IV. Newton

- An inflection point (f"(x)=0) at the vicinity of a root causes divergence.
- A local maximum or minimum causes oscillations.
- It may jump from one location close to one root to a location that is several roots away.
- A zero slope causes division by zero.
- No general convergence criteria for Newton Raphson method.
- it diverges when the root is a multiple roots.





V. Bierge Vieta

- it detects only one root even though there are multiple roots -same behavior obtained from bracketing methods-.
- Able to deal *only* with polynomial functions and cannot deal with exponential or trigonometric functions.

VI. General Method

- General Method depends mainly on bisection method therefore it is simply expectable to have the dame pitfalls like that one previously discussed.

Part II

System of linear equations solving techniques

1. Methods' Pseudo Codes

I. Removing Lower Triangle Function

Although we are implementing 4 different methods / techniques to solve systems of linear equations, 3 of them have the same core of using gauss' technique to remove the lower triangle of the coefficients matrix, these techniques are:

- Gauss Elimination
- Gauss - Jordan
- LU Decomposition[6]

The function takes 3 parameters which are:

- The matrix to eliminate its lower triangle (gaussMatrix)
- Values after equal sign matrix (answer)
- Number of rows / columns of the square matrix (dimension)

The function returns the same 2 input after changes:

- The matrix to eliminate its lower triangle (gaussMatrix)
- Values after equal sign matrix (answer)

```
removeLowerTriangle
row = 2 // starting by the second row
for diagonal = 1 to dimension
    pivotVector = the pivot's row
    pivotAnswerVector = answer_diagonal
    while row <= dimension
        delta =  (element value)_row,diagonal / (pivot value)_diagonal,diagonal
        toBeAddedVector = delta *  pivotVector
        toBeAddedAnswerVector = delta * pivotAnswerVector
        gaussMatrix_row =  gaussMatrix_row - toBeAddedVector
        answer_row =  answer_row - toBeAddedAnswerVector
        row = row + 1
    end while
    row = the following pivot's row number
end for
end removeLowerTriangle
```

[6] At this case we store the values of delta at the lower triangle matrix.

II. Ordinary Gauss Elimination[7]

The function takes 2 parameters which are:

- Coefficient's 2D array (inputArray)
- Values after equal sign matrix (answer)

The function returns 1 Column matrix containing the values of unknowns.

```
Gauss
dimension = number of rows / columns of the matrix
call to removeLowerTriangle function (explained above)
finalAnswer(last column) = answer(last column) / gaussMatrix(dimension,dimension)
rowNum = dimension - 1 // starting from the row before last.
while rowNum > 0
    sigma = 0 // reset value of the sum each time
    for nextRows =  rowNum  + 1 to dimension
        sigma = sigma + gaussMatrix(rowNum, nextRows) *
finalAnswer(nextRows)
    end for
    finalAnswer(rowNum) = (answer(rowNum) - sigma) / pivot
    rowNum = rowNum - 1
end while
end Gauss
```

III. Gauss - Jordan

The function takes 2 parameters which are:

- Coefficient's 2D array (inputArray)
- Values after equal sign matrix (answer)

The function returns 1 Column matrix containing the values of unknowns.

```
Gauss-Jordan
dimension = number of rows / columns of the matrix
call to removeLowerTriangle function
row = 1
for diagonal = 1 to dimension
```

[7] i.e. No pivoting process

```
    pivot = 1 / gaussMatrix_diagonal,diagonal
    gaussMatrixRow_row = pivot *  gaussMatrixRow_row
     answer_row =  pivot * answer_row
    row = row + 1
end for
flip gaussMatrix vertically
flip gaussMatrix horizontally
flip answer matrix horizontally
remove the lower triangle of gaussMatrix again
call to removeLowerTriangle function
re-flip answer matrix horizontally
end Gauss-Jordan
```

IV. LU Decomposition

The function takes 2 parameters which are:

- Coefficient's 2D array (inputArray)
- Values after equal sign matrix (answer)

The function returns 1 Column matrix containing the values of unknowns.

```
LU-Decomposition
Initializing the matrices.
call to removeLowerTriangle function
getting upper and lower triangles matrices
stage1 = zeros(dimension,1)
stage1(1) = answer(1)
for i = 2 to dimension
    sigma = 0
    for j = 1 to i - 1
        sigma = sigma + lower(i,j) * stage1(j)
    end
    stage1(i) = answer(i) - sigma
end
finalAnswer = zeros(dimension,1)
finalAnswer(dimension) = stage1(dimension) /
upper(dimension,dimension)
i = dimension - 1
```

```
while i > 0
    sigma = 0
    for j = i + 1 : dimension
        sigma = sigma + upper(i,j) * finalAnswer(j)
    end for
    finalAnswer(i) = (stage1(i) - sigma ) / upper(i,i)
    i = i - 1
end while
end LU-Decomposition
```

V. Gauss Seidel

The function takes 5 parameters which are:

- Coefficient's 2D array (inputArray)
- Values after equal sign matrix (answer)
- Initial values matrix (initalMatrix)
- Max iterations
- Allowed error (epsilon)

The function returns 2 items which are:

- Answer Matrix
- Erros

```
GaussSeidel
initialize variables and matrices
// check the first  condition of convergence
for diagonal= 1 : dimension
    dominant = abs(coffMatrix(diagonal,diagonal))
    dominantCounter= 1
    sum=0
    while( dominantCounter <= dimension)
        if dominantCounter ≠  diagonal
            sum = sum+abs(coffMatrix(diagonal,dominantCounter))
            dominantCounter = dominantCounter+1
        else

            dominantCounter = dominantCounter+1

        end
    end if
```

```
    if dominant< sum
        error
    elseif dominant > sum
        checkSecondConditon=checkSecondConditon+1
    end if
   end while

// check the second  condition of convergence
if checkSecondConditon = 0
    error
end
initialize error = 100, iterate = 1
while iterate <= maxIterations AND maxError > epsilon
   for diagonal= 1 to dimension
        numerator= answerMatrix(diagonal)
        counter=1
 // calculate numerator for the  current  variable
        while counter<=dimension
            if counter > diagonal
            numerator=numerator-coffMatrix(diagonal,counter) * prevCalculatedRoots(counter)
                counter= counter+1
            elseif counter < diagonal
            numerator=numerator-coffMatrix(diagonal,counter) * calculatedRoots(counter)
                counter= counter+1
            else
                counter= counter+1
            end if
        end while
 // end calculating numerator for the  current  variable
        calculatedRoots(diagonal)= numerator / coffMatrix(diagonal,diagonal)
        errorColumn(diagonal)= abs(calculatedRoots(diagonal)-
prevCalculatedRoots(diagonal))
    end for
    maxError= max(errorColumn)
    prevCalculatedRoots= calculatedRoots
    if  iterate==1
        rootsMatrix = calculatedRoots
        errors=errorColumn
    else
        rootsMatrix =[rootsMatrix  calculatedRoots']
        errors=[ errors errorColumn']
    end if
    iterate= iterate+1
end while
end GaussSeidel
```

2. Data Structures Used

I. Matrices

Matrices where used broadly in this part of the assignment, since, we deal with the input equations coefficients as matrices to perform the desired processes on them in almost all methods.

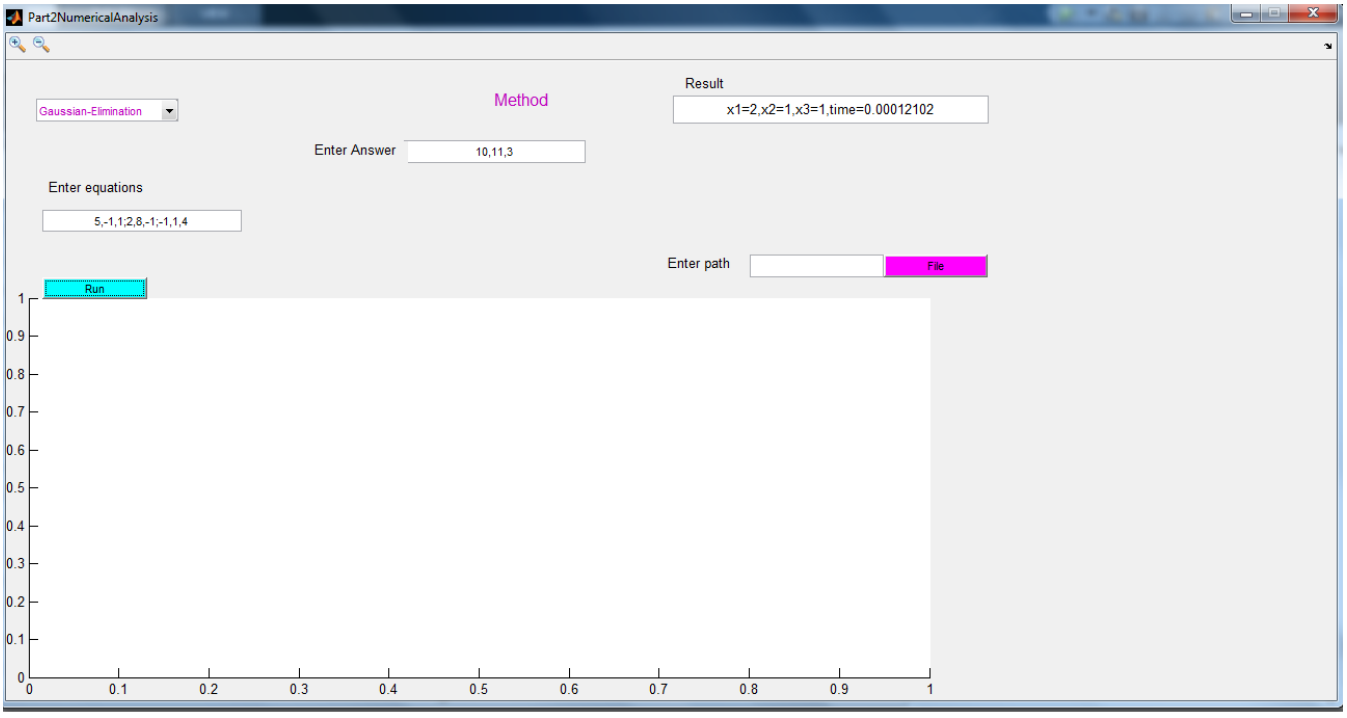
II. Arrays

Arrays were used in a tiny scope to export the final answers of the input equations.

3. Sample Tests

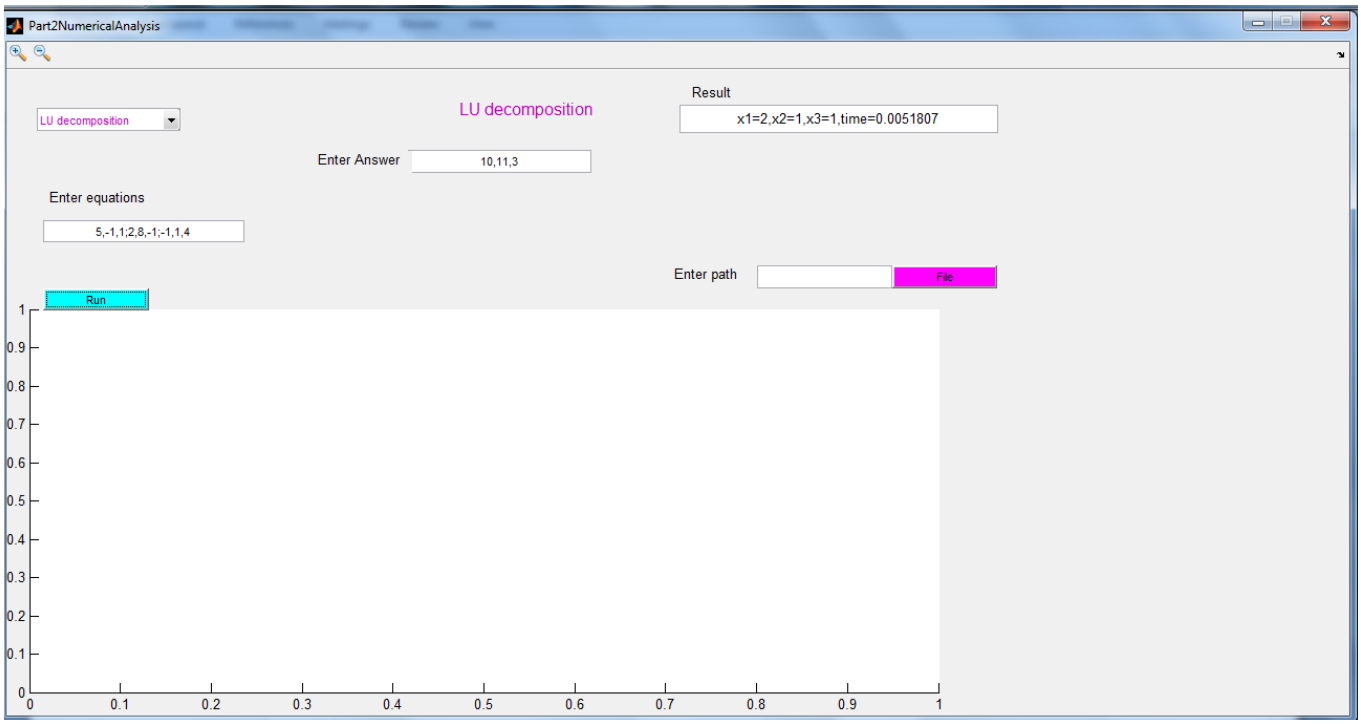
I. Via GUI Operations

- **Gaussian Elimination**



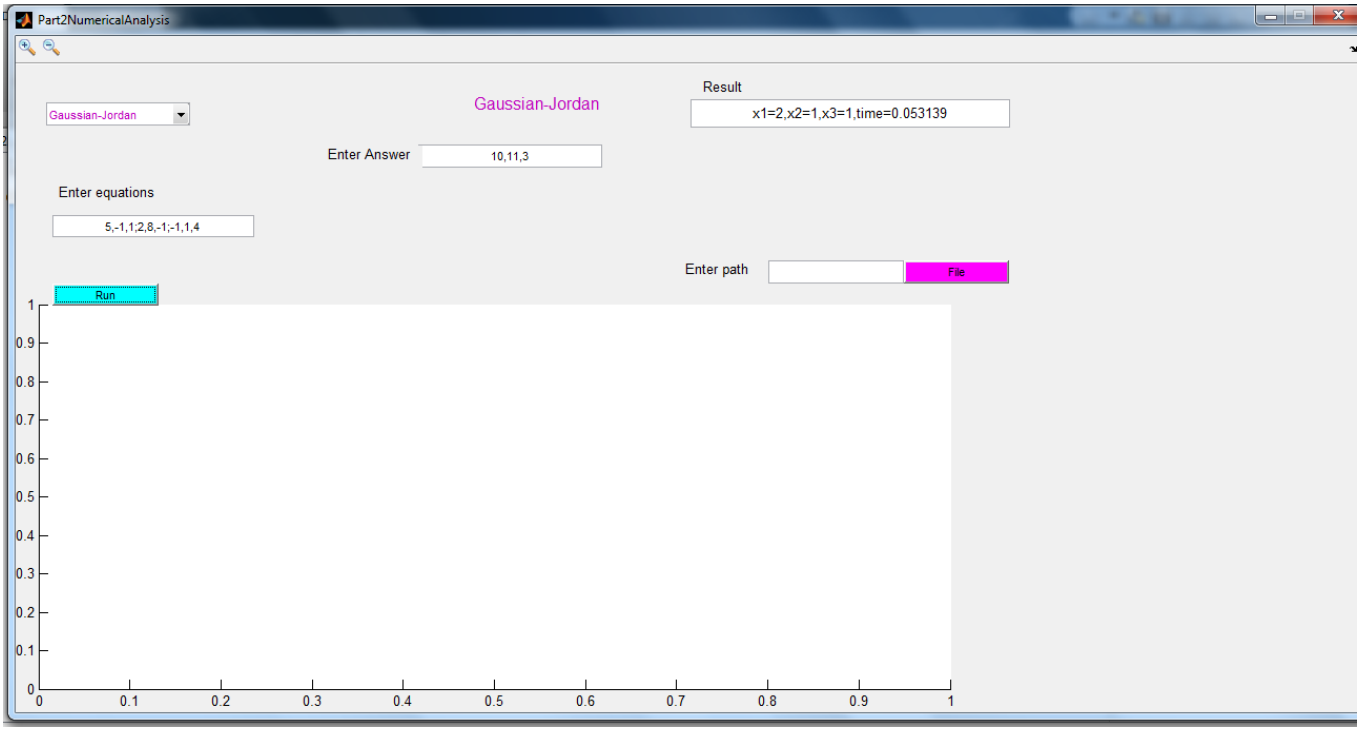

```
1  Gauss Elimination
2  execution time =  0.000121
3       X 1  = 2.000000
4       X 2  = 1.000000
5       X 3  = 1.000000
6
```

- **Lower-Upper Decomposition**



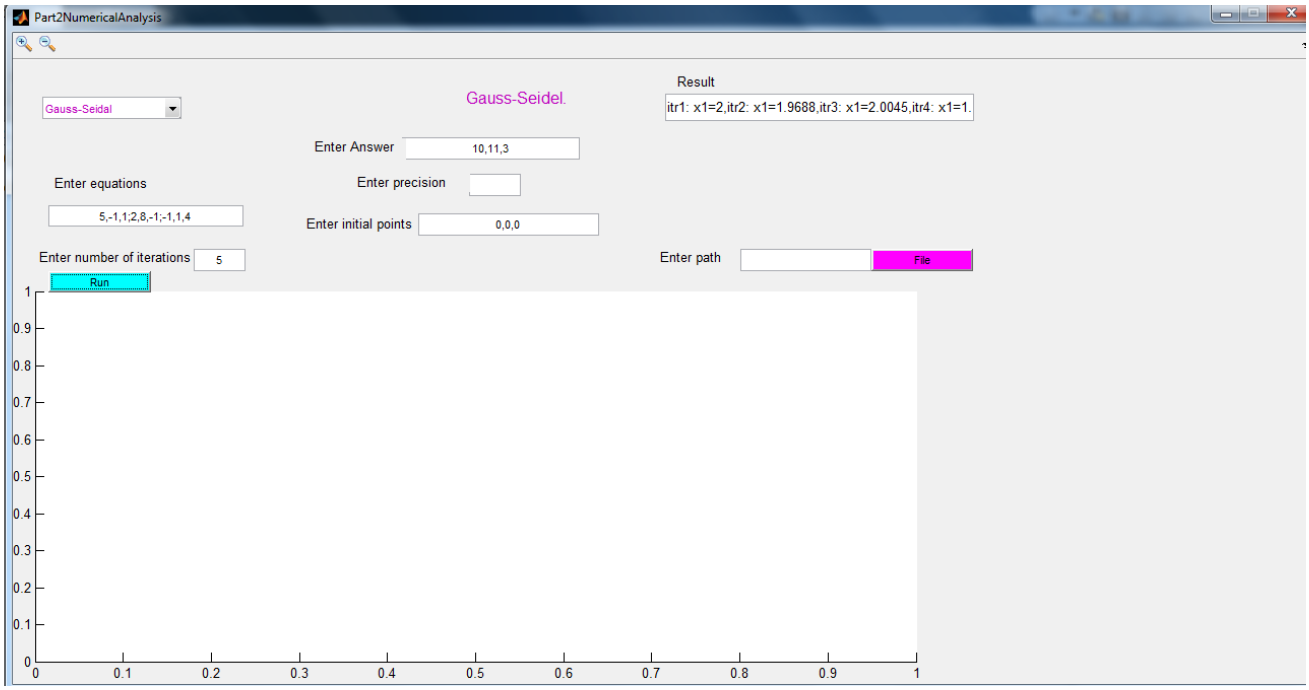

```
1  LU Decomposition
2  execution time =  0.005181
3       X 1  = 2.000000
4       X 2  = 1.000000
5       X 3  = 1.000000
6
```

- **Gauss Jordan**



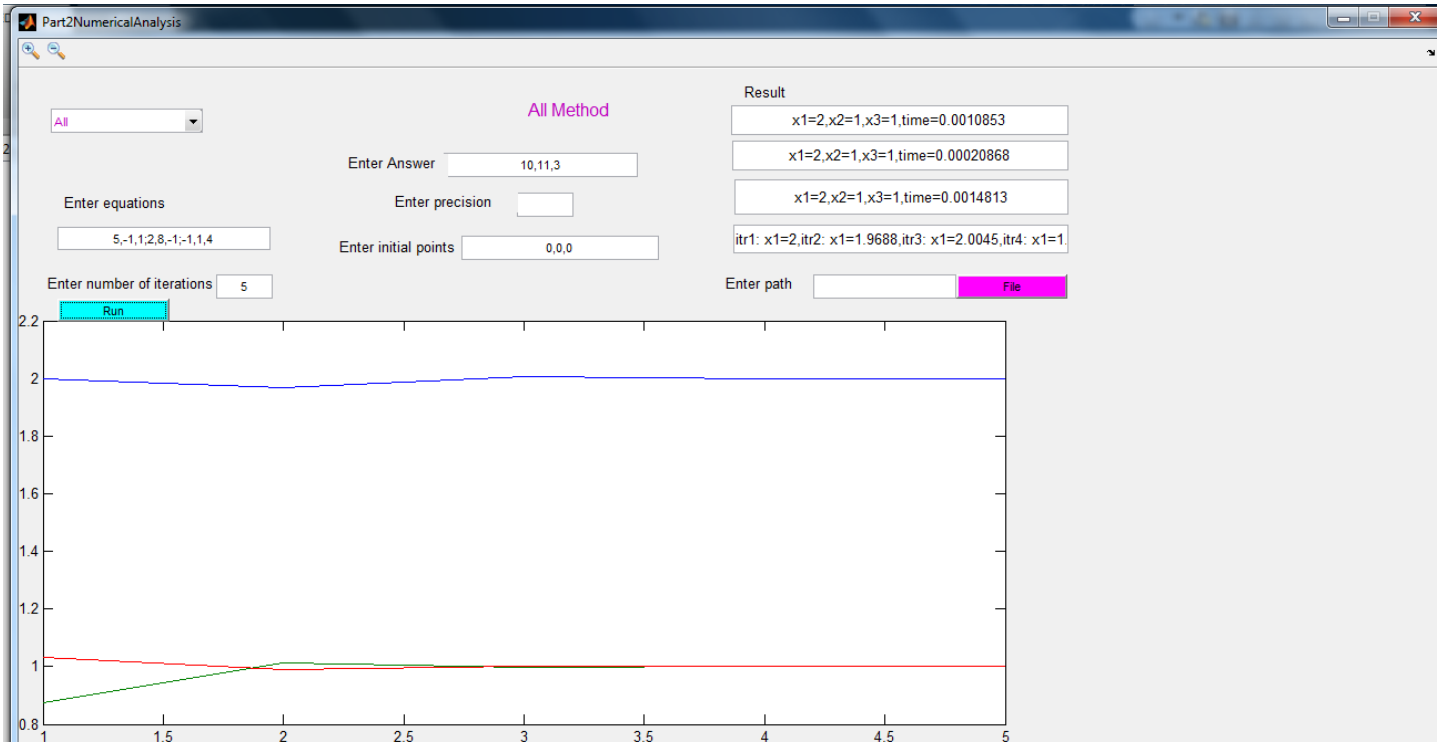

```
Gauss_Jordan
execution time =  0.053139
     X 1  = 2.000000
     X 2  = 1.000000
     X 3  = 1.000000

```

- **Gauss Seidel**




```
Gauss_Seidel
execution time =  0.011846
rootsMatrix
2.000000   1.968750   2.004492   1.999159   2.000149
0.875000   1.011719   0.997534   1.000428   0.999923
1.031250   0.989258   1.001740   0.999683   1.000056
errors
2.000000   0.031250   0.035742   0.005333   0.000990
0.875000   0.136719   0.014185   0.002894   0.000505
1.031250   0.041992   0.012482   0.002057   0.000374
```

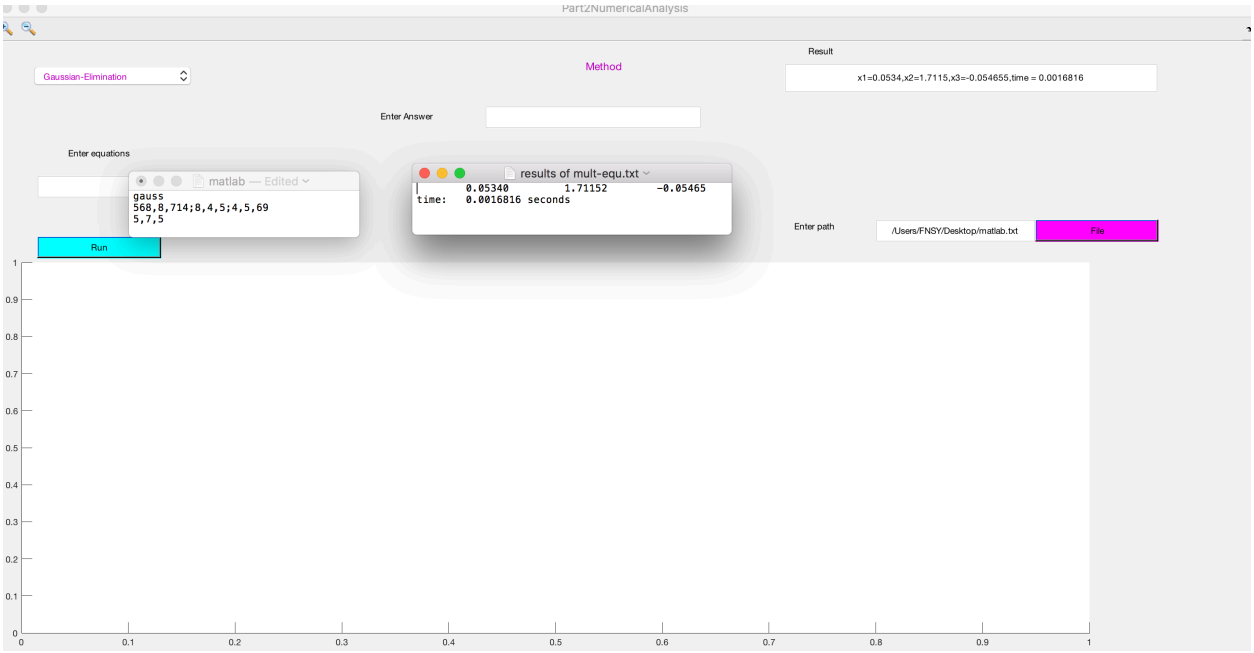
- **All Methods**



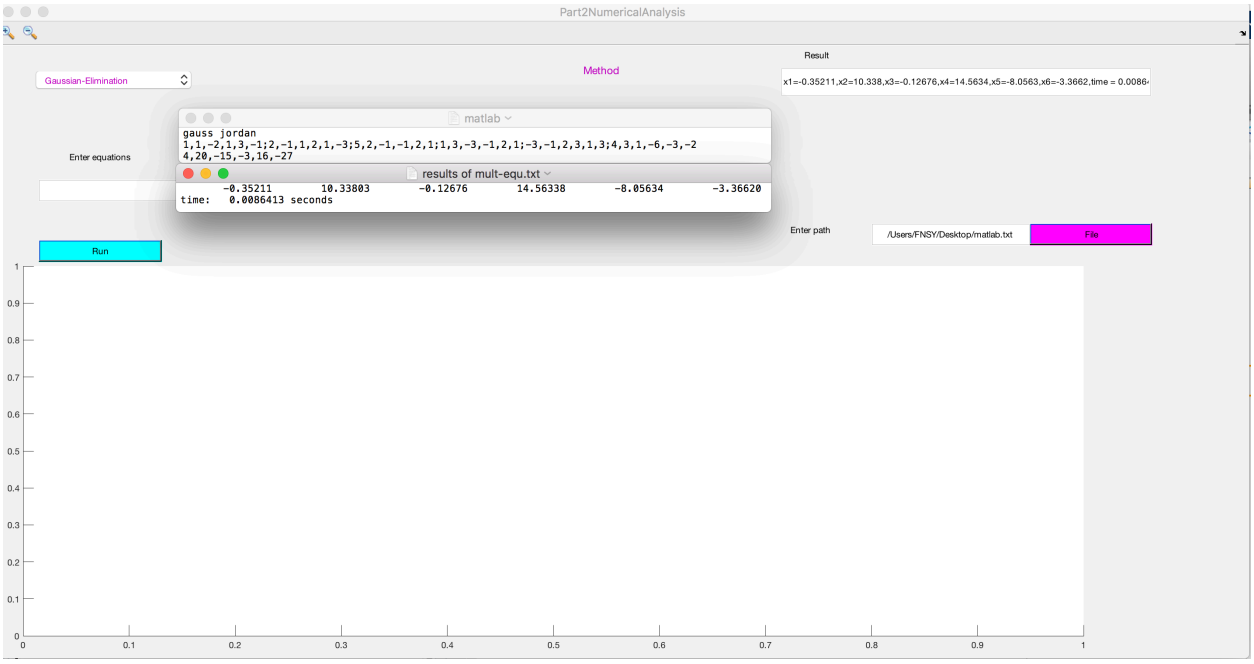

```
Gauss Elimination
execution time =  0.001085
     X 1  = 2.000000
     X 2  = 1.000000
     X 3  = 1.000000

LU Decomposition
execution time =  0.000209
     X 1  = 2.000000
     X 2  = 1.000000
     X 3  = 1.000000

Gauss_Jordan
execution time =  0.001481
     X 1  = 2.000000
     X 2  = 1.000000
     X 3  = 1.000000

Gauss_Seidel
execution time =  0.000342
rootsMatrix
2.000000   1.968750   2.004492   1.999159   2.000149
0.875000   1.011719   0.997534   1.000428   0.999923
1.031250   0.989258   1.001740   0.999683   1.000056
errors
2.000000   0.031250   0.035742   0.005333   0.000990
0.875000   0.136719   0.014185   0.002894   0.000505
1.031250   0.041992   0.012482   0.002057   0.000374

```

I. Via Files Operations

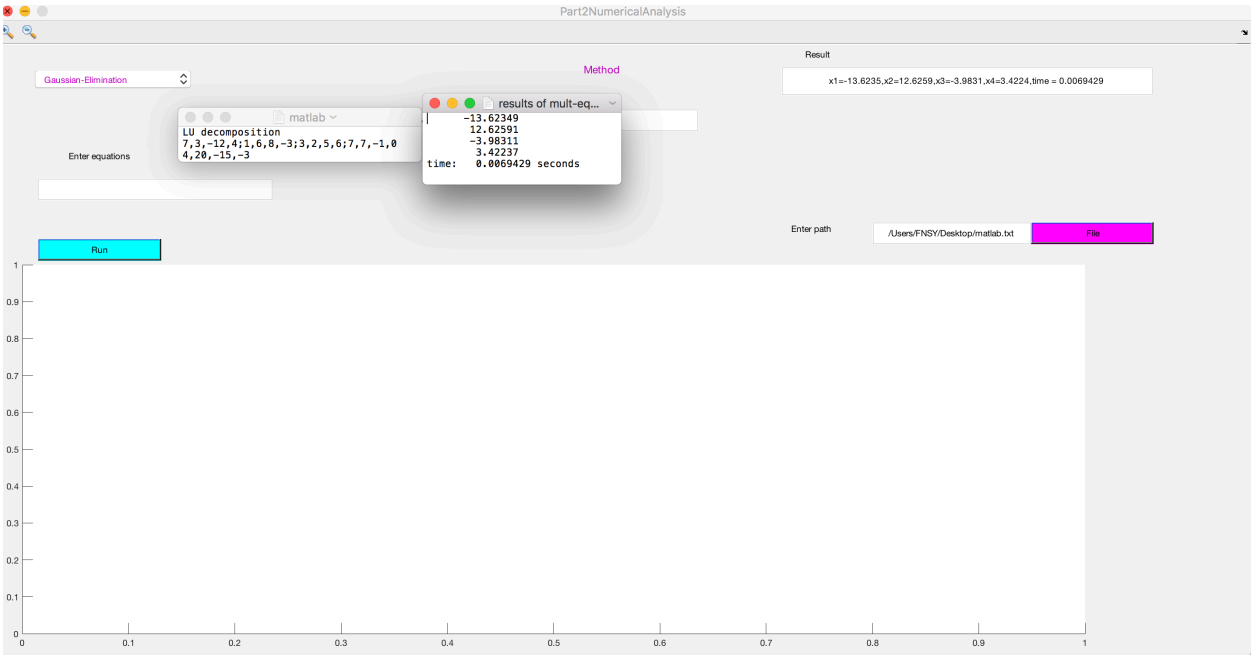
▸ Gaussian Elimination



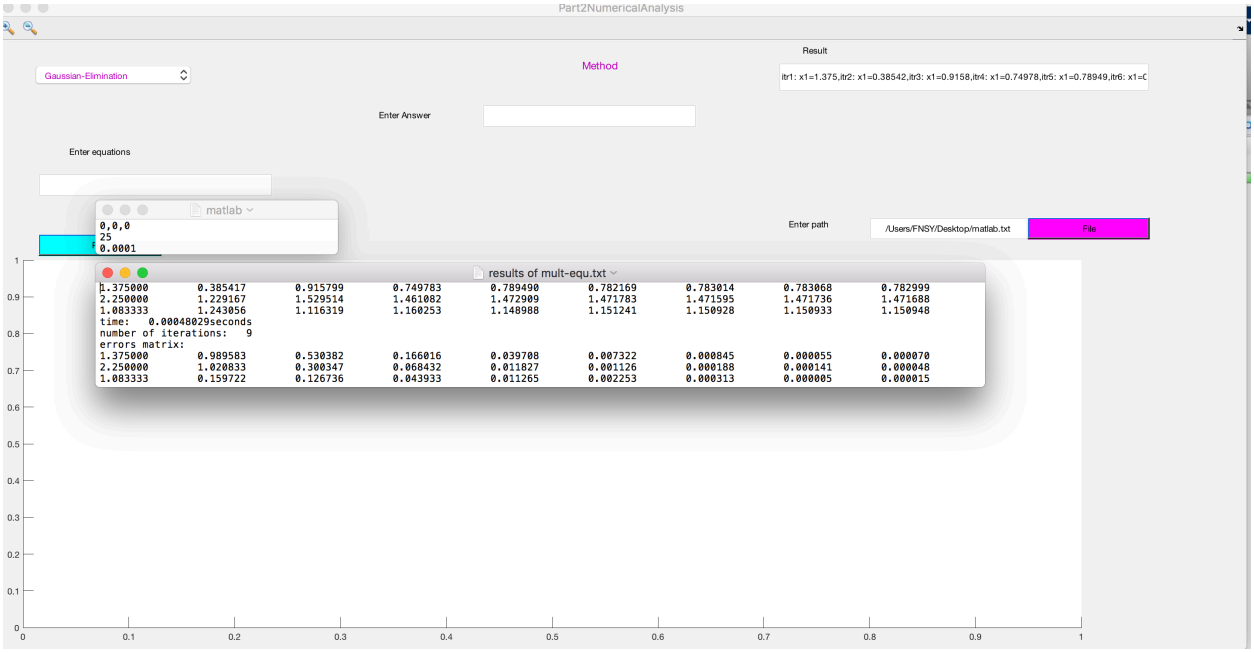
▸ Gauss Jordan



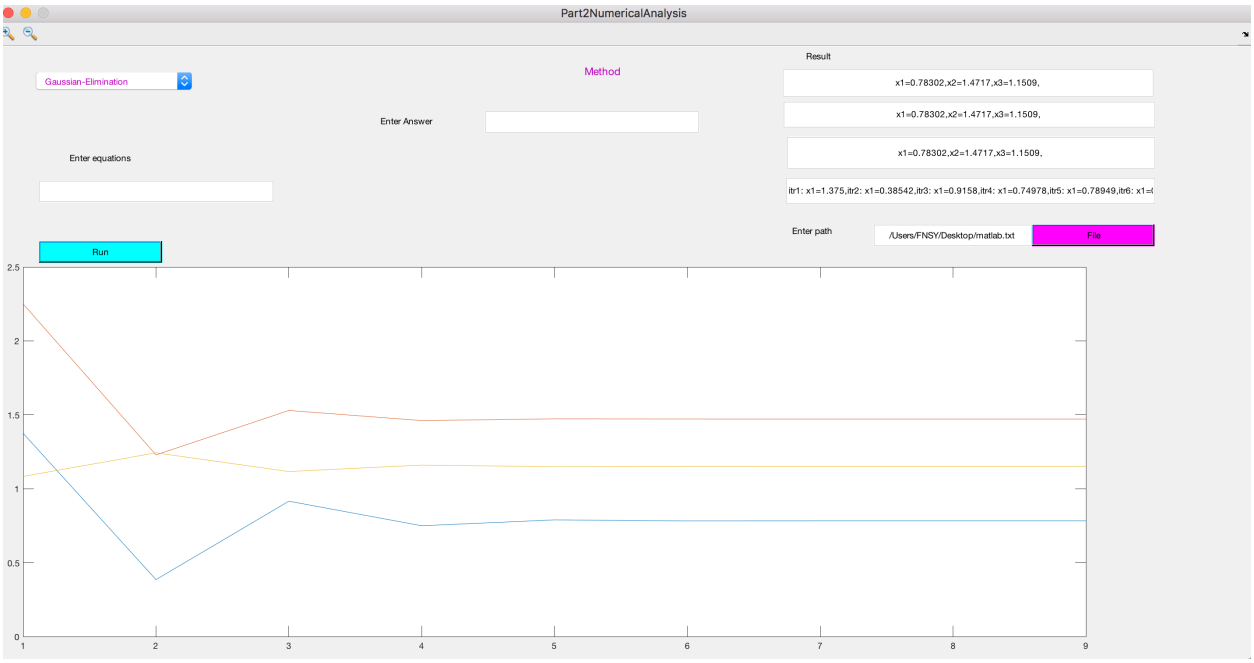
▸ Lower-Upper Decomposition

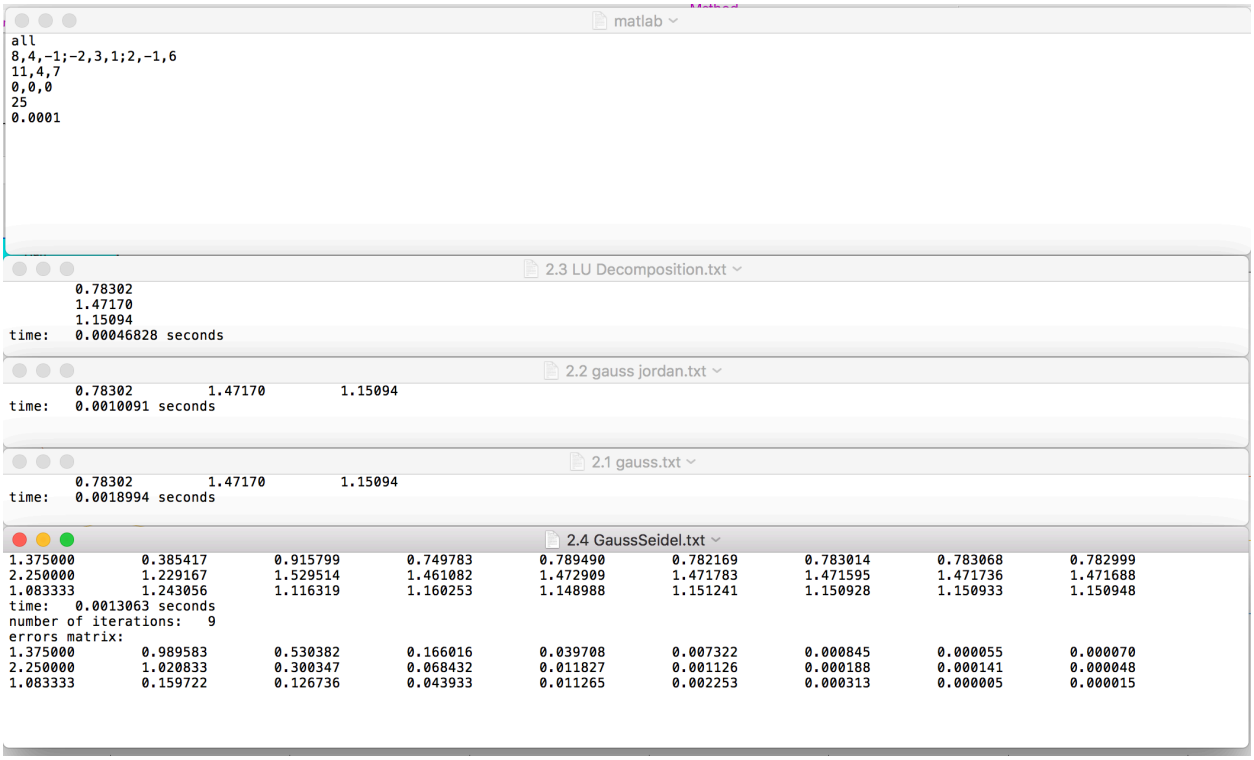


▸ Gauss Seidel



- **All Methods**

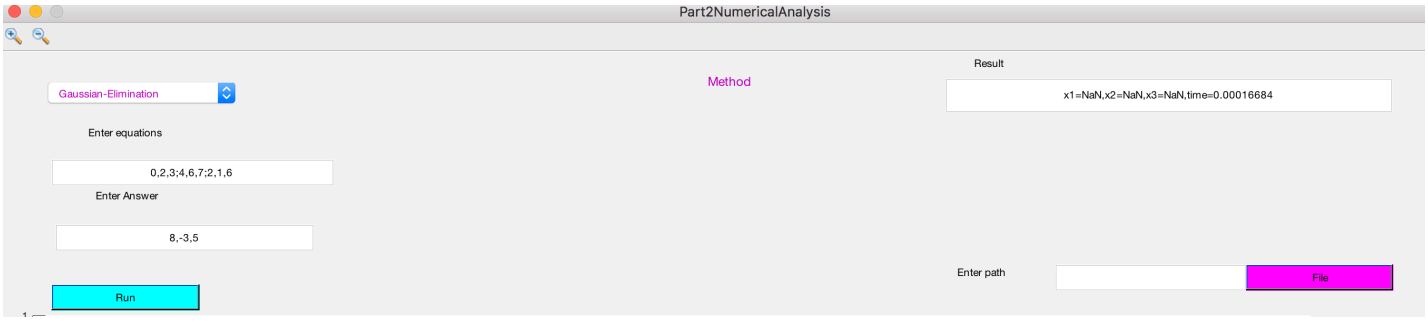

Part2NumericalAnalysis
Method
Gaussian-Elimination
Enter Answer
Enter equations
Result
x1=0.78302,x2=1.4717,x3=1.1509,
x1=0.78302,x2=1.4717,x3=1.1509,
x1=0.78302,x2=1.4717,x3=1.1509,
itr1: x1=1.375,itr2: x1=0.38542,itr3: x1=0.9158,itr4: x1=0.74978,itr5: x1=0.78949,itr6: x1=
Enter path
/Users/FNSY/Desktop/matlab.txt
File
Run


matlab
all
8,4,-1;-2,3,1;2,-1,6
11,4,7
0,0,0
25
0.0001
2.3 LU Decomposition.txt
0.78302
1.47170
1.15094
time: 0.00046828 seconds
2.2 gauss jordan.txt
0.78302 1.47170 1.15094
time: 0.0010091 seconds
2.1 gauss.txt
0.78302 1.47170 1.15094
time: 0.0018994 seconds
2.4 GaussSeidel.txt
1.375000 0.385417 0.915799 0.749783 0.789490 0.782169 0.783014 0.783068 0.782999
2.250000 1.229167 1.529514 1.461082 1.472909 1.471783 1.471595 1.471736 1.471688
1.083333 1.243056 1.116319 1.160253 1.148988 1.151241 1.150928 1.150933 1.150948
time: 0.0013063 seconds
number of iterations: 9
errors matrix:
1.375000 0.989583 0.530382 0.166016 0.039708 0.007322 0.000845 0.000055 0.000070
2.250000 1.020833 0.300347 0.068432 0.011827 0.001126 0.000188 0.000141 0.000048
1.083333 0.159722 0.126736 0.043933 0.011265 0.002253 0.000313 0.000005 0.000015

4. *Pitfalls*

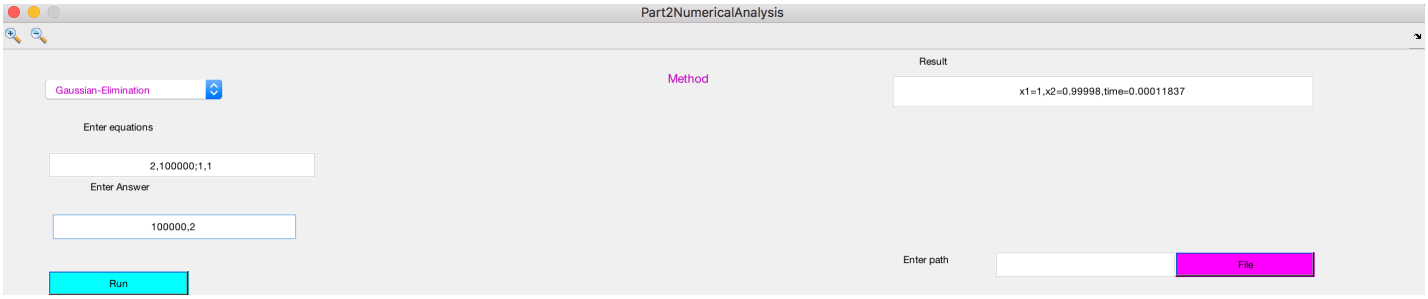
I. Gaussian Elimination / Gauss Jordan

- Dividing by zero is one of the most irritative pitfalls of the *naive gaussian elimination* and *gauss-jordan technique,* if the starting pivot is zero or one of the internal pivots tended to be zero during calculations; a division by zero occurs and the whole process is demolished.
  - Example:
    - **2y + 3z = 8**
    - **4x + 6y +7z = -3**
    - **2x + y +6z = 5**



Solution: pivoting is used to resolve this pitfall, where the largest element's row in the pivot column substitutes the current ill-pivot.

- Large round-off errors may cause catastrophic errors.
  - Example
    - **2x + 100000y =100000**
    - **x + y = 2**



- You can easily notice that the *naive gaussian elimination* returned values of [x = 1 & y = 0.99998], where the real values should be [x = 1.00002 & y = 0.99998]

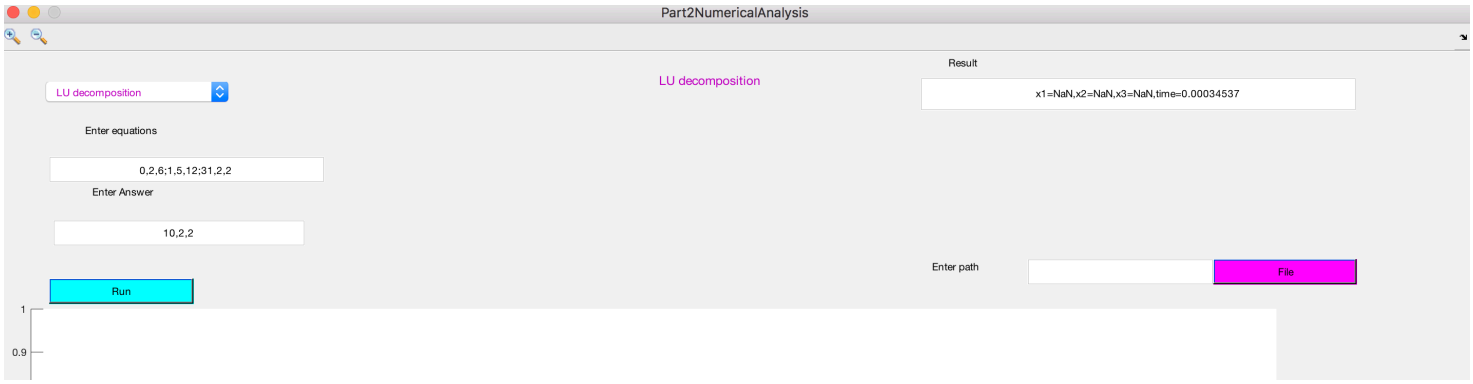
Solution: Apply scaling techniques on the coefficients to avoid the catastrophic errors.

- Ill conditioned system of equations is an another irritative pitfall for both methods, where the determinant of the matrix of coefficients is **not** zero but approaches zero.

Solution: after calculations, substitute using the obtained values and check the answers.

II. Lower-Upper Decomposition

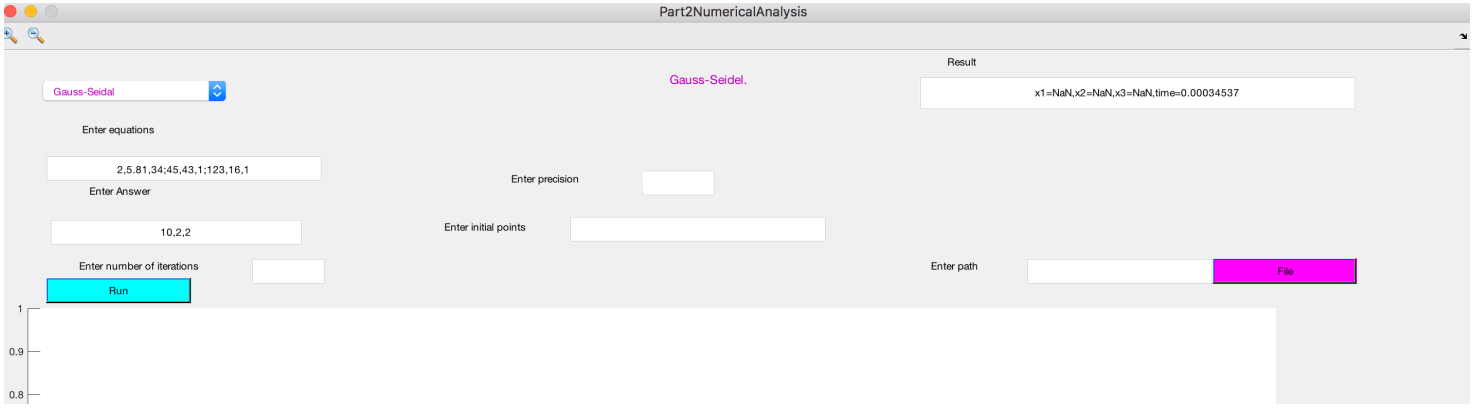
- Zero pivot issue, same as gauss and gauss jordan.



Solution: same trend of solution followed in gauss . gauss-jordan techniques which is the LU Decomposition with pivoting.

III. Gauss Seidel

- 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.
    - Example:
        - **2x + 5.81y + 34z = 10**
        - **45x + 43y + z = 2**
        - **123x + 16y + z = 2**


Part2NumericalAnalysis
Gauss-Seidal
Gauss-Seidel.
Result
x1=NaN,x2=NaN,x3=NaN,time=0.00034537
Enter equations
2,5.81,34;45,43,1;123,16,1
Enter precision
Enter Answer
10,2,2
Enter initial points
Enter number of iterations
Enter path
File
Run
1
0.9
0.8

Thank You