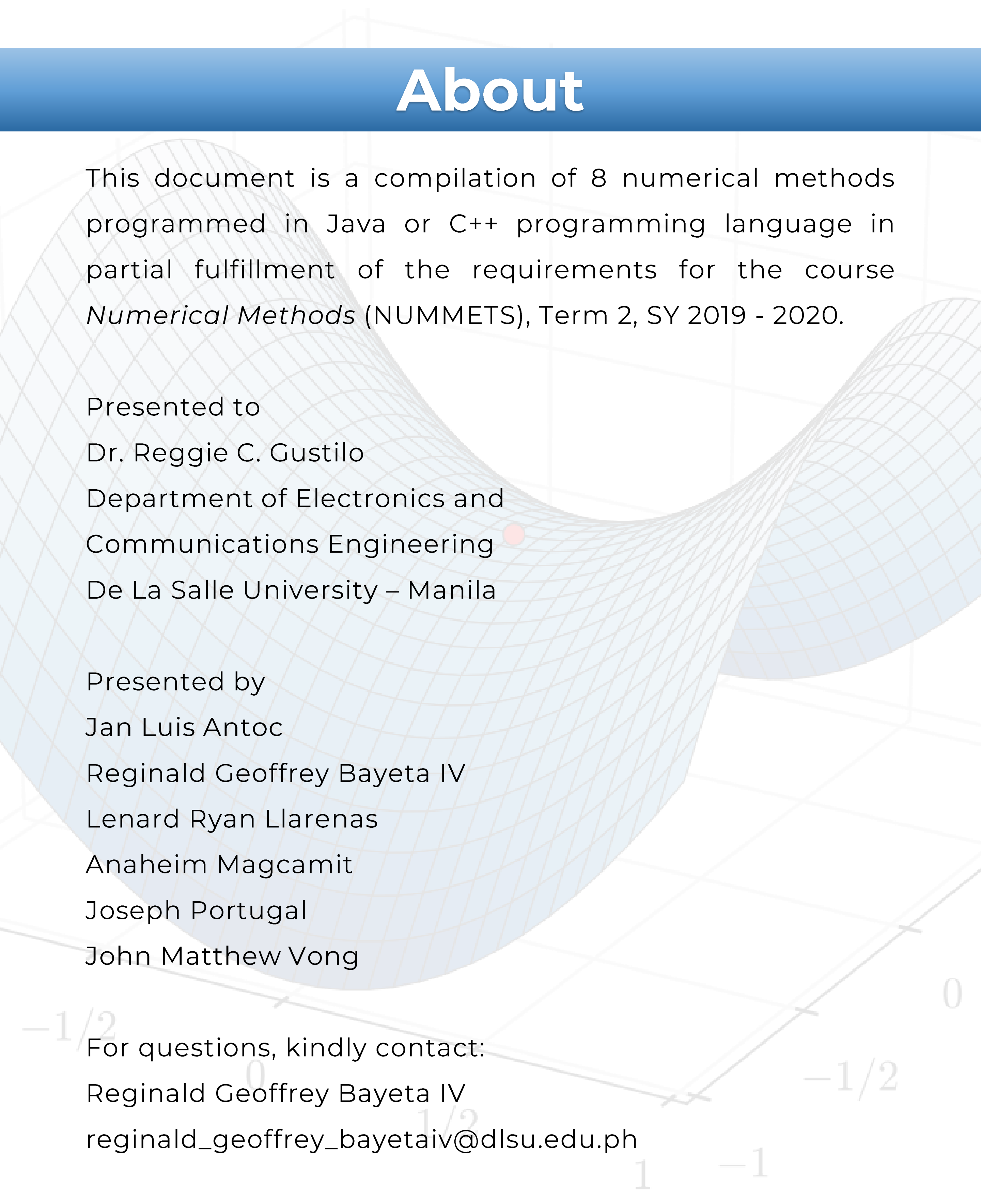
NUMMETS
Programming Numerical Methods in Java and C++

Gokongwei College of Engineering

DE LA SALLE UNIVERSITY

About



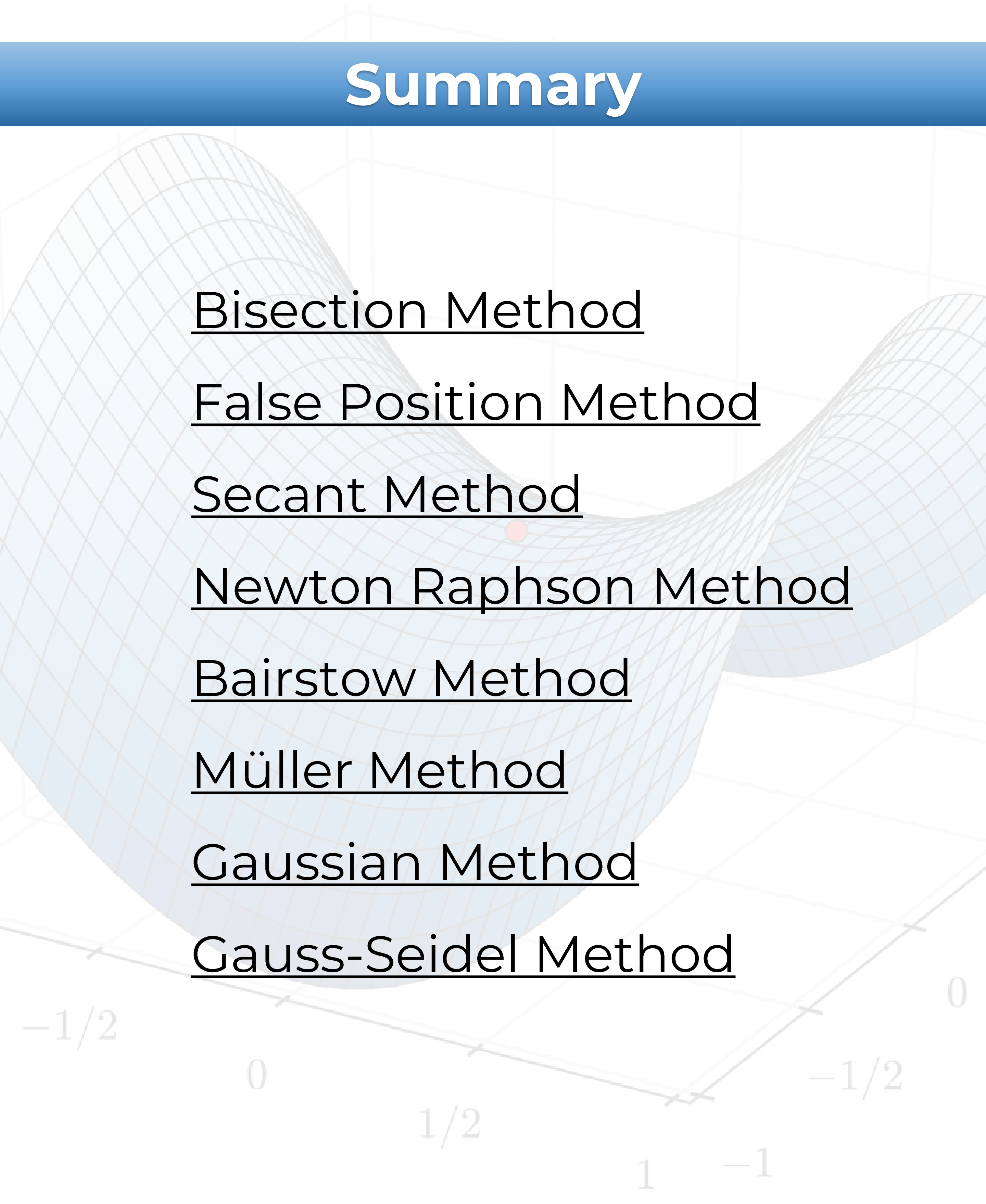
This document is a compilation of 8 numerical methods programmed in Java or C++ programming language in partial fulfillment of the requirements for the course *Numerical Methods* (NUMMETS), Term 2, SY 2019 - 2020.

Presented to
Dr. Reggie C. Gustilo
Department of Electronics and
Communications Engineering
De La Salle University – Manila

Presented by
Jan Luis Antoc
Reginald Geoffrey Bayeta IV
Lenard Ryan Llarenas
Anaheim Magcamit
Joseph Portugal
John Matthew Vong

For questions, kindly contact:
Reginald Geoffrey Bayeta IV
reginald_geoffrey_bayetaiv@dlsu.edu.ph

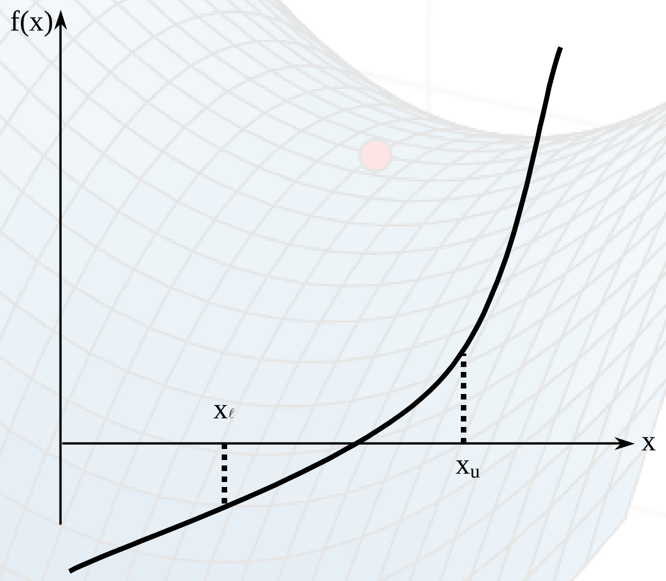
Summary



Bisection Method Solver

by **BAYETA IV, Reginald G.L.**

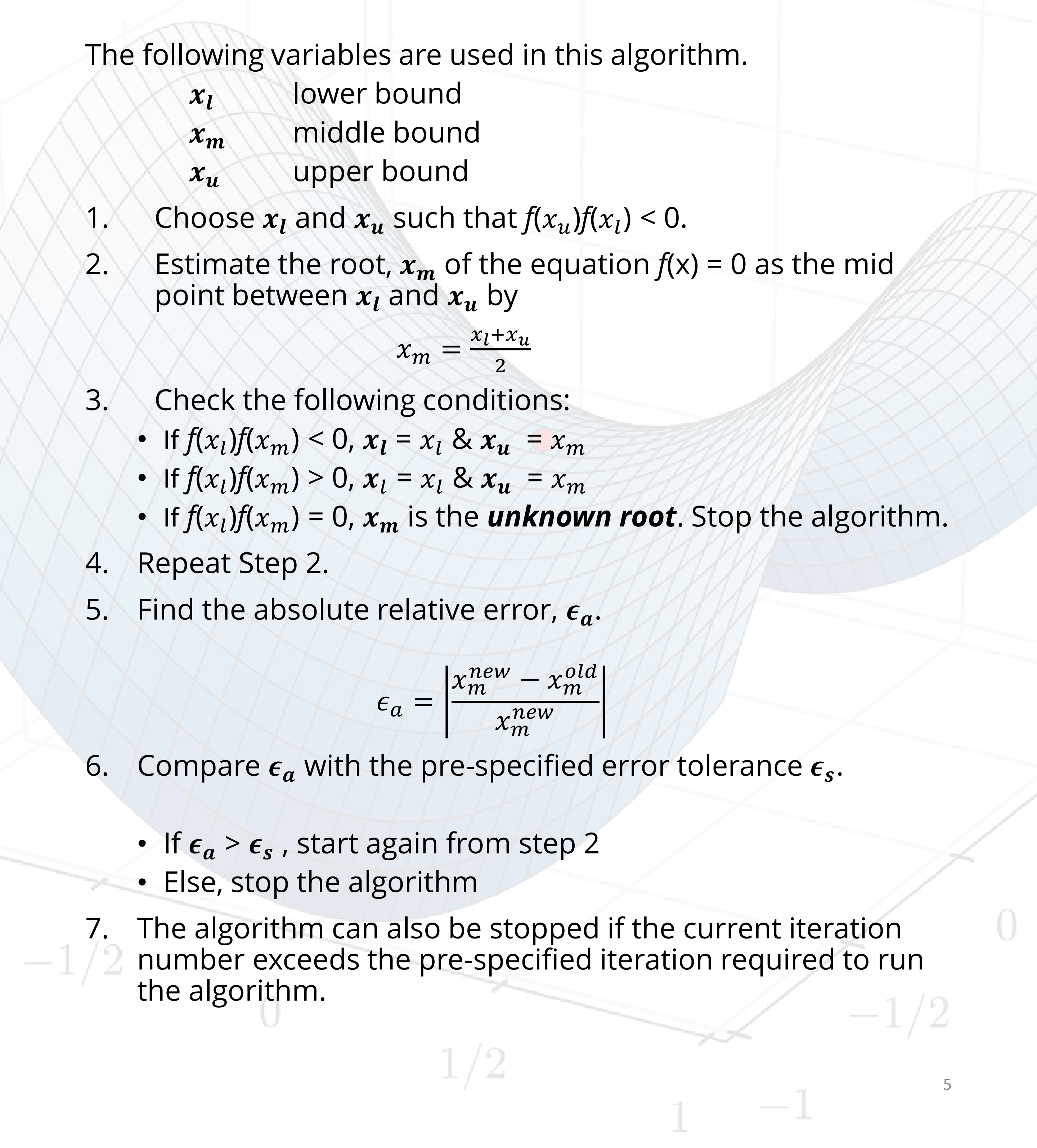
The **bisection method** is a recursive, approximation method in finding a root of a given polynomial equation by repeatedly dividing the interval per iteration.



An equation, *f*(*x*) = 0, where x is a real continuous function, has at least one root between x_u, x_l if $f(x_u)f(x_l) < 0$.

In computer science, this is similar to **binary searching** wherein the sample space is reduced to half until a desired value is found in a given data set.

Bisection Method Algorithm



The following variables are used in this algorithm.

$\boldsymbol{x_l}$	lower bound
$\boldsymbol{x_m}$	middle bound
$\boldsymbol{x_u}$	upper bound

1. Choose $\boldsymbol{x_l}$ and $\boldsymbol{x_u}$ such that $f(x_u)f(x_l) < 0$.
2. Estimate the root, $\boldsymbol{x_m}$ of the equation f(x) = 0 as the mid point between $\boldsymbol{x_l}$ and $\boldsymbol{x_u}$ by

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

3. Check the following conditions:
   - If $f(x_l)f(x_m) < 0$, $\boldsymbol{x_l} = x_l$ & $\boldsymbol{x_u} = x_m$
   - If $f(x_l)f(x_m) > 0$, $\boldsymbol{x_l} = x_l$ & $\boldsymbol{x_u} = x_m$
   - If $f(x_l)f(x_m) = 0$, $\boldsymbol{x_m}$ is the ***unknown root***. Stop the algorithm.
4. Repeat Step 2.
5. Find the absolute relative error, $\boldsymbol{\epsilon_a}$.

$$\epsilon_a = \left|\frac{x_m^{new} - x_m^{old}}{x_m^{new}}\right|$$

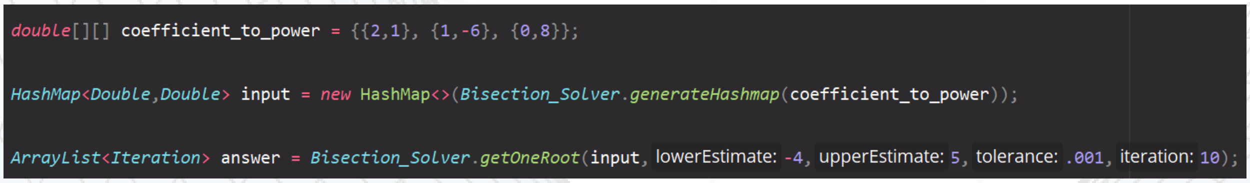
6. Compare $\boldsymbol{\epsilon_a}$ with the pre-specified error tolerance $\boldsymbol{\epsilon_s}$.

   - If $\boldsymbol{\epsilon_a} > \boldsymbol{\epsilon_s}$, start again from step 2
   - Else, stop the algorithm
7. The algorithm can also be stopped if the current iteration number exceeds the pre-specified iteration required to run the algorithm.

Bisection Method Sample Computations

The ***Bisection Method Solver*** is written in Java programming language. The program accepts 5 inputs from the user: the equation represented by a 2D array data structure, 2 initial guess x_l and x_u, error tolerance, and number of iterations.

Sample Computation 1.



```java
double[][] coefficient_to_power = {{2,1}, {1,-6}, {0,8}};

HashMap<Double,Double> input = new HashMap<>(Bisection_Solver.generateHashmap(coefficient_to_power));

ArrayList<Iteration> answer = Bisection_Solver.getOneRoot(input, lowerEstimate: -4, upperEstimate: 5, tolerance: .001, iteration: 10);
```

Given:

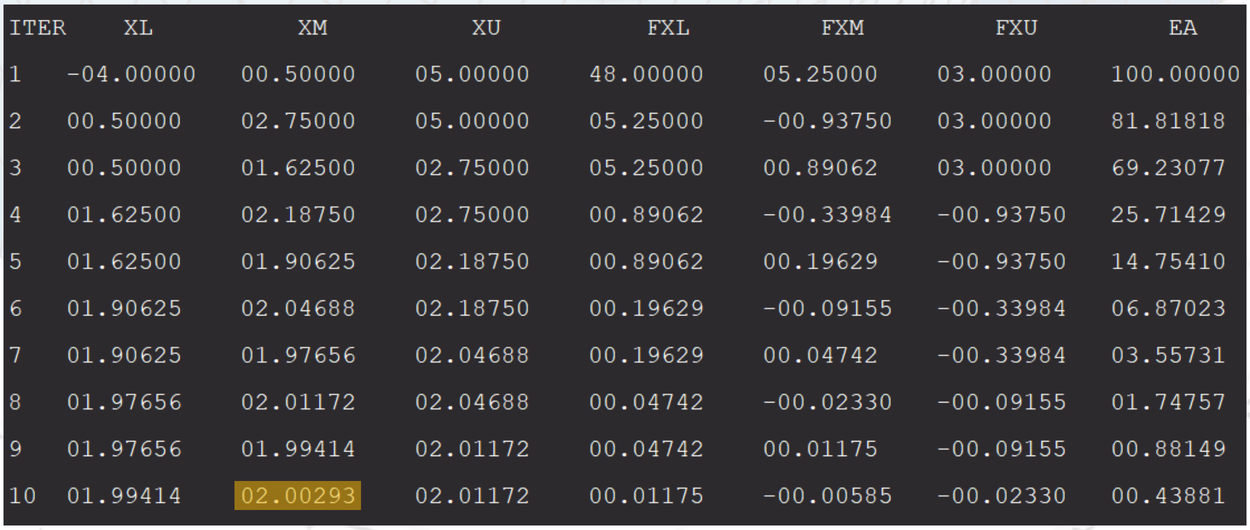
$f(x) = x^2 - 6x + 8$

x_l = -4

x_u = 5

ϵ_s = .001

10 iteration count limit

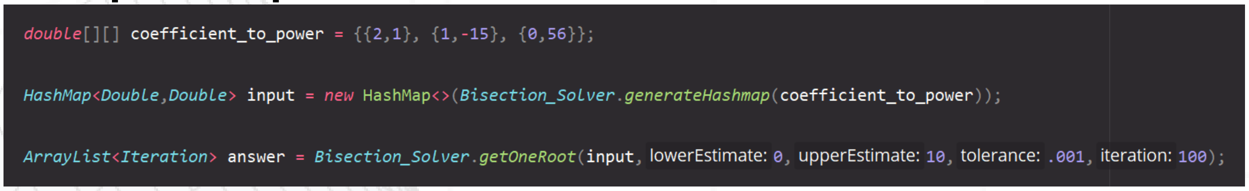


```
ITER   XL          XM          XU          FXL         FXM          FXU          EA
1    -04.00000   00.50000    05.00000    48.00000    05.25000     03.00000     100.00000
2     00.50000   02.75000    05.00000    05.25000    -00.93750    03.00000     81.81818
3     00.50000   01.62500    02.75000    05.25000    00.89062     03.00000     69.23077
4     01.62500   02.18750    02.75000    00.89062    -00.33984    -00.93750    25.71429
5     01.62500   01.90625    02.18750    00.89062    00.19629     -00.93750    14.75410
6     01.90625   02.04688    02.18750    00.19629    -00.09155    -00.33984    06.87023
7     01.90625   01.97656    02.04688    00.19629    00.04742     -00.33984    03.55731
8     01.97656   02.01172    02.04688    00.04742    -00.02330    -00.09155    01.74757
9     01.97656   01.99414    02.01172    00.04742    00.01175     -00.09155    00.88149
10    01.99414   02.00293    02.01172    00.01175    -00.00585    -00.02330    00.43881
```

The algorithm was stopped because it reached 10 iterations and value of the root is converging to ***2***. We know that the factors of f(x) are (x-4) (x-2) giving the roots of (4,2). Therefore, we found the correct root.

Bisection Method Sample Computations

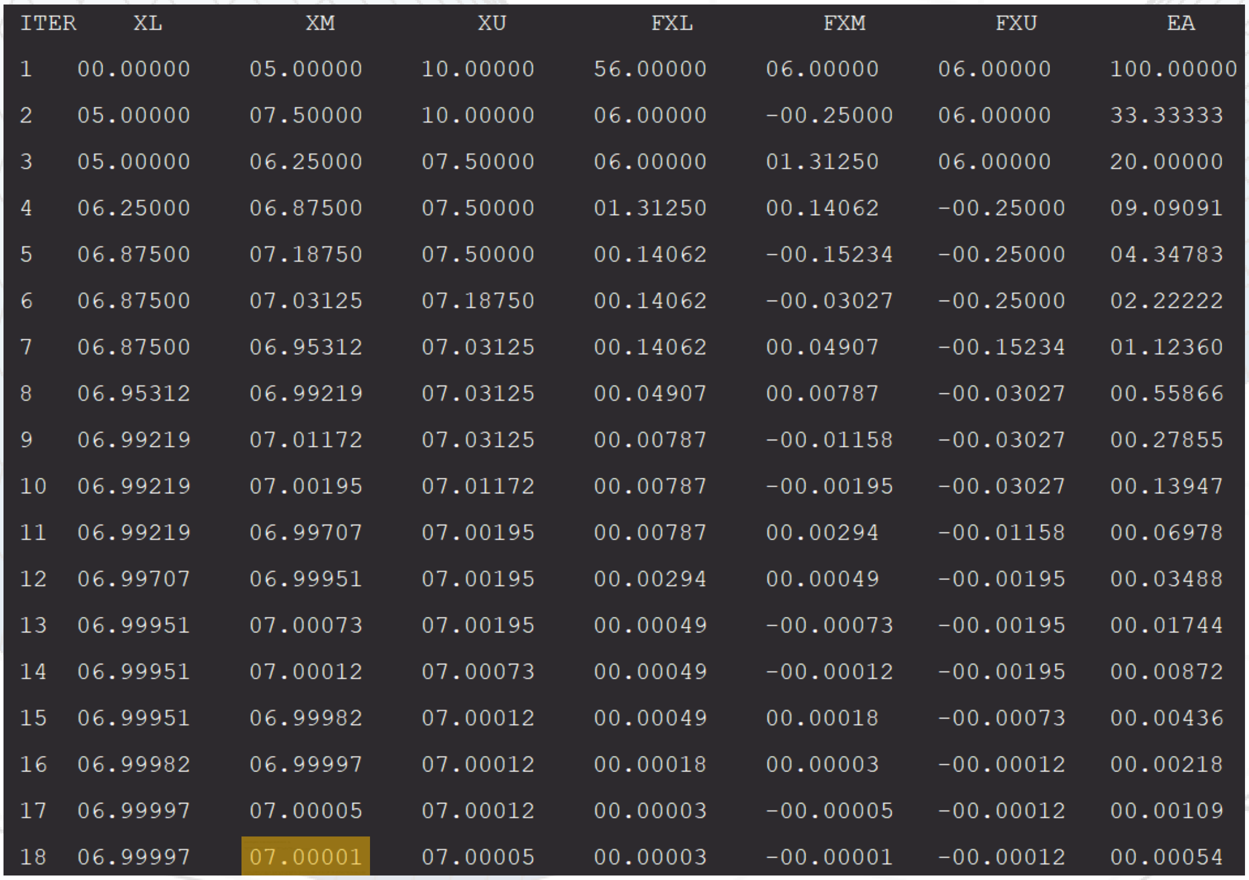
Sample Computation 2.



```
double[][] coefficient_to_power = {{2,1}, {1,-15}, {0,56}};

HashMap<Double,Double> input = new HashMap<>(Bisection_Solver.generateHashmap(coefficient_to_power));

ArrayList<Iteration> answer = Bisection_Solver.getOneRoot(input, lowerEstimate: 0, upperEstimate: 10, tolerance: .001, iteration: 100);
```

Given: $f(x) = x^2 - 15x + 56$

x_l = 0, x_u = 10, ϵ_s = .001,100 iterations

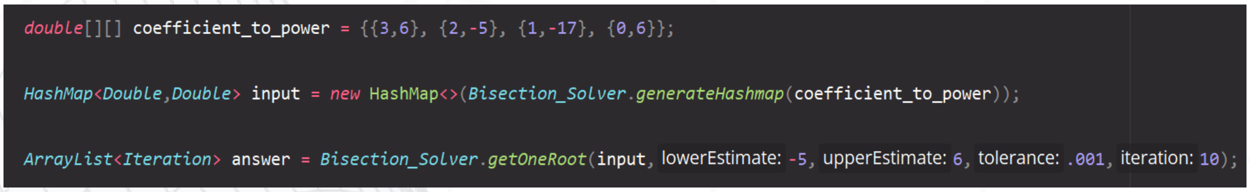


```
ITER  XL        XM        XU        FXL       FXM        FXU        EA
1     00.00000  05.00000  10.00000  56.00000  06.00000   06.00000   100.00000
2     05.00000  07.50000  10.00000  06.00000  -00.25000  06.00000   33.33333
3     05.00000  06.25000  07.50000  06.00000  01.31250   06.00000   20.00000
4     06.25000  06.87500  07.50000  01.31250  00.14062   -00.25000  09.09091
5     06.87500  07.18750  07.50000  00.14062  -00.15234  -00.25000  04.34783
6     06.87500  07.03125  07.18750  00.14062  -00.03027  -00.25000  02.22222
7     06.87500  06.95312  07.03125  00.14062  00.04907   -00.15234  01.12360
8     06.95312  06.99219  07.03125  00.04907  00.00787   -00.03027  00.55866
9     06.99219  07.01172  07.03125  00.00787  -00.01158  -00.03027  00.27855
10    06.99219  07.00195  07.01172  00.00787  -00.00195  -00.03027  00.13947
11    06.99219  06.99707  07.00195  00.00787  00.00294   -00.01158  00.06978
12    06.99707  06.99951  07.00195  00.00294  00.00049   -00.00195  00.03488
13    06.99951  07.00073  07.00195  00.00049  -00.00073  -00.00195  00.01744
14    06.99951  07.00012  07.00073  00.00049  -00.00012  -00.00195  00.00872
15    06.99951  06.99982  07.00012  00.00049  00.00018   -00.00073  00.00436
16    06.99982  06.99997  07.00012  00.00018  00.00003   -00.00012  00.00218
17    06.99997  07.00005  07.00012  00.00003  -00.00005  -00.00012  00.00109
18    06.99997  07.00001  07.00005  00.00003  -00.00001  -00.00012  00.00054
```

The algorithm was stopped at the 18th iteration because $\epsilon_s > \epsilon_a$ with the value of the root is converging to ***7***. We know that the factors of f(x) are (x-8) (x-7) giving the roots of (8,7). Therefore, we found the correct root.

Bisection Method Sample Computations

Sample Computation 3.



```
double[][] coefficient_to_power = {{3,6}, {2,-5}, {1,-17}, {0,6}};

HashMap<Double,Double> input = new HashMap<>(Bisection_Solver.generateHashmap(coefficient_to_power));

ArrayList<Iteration> answer = Bisection_Solver.getOneRoot(input, lowerEstimate: -5, upperEstimate: 6, tolerance: .001, iteration: 10);
```

Given: $f(x) = 6x^3 - 5x^2 - 17x + 6$,

x_l = -5

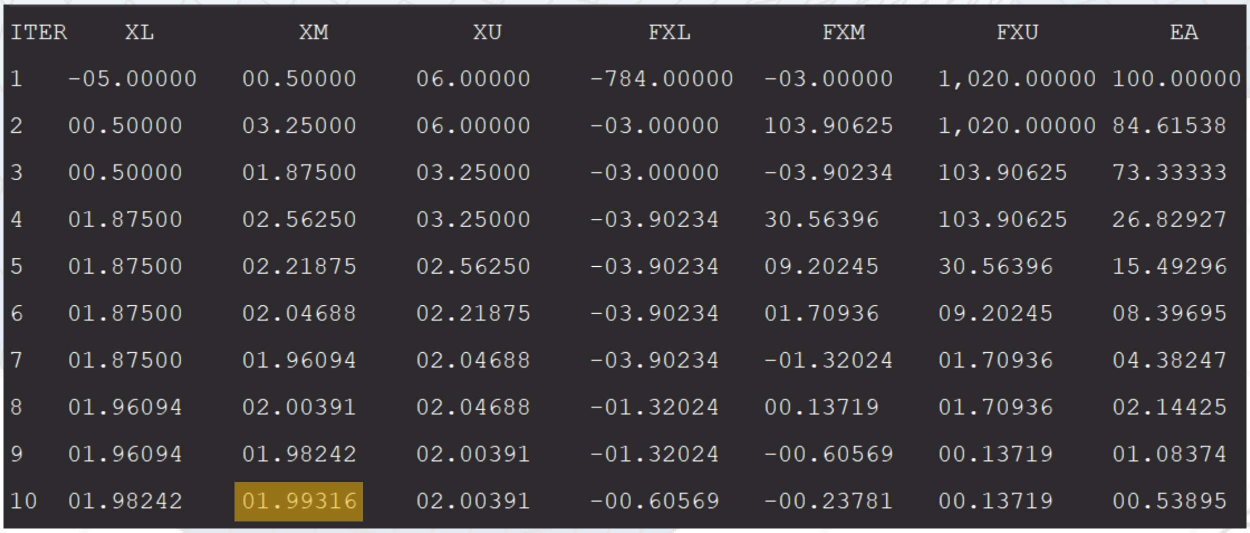
x_u = 6

ϵ_s = .001

10 iterations

Factoring

$$\begin{gathered} 6x^3 - 5x^2 - 17x + 6 = 0 \\ (x-2)(2x+3)(3x-1) = 0 \\ x = 2 \text{ or } x = \frac{1}{3} \text{ or } x = -\frac{3}{2} \end{gathered}$$



```
ITER  XL         XM         XU         FXL         FXM         FXU           EA
1     -05.00000  00.50000   06.00000   -784.00000  -03.00000   1,020.00000   100.00000
2     00.50000   03.25000   06.00000   -03.00000   103.90625   1,020.00000   84.61538
3     00.50000   01.87500   03.25000   -03.00000   -03.90234   103.90625     73.33333
4     01.87500   02.56250   03.25000   -03.90234   30.56396    103.90625     26.82927
5     01.87500   02.21875   02.56250   -03.90234   09.20245    30.56396      15.49296
6     01.87500   02.04688   02.21875   -03.90234   01.70936    09.20245      08.39695
7     01.87500   01.96094   02.04688   -03.90234   -01.32024   01.70936      04.38247
8     01.96094   02.00391   02.04688   -01.32024   00.13719    01.70936      02.14425
9     01.96094   01.98242   02.00391   -01.32024   -00.60569   00.13719      01.08374
10    01.98242   01.99316   02.00391   -00.60569   -00.23781   00.13719      00.53895
```

The algorithm was stopped because it reached 10 iterations and value of the root is converging to ***2***. We know that the factors of f(x) are (x-2) (2x+3) (3x-1) giving the roots of (2,1/3,-3/2). Therefore, we found the correct root.

False Position Method Solver

by **ANTOC, Jan Luis**

False Position Method is a bracketing method wherein it requires two initial guesses for the root in order to obtain the real root of a nonlinear equation. These two initial guesses, when substituted to the equation, should have two different outputs: one which is a negative value and another positive value, indicating a sign change. This method aims to locate the root by determining the midpoint of the subinterval within which the sign change occurs.

This method is **almost the same** with the bisection method except for the equation used to find the midpoint between the guesses for the root.

False Position Method Algorithm

The following variables are used in this algorithm.

$\boldsymbol{x_l}$	lower bound
$\boldsymbol{x_m}$	middle bound
$\boldsymbol{x_u}$	upper bound

1. Choose $\boldsymbol{x_l}$ and $\boldsymbol{x_u}$ such that $f(x_u)f(x_l) < 0$.
2. Estimate the root, $\boldsymbol{x_m}$ of the equation f(x) = 0 as the mid point between $\boldsymbol{x_l}$ and $\boldsymbol{x_u}$ by

$$x_m = \frac{x_u f(x_l) - x_l f(x_u)}{f(x_l) - f(x_u)} \longleftarrow$$

Main difference with Bisection Method

3. Check the following conditions:
   - If $f(x_l)f(x_m) < 0$, $\boldsymbol{x_l} = x_l$ & $\boldsymbol{x_u} = x_m$
   - If $f(x_l)f(x_m) > 0$, $\boldsymbol{x_l} = x_l$ & $\boldsymbol{x_u} = x_m$
   - If $f(x_l)f(x_m) = 0$, $\boldsymbol{x_m}$ is the ***unknown root***. Stop the algorithm.
4. Repeat Step 2.
5. Find the absolute relative error, $\boldsymbol{\epsilon_a}$.

$$\epsilon_a = \left| \frac{x_m^{new} - x_m^{old}}{x_m^{new}} \right|$$

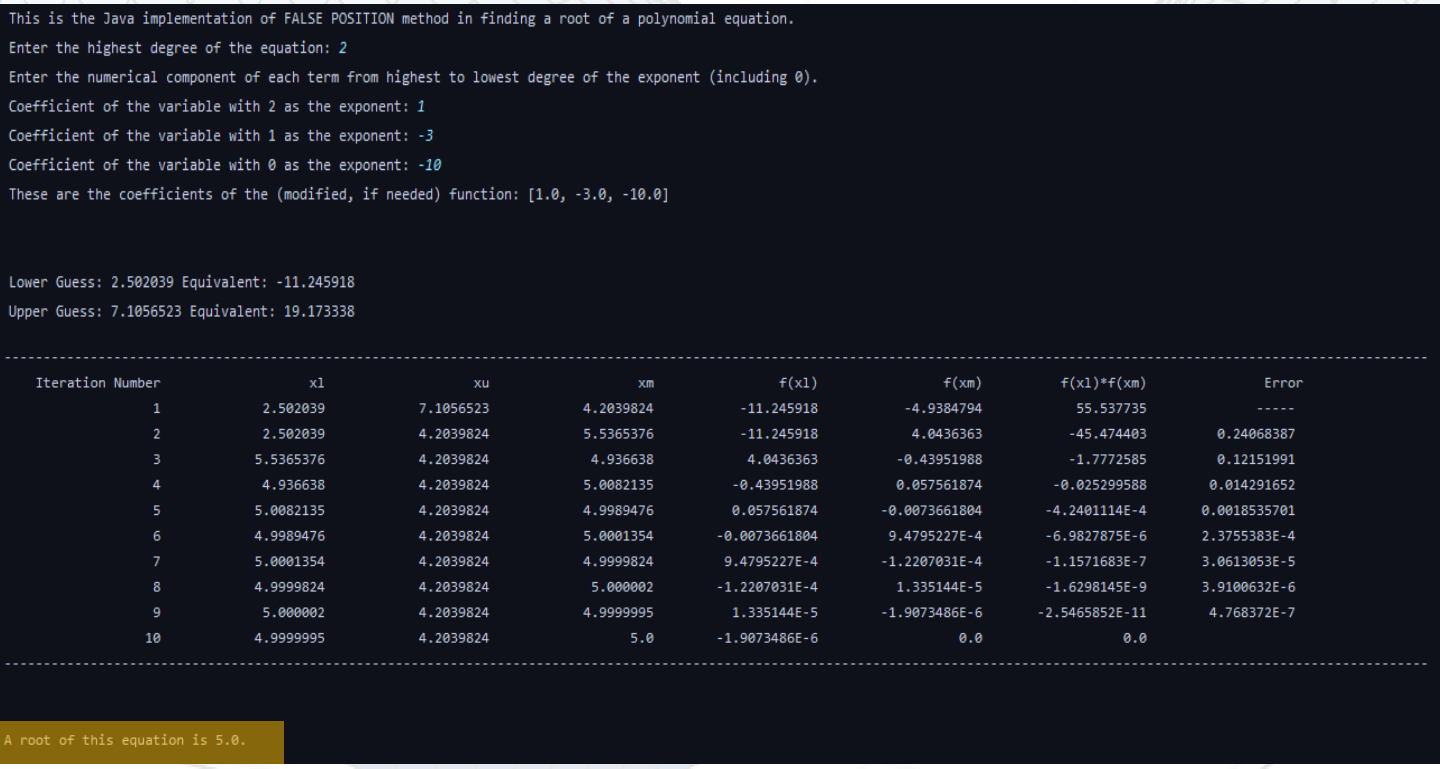
6. Compare $\boldsymbol{\epsilon_a}$ with the pre-specified error tolerance $\boldsymbol{\epsilon_s}$.
   - If $\boldsymbol{\epsilon_a} > \boldsymbol{\epsilon_s}$, start again from step 2
   - Else, stop the algorithm
7. The algorithm can also be stopped if the current iteration number exceeds the pre-specified iteration required to run the algorithm.

False Position Sample Computations

The ***False Position Method Solver*** is written in Java programming language. User is prompted to input the equation to be solved and the program will randomly pick the initial lower and upper guesses.

Sample Computation 1.

Given	Stopping Criteria	Expected Values
f(x) = $x^2 - 3x - 10$	iterations > 10000	f(x) = (x-5) (x+2)
x_l = 2.502039 x_u = 7.105623	error < 1E-7	x = 5, x = -2



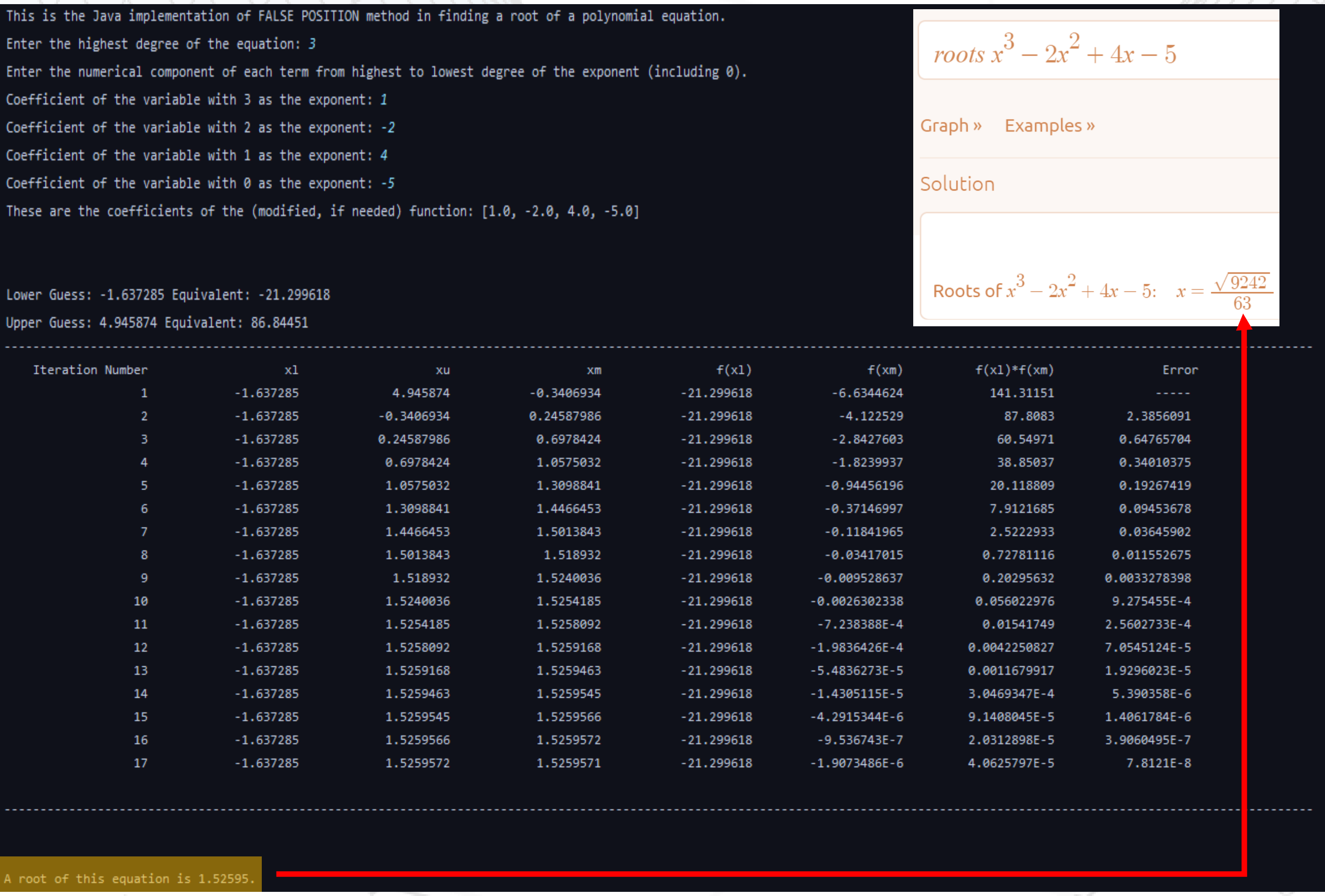
```
This is the Java implementation of FALSE POSITION method in finding a root of a polynomial equation.
Enter the highest degree of the equation: 2
Enter the numerical component of each term from highest to lowest degree of the exponent (including 0).
Coefficient of the variable with 2 as the exponent: 1
Coefficient of the variable with 1 as the exponent: -3
Coefficient of the variable with 0 as the exponent: -10
These are the coefficients of the (modified, if needed) function: [1.0, -3.0, -10.0]



Lower Guess: 2.502039 Equivalent: -11.245918
Upper Guess: 7.1056523 Equivalent: 19.173338

----------------------------------------------------------------------------------------------------------------------------------

  Iteration Number          x1            xu            xm          f(x1)          f(xm)      f(x1)*f(xm)          Error
                 1    2.502039     7.1056523     4.2039824     -11.245918     -4.9384794        55.537735          -----
                 2    2.502039     4.2039824     5.5365376     -11.245918      4.0436363       -45.474403     0.24068387
                 3   5.5365376     4.2039824      4.936638      4.0436363    -0.43951988       -1.7772585     0.12151991
                 4    4.936638     4.2039824     5.0082135    -0.43951988    0.057561874     -0.025299588    0.014291652
                 5   5.0082135     4.2039824     4.9989476    0.057561874  -0.0073661804    -4.2401114E-4   0.0018535701
                 6   4.9989476     4.2039824     5.0001354  -0.0073661804   9.4795227E-4    -6.9827875E-6   2.3755383E-4
                 7   5.0001354     4.2039824     4.9999824   9.4795227E-4  -1.2207031E-4    -1.1571683E-7   3.0613053E-5
                 8   4.9999824     4.2039824      5.000002  -1.2207031E-4    1.335144E-5    -1.6298145E-9   3.9100632E-6
                 9    5.000002     4.2039824     4.9999995    1.335144E-5  -1.9073486E-6   -2.5465852E-11    4.768372E-7
                10   4.9999995     4.2039824           5.0  -1.9073486E-6            0.0              0.0

----------------------------------------------------------------------------------------------------------------------------------


A root of this equation is 5.0.
```

The algorithm was stopped at the 10th iteration wherein the approximate value of the root is equal to 5.00000. We know that the factors of f(x) are (x-5) (x+2) giving the roots of (5,-2). **5 = 5**. Therefore, we found the correct root.

False Position Sample Computations

Sample Computation 2.

Given	Stopping Criteria	Expected Values
f(x) = $x^3 - 2x^2 + 4x - 5$	iterations > 10000	x = $\frac{\sqrt{9242}}{63}$ ≈ 1.52595
x_l = -1.637285 x_u = 4.945874	error < 1E-7	



```
This is the Java implementation of FALSE POSITION method in finding a root of a polynomial equation.
Enter the highest degree of the equation: 3
Enter the numerical component of each term from highest to lowest degree of the exponent (including 0).
Coefficient of the variable with 3 as the exponent: 1
Coefficient of the variable with 2 as the exponent: -2
Coefficient of the variable with 1 as the exponent: 4
Coefficient of the variable with 0 as the exponent: -5
These are the coefficients of the (modified, if needed) function: [1.0, -2.0, 4.0, -5.0]


Lower Guess: -1.637285 Equivalent: -21.299618
Upper Guess: 4.945874 Equivalent: 86.84451
-------------------------------------------------------------------------------------------------------------------------------------
    Iteration Number          xl            xu            xm          f(xl)           f(xm)        f(xl)*f(xm)          Error
                   1    -1.637285      4.945874    -0.3406934    -21.299618      -6.6344624       141.31151          -----
                   2    -1.637285    -0.3406934    0.24587986    -21.299618       -4.122529         87.8083      2.3856091
                   3    -1.637285    0.24587986     0.6978424    -21.299618      -2.8427603        60.54971     0.64765704
                   4    -1.637285     0.6978424     1.0575032    -21.299618      -1.8239937        38.85037     0.34010375
                   5    -1.637285     1.0575032     1.3098841    -21.299618     -0.94456196       20.118809     0.19267419
                   6    -1.637285     1.3098841     1.4466453    -21.299618     -0.37146997       7.9121685     0.09453678
                   7    -1.637285     1.4466453     1.5013843    -21.299618     -0.11841965       2.5222933     0.03645902
                   8    -1.637285     1.5013843      1.518932    -21.299618     -0.03417015      0.72781116    0.011552675
                   9    -1.637285      1.518932     1.5240036    -21.299618    -0.009528637      0.20295632   0.0033278398
                  10    -1.637285     1.5240036     1.5254185    -21.299618   -0.0026302338     0.056022976     9.275455E-4
                  11    -1.637285     1.5254185     1.5258092    -21.299618    -7.238388E-4      0.01541749    2.5602733E-4
                  12    -1.637285     1.5258092     1.5259168    -21.299618  -1.9836426E-4    0.0042250827    7.0545124E-5
                  13    -1.637285     1.5259168     1.5259463    -21.299618  -5.4836273E-5    0.0011679917    1.9296023E-5
                  14    -1.637285     1.5259463     1.5259545    -21.299618  -1.4305115E-5    3.0469347E-4     5.390358E-6
                  15    -1.637285     1.5259545     1.5259566    -21.299618  -4.2915344E-6    9.1408045E-5    1.4061784E-6
                  16    -1.637285     1.5259566     1.5259572    -21.299618   -9.536743E-7    2.0312898E-5    3.9060495E-7
                  17    -1.637285     1.5259572     1.5259571    -21.299618  -1.9073486E-6    4.0625797E-5       7.8121E-8

-------------------------------------------------------------------------------------------------------------------------------------

A root of this equation is 1.52595.
```

roots $x^3 - 2x^2 + 4x - 5$

Graph » Examples »

Solution

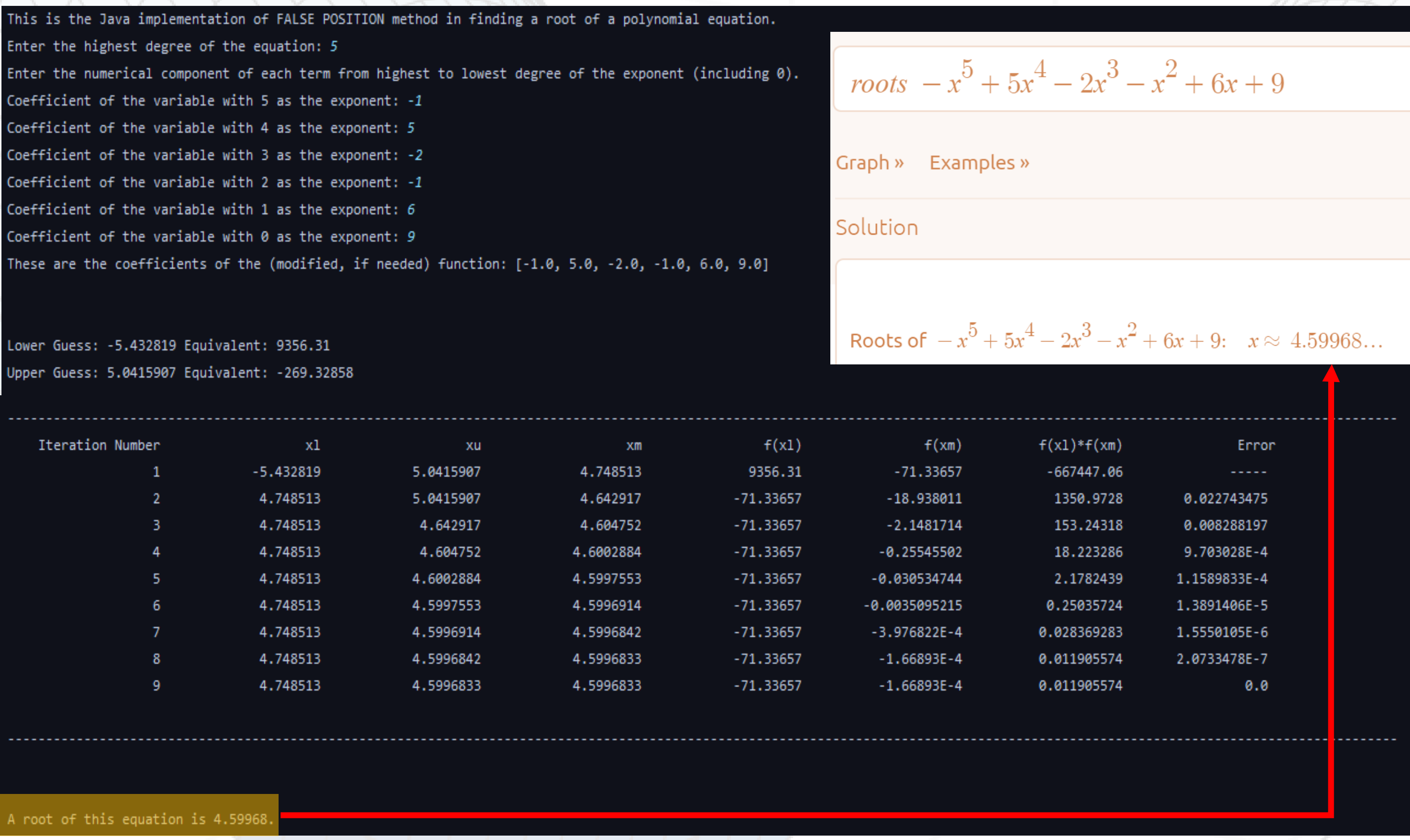
Roots of $x^3 - 2x^2 + 4x - 5$: $x = \frac{\sqrt{9242}}{63}$

The algorithm was stopped at the 17th iteration because it satisfies the stopping criteria: error is less then 1.0×10^{-7}. Since 1.52595 ≈ 1.52595…, we can conclude that we found the correct root.

False Position Sample Computations

Sample Computation 3.

Given	Stopping Criteria	Expected Values
f(x) = $-x^5 + 5x^4 - 2x^3 - x^2 + 6x + 9$	iterations > 10000	x≈ 4.59968…
x_l = -5.432819 x_u = 5.0415907	error < 1E-7	



```
This is the Java implementation of FALSE POSITION method in finding a root of a polynomial equation.
Enter the highest degree of the equation: 5
Enter the numerical component of each term from highest to lowest degree of the exponent (including 0).
Coefficient of the variable with 5 as the exponent: -1
Coefficient of the variable with 4 as the exponent: 5
Coefficient of the variable with 3 as the exponent: -2
Coefficient of the variable with 2 as the exponent: -1
Coefficient of the variable with 1 as the exponent: 6
Coefficient of the variable with 0 as the exponent: 9
These are the coefficients of the (modified, if needed) function: [-1.0, 5.0, -2.0, -1.0, 6.0, 9.0]


Lower Guess: -5.432819 Equivalent: 9356.31
Upper Guess: 5.0415907 Equivalent: -269.32858

------------------------------------------------------------------------------------------------------------------------------------------
    Iteration Number          xl           xu           xm          f(xl)          f(xm)      f(x1)*f(xm)          Error
                   1   -5.432819    5.0415907     4.748513        9356.31      -71.33657      -667447.06          -----
                   2    4.748513    5.0415907     4.642917      -71.33657     -18.938011       1350.9728    0.022743475
                   3    4.748513     4.642917     4.604752      -71.33657     -2.1481714       153.24318    0.008288197
                   4    4.748513     4.604752    4.6002884      -71.33657    -0.25545502       18.223286    9.703028E-4
                   5    4.748513    4.6002884    4.5997553      -71.33657   -0.030534744       2.1782439   1.1589833E-4
                   6    4.748513    4.5997553    4.5996914      -71.33657  -0.0035095215      0.25035724   1.3891406E-5
                   7    4.748513    4.5996914    4.5996842      -71.33657    -3.976822E-4     0.028369283   1.5550105E-6
                   8    4.748513    4.5996842    4.5996833      -71.33657    -1.66893E-4     0.011905574   2.0733478E-7
                   9    4.748513    4.5996833    4.5996833      -71.33657    -1.66893E-4     0.011905574            0.0

------------------------------------------------------------------------------------------------------------------------------------------

A root of this equation is 4.59968.
```

roots $-x^5 + 5x^4 - 2x^3 - x^2 + 6x + 9$

Solution

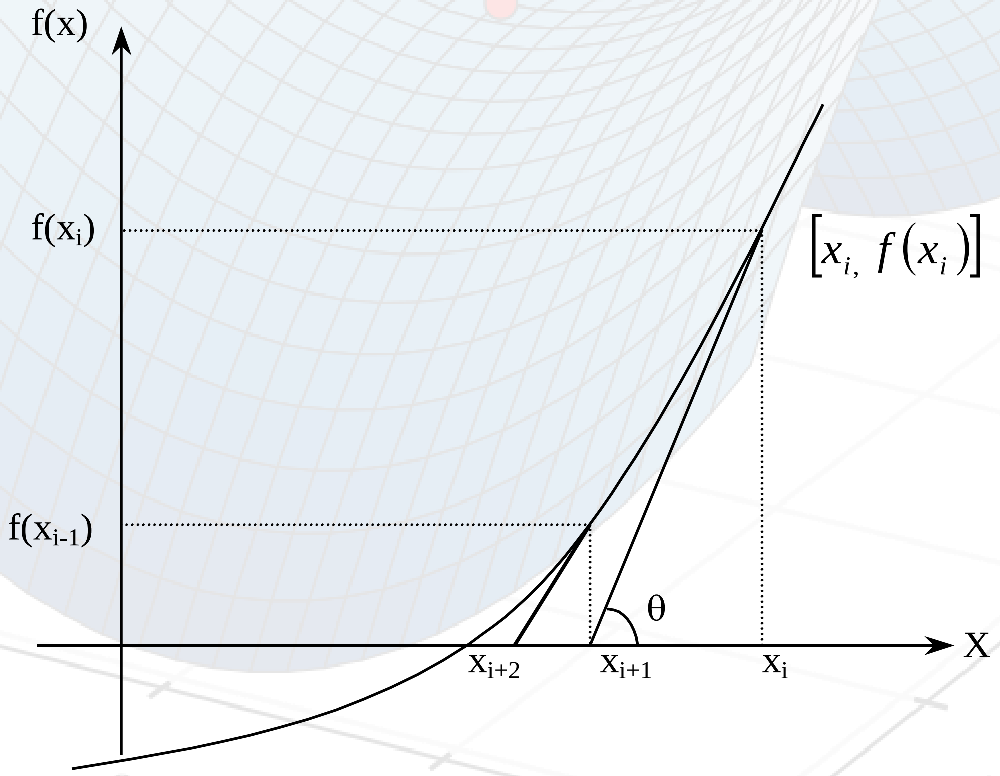
Roots of $-x^5 + 5x^4 - 2x^3 - x^2 + 6x + 9$: $x \approx 4.59968\ldots$

The algorithm was stopped at the 9th iteration because it satisfies the stopping criteria error < 1E-7. Since 4.59968 ≈ 4.59968… , we can conclude that we found the correct root.

Secant Method Solver

***by* LLARENAS, Lenard Ryan**

The **secant method** is a technique for finding the root of a scalar-valued function f(*x*) of a single variable *x* when no information about the derivative exists. It is similar in many ways to the false-position method but trades the possibility of non-convergence for faster convergence [1].



Geometrical illustration of the secant method

[1] https://ece.uwaterloo.ca/~dwharder/NumericalAnalysis/10RootFinding/secant/

Secant Method Algorithm

The secant method is a root-finding algorithm which operates on a succession of roots of secant lines to effectively approximate a root of a function. At the start of the algorithm, the equation has to be inputted. The equation is then passed to a function to convert it into the syntax required for Java mathematical computations. From here, ScriptEngineManager was used to evaluate the new modified mathematical expression using the JavaScript engine. After so, two initial guesses (of variable x) are required for Iteration -1 and Iteration 0. Subsequently, the program will then progress through a for-loop that iterates 8 times.

Within this loop are conditional statements. Each statement has a condition that depends on which iteration the loop is currently in. For Iteration -1, the first initial guess (x_{-1}) is substituted to the equation in order to determine the f(x) for that iteration. For Iteration 0, the process is similar, though it varies on the existence of more variables. Again, the second initial guess (x_0) is substituted to the equation to obtain the f(x) for this iteration. Next, the value of the f(x) from Iteration -1 is now assigned to $f(x_{i-1})$, since it is the f(x) of the previous iteration. Eventually, the error is computed using the equation below:

$$\left|\in_a\right| = \left|\frac{x_{i+1} - x_i}{x_{i+1}}\right|$$

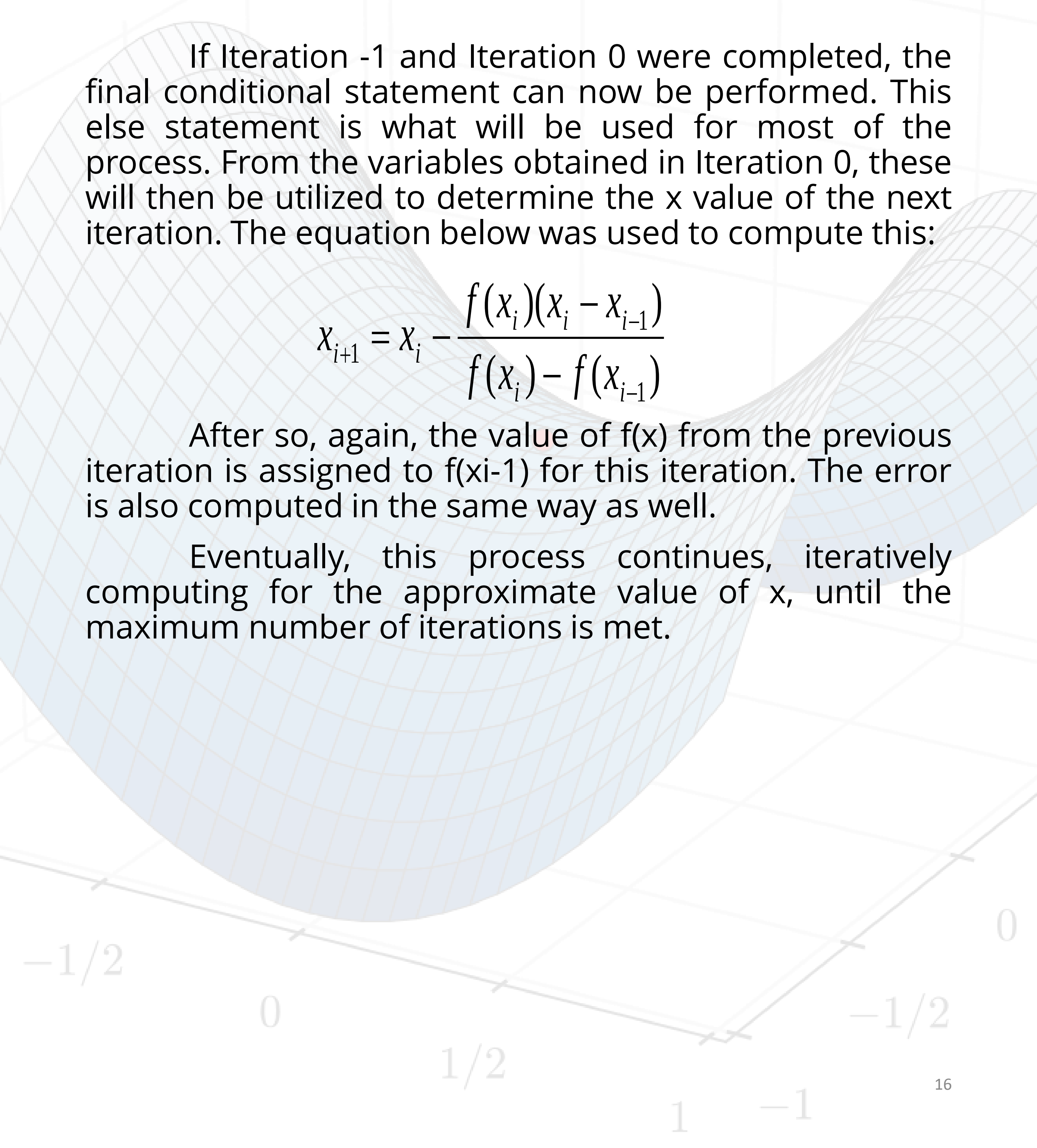
Secant Method Algorithm

If Iteration -1 and Iteration 0 were completed, the final conditional statement can now be performed. This else statement is what will be used for most of the process. From the variables obtained in Iteration 0, these will then be utilized to determine the x value of the next iteration. The equation below was used to compute this:

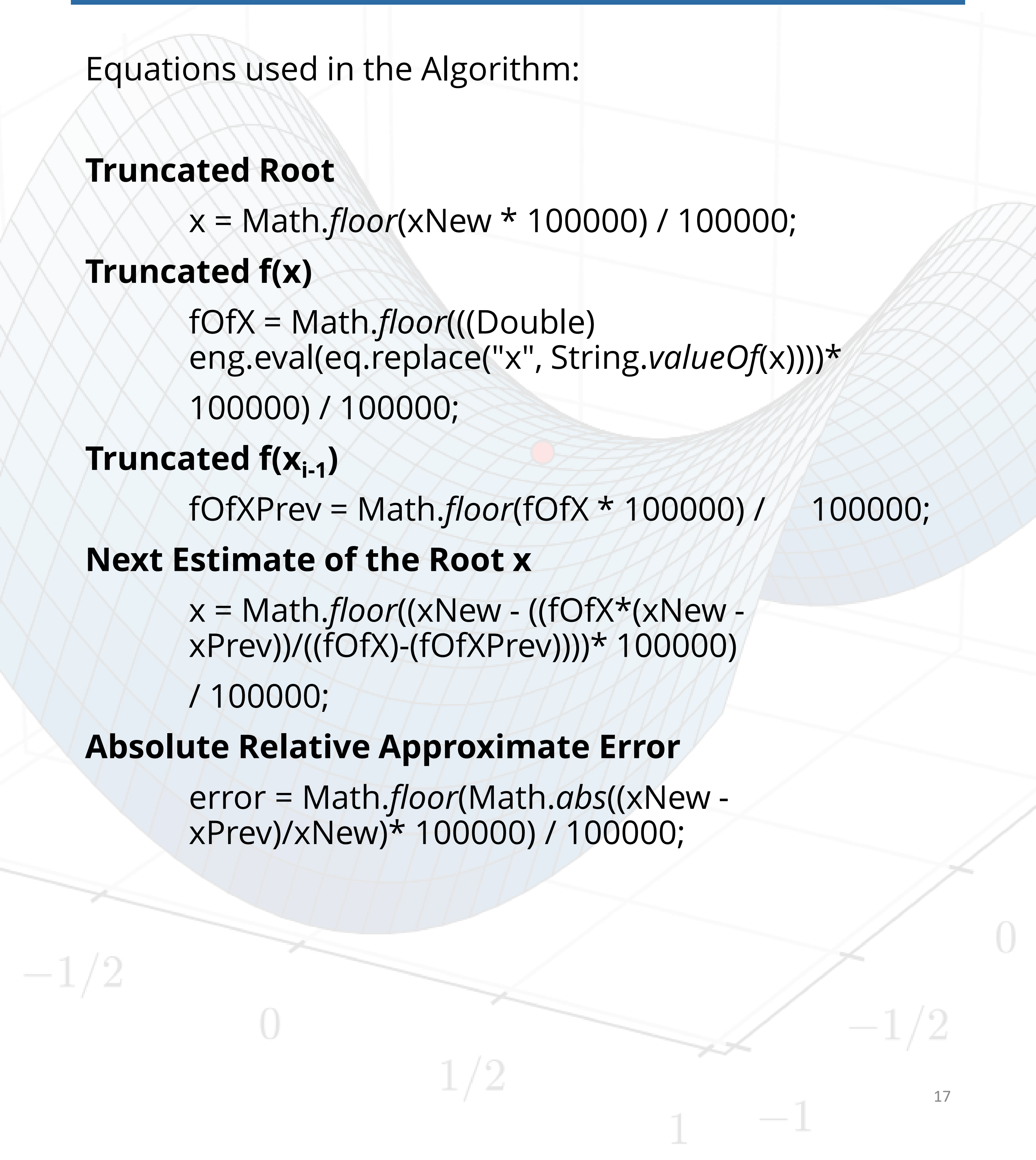
$$x_{i+1} = x_i - \frac{f(x_i)(x_i - x_{i-1})}{f(x_i) - f(x_{i-1})}$$

After so, again, the value of f(x) from the previous iteration is assigned to f(xi-1) for this iteration. The error is also computed in the same way as well.

Eventually, this process continues, iteratively computing for the approximate value of x, until the maximum number of iterations is met.



Secant Method Algorithm



Equations used in the Algorithm:

Truncated Root

```
x = Math.floor(xNew * 100000) / 100000;
```

Truncated f(x)

```
fOfX = Math.floor(((Double)
eng.eval(eq.replace("x", String.valueOf(x))))*
100000) / 100000;
```

Truncated f(x_{i-1})

```
fOfXPrev = Math.floor(fOfX * 100000) /     100000;
```

Next Estimate of the Root x

```
x = Math.floor((xNew - ((fOfX*(xNew -
xPrev))/((fOfX)-(fOfXPrev))))* 100000)
/ 100000;
```

Absolute Relative Approximate Error

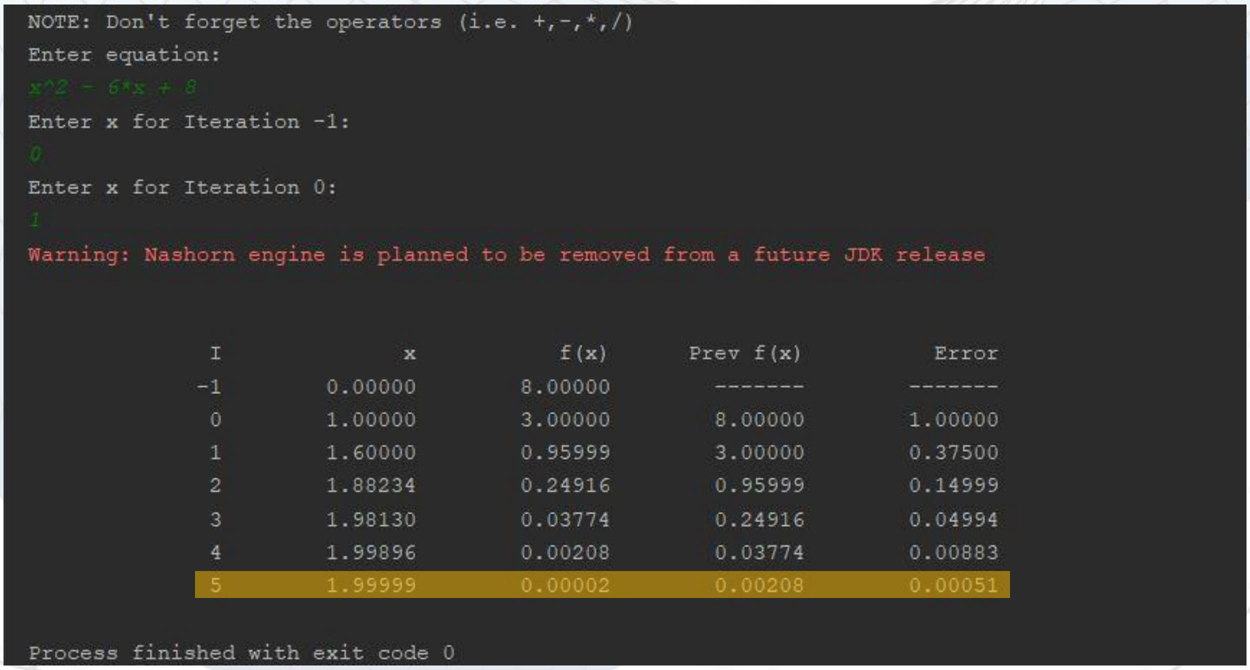
```
error = Math.floor(Math.abs((xNew -
xPrev)/xNew)* 100000) / 100000;
```

Secant Sample Computations

The ***Secant Solver*** is written in Java programming language. The program accepts 2 main inputs from the user: an equation and the initial guess.

Sample Computation 1.

Given	**Stopping Criteria**
f(x) = $x^2 - 6x + 8$	iterations = 11
Initial guess at iteration (-1): **0**	error = 0.001
Initial guess at iteration (0): **1**	$f(x)$ = 0.001



```
NOTE: Don't forget the operators (i.e. +,-,*,/)
Enter equation:
x^2 - 6*x + 8
Enter x for Iteration -1:
0
Enter x for Iteration 0:
1
Warning: Nashorn engine is planned to be removed from a future JDK release


         I            x           f(x)      Prev f(x)          Error
        -1      0.00000        8.00000        -------        -------
         0      1.00000        3.00000        8.00000        1.00000
         1      1.60000        0.95999        3.00000        0.37500
         2      1.88234        0.24916        0.95999        0.14999
         3      1.98130        0.03774        0.24916        0.04994
         4      1.99896        0.00208        0.03774        0.00883
         5      1.99999        0.00002        0.00208        0.00051

Process finished with exit code 0
```

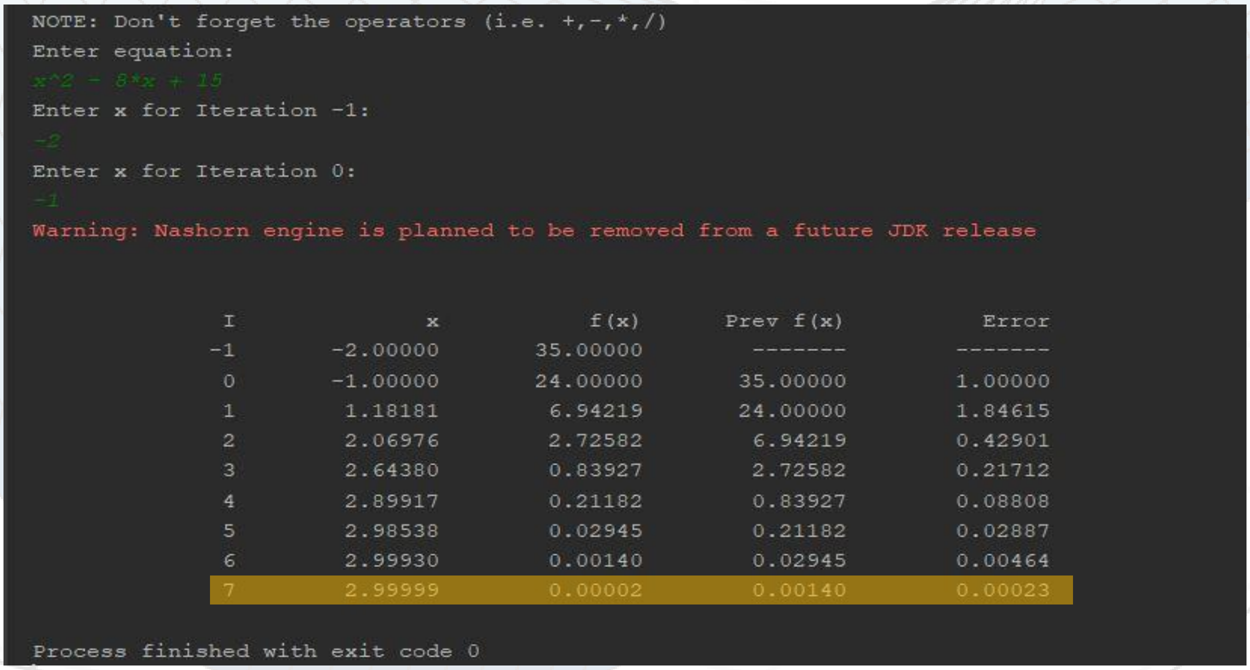
The algorithm stopped at the 5th iteration because it satisfied 2 stopping criteria: Error at *x = 1.99999* is equal to 0.00051 which is less than 0.001 ;and *f(1.99999)* is equal to 0.00002 which is less than 0.001. When calculated, we know that the factor of *f(x)* is (x-4) and (x-2) giving the roots x = 4, and **x = 2**. *x = 1.99999 ≈ 2* so we can conclude that we got the correct root.

Secant Sample Computations

The ***Secant Solver*** is written in Java programming language. The program accepts 2 main inputs from the user: an equation and the initial guess.

Sample Computation 2.

Given	Stopping Criteria
f(x) = $x^2 - 8x + 15$	iterations = 11
Initial guess at iteration (-1): **-2**	error = 0.001
Initial guess at iteration (0): **-1**	$f(x)$ = 0.001



```
NOTE: Don't forget the operators (i.e. +,-,*,/)
Enter equation:
x^2 - 8*x + 15
Enter x for Iteration -1:
-2
Enter x for Iteration 0:
-1
Warning: Nashorn engine is planned to be removed from a future JDK release


              I            x          f(x)      Prev f(x)        Error
             -1     -2.00000      35.00000       --------     --------
              0     -1.00000      24.00000       35.00000      1.00000
              1      1.18181       6.94219       24.00000      1.84615
              2      2.06976       2.72582        6.94219      0.42901
              3      2.64380       0.83927        2.72582      0.21712
              4      2.89917       0.21182        0.83927      0.08808
              5      2.98538       0.02945        0.21182      0.02887
              6      2.99930       0.00140        0.02945      0.00464
              7      2.99999       0.00002        0.00140      0.00023

Process finished with exit code 0
```

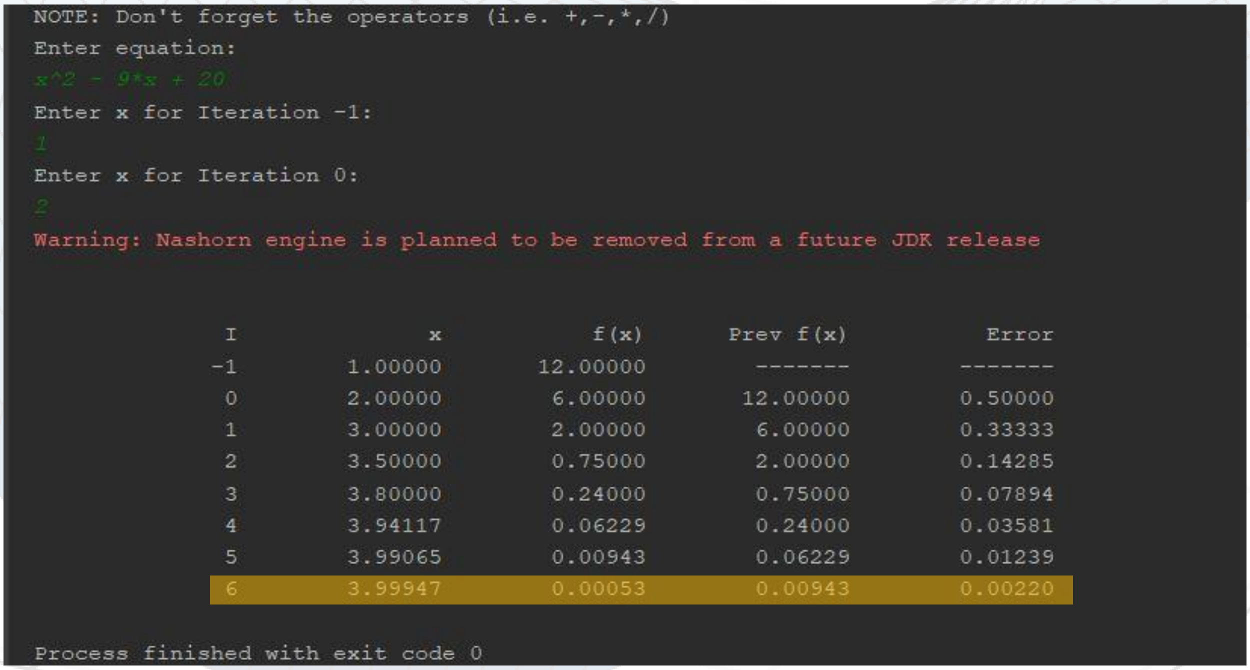
The algorithm stopped at the 7th iteration because it satisfied 2 stopping criteria: Error at *x = 2.99999* is equal to 0.00023 which is less than 0.001 ;and *f(2.99999)* is equal to 0.00002 which is less than 0.001. When calculated, we know that the factor of *f(x)* is (x-5) and (x-3) giving the roots x = 5, and **x = 3**. *x = 2.99999 ≈ 3* so we can conclude that we got the correct root.

Secant Sample Computations

The ***Secant Solver*** is written in Java programming language. The program accepts 2 main inputs from the user: an equation and the initial guess.

Sample Computation 3.

Given	**Stopping Criteria**
f(x) = $x^2 - 9x + 20$	iterations = 11
Initial guess at iteration (-1): **1**	error = 0.001
Initial guess at iteration (0): **2**	$f(x)$ = 0.001



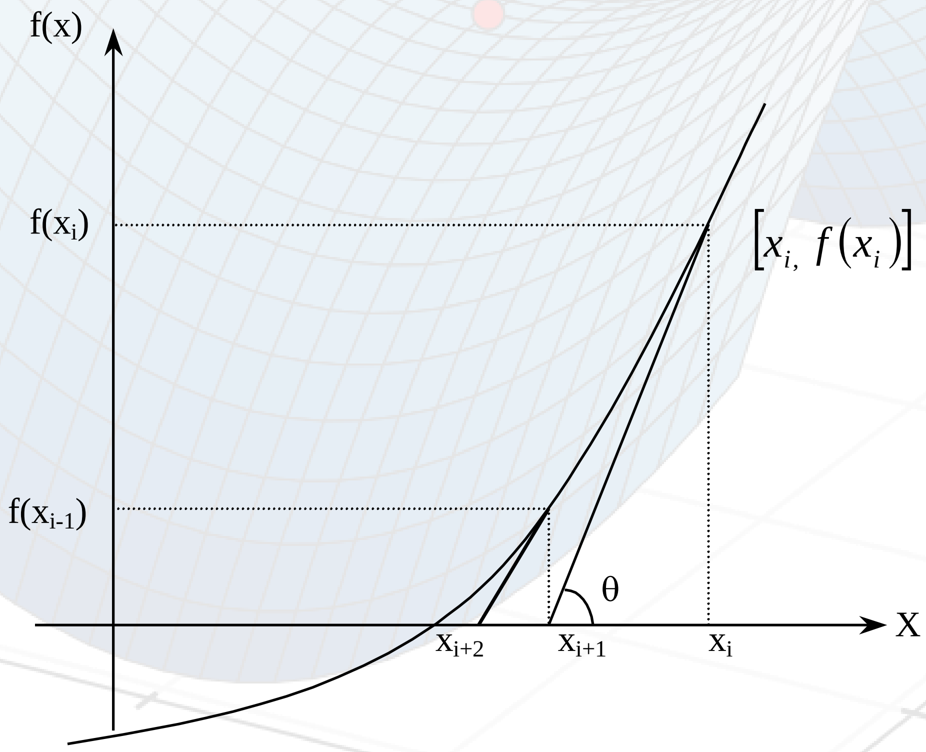
```
NOTE: Don't forget the operators (i.e. +,-,*,/)
Enter equation:
x^2 - 9*x + 20
Enter x for Iteration -1:
1
Enter x for Iteration 0:
2
Warning: Nashorn engine is planned to be removed from a future JDK release


          I              x            f(x)        Prev f(x)          Error
         -1        1.00000        12.00000         --------       --------
          0        2.00000         6.00000         12.00000        0.50000
          1        3.00000         2.00000          6.00000        0.33333
          2        3.50000         0.75000          2.00000        0.14285
          3        3.80000         0.24000          0.75000        0.07894
          4        3.94117         0.06229          0.24000        0.03581
          5        3.99065         0.00943          0.06229        0.01239
          6        3.99947         0.00053          0.00943        0.00220

Process finished with exit code 0
```

The algorithm stopped at the 6th iteration because it satisfied 1 stopping criteria: *f(3.99947)* is equal to 0.00053 which is less than 0.001. When calculated, we know that the factor of *f(x)* is (x-5) and (x-4) giving the roots x = 5, and **x = 4**. *x = 3.99947 ≈ 4* so we can conclude that we got the correct root.

Newton Raphson Method Solver

***by* PORTUGAL, Joseph**

The **Newton-Raphson method** (also known as Newton's method) is a way to quickly find a good approximation for the root of a real-valued function *f*(*x*)=0. It uses the idea that a continuous and differentiable function can be approximated by a straight-line tangent to it **[1]**.



Geometrical illustration of the Newton-Raphson method

[1] https://brilliant.org/wiki/newton-raphson-method/

Newton Raphson Method Algorithm

Initially, the user would input the degree of the polynomial (variable: degree) to be utilized by the algorithm. Then the program will declare the array variable which holds the coefficients of the polynomial (based on the degree of the polynomial) called poly[degree]. Then the user inputs the coefficients of said polynomial per index of the aforementioned coefficient array (poly[i]). The program will then ask the user to input the user's initial guess (initGuess), which the user will input the value of said initial guess. A function to commence the generation of the approximate root value using Newton-Raphson's approximation - will then be called and taken place using the newtonRaphson() function.

Before everything else the program will start with a clean slate by clearing the console, then declare the variable for the iteration counter (iter). After that it will also declare variables for the previous approximation (xOld), the array for the coefficients of the differentiated polynomial (derivativePolynomial[]), and the error size (err). Do note that array size of the differentiated polynomial will be lower by ‘1’, since naturally, the degree of the differentiated polynomial is lower by a degree. The program will then set the array elements for the coefficients of the differentiated polynomial - by multiplying the coefficients of our polynomial to its respective exponent. After which it will display the table header and the current iteration number

Newton Raphson Method Algorithm

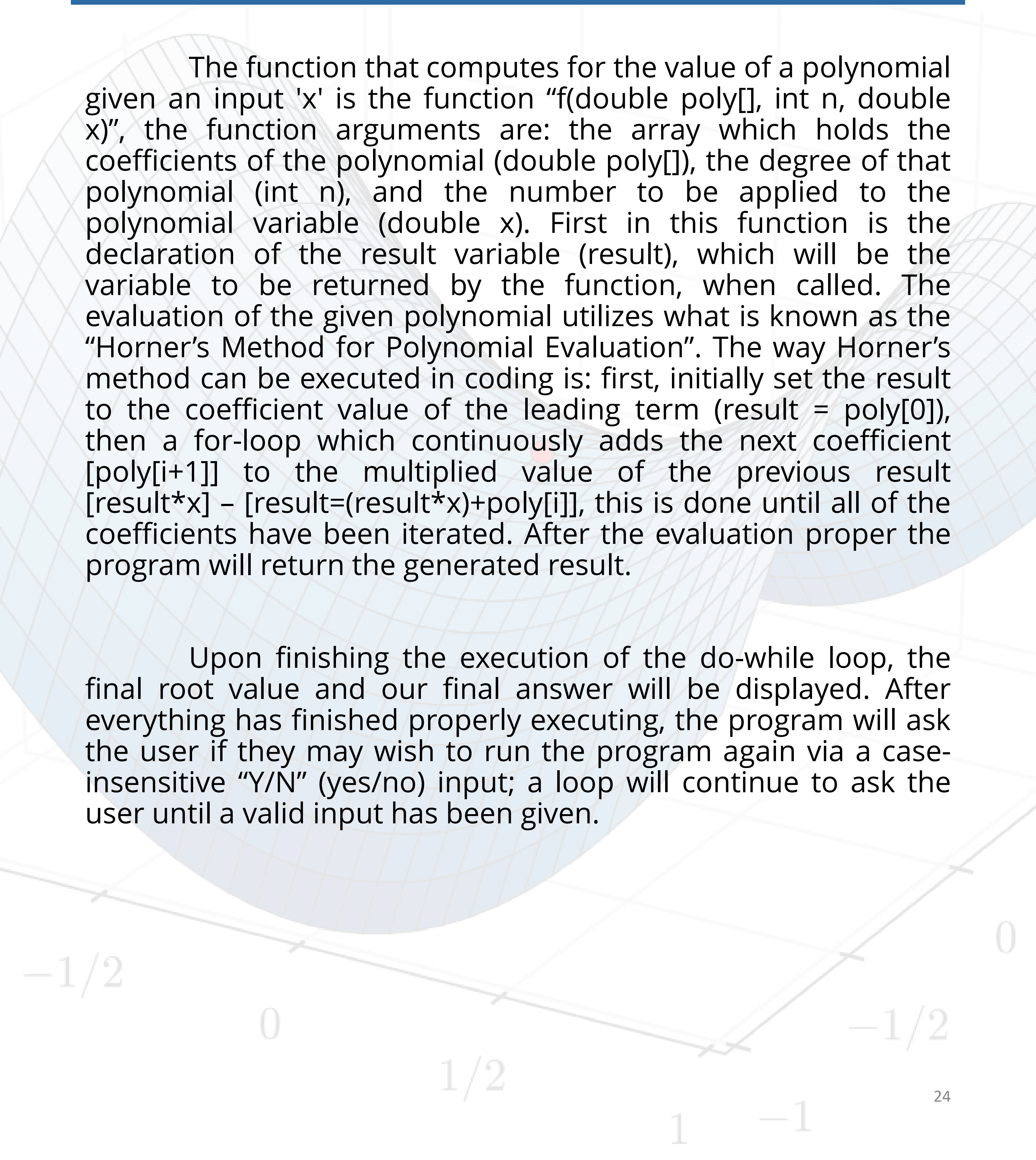
The program will then set the variable (xPlusOne) that will give us the answer to f(x) / f’(x) (or the number that will be subtracted to the current approximation) [f(x)/f’(x)]. It will then display the current root approximation, the result of our polynomial with its variables set to the current approximation (utilizing the f() function), and the result of the differentiated polynomial with its variables also set to the current approximation (also utilizing the f() function). The program will now set the value of the current absolute error (err), and it will then display the current absolute error, this section will display “[N/A]” if it is the first iteration. It will also set the current approximation iteration (x) as 'xOld' (for absolute error computation and final answer display purposes).

After that it will now also set the value of the next approximation (x - xPlusOne). After everything has been finished, the final thing to do before the next loop iteration is to increment the iteration count for the next loop (iter++). Do note that everything in the while loop is repeated until any one of the do-while loop conditions are not met. The do-while loop will continue to execute everything in it as long as the absolute value of the next iteration is more than 0.001, the absolute value of the error is more than 0.001, and the amount of iterations is not more than 20.

Newton Raphson Method Algorithm

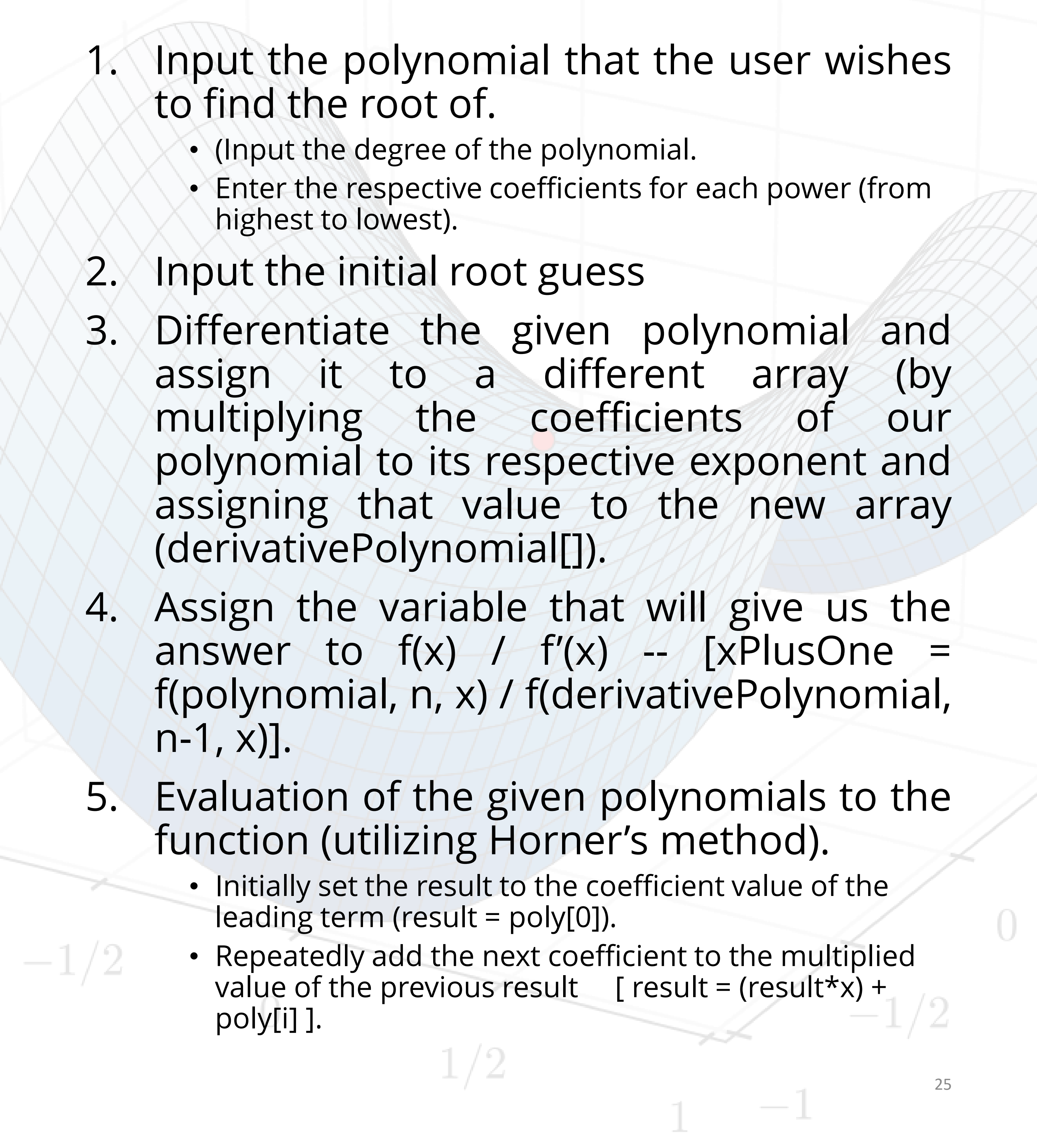
The function that computes for the value of a polynomial given an input 'x' is the function “f(double poly[], int n, double x)”, the function arguments are: the array which holds the coefficients of the polynomial (double poly[]), the degree of that polynomial (int n), and the number to be applied to the polynomial variable (double x). First in this function is the declaration of the result variable (result), which will be the variable to be returned by the function, when called. The evaluation of the given polynomial utilizes what is known as the “Horner’s Method for Polynomial Evaluation”. The way Horner’s method can be executed in coding is: first, initially set the result to the coefficient value of the leading term (result = poly[0]), then a for-loop which continuously adds the next coefficient [poly[i+1]] to the multiplied value of the previous result [result*x] – [result=(result*x)+poly[i]], this is done until all of the coefficients have been iterated. After the evaluation proper the program will return the generated result.

Upon finishing the execution of the do-while loop, the final root value and our final answer will be displayed. After everything has finished properly executing, the program will ask the user if they may wish to run the program again via a case-insensitive “Y/N” (yes/no) input; a loop will continue to ask the user until a valid input has been given.



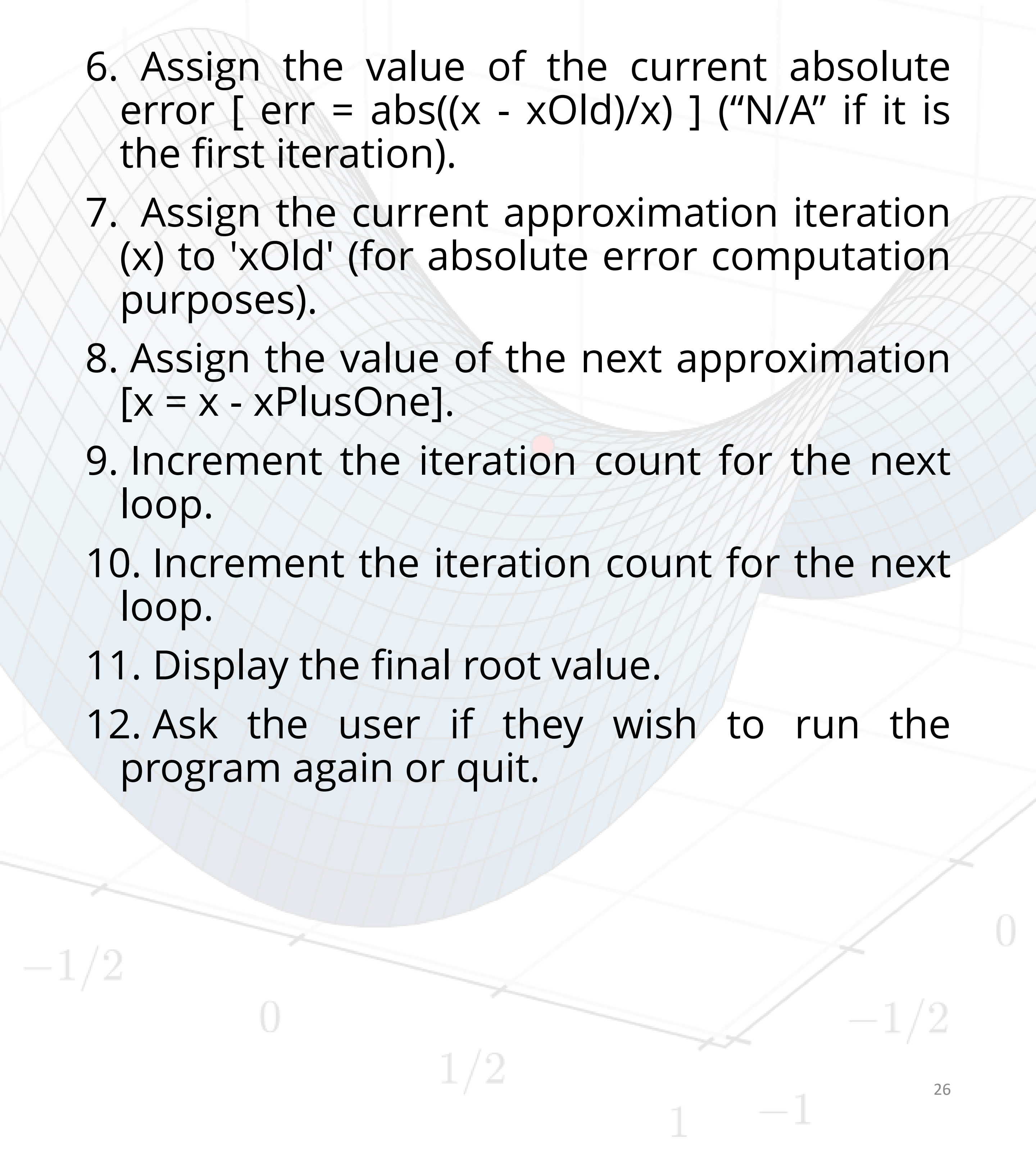
Newton Raphson Method Algorithm - Summary

1. Input the polynomial that the user wishes to find the root of.
   - (Input the degree of the polynomial.
   - Enter the respective coefficients for each power (from highest to lowest).
2. Input the initial root guess
3. Differentiate the given polynomial and assign it to a different array (by multiplying the coefficients of our polynomial to its respective exponent and assigning that value to the new array (derivativePolynomial[]).
4. Assign the variable that will give us the answer to f(x) / f'(x) -- [xPlusOne = f(polynomial, n, x) / f(derivativePolynomial, n-1, x)].
5. Evaluation of the given polynomials to the function (utilizing Horner's method).
   - Initially set the result to the coefficient value of the leading term (result = poly[0]).
   - Repeatedly add the next coefficient to the multiplied value of the previous result [result = (result*x) + poly[i]].



Newton Raphson Method Algorithm - Summary

6. Assign the value of the current absolute error [err = abs((x - xOld)/x)] ("N/A" if it is the first iteration).
7. Assign the current approximation iteration (x) to 'xOld' (for absolute error computation purposes).
8. Assign the value of the next approximation [x = x - xPlusOne].
9. Increment the iteration count for the next loop.
10. Increment the iteration count for the next loop.
11. Display the final root value.
12. Ask the user if they wish to run the program again or quit.

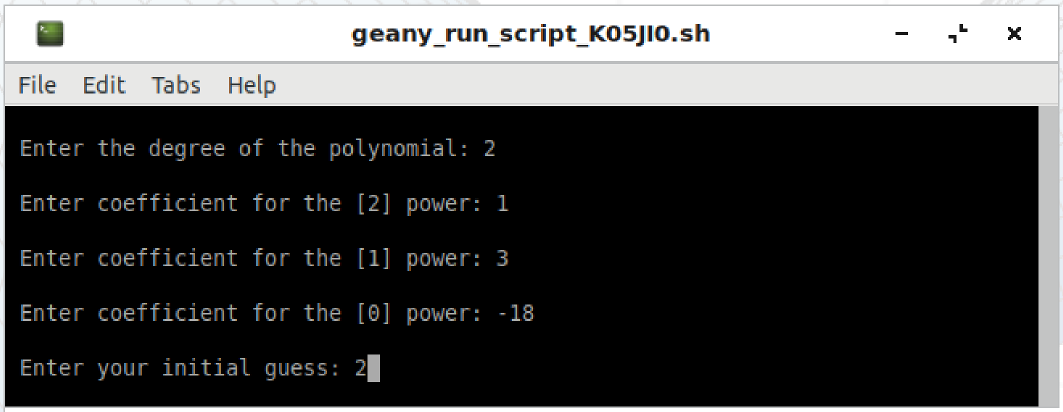


Newton Raphson Sample Computations

The ***Newton-Raphson Solver*** is written in C++ programming language. The program accepts 2 main inputs from the user: an equation and the initial guess.

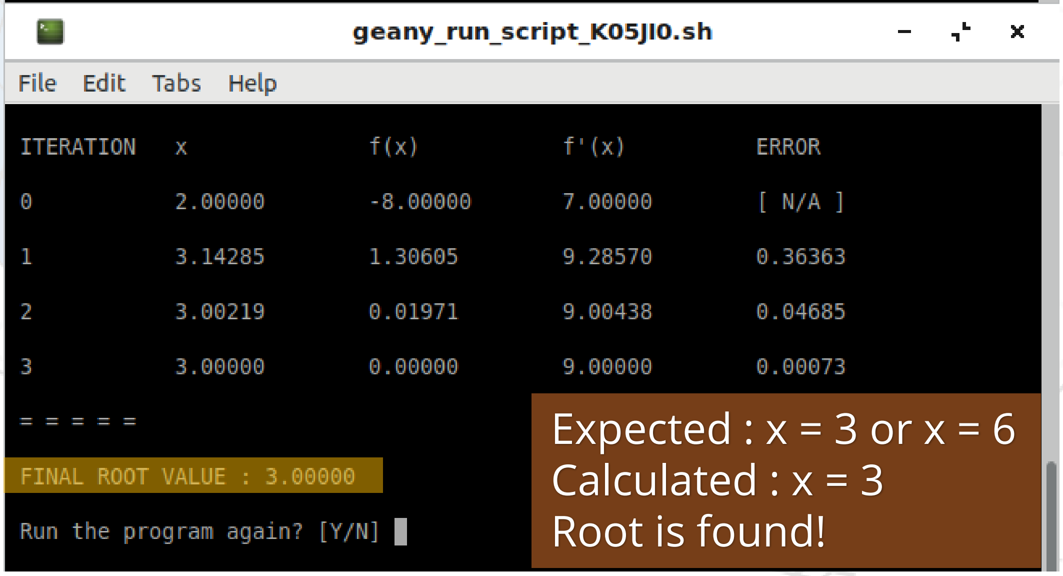
Sample Computation 1.

Given	Stopping Criteria	Expected Values
f(x) = $x^2 + 3x - 18$	iterations = 20	f(x) = (x+6) (x-3)
Initial guess : 2	error = 0.001	x = -6, x = 3



geany_run_script_K05JI0.sh

```
Enter the degree of the polynomial: 2
Enter coefficient for the [2] power: 1
Enter coefficient for the [1] power: 3
Enter coefficient for the [0] power: -18
Enter your initial guess: 2
```



geany_run_script_K05JI0.sh

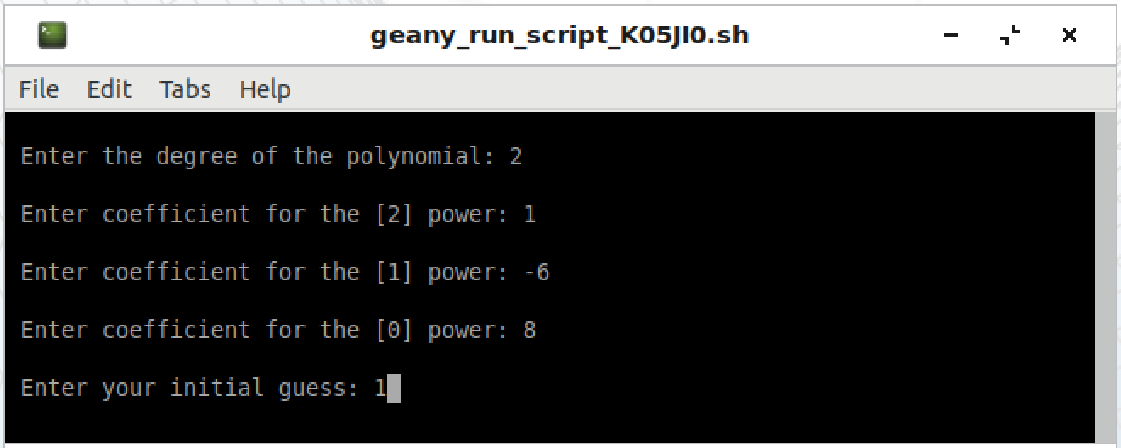
```
ITERATION   x          f(x)        f'(x)       ERROR
0           2.00000    -8.00000    7.00000     [ N/A ]
1           3.14285    1.30605     9.28570     0.36363
2           3.00219    0.01971     9.00438     0.04685
3           3.00000    0.00000     9.00000     0.00073
= = = = =
FINAL ROOT VALUE : 3.00000
Run the program again? [Y/N]
```

Expected : x = 3 or x = 6
Calculated : x = 3
Root is found!

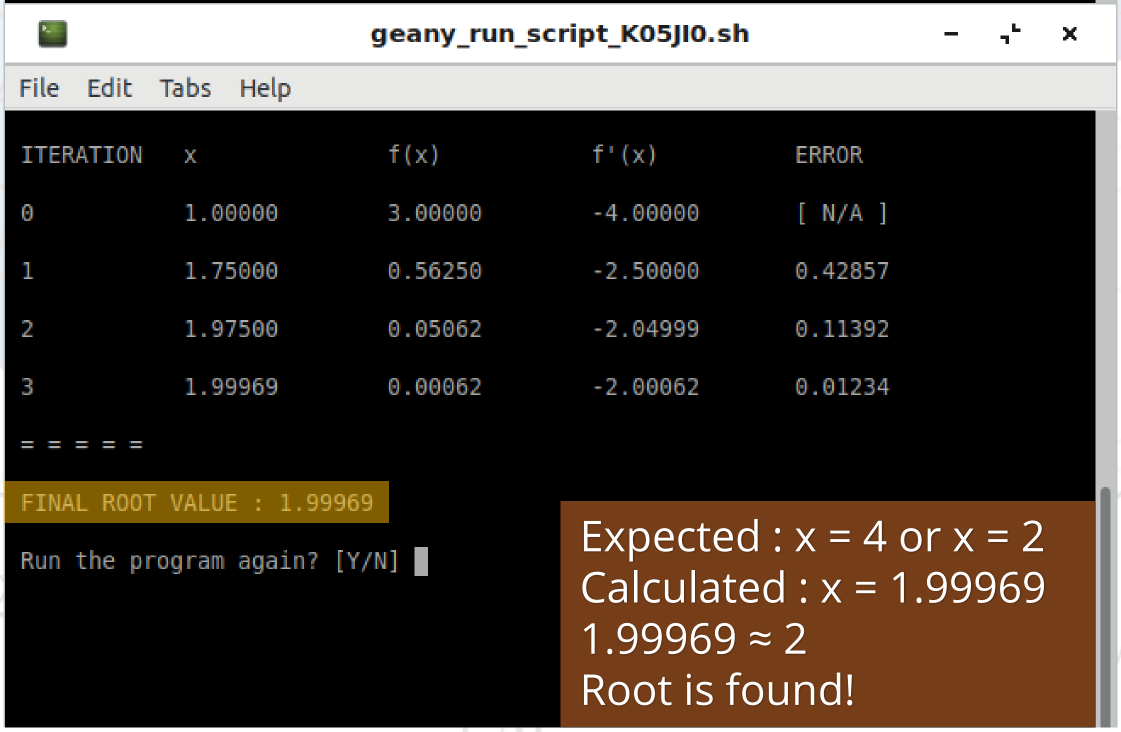
Newton Raphson Sample Computations

Sample Computation 2.

Given	Stopping Criteria	Expected Values
f(x) = $x^2 - 6x + 8$	iterations = 20	f(x) = (x-4) (x-2)
Initial guess : 1	error = 0.001	x = 4, x = 2



```
Enter the degree of the polynomial: 2

Enter coefficient for the [2] power: 1

Enter coefficient for the [1] power: -6

Enter coefficient for the [0] power: 8

Enter your initial guess: 1
```



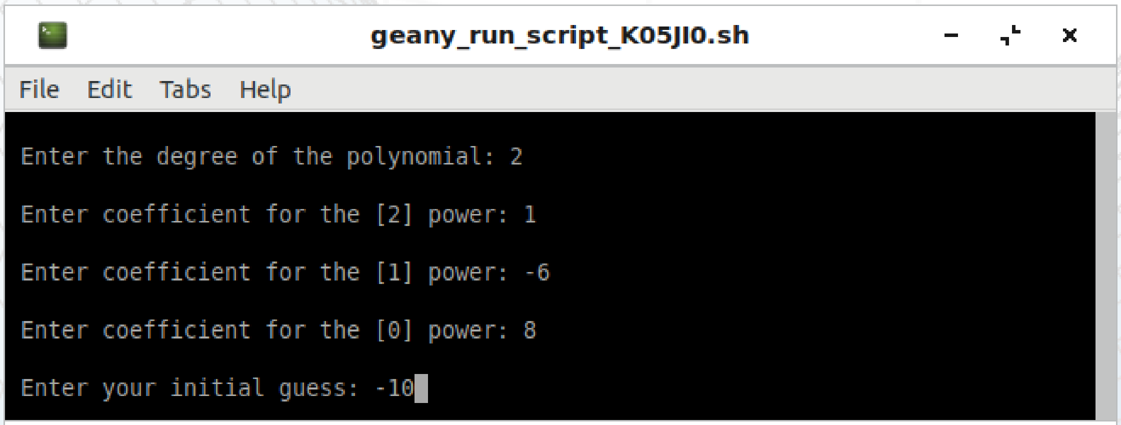
```
ITERATION    x           f(x)         f'(x)        ERROR

0            1.00000     3.00000      -4.00000     [ N/A ]

1            1.75000     0.56250      -2.50000     0.42857

2            1.97500     0.05062      -2.04999     0.11392

3            1.99969     0.00062      -2.00062     0.01234

= = = = =

FINAL ROOT VALUE : 1.99969

Run the program again? [Y/N]
```

Expected : x = 4 or x = 2
Calculated : x = 1.99969
1.99969 ≈ 2
Root is found!

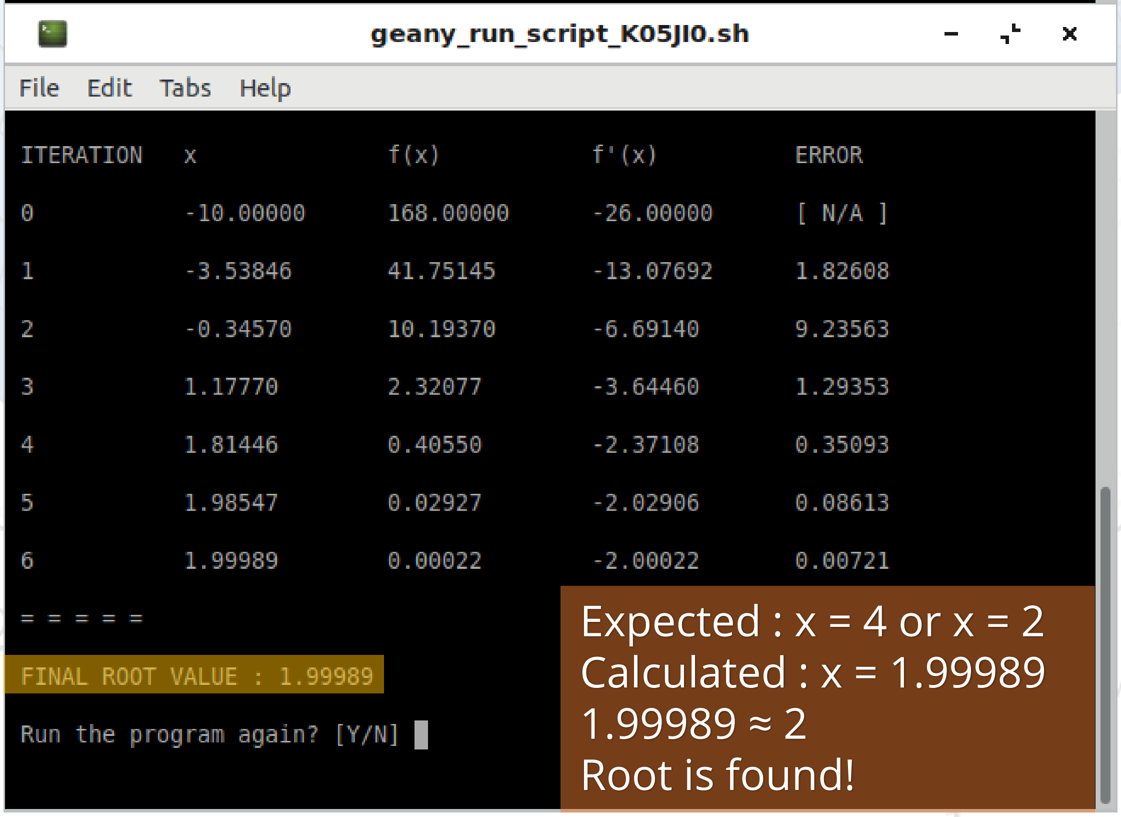
Newton Raphson Sample Computations

Sample Computation 3.

Given	Stopping Criteria	Expected Values
f(x) = $x^2 - 6x + 8$	iterations = 20	f(x) = (x-4) (x-2)
Initial guess : -10	error = 0.001	x = 4, x = 2



```
Enter the degree of the polynomial: 2

Enter coefficient for the [2] power: 1

Enter coefficient for the [1] power: -6

Enter coefficient for the [0] power: 8

Enter your initial guess: -10
```



```
ITERATION   x           f(x)        f'(x)       ERROR

0           -10.00000   168.00000   -26.00000   [ N/A ]

1           -3.53846    41.75145    -13.07692   1.82608

2           -0.34570    10.19370    -6.69140    9.23563

3           1.17770     2.32077     -3.64460    1.29353

4           1.81446     0.40550     -2.37108    0.35093

5           1.98547     0.02927     -2.02906    0.08613

6           1.99989     0.00022     -2.00022    0.00721

= = = = =

FINAL ROOT VALUE : 1.99989

Run the program again? [Y/N]
```

Expected : x = 4 or x = 2
Calculated : x = 1.99989
1.99989 ≈ 2
Root is found!

Bairstow Method Solver

by **VONG, John Matthew**

The **Bairstow method** is an algorithm used to solve for the roots of polynomial equations. It is an iterative algorithm which will continuously try to find the roots of a polynomial equation. It works by having the quadratic equation $x^2 + rx + s$ where r and s are estimated roots for the equation. The algorithm will have values for the variables a_0 up to a_n, b_0 up to b_n, and c_0 up to c_n. The values of a_0 up to a_n are the coefficients of the polynomial, the values of b_0 up to b_n and the values of c_0 upto c_n are taken using the equations: $b_n = c_n = a_n$; $b(i) = a(i) + rb(i+1) + sb(i+2)$; $c(i) = b(i) + rc(i+1)sc(i+2)$. The values of the b and c variables will be adjusted with the current estimate values of r and s to change the values of r and s to a closer estimate. This will continue to iterate until the values of r and s meet the desired conditions and will be used to find the roots of the given polynomial with the equation: $\frac{r \pm \sqrt{r^2+4s}}{2}$.

Bairstow Method Algorithm

The following equations are used in this algorithm.

a. $b_n = c_n = a_n$

b. $b(i) = a(i) + rb(i+1) + sb(i+2)$

c. $c(i) = b(i) + rc(i+1)sc(i+2)$.

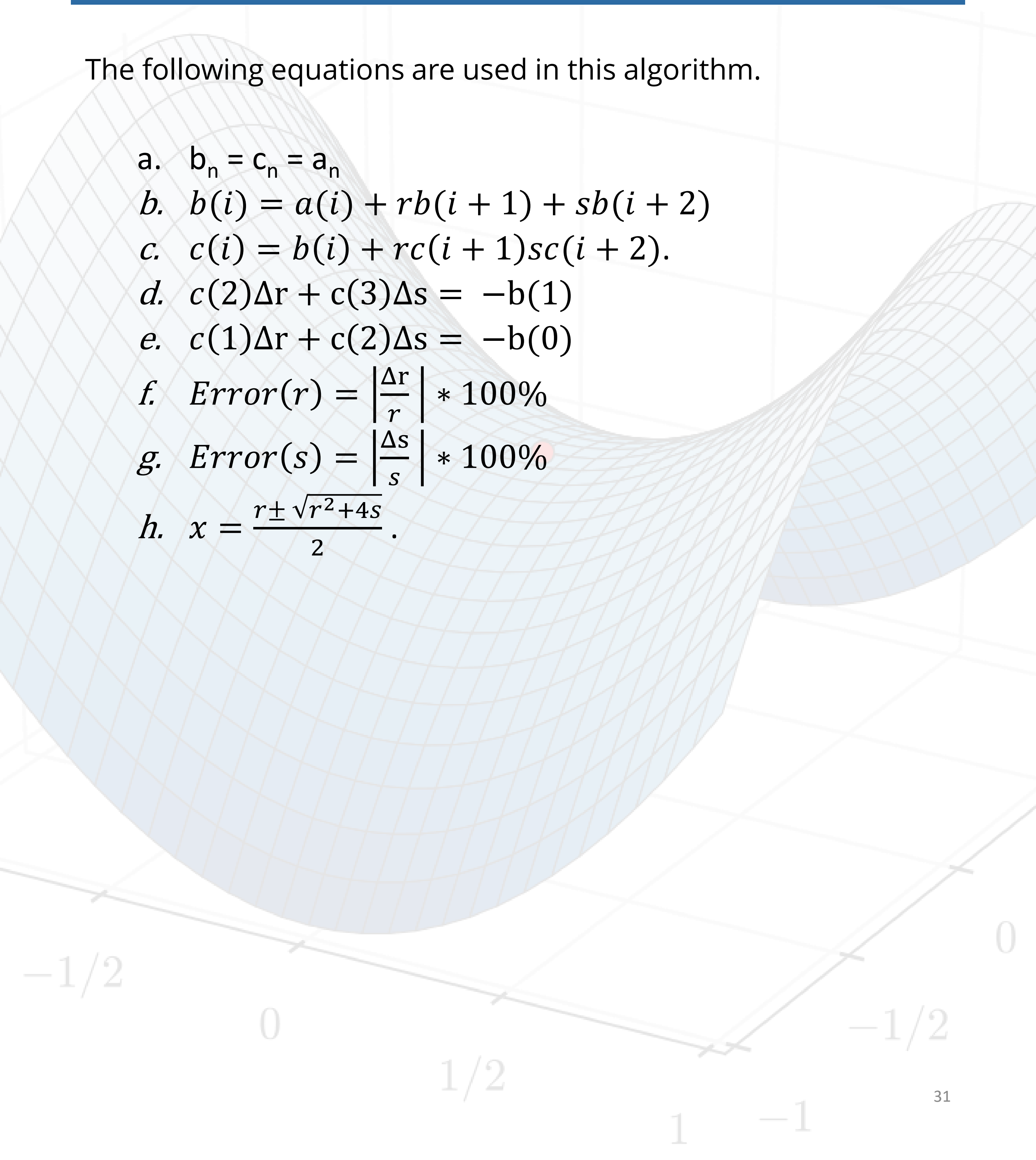
d. $c(2)\Delta\mathrm{r} + \mathrm{c}(3)\Delta\mathrm{s} = -\mathrm{b}(1)$

e. $c(1)\Delta\mathrm{r} + \mathrm{c}(2)\Delta\mathrm{s} = -\mathrm{b}(0)$

f. $Error(r) = \left|\frac{\Delta\mathrm{r}}{r}\right| * 100\%$

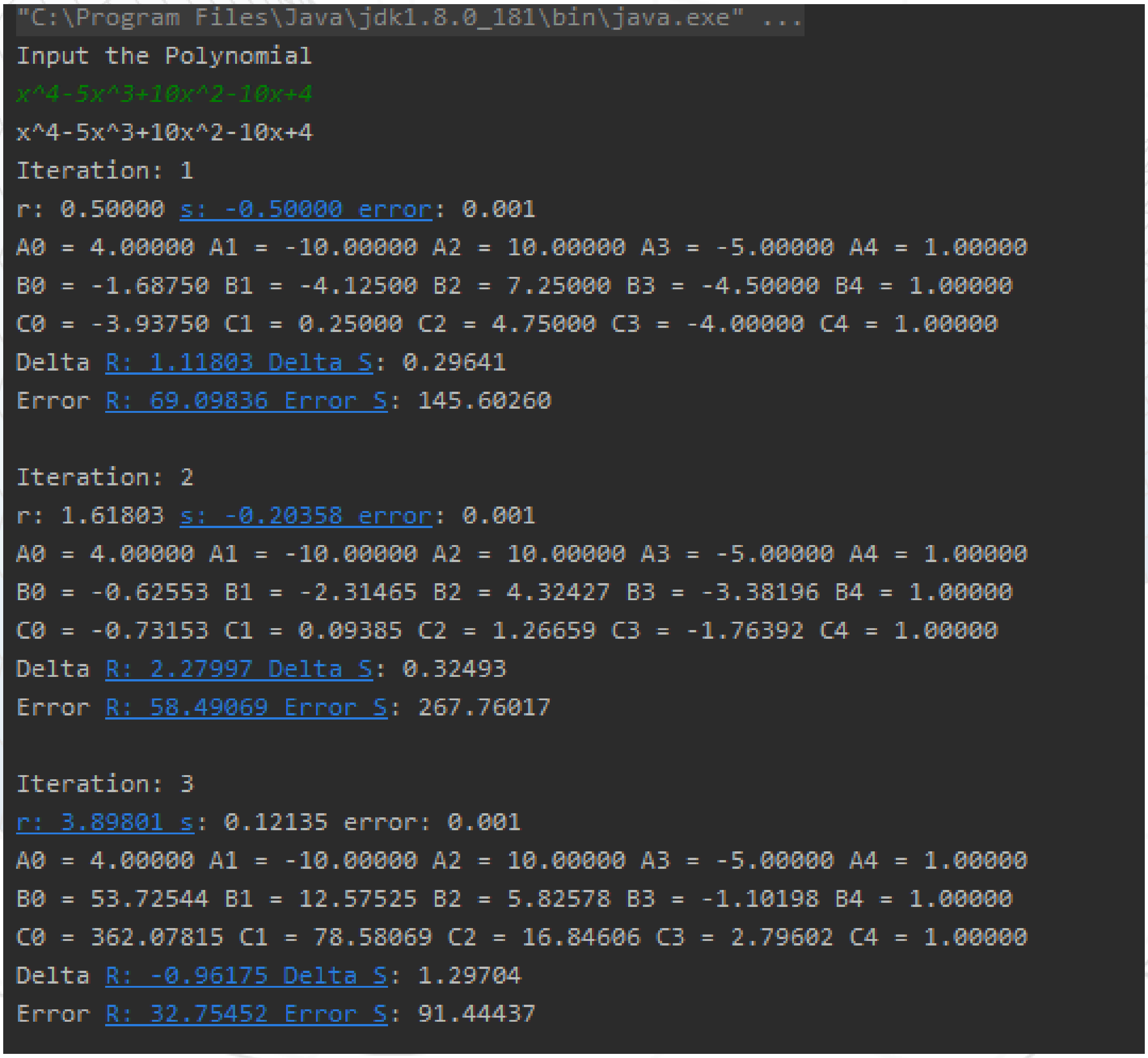
g. $Error(s) = \left|\frac{\Delta\mathrm{s}}{s}\right| * 100\%$

h. $x = \frac{r \pm \sqrt{r^2+4s}}{2}$.



Bairstow Sample Computations

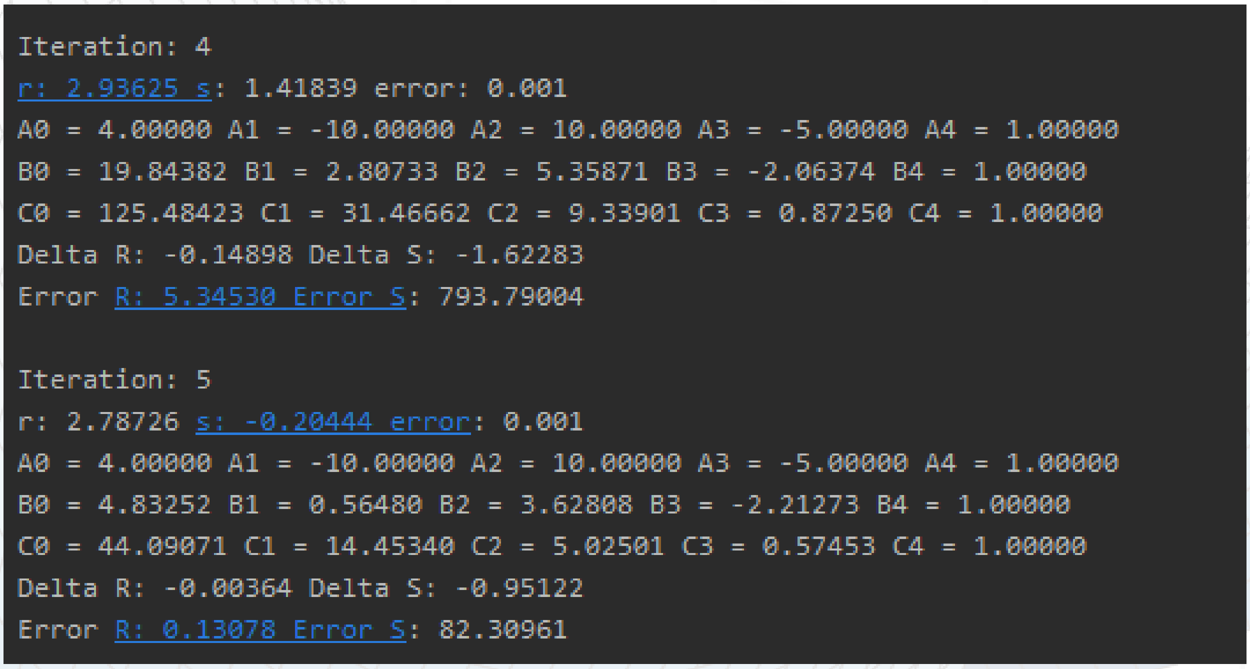
Sample Computation 1.



```
"C:\Program Files\Java\jdk1.8.0_181\bin\java.exe" ...
Input the Polynomial
x^4-5x^3+10x^2-10x+4
x^4-5x^3+10x^2-10x+4
Iteration: 1
r: 0.50000 s: -0.50000 error: 0.001
A0 = 4.00000 A1 = -10.00000 A2 = 10.00000 A3 = -5.00000 A4 = 1.00000
B0 = -1.68750 B1 = -4.12500 B2 = 7.25000 B3 = -4.50000 B4 = 1.00000
C0 = -3.93750 C1 = 0.25000 C2 = 4.75000 C3 = -4.00000 C4 = 1.00000
Delta R: 1.11803 Delta S: 0.29641
Error R: 69.09836 Error S: 145.60260

Iteration: 2
r: 1.61803 s: -0.20358 error: 0.001
A0 = 4.00000 A1 = -10.00000 A2 = 10.00000 A3 = -5.00000 A4 = 1.00000
B0 = -0.62553 B1 = -2.31465 B2 = 4.32427 B3 = -3.38196 B4 = 1.00000
C0 = -0.73153 C1 = 0.09385 C2 = 1.26659 C3 = -1.76392 C4 = 1.00000
Delta R: 2.27997 Delta S: 0.32493
Error R: 58.49069 Error S: 267.76017

Iteration: 3
r: 3.89801 s: 0.12135 error: 0.001
A0 = 4.00000 A1 = -10.00000 A2 = 10.00000 A3 = -5.00000 A4 = 1.00000
B0 = 53.72544 B1 = 12.57525 B2 = 5.82578 B3 = -1.10198 B4 = 1.00000
C0 = 362.07815 C1 = 78.58069 C2 = 16.84606 C3 = 2.79602 C4 = 1.00000
Delta R: -0.96175 Delta S: 1.29704
Error R: 32.75452 Error S: 91.44437
```

Bairstow Sample Computations

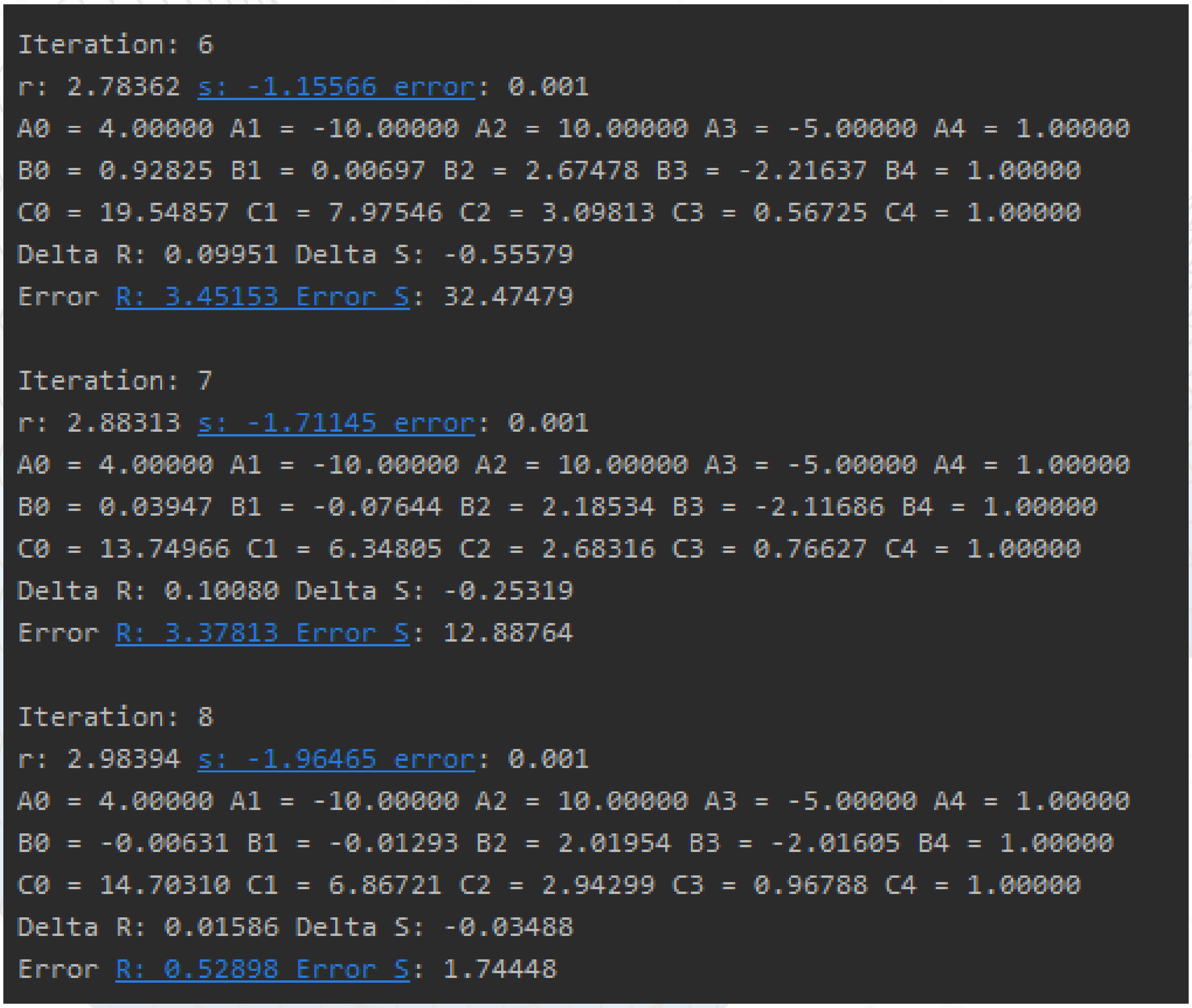
Sample Computation 1. (continued)



```
Iteration: 4
r: 2.93625 s: 1.41839 error: 0.001
A0 = 4.00000 A1 = -10.00000 A2 = 10.00000 A3 = -5.00000 A4 = 1.00000
B0 = 19.84382 B1 = 2.80733 B2 = 5.35871 B3 = -2.06374 B4 = 1.00000
C0 = 125.48423 C1 = 31.46662 C2 = 9.33901 C3 = 0.87250 C4 = 1.00000
Delta R: -0.14898 Delta S: -1.62283
Error R: 5.34530 Error S: 793.79004

Iteration: 5
r: 2.78726 s: -0.20444 error: 0.001
A0 = 4.00000 A1 = -10.00000 A2 = 10.00000 A3 = -5.00000 A4 = 1.00000
B0 = 4.83252 B1 = 0.56480 B2 = 3.62808 B3 = -2.21273 B4 = 1.00000
C0 = 44.09071 C1 = 14.45340 C2 = 5.02501 C3 = 0.57453 C4 = 1.00000
Delta R: -0.00364 Delta S: -0.95122
Error R: 0.13078 Error S: 82.30961
```

Bairstow Sample Computations

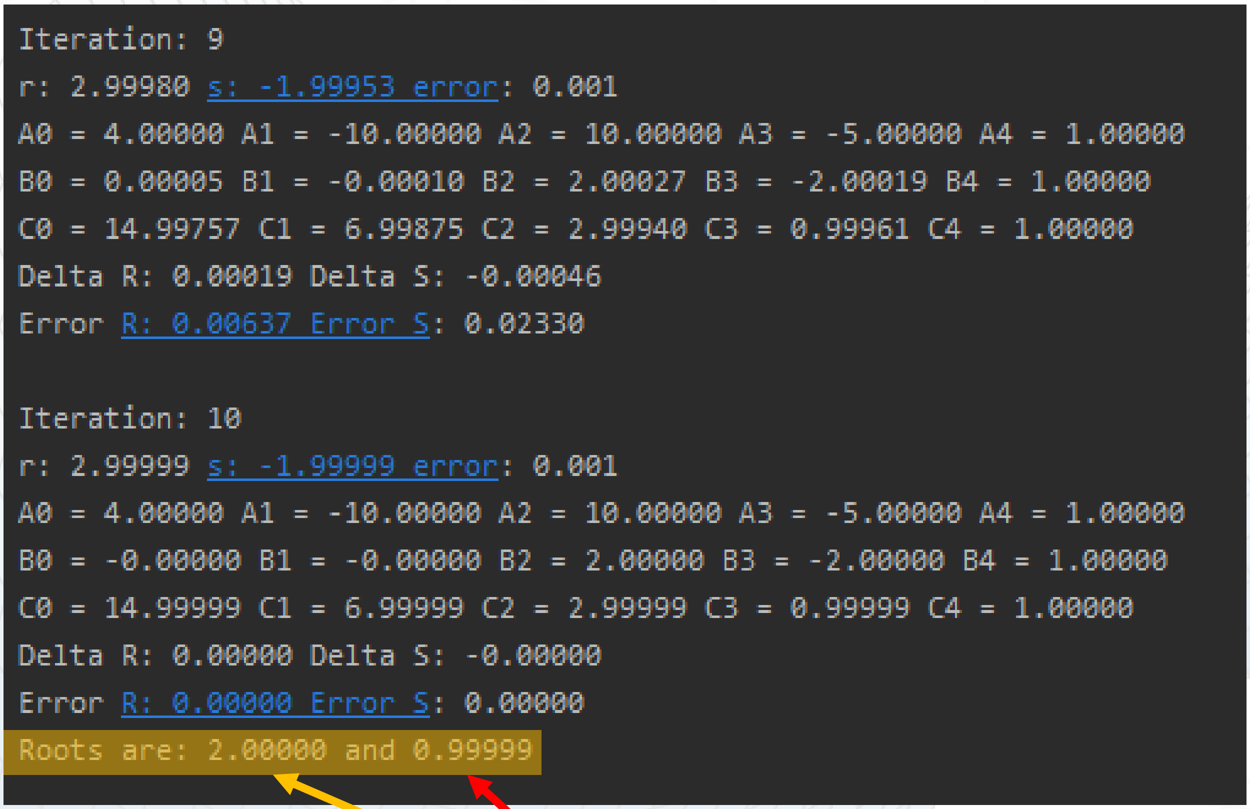
Sample Computation 1. (continued)



```
Iteration: 6
r: 2.78362 s: -1.15566 error: 0.001
A0 = 4.00000 A1 = -10.00000 A2 = 10.00000 A3 = -5.00000 A4 = 1.00000
B0 = 0.92825 B1 = 0.00697 B2 = 2.67478 B3 = -2.21637 B4 = 1.00000
C0 = 19.54857 C1 = 7.97546 C2 = 3.09813 C3 = 0.56725 C4 = 1.00000
Delta R: 0.09951 Delta S: -0.55579
Error R: 3.45153 Error S: 32.47479

Iteration: 7
r: 2.88313 s: -1.71145 error: 0.001
A0 = 4.00000 A1 = -10.00000 A2 = 10.00000 A3 = -5.00000 A4 = 1.00000
B0 = 0.03947 B1 = -0.07644 B2 = 2.18534 B3 = -2.11686 B4 = 1.00000
C0 = 13.74966 C1 = 6.34805 C2 = 2.68316 C3 = 0.76627 C4 = 1.00000
Delta R: 0.10080 Delta S: -0.25319
Error R: 3.37813 Error S: 12.88764

Iteration: 8
r: 2.98394 s: -1.96465 error: 0.001
A0 = 4.00000 A1 = -10.00000 A2 = 10.00000 A3 = -5.00000 A4 = 1.00000
B0 = -0.00631 B1 = -0.01293 B2 = 2.01954 B3 = -2.01605 B4 = 1.00000
C0 = 14.70310 C1 = 6.86721 C2 = 2.94299 C3 = 0.96788 C4 = 1.00000
Delta R: 0.01586 Delta S: -0.03488
Error R: 0.52898 Error S: 1.74448
```

Bairstow Sample Computations

Sample Computation 1. (continued)

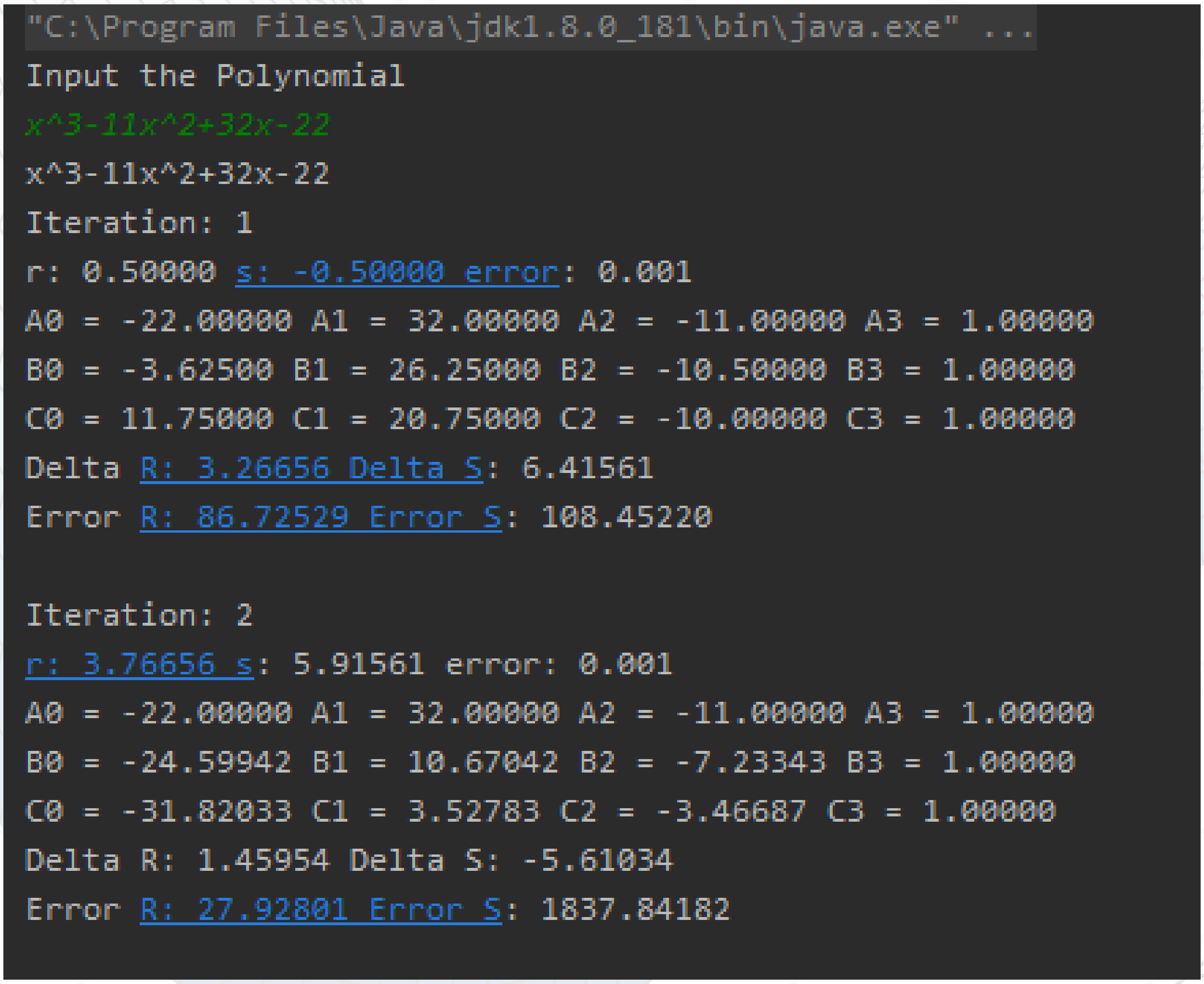


```
Iteration: 9
r: 2.99980 s: -1.99953 error: 0.001
A0 = 4.00000 A1 = -10.00000 A2 = 10.00000 A3 = -5.00000 A4 = 1.00000
B0 = 0.00005 B1 = -0.00010 B2 = 2.00027 B3 = -2.00019 B4 = 1.00000
C0 = 14.99757 C1 = 6.99875 C2 = 2.99940 C3 = 0.99961 C4 = 1.00000
Delta R: 0.00019 Delta S: -0.00046
Error R: 0.00637 Error S: 0.02330

Iteration: 10
r: 2.99999 s: -1.99999 error: 0.001
A0 = 4.00000 A1 = -10.00000 A2 = 10.00000 A3 = -5.00000 A4 = 1.00000
B0 = -0.00000 B1 = -0.00000 B2 = 2.00000 B3 = -2.00000 B4 = 1.00000
C0 = 14.99999 C1 = 6.99999 C2 = 2.99999 C3 = 0.99999 C4 = 1.00000
Delta R: 0.00000 Delta S: -0.00000
Error R: 0.00000 Error S: 0.00000
Roots are: 2.00000 and 0.99999
```

Roots of $x^4 - 5x^3 + 10x^2 - 10x + 4$: $\quad x = 1, x = 2, x = 1 + i, x = 1 - i$

Bairstow Sample Computations

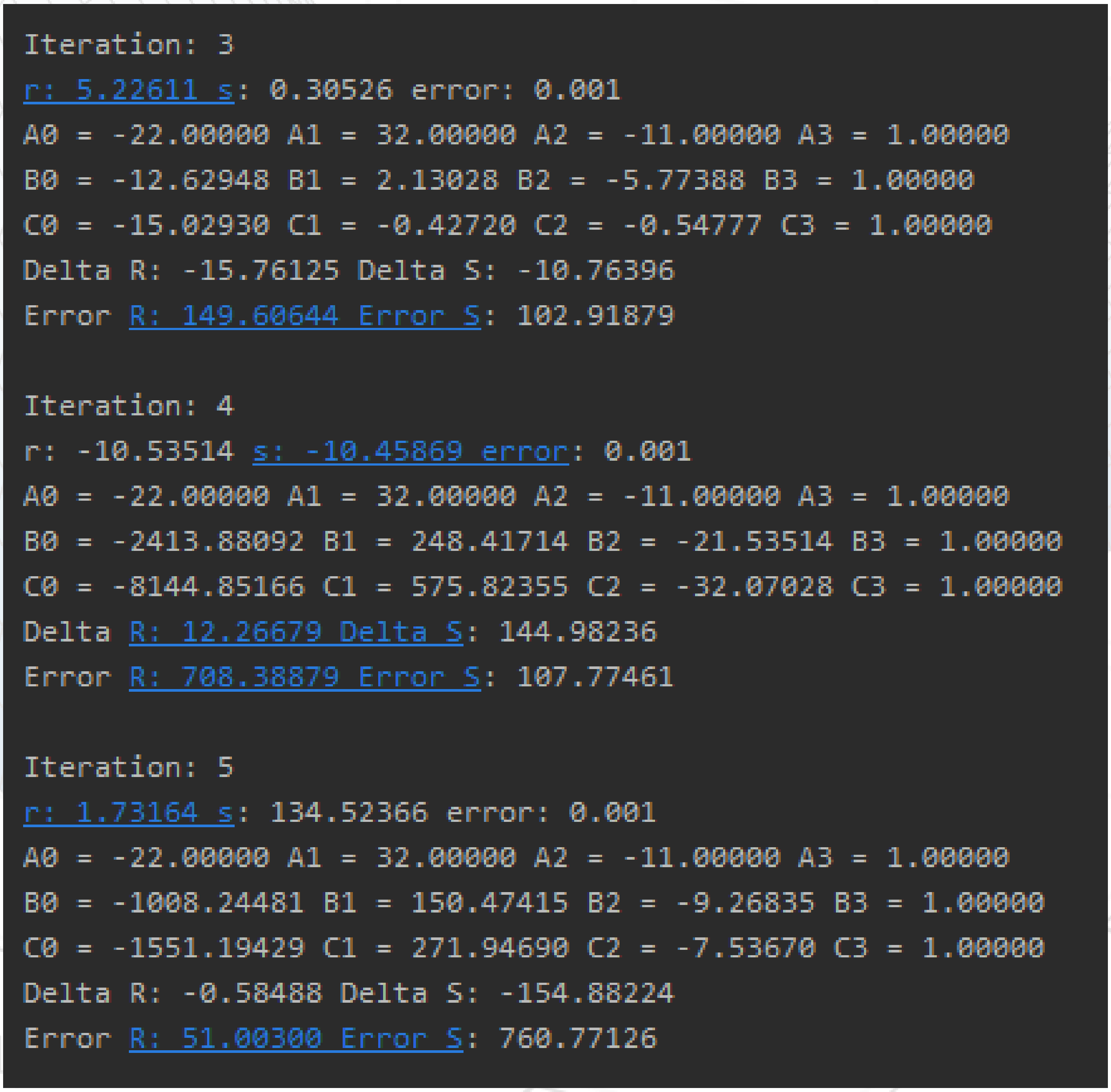
Sample Computation 2.



```
"C:\Program Files\Java\jdk1.8.0_181\bin\java.exe" ...
Input the Polynomial
x^3-11x^2+32x-22
x^3-11x^2+32x-22
Iteration: 1
r: 0.50000 s: -0.50000 error: 0.001
A0 = -22.00000 A1 = 32.00000 A2 = -11.00000 A3 = 1.00000
B0 = -3.62500 B1 = 26.25000 B2 = -10.50000 B3 = 1.00000
C0 = 11.75000 C1 = 20.75000 C2 = -10.00000 C3 = 1.00000
Delta R: 3.26656 Delta S: 6.41561
Error R: 86.72529 Error S: 108.45220

Iteration: 2
r: 3.76656 s: 5.91561 error: 0.001
A0 = -22.00000 A1 = 32.00000 A2 = -11.00000 A3 = 1.00000
B0 = -24.59942 B1 = 10.67042 B2 = -7.23343 B3 = 1.00000
C0 = -31.82033 C1 = 3.52783 C2 = -3.46687 C3 = 1.00000
Delta R: 1.45954 Delta S: -5.61034
Error R: 27.92801 Error S: 1837.84182
```

Bairstow Sample Computations

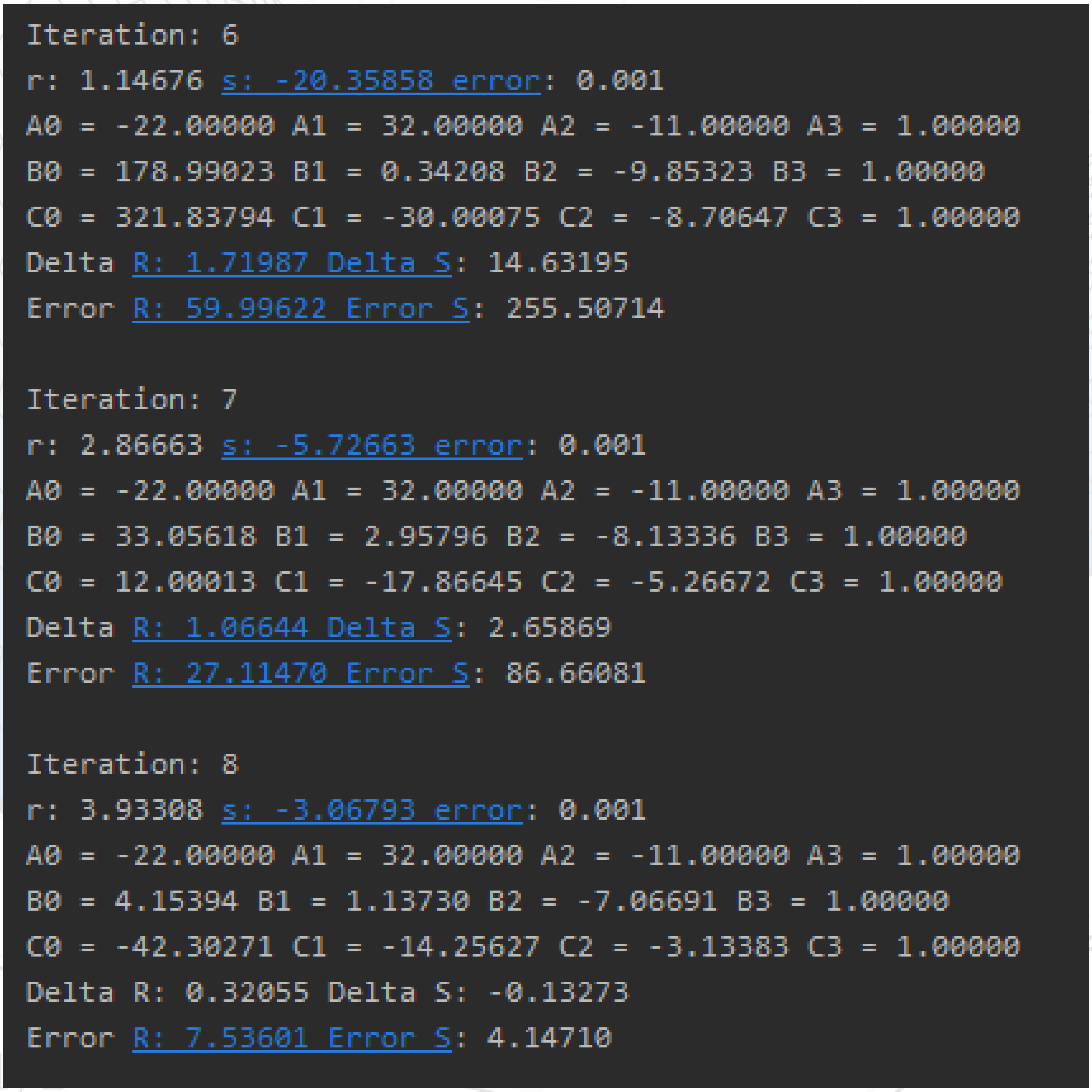
Sample Computation 2. (continued)



```
Iteration: 3
r: 5.22611 s: 0.30526 error: 0.001
A0 = -22.00000 A1 = 32.00000 A2 = -11.00000 A3 = 1.00000
B0 = -12.62948 B1 = 2.13028 B2 = -5.77388 B3 = 1.00000
C0 = -15.02930 C1 = -0.42720 C2 = -0.54777 C3 = 1.00000
Delta R: -15.76125 Delta S: -10.76396
Error R: 149.60644 Error S: 102.91879

Iteration: 4
r: -10.53514 s: -10.45869 error: 0.001
A0 = -22.00000 A1 = 32.00000 A2 = -11.00000 A3 = 1.00000
B0 = -2413.88092 B1 = 248.41714 B2 = -21.53514 B3 = 1.00000
C0 = -8144.85166 C1 = 575.82355 C2 = -32.07028 C3 = 1.00000
Delta R: 12.26679 Delta S: 144.98236
Error R: 708.38879 Error S: 107.77461

Iteration: 5
r: 1.73164 s: 134.52366 error: 0.001
A0 = -22.00000 A1 = 32.00000 A2 = -11.00000 A3 = 1.00000
B0 = -1008.24481 B1 = 150.47415 B2 = -9.26835 B3 = 1.00000
C0 = -1551.19429 C1 = 271.94690 C2 = -7.53670 C3 = 1.00000
Delta R: -0.58488 Delta S: -154.88224
Error R: 51.00300 Error S: 760.77126
```

Bairstow Sample Computations

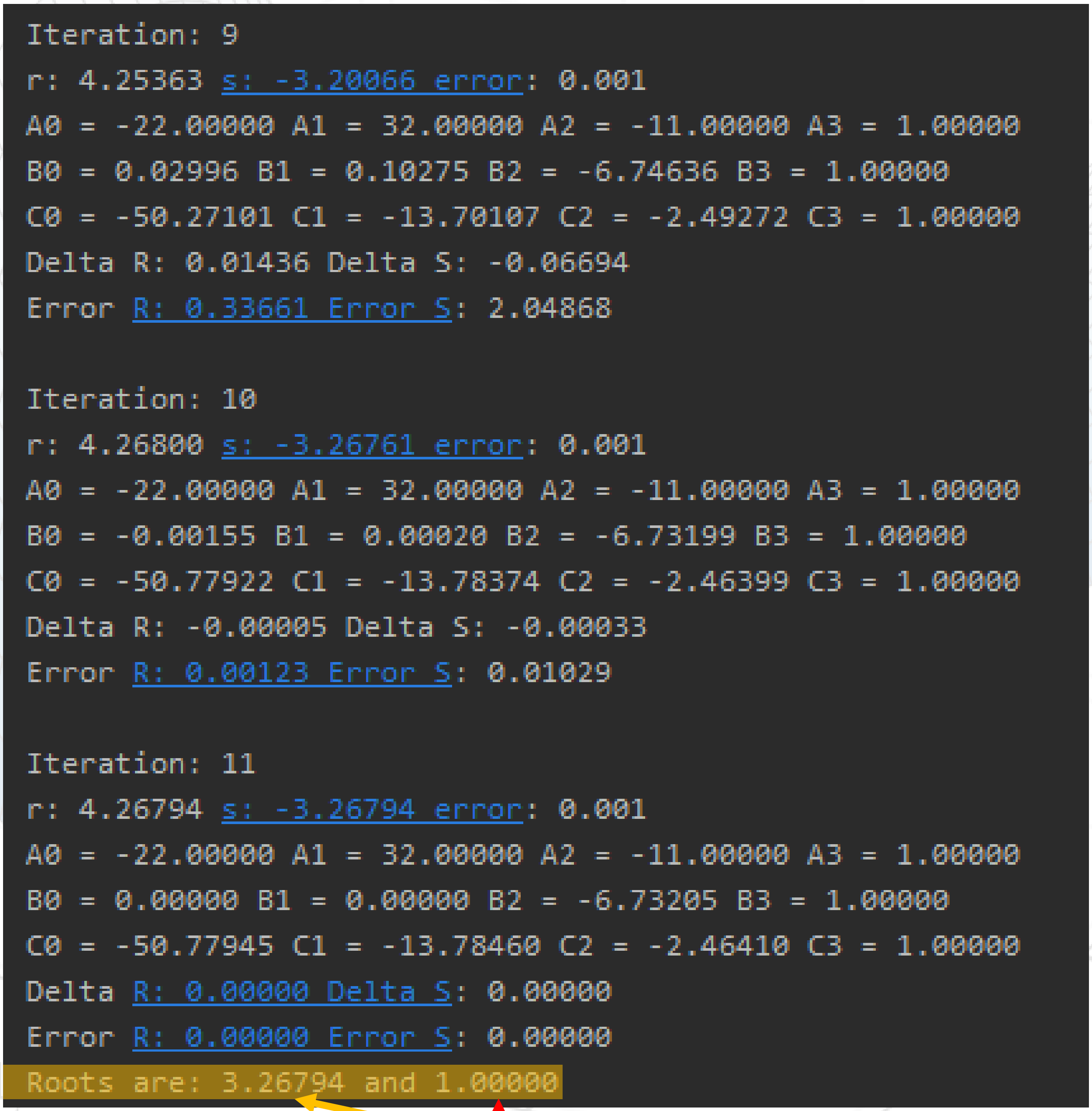
Sample Computation 2. (continued)



```
Iteration: 6
r: 1.14676 s: -20.35858 error: 0.001
A0 = -22.00000 A1 = 32.00000 A2 = -11.00000 A3 = 1.00000
B0 = 178.99023 B1 = 0.34208 B2 = -9.85323 B3 = 1.00000
C0 = 321.83794 C1 = -30.00075 C2 = -8.70647 C3 = 1.00000
Delta R: 1.71987 Delta S: 14.63195
Error R: 59.99622 Error S: 255.50714

Iteration: 7
r: 2.86663 s: -5.72663 error: 0.001
A0 = -22.00000 A1 = 32.00000 A2 = -11.00000 A3 = 1.00000
B0 = 33.05618 B1 = 2.95796 B2 = -8.13336 B3 = 1.00000
C0 = 12.00013 C1 = -17.86645 C2 = -5.26672 C3 = 1.00000
Delta R: 1.06644 Delta S: 2.65869
Error R: 27.11470 Error S: 86.66081

Iteration: 8
r: 3.93308 s: -3.06793 error: 0.001
A0 = -22.00000 A1 = 32.00000 A2 = -11.00000 A3 = 1.00000
B0 = 4.15394 B1 = 1.13730 B2 = -7.06691 B3 = 1.00000
C0 = -42.30271 C1 = -14.25627 C2 = -3.13383 C3 = 1.00000
Delta R: 0.32055 Delta S: -0.13273
Error R: 7.53601 Error S: 4.14710
```

Bairstow Sample Computations

Sample Computation 2. (continued)

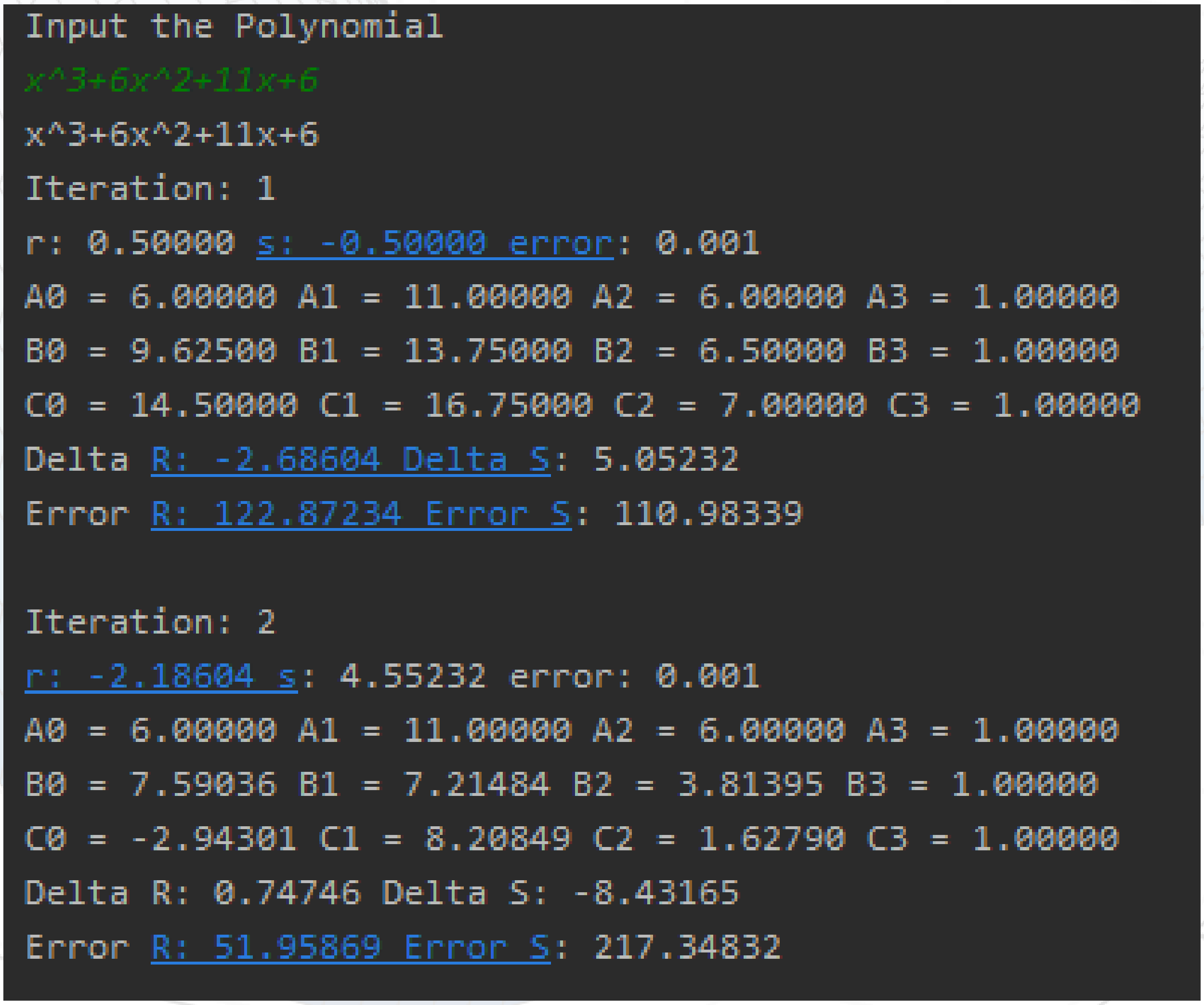


```
Iteration: 9
r: 4.25363 s: -3.20066 error: 0.001
A0 = -22.00000 A1 = 32.00000 A2 = -11.00000 A3 = 1.00000
B0 = 0.02996 B1 = 0.10275 B2 = -6.74636 B3 = 1.00000
C0 = -50.27101 C1 = -13.70107 C2 = -2.49272 C3 = 1.00000
Delta R: 0.01436 Delta S: -0.06694
Error R: 0.33661 Error S: 2.04868

Iteration: 10
r: 4.26800 s: -3.26761 error: 0.001
A0 = -22.00000 A1 = 32.00000 A2 = -11.00000 A3 = 1.00000
B0 = -0.00155 B1 = 0.00020 B2 = -6.73199 B3 = 1.00000
C0 = -50.77922 C1 = -13.78374 C2 = -2.46399 C3 = 1.00000
Delta R: -0.00005 Delta S: -0.00033
Error R: 0.00123 Error S: 0.01029

Iteration: 11
r: 4.26794 s: -3.26794 error: 0.001
A0 = -22.00000 A1 = 32.00000 A2 = -11.00000 A3 = 1.00000
B0 = 0.00000 B1 = 0.00000 B2 = -6.73205 B3 = 1.00000
C0 = -50.77945 C1 = -13.78460 C2 = -2.46410 C3 = 1.00000
Delta R: 0.00000 Delta S: 0.00000
Error R: 0.00000 Error S: 0.00000
Roots are: 3.26794 and 1.00000
```

Roots of $x^3 - 11x^2 + 32x - 22$: $\quad x = 1, x = 5 + \sqrt{3}, x = 5 - \sqrt{3}$

Bairstow Sample Computations

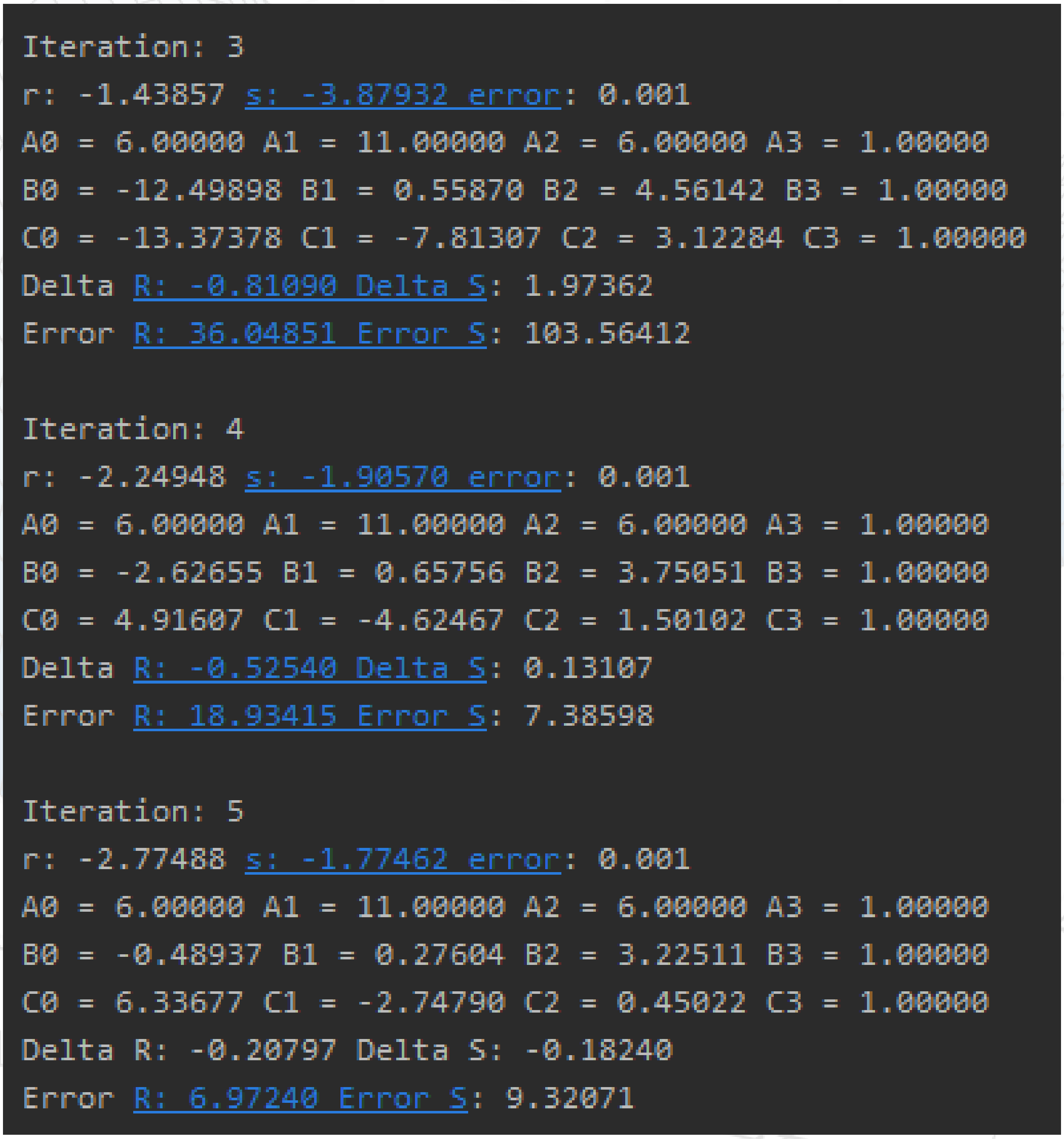
Sample Computation 3.



```
Input the Polynomial
x^3+6x^2+11x+6
x^3+6x^2+11x+6
Iteration: 1
r: 0.50000 s: -0.50000 error: 0.001
A0 = 6.00000 A1 = 11.00000 A2 = 6.00000 A3 = 1.00000
B0 = 9.62500 B1 = 13.75000 B2 = 6.50000 B3 = 1.00000
C0 = 14.50000 C1 = 16.75000 C2 = 7.00000 C3 = 1.00000
Delta R: -2.68604 Delta S: 5.05232
Error R: 122.87234 Error S: 110.98339

Iteration: 2
r: -2.18604 s: 4.55232 error: 0.001
A0 = 6.00000 A1 = 11.00000 A2 = 6.00000 A3 = 1.00000
B0 = 7.59036 B1 = 7.21484 B2 = 3.81395 B3 = 1.00000
C0 = -2.94301 C1 = 8.20849 C2 = 1.62790 C3 = 1.00000
Delta R: 0.74746 Delta S: -8.43165
Error R: 51.95869 Error S: 217.34832
```

Bairstow Sample Computations

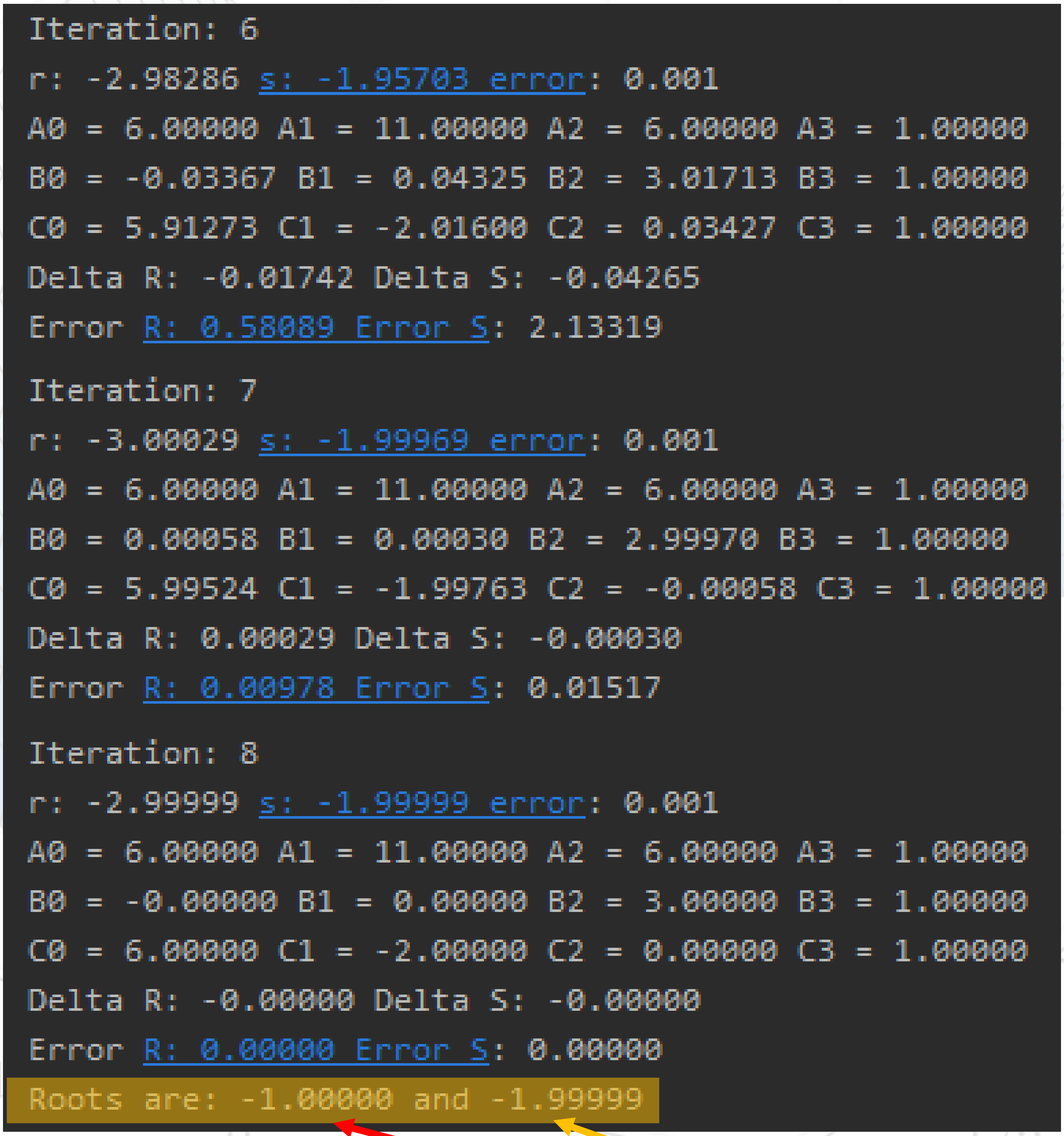
Sample Computation 3. (continued)



```
Iteration: 3
r: -1.43857 s: -3.87932 error: 0.001
A0 = 6.00000 A1 = 11.00000 A2 = 6.00000 A3 = 1.00000
B0 = -12.49898 B1 = 0.55870 B2 = 4.56142 B3 = 1.00000
C0 = -13.37378 C1 = -7.81307 C2 = 3.12284 C3 = 1.00000
Delta R: -0.81090 Delta S: 1.97362
Error R: 36.04851 Error S: 103.56412

Iteration: 4
r: -2.24948 s: -1.90570 error: 0.001
A0 = 6.00000 A1 = 11.00000 A2 = 6.00000 A3 = 1.00000
B0 = -2.62655 B1 = 0.65756 B2 = 3.75051 B3 = 1.00000
C0 = 4.91607 C1 = -4.62467 C2 = 1.50102 C3 = 1.00000
Delta R: -0.52540 Delta S: 0.13107
Error R: 18.93415 Error S: 7.38598

Iteration: 5
r: -2.77488 s: -1.77462 error: 0.001
A0 = 6.00000 A1 = 11.00000 A2 = 6.00000 A3 = 1.00000
B0 = -0.48937 B1 = 0.27604 B2 = 3.22511 B3 = 1.00000
C0 = 6.33677 C1 = -2.74790 C2 = 0.45022 C3 = 1.00000
Delta R: -0.20797 Delta S: -0.18240
Error R: 6.97240 Error S: 9.32071
```

Bairstow Sample Computations

Sample Computation 3. (continued)



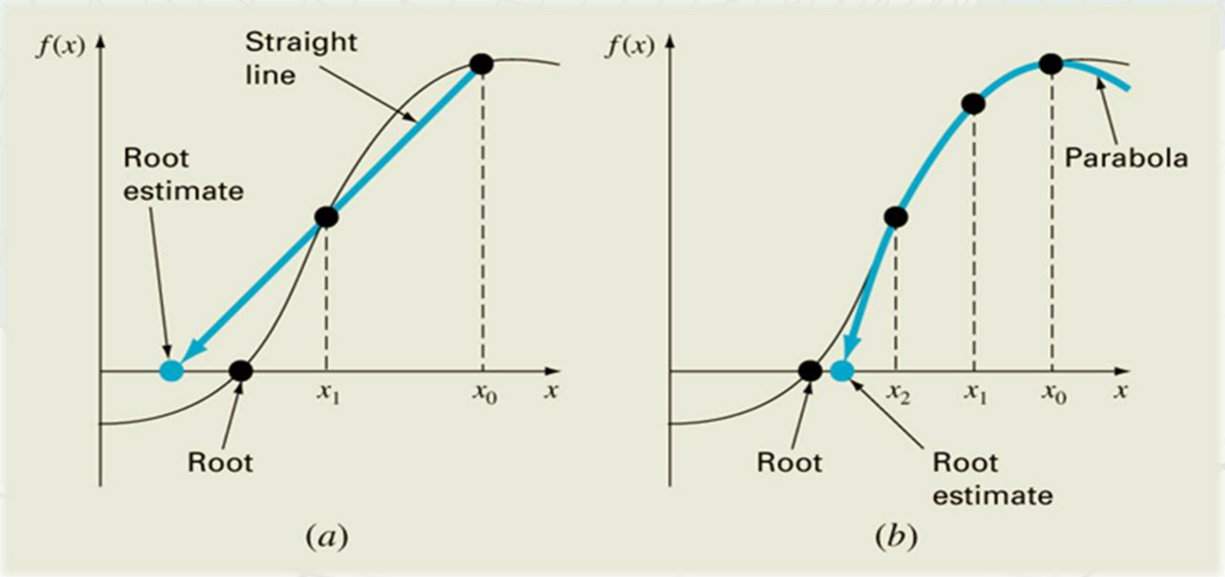
```
Iteration: 6
r: -2.98286 s: -1.95703 error: 0.001
A0 = 6.00000 A1 = 11.00000 A2 = 6.00000 A3 = 1.00000
B0 = -0.03367 B1 = 0.04325 B2 = 3.01713 B3 = 1.00000
C0 = 5.91273 C1 = -2.01600 C2 = 0.03427 C3 = 1.00000
Delta R: -0.01742 Delta S: -0.04265
Error R: 0.58089 Error S: 2.13319

Iteration: 7
r: -3.00029 s: -1.99969 error: 0.001
A0 = 6.00000 A1 = 11.00000 A2 = 6.00000 A3 = 1.00000
B0 = 0.00058 B1 = 0.00030 B2 = 2.99970 B3 = 1.00000
C0 = 5.99524 C1 = -1.99763 C2 = -0.00058 C3 = 1.00000
Delta R: 0.00029 Delta S: -0.00030
Error R: 0.00978 Error S: 0.01517

Iteration: 8
r: -2.99999 s: -1.99999 error: 0.001
A0 = 6.00000 A1 = 11.00000 A2 = 6.00000 A3 = 1.00000
B0 = -0.00000 B1 = 0.00000 B2 = 3.00000 B3 = 1.00000
C0 = 6.00000 C1 = -2.00000 C2 = 0.00000 C3 = 1.00000
Delta R: -0.00000 Delta S: -0.00000
Error R: 0.00000 Error S: 0.00000
Roots are: -1.00000 and -1.99999
```

Roots of $x^3 + 6x^2 + 11x + 6$: $\quad x = -1, x = -2, x = -3$

Müller Method Solver

by **MAGCAMIT, Anaheim**

The **Müller Method** is a special method that finds the real and complex roots of polynomials by projecting a parabola to the x-axis through three initial values. The root will be determined by considering the intersection of the parabola with the three points *[x_o, $f(x_o)$], [x_1, $f(x_1)$], [x_2, $f(x_2)$].*



Geometrical illustration of the Müller Method

Müller Method Algorithm

The following variables are used in this algorithm.

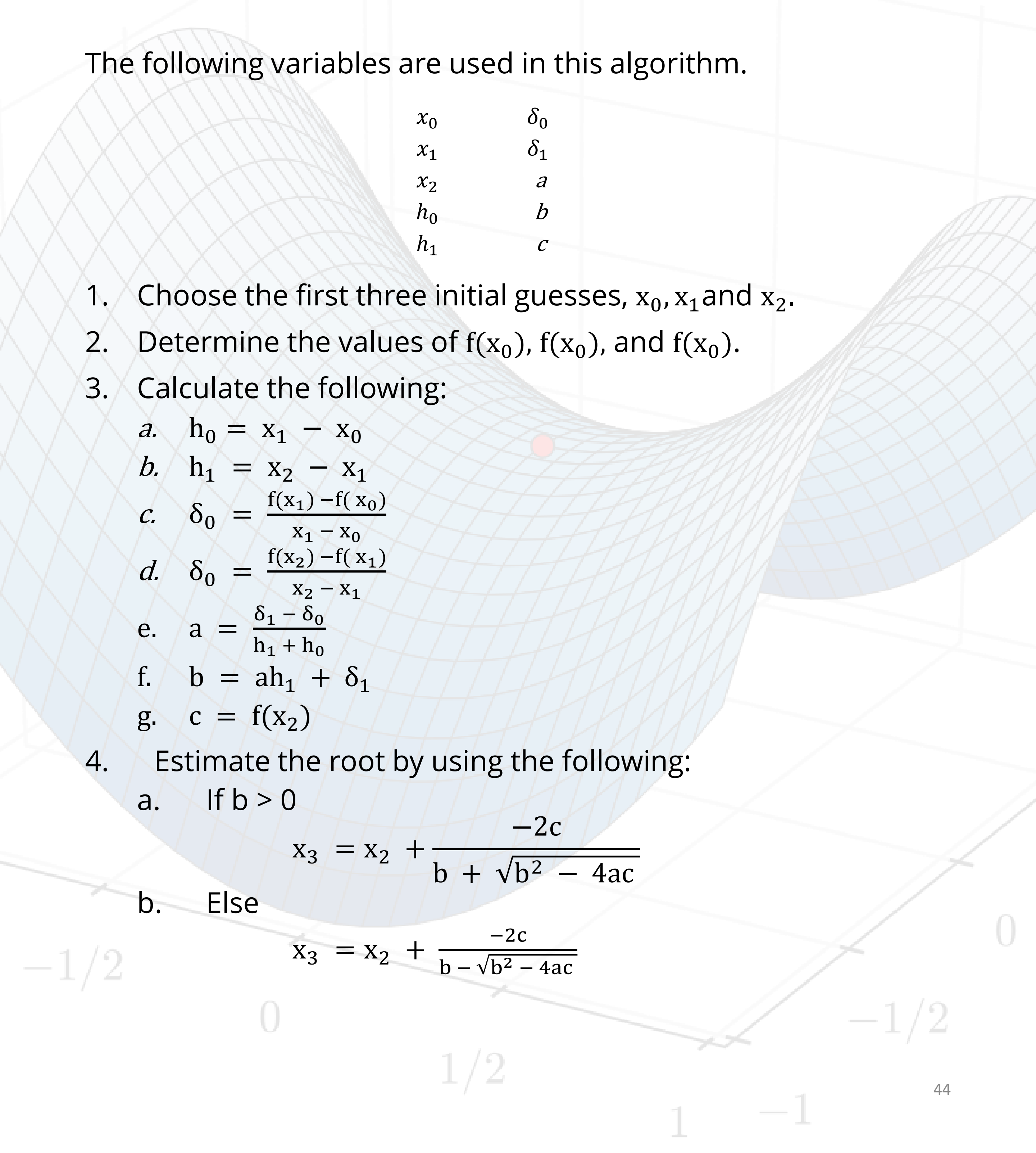
x_0	δ_0
x_1	δ_1
x_2	a
h_0	b
h_1	c

1. Choose the first three initial guesses, x_0, x_1 and x_2.
2. Determine the values of $f(x_0)$, $f(x_0)$, and $f(x_0)$.
3. Calculate the following:
   a. $h_0 = x_1 - x_0$
   b. $h_1 = x_2 - x_1$
   c. $\delta_0 = \frac{f(x_1) - f(x_0)}{x_1 - x_0}$
   d. $\delta_0 = \frac{f(x_2) - f(x_1)}{x_2 - x_1}$
   e. $a = \frac{\delta_1 - \delta_0}{h_1 + h_0}$
   f. $b = ah_1 + \delta_1$
   g. $c = f(x_2)$
4. Estimate the root by using the following:
   a. If b > 0

$$x_3 = x_2 + \frac{-2c}{b + \sqrt{b^2 - 4ac}}$$

   b. Else

$$x_3 = x_2 + \frac{-2c}{b - \sqrt{b^2 - 4ac}}$$

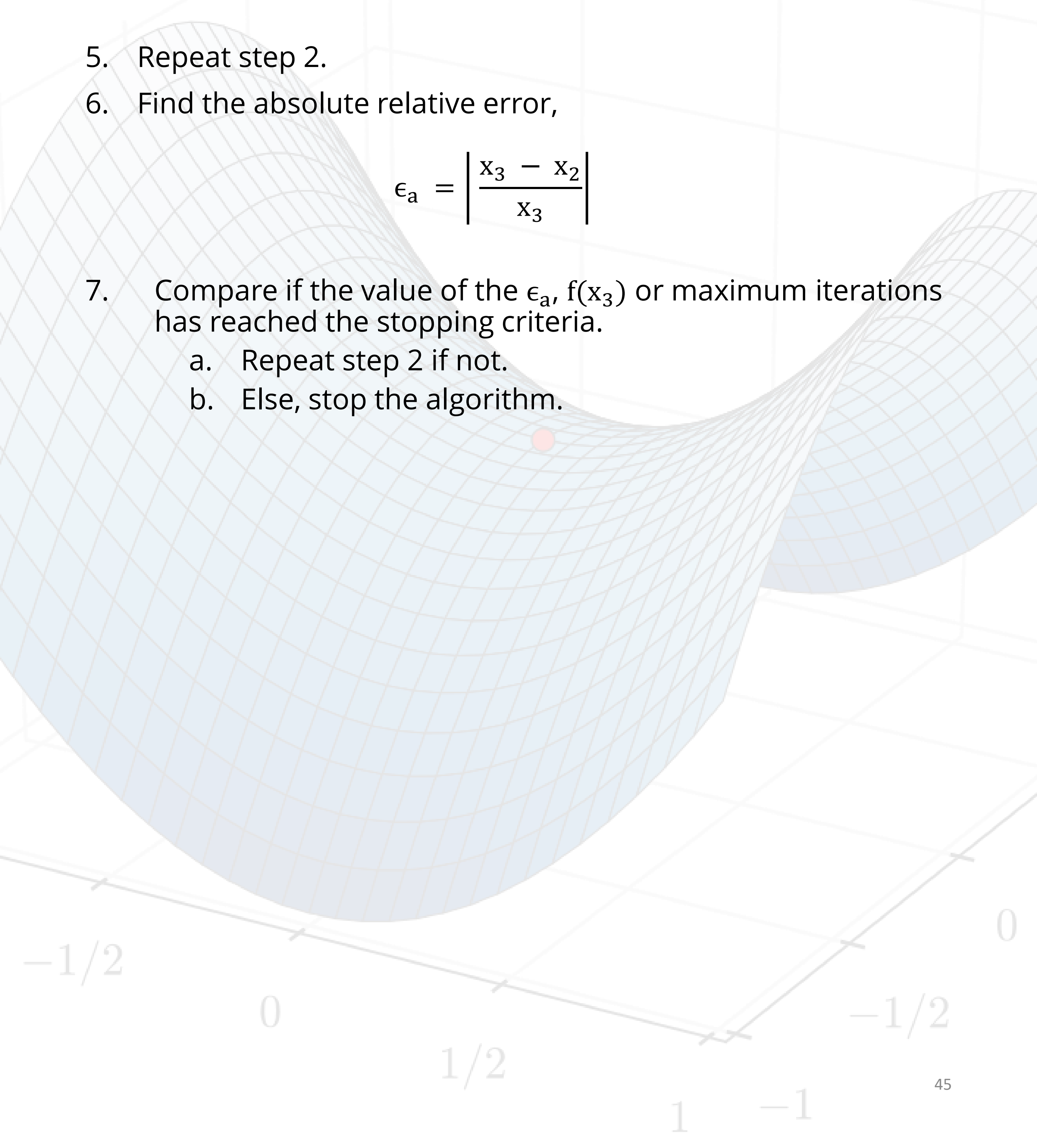


Müller Method Algorithm

5. Repeat step 2.
6. Find the absolute relative error,

$$\epsilon_a = \left| \frac{x_3 - x_2}{x_3} \right|$$

7. Compare if the value of the ϵ_a, $f(x_3)$ or maximum iterations has reached the stopping criteria.
   a. Repeat step 2 if not.
   b. Else, stop the algorithm.

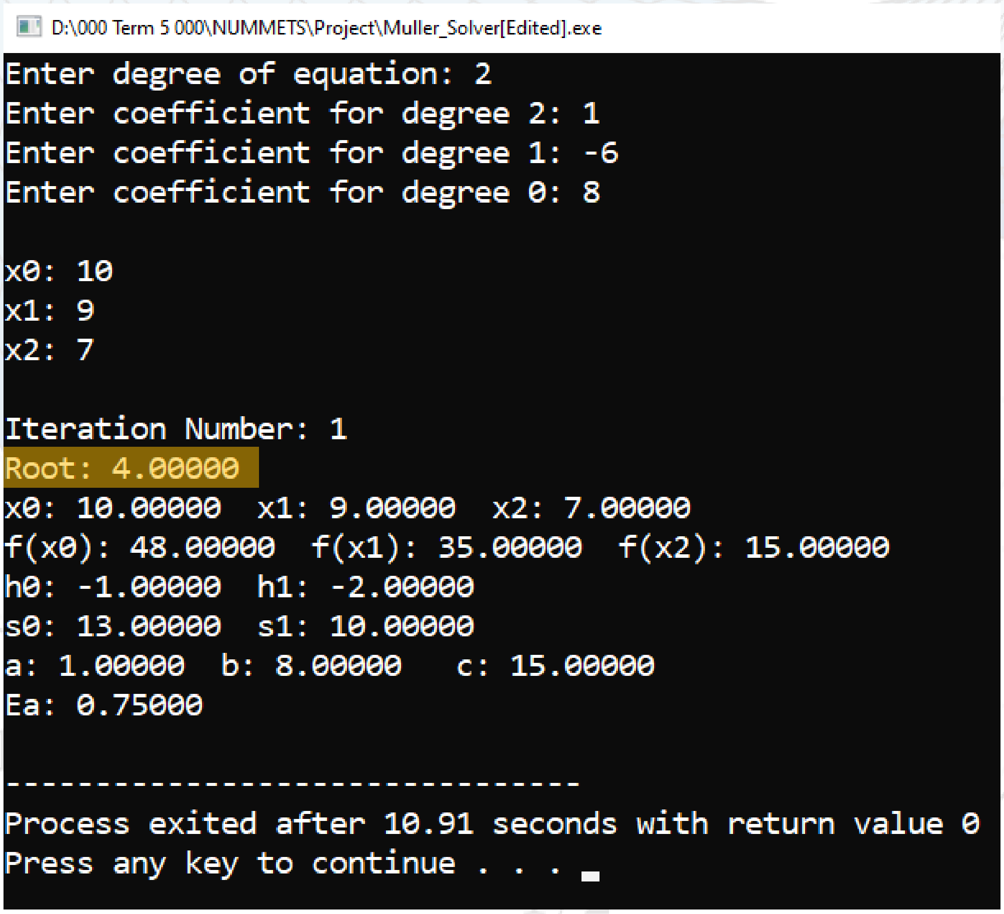


Müller Sample Computations

The Muller Solver is created using C/C++ Programming Language. For the program to run, the user must provide 6 inputs: the equation, the three initial guesses, x_0, x_1 and x_2, and the stopping criteria, the maximum iterations, error and $f(x_3)$.

Sample Computation 1.

Given	Stopping Criteria
f(x) = $x^2 - 6x + 8$	iterations = 100
$\boldsymbol{x_0}$ = 10	error = 0.00001
$\boldsymbol{x_1}$ = 9	$f(x)$ = 0.0001
$\boldsymbol{x_2}$ = 7	



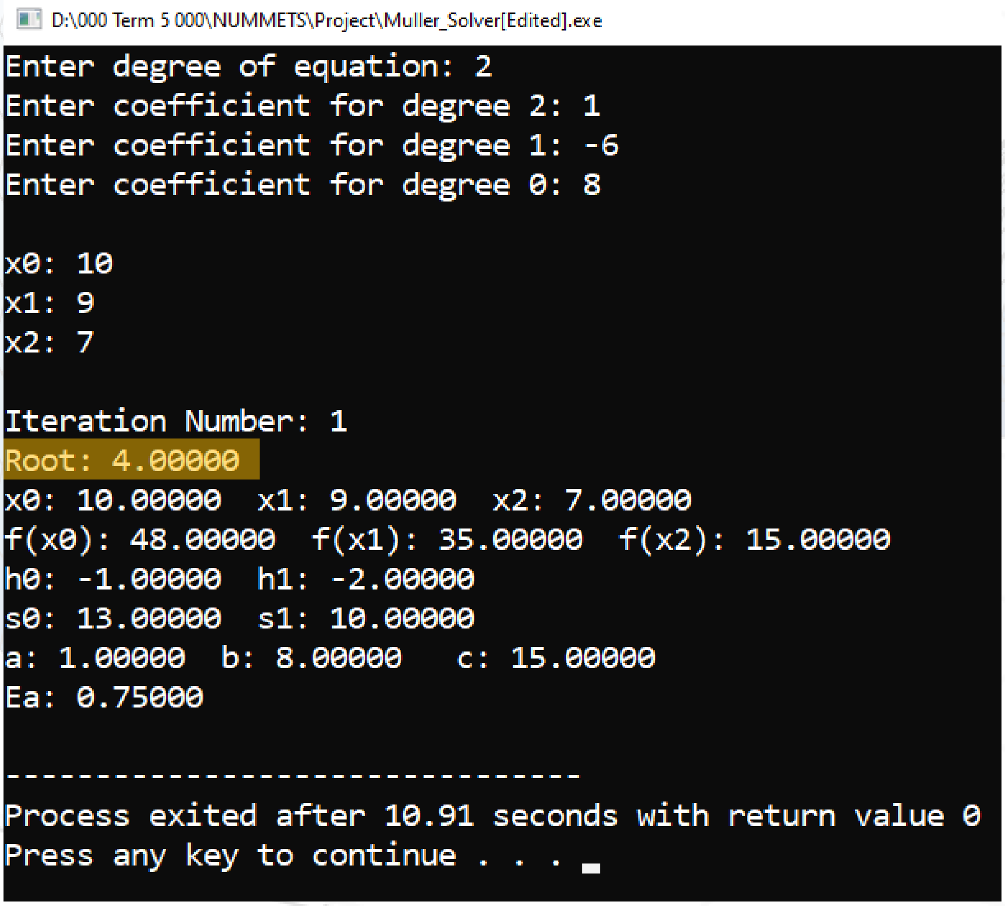
D:\000 Term 5 000\NUMMETS\Project\Muller_Solver[Edited].exe

```
Enter degree of equation: 2
Enter coefficient for degree 2: 1
Enter coefficient for degree 1: -6
Enter coefficient for degree 0: 8

x0: 10
x1: 9
x2: 7

Iteration Number: 1
Root: 4.00000
x0: 10.00000  x1: 9.00000  x2: 7.00000
f(x0): 48.00000  f(x1): 35.00000  f(x2): 15.00000
h0: -1.00000  h1: -2.00000
s0: 13.00000  s1: 10.00000
a: 1.00000  b: 8.00000   c: 15.00000
Ea: 0.75000


--------------------------------
Process exited after 10.91 seconds with return value 0
Press any key to continue . . .
```

Müller Sample Computations

Sample Computation 1. (continued)

Iter	x	f(x)	h0	h1	s0	s1	a	b	c	Ea
-2	10	48								
-1	9	35								
0	7	15	-1	-2	13	10	1	8	15	0.75
1	4	0								

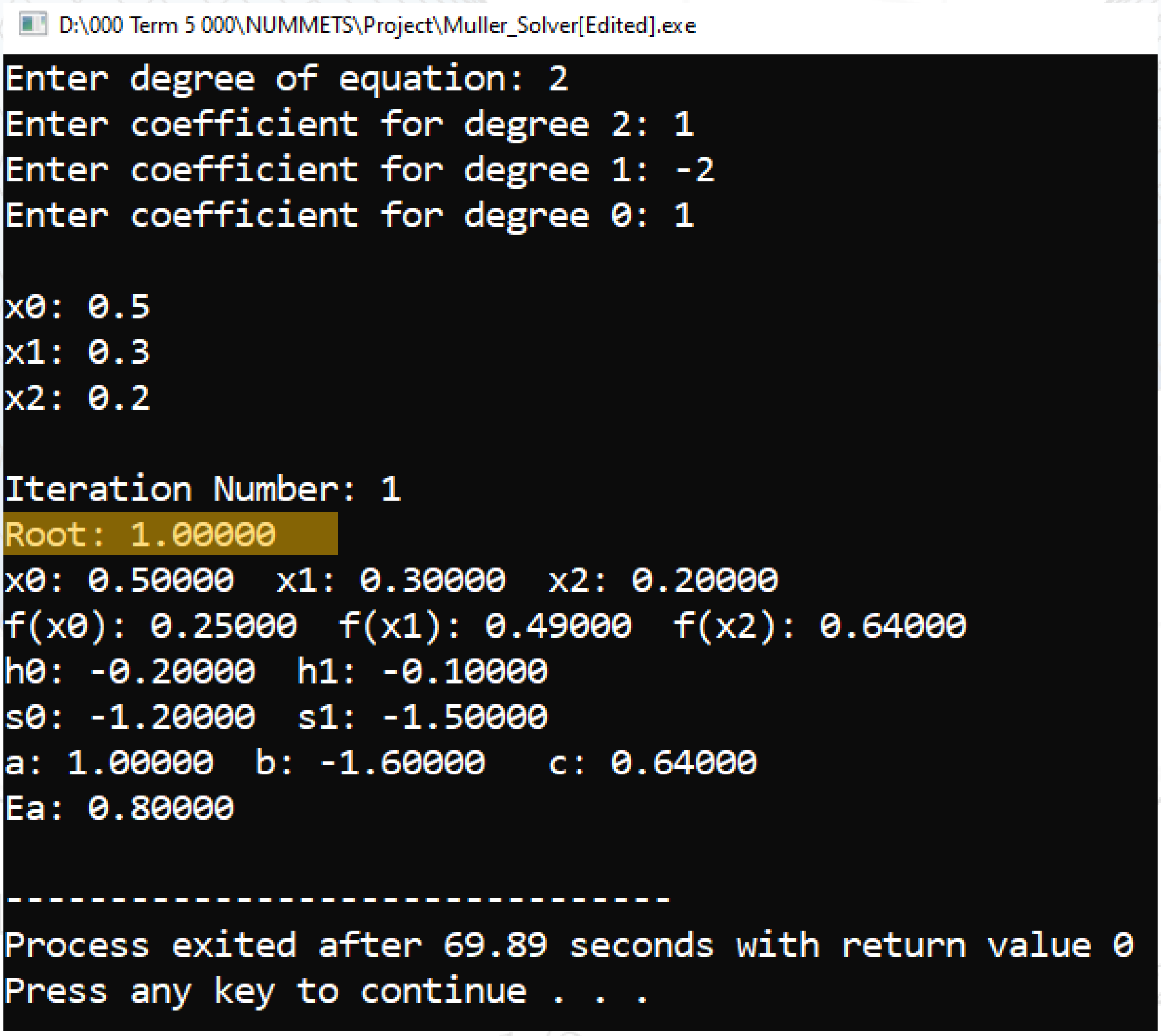

```
D:\000 Term 5 000\NUMMETS\Project\Muller_Solver[Edited].exe
Enter degree of equation: 2
Enter coefficient for degree 2: 1
Enter coefficient for degree 1: -6
Enter coefficient for degree 0: 8

x0: 10
x1: 9
x2: 7

Iteration Number: 1
Root: 4.00000
x0: 10.00000  x1: 9.00000  x2: 7.00000
f(x0): 48.00000  f(x1): 35.00000  f(x2): 15.00000
h0: -1.00000  h1: -2.00000
s0: 13.00000  s1: 10.00000
a: 1.00000  b: 8.00000   c: 15.00000
Ea: 0.75000


--------------------------------
Process exited after 10.91 seconds with return value 0
Press any key to continue . . .
```

Since $f(4) = 0$, is less than the stopping criteria $f(x) = 0.0001$, the algorithm stopped. When computed, the roots of $f(x) = x^2 - 6x + 8$ are 4 and 2. With that, we can conclude that we have found the correct root.

Müller Sample Computations

Sample Computation 2.

Given	Stopping Criteria
f(x) = $x^2 - 2x + 1$	iterations = 100
x_0 = 0.5	error = 0.00001
x_1 = 0.3	$f(x)$ = 0.0001
x_2 = 0.2	



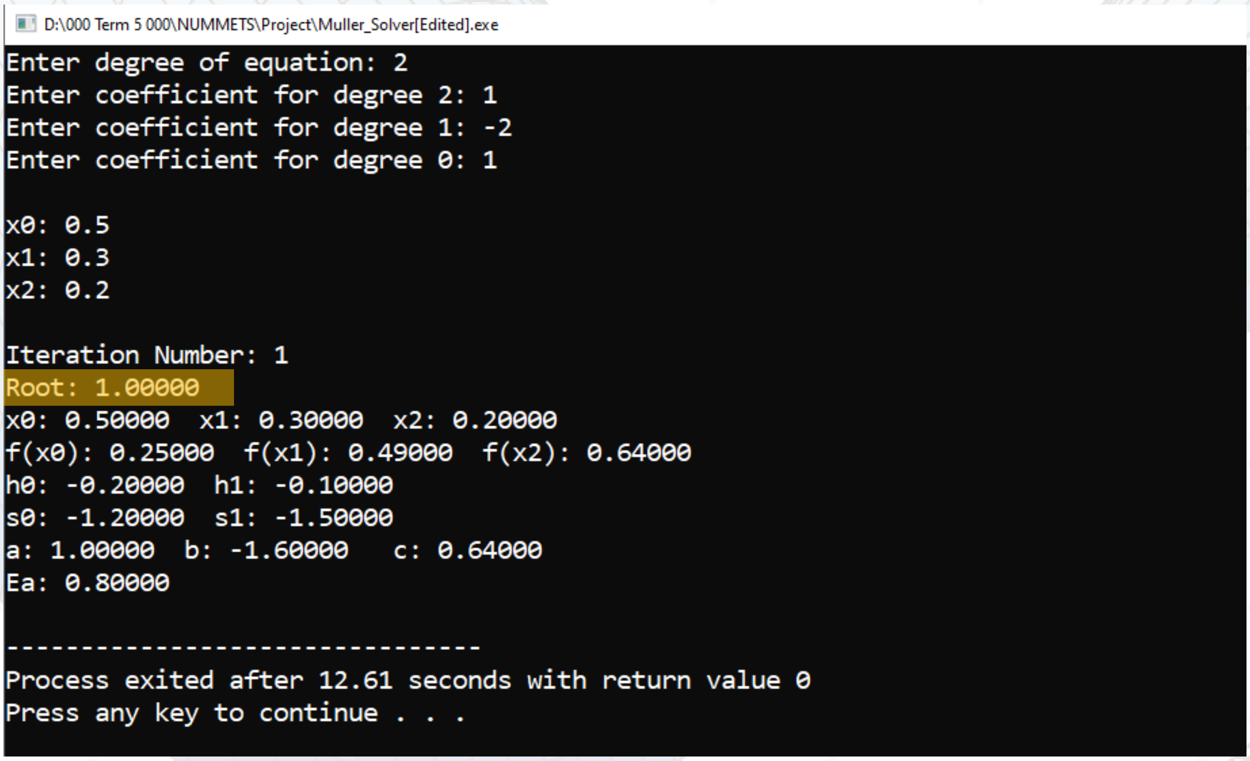
D:\000 Term 5 000\NUMMETS\Project\Muller_Solver[Edited].exe

```
Enter degree of equation: 2
Enter coefficient for degree 2: 1
Enter coefficient for degree 1: -2
Enter coefficient for degree 0: 1

x0: 0.5
x1: 0.3
x2: 0.2

Iteration Number: 1
Root: 1.00000
x0: 0.50000  x1: 0.30000  x2: 0.20000
f(x0): 0.25000  f(x1): 0.49000  f(x2): 0.64000
h0: -0.20000  h1: -0.10000
s0: -1.20000  s1: -1.50000
a: 1.00000  b: -1.60000   c: 0.64000
Ea: 0.80000


--------------------------------
Process exited after 69.89 seconds with return value 0
Press any key to continue . . .
```

Müller Sample Computations

Sample Computation 2. (continued)

Iter	x	f(x)	h0	h1	s0	s1	a	b	c	Ea
-2	0.5	2.25								
-1	0.3	1.69								
0	0.2	1.44	-0.2	-0.1	2.8	2.5	1	2.4	1.44	1.2
1	1	0								


D:\000 Term 5 000\NUMMETS\Project\Muller_Solver[Edited].exe

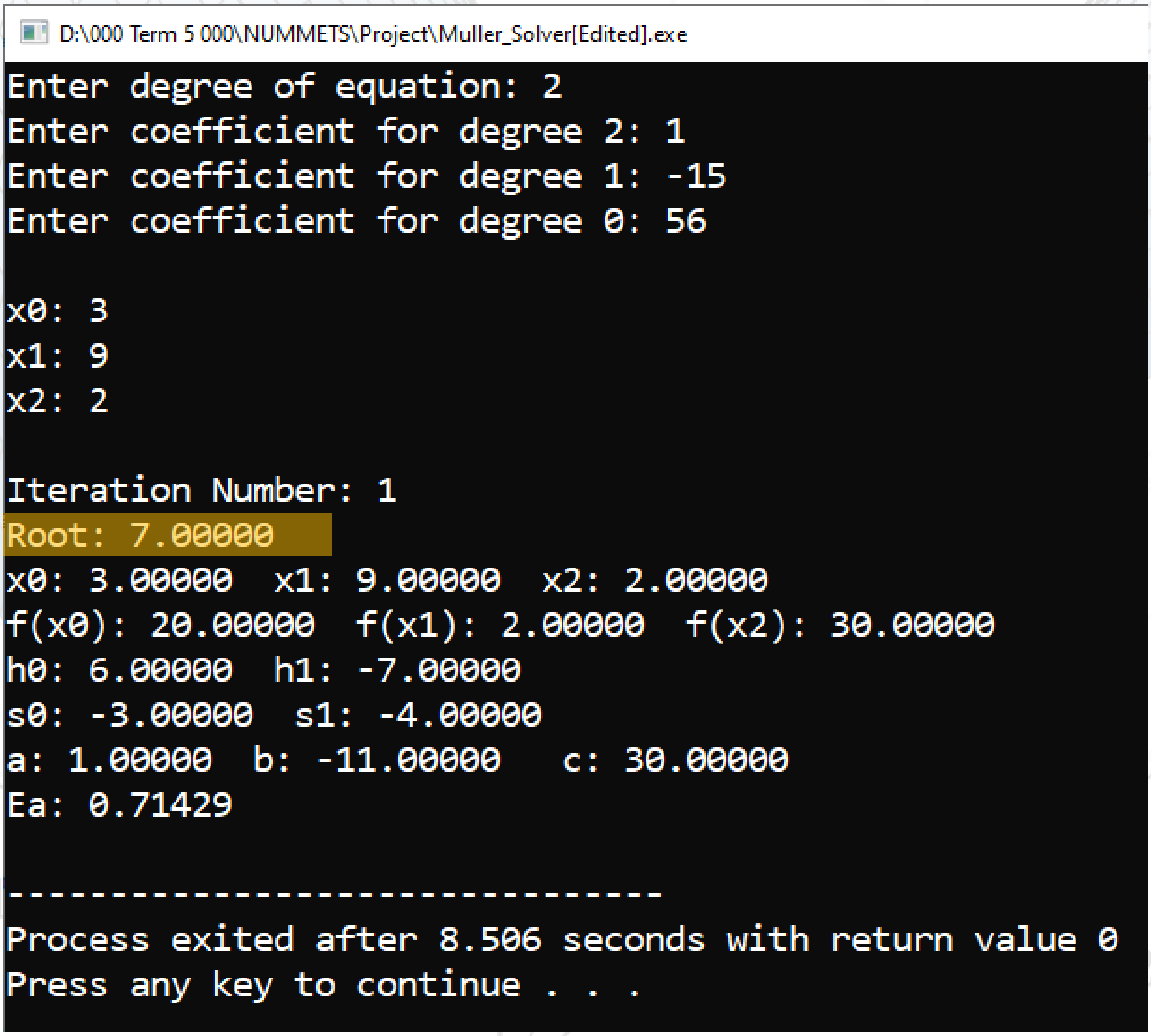
```
Enter degree of equation: 2
Enter coefficient for degree 2: 1
Enter coefficient for degree 1: -2
Enter coefficient for degree 0: 1

x0: 0.5
x1: 0.3
x2: 0.2

Iteration Number: 1
Root: 1.00000
x0: 0.50000  x1: 0.30000  x2: 0.20000
f(x0): 0.25000  f(x1): 0.49000  f(x2): 0.64000
h0: -0.20000  h1: -0.10000
s0: -1.20000  s1: -1.50000
a: 1.00000  b: -1.60000   c: 0.64000
Ea: 0.80000

--------------------------------
Process exited after 12.61 seconds with return value 0
Press any key to continue . . .
```

Since $f(1) = 0$, is less than the stopping criteria $f(x) = 0.0001$, the algorithm stopped. Since $f(x) = x^2 - 2x + 1$ is a perfect square binomial, it has only one root $x = 1$. With the result, we can conclude that we have found the correct root.

Müller Sample Computations

Sample Computation 3.

Given	**Stopping Criteria**
f(x) = $x^2 - 15x + 56$	iterations = 100
x_0 = 3	error = 0.00001
x_1 = 9	$f(x)$ = 0.0001
x_2 = 2	

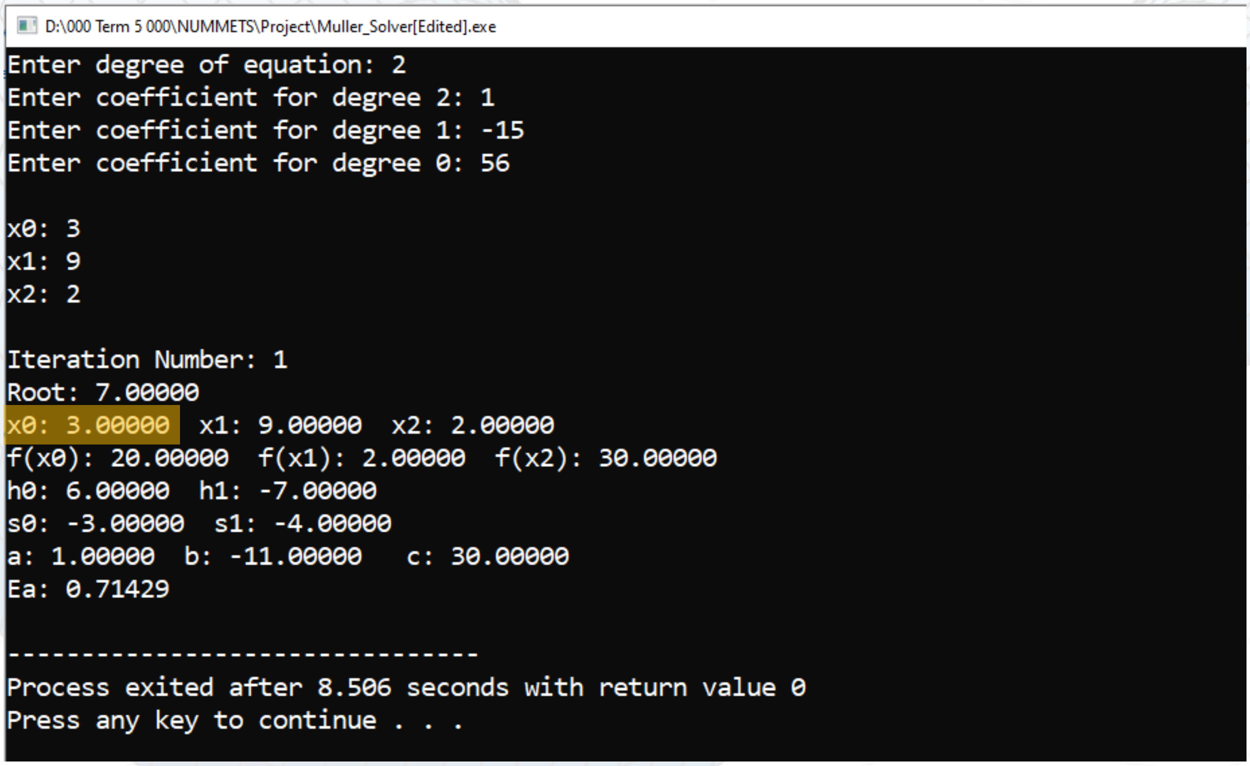

D:\000 Term 5 000\NUMMETS\Project\Muller_Solver[Edited].exe

```
Enter degree of equation: 2
Enter coefficient for degree 2: 1
Enter coefficient for degree 1: -15
Enter coefficient for degree 0: 56

x0: 3
x1: 9
x2: 2

Iteration Number: 1
Root: 7.00000
x0: 3.00000  x1: 9.00000  x2: 2.00000
f(x0): 20.00000  f(x1): 2.00000  f(x2): 30.00000
h0: 6.00000  h1: -7.00000
s0: -3.00000  s1: -4.00000
a: 1.00000  b: -11.00000   c: 30.00000
Ea: 0.71429


--------------------------------
Process exited after 8.506 seconds with return value 0
Press any key to continue . . .
```

Müller Sample Computations

Sample Computation 3. (continued)

Iter	x	f(x)	h0	h1	s0	s1	a	b	c	Ea
-2	3	20								
-1	9	2								
0	2	30	6	-7	-3	-4	1	-11	30	0.71429
1	7	0								



D:\000 Term 5 000\NUMMETS\Project\Muller_Solver[Edited].exe

```
Enter degree of equation: 2
Enter coefficient for degree 2: 1
Enter coefficient for degree 1: -15
Enter coefficient for degree 0: 56

x0: 3
x1: 9
x2: 2

Iteration Number: 1
Root: 7.00000
x0: 3.00000  x1: 9.00000  x2: 2.00000
f(x0): 20.00000  f(x1): 2.00000  f(x2): 30.00000
h0: 6.00000  h1: -7.00000
s0: -3.00000  s1: -4.00000
a: 1.00000  b: -11.00000   c: 30.00000
Ea: 0.71429

--------------------------------
Process exited after 8.506 seconds with return value 0
Press any key to continue . . .
```

Since $f(7) = 0$, is less than the stopping criteria $f(x) = 0.0001$, the algorithm stopped. Since the factor of $f(x) = x^2 - 15x + 56$ is (x-7)(x-8), x = 7 and x = 8. With the result, we can conclude that we have found the correct root.

Naïve Gaussian Method Solver

by **BAYETA IV, Reginald G.L.**

The **Naïve Gaussian Elimination Method** is a method used to solve simultaneous linear equation in the form [A][X] = [C] wherein [A] and [C] are given, and [X] are the unknowns.

There are two steps in the Naïve Gaussian method.

1. **Forward Elimination**

   the goal of forward elimination is to transform the coefficient matrix into upper triangular matrix such that the lower triangular matrix has values of **zeroes** as shown below:

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} & \cdots & a_{1n} \\ 0 & a'_{22} & a'_{23} & \cdots & a'_{2n} \\ 0 & 0 & a''_{33} & \cdots & a''_{3n} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & 0 & a_{nn}^{(n-1)} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ \vdots \\ x_n \end{bmatrix} = \begin{bmatrix} b_1 \\ b'_2 \\ b''_3 \\ \vdots \\ b_n^{(n-1)} \end{bmatrix}$$

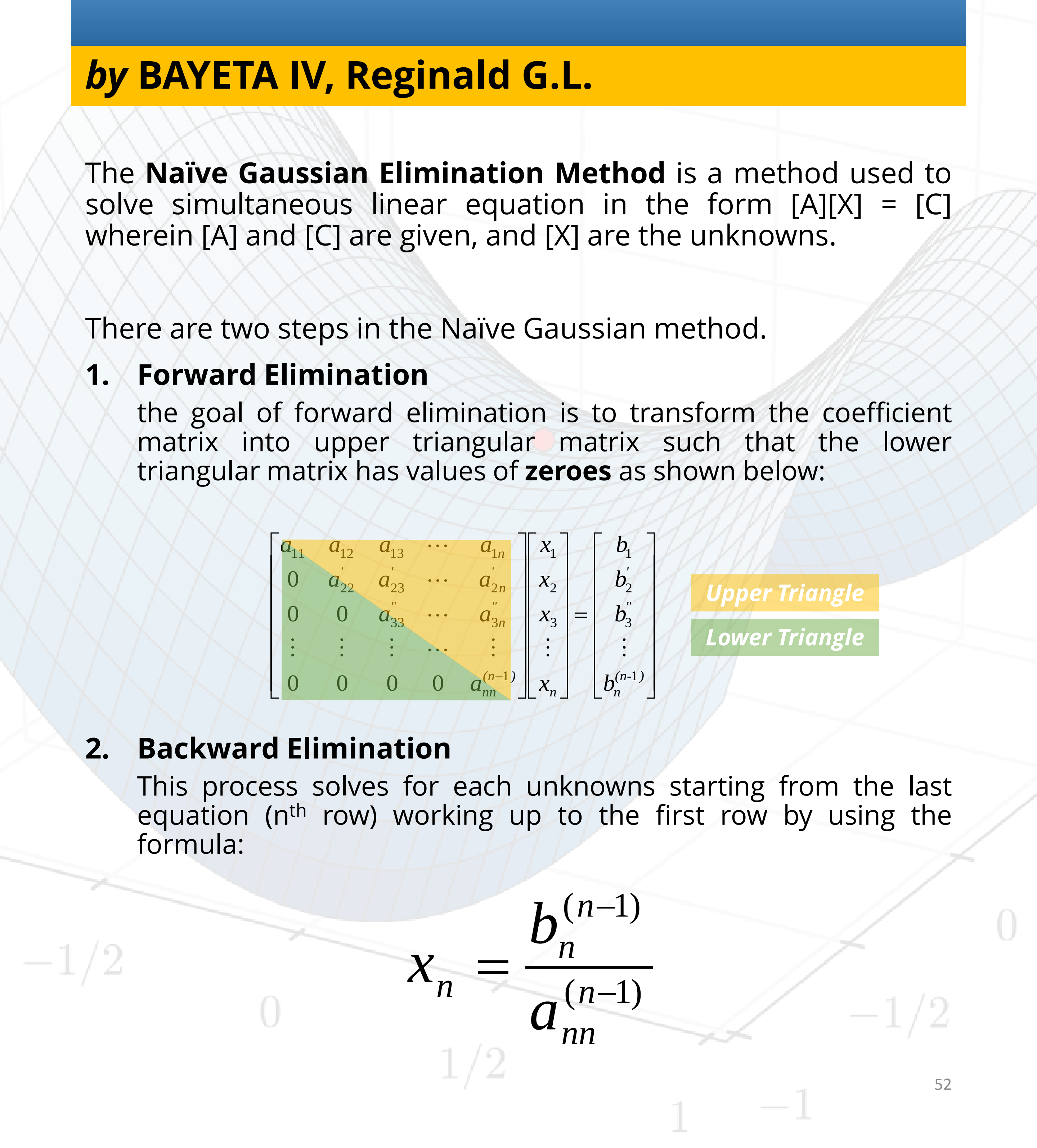
🟨 *Upper Triangle*

🟩 *Lower Triangle*

2. **Backward Elimination**

   This process solves for each unknowns starting from the last equation (nth row) working up to the first row by using the formula:

$$x_n = \frac{b_n^{(n-1)}}{a_{nn}^{(n-1)}}$$



Naïve Gaussian Method Algorithm

$$\underset{[A]}{\begin{bmatrix} a_{11} & a_{12} & a_{13} & \cdots & a_{1n} \\ 0 & a'_{22} & a'_{23} & \cdots & a'_{2n} \\ 0 & 0 & a''_{33} & \cdots & a''_{3n} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & 0 & a_{nn}^{(n-1)} \end{bmatrix}} \underset{[X]}{\begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ \vdots \\ x_n \end{bmatrix}} \underset{=}{=} \underset{[C]}{\begin{bmatrix} b_1 \\ b'_2 \\ b''_3 \\ \vdots \\ b_n^{(n-1)} \end{bmatrix}}$$

- 🟨 Upper Triangle
- 🟩 Lower Triangle
- ⬛ Diagonal

The matrix [U] shown above follows the indexing system [(i,j) | i={1,2,3,...,n} for row, j={1,2,3,...,n}] for column. The lower triangle matrix that we want to turn into zeroes have indices such that i < j. Each row has each unique equations [A][X] = [C] (hereinafter referred to as simply equation)

```
1.  For i ← 1 to n do
2.  For j ← 1 to n do
3.  If i < j do          ○ Check if current matrix value is part of the lower triangle
4.  multiplier = $a_{i,j}$ / $\boldsymbol{a_{j,j}}$      ○ $\boldsymbol{a_{j,j}}$ is part of the diagonal of the matrix [U]
5.  equationMultiplied = $\underbrace{\left[\frac{a_{i,j}}{a_{j,j}}\right]}_{multiplier}\underbrace{(a_i x_j + a_i x_{j+1} + \cdots a_i x_{j=n} = b_i)}_{equation_j}$
6.  newUpdatedEquation = $equation_i - equationMultiplied$
        Example newUpdatedEquation at i = 2
```

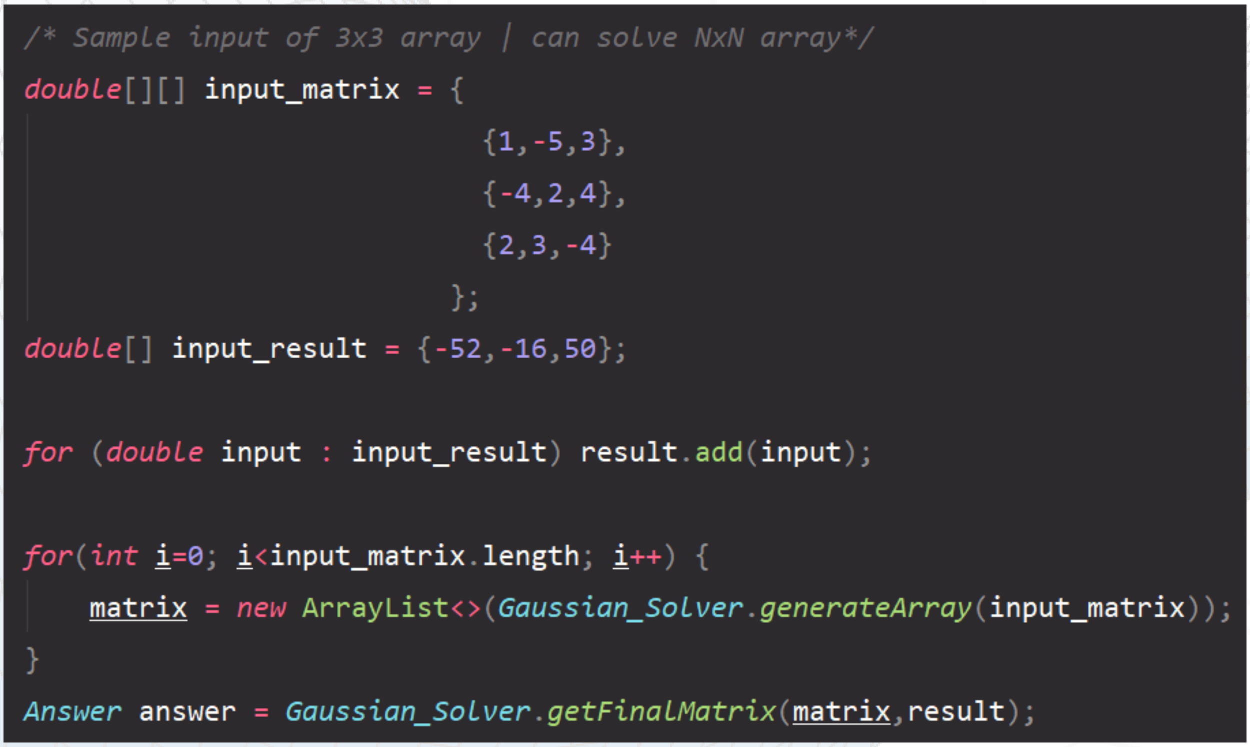
$$\begin{array}{rl} & a_{21}x_1 + a_{22}x_2 + a_{23}x_3 + \ldots + a_{2n}x_n = b_2 \\ - & a_{21}x_1 + \frac{a_{21}}{a_{11}}a_{12}x_2 + \ldots + \frac{a_{21}}{a_{11}}a_{1n}x_n = \frac{a_{21}}{a_{11}}b_1 \\ \hline & \left(a_{22} - \frac{a_{21}}{a_{11}}a_{12}\right)x_2 + \ldots + \left(a_{2n} - \frac{a_{21}}{a_{11}}a_{1n}\right)x_n = b_2 - \frac{a_{21}}{a_{11}}b_1 \\ & \\ \text{or} & a'_{22}x_2 + \ldots + a'_{2n}x_n = b'_2 \end{array}$$

```
7.  End for at 2, continue at 1.
8.  End for at 1, end forward elimination.
9.  Do backward elimination starting from the last equation towards the first row.
```

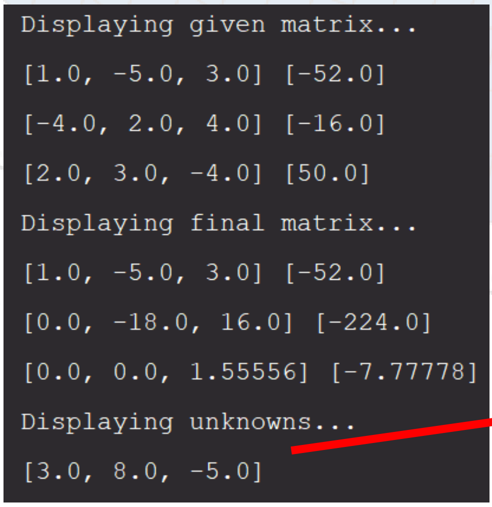
Naïve Gaussian Sample Computations

The ***Gaussian Solver*** is written in Java programming language. The program accepts 2 inputs from the user: the matrix [A] represented by a 2D array data structure, and matrix [C] represented by an ArrayList data structure.

Sample Computation 1.



```java
/* Sample input of 3x3 array | can solve NxN array*/
double[][] input_matrix = {
                            {1,-5,3},
                            {-4,2,4},
                            {2,3,-4}
                          };
double[] input_result = {-52,-16,50};

for (double input : input_result) result.add(input);

for(int i=0; i<input_matrix.length; i++) {
    matrix = new ArrayList<>(Gaussian_Solver.generateArray(input_matrix));
}
Answer answer = Gaussian_Solver.getFinalMatrix(matrix,result);
```



```
Displaying given matrix...
[1.0, -5.0, 3.0] [-52.0]
[-4.0, 2.0, 4.0] [-16.0]
[2.0, 3.0, -4.0] [50.0]
Displaying final matrix...
[1.0, -5.0, 3.0] [-52.0]
[0.0, -18.0, 16.0] [-224.0]
[0.0, 0.0, 1.55556] [-7.77778]
Displaying unknowns...
[3.0, 8.0, -5.0]
```

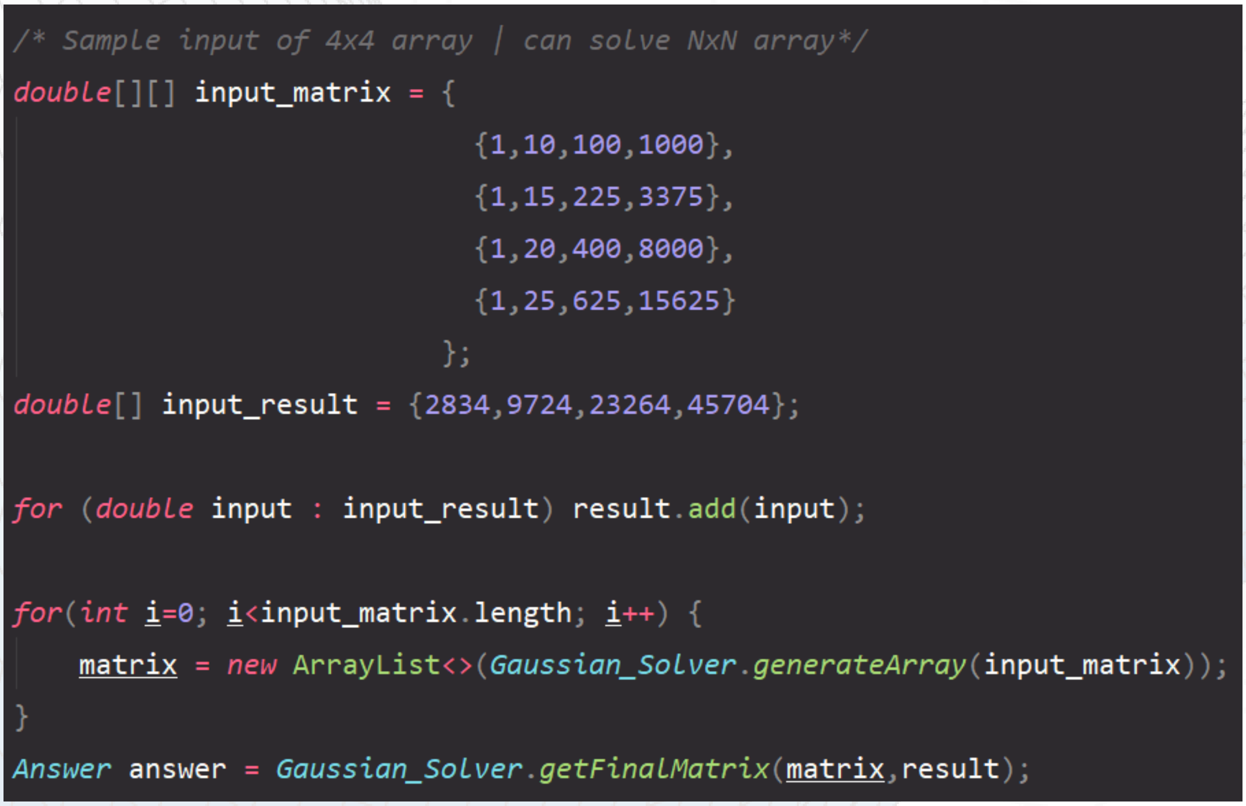
Given:
$f_1(x) = 1 - 5x + 3x^2 = -52$
$f_2(x) = -4 + 2x + 4x^2 = -16$
$f_3(x) = 2 + 3x - 4x^2 = 50$

Values of the unknowns are
$x_1 = 3$, $x_2 = 8$, $x_3 = -5$.

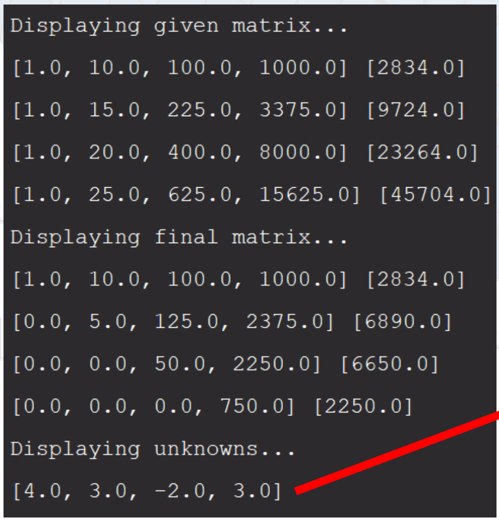
Naïve Gaussian Sample Computations

The ***Gaussian Solver*** accepts matrix size of ***n*** x ***n*** size. Sample computation 2 shows an example of solving a matrix of 4x4 size.

Sample Computation 2.



```java
/* Sample input of 4x4 array | can solve NxN array*/
double[][] input_matrix = {
                            {1,10,100,1000},
                            {1,15,225,3375},
                            {1,20,400,8000},
                            {1,25,625,15625}
                          };
double[] input_result = {2834,9724,23264,45704};

for (double input : input_result) result.add(input);

for(int i=0; i<input_matrix.length; i++) {
    matrix = new ArrayList<>(Gaussian_Solver.generateArray(input_matrix));
}
Answer answer = Gaussian_Solver.getFinalMatrix(matrix,result);
```



```
Displaying given matrix...
[1.0, 10.0, 100.0, 1000.0] [2834.0]
[1.0, 15.0, 225.0, 3375.0] [9724.0]
[1.0, 20.0, 400.0, 8000.0] [23264.0]
[1.0, 25.0, 625.0, 15625.0] [45704.0]
Displaying final matrix...
[1.0, 10.0, 100.0, 1000.0] [2834.0]
[0.0, 5.0, 125.0, 2375.0] [6890.0]
[0.0, 0.0, 50.0, 2250.0] [6650.0]
[0.0, 0.0, 0.0, 750.0] [2250.0]
Displaying unknowns...
[4.0, 3.0, -2.0, 3.0]
```

Given:

f_1(x) = $1 + 10x + 100x^2 + 1000x^3 = 2834$

f_2(x) = $1 + 15x + 225x^2 + 3375x^3 = 9724$

f_3(x) = $1 + 20x + 400x^2 + 8000x^3 = 23264$

f_4(x) = $1 + 25x + 625x^2 + 15625x^3 = 45704$

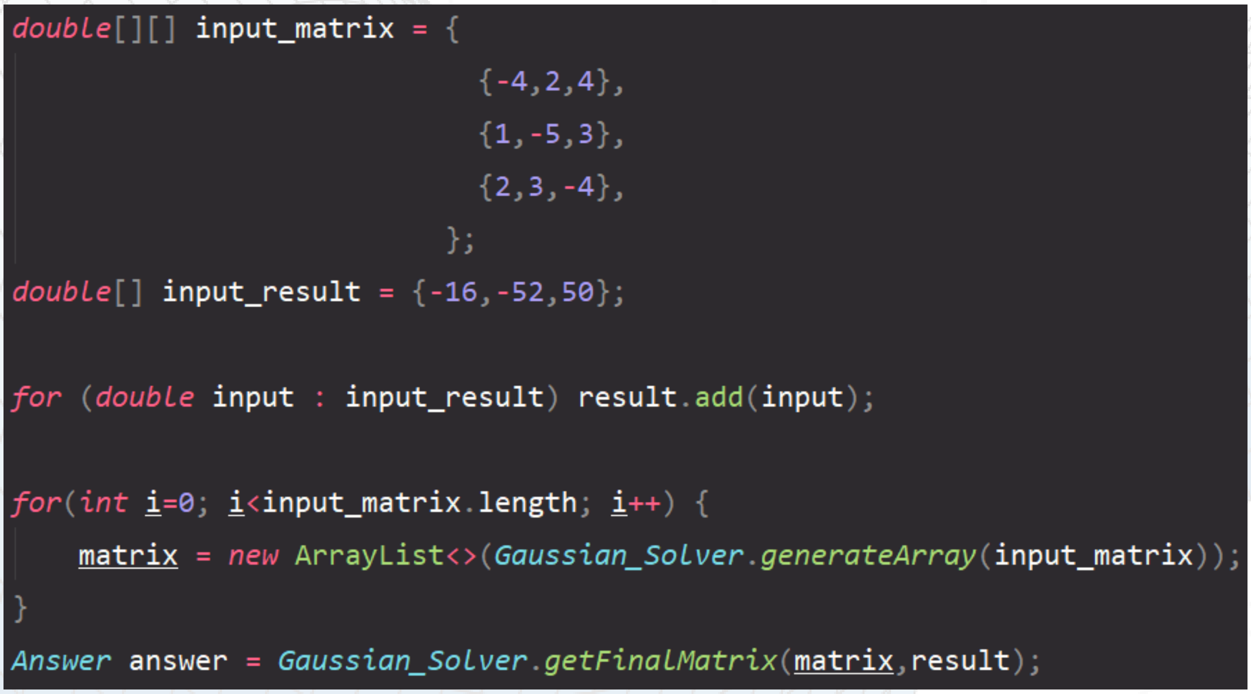
Values of the unknowns are

$x_1 = 4$, $x_2 = 3,\ x_3 = -2,\ x_4 = 3$

Naïve Gaussian Sample Computations

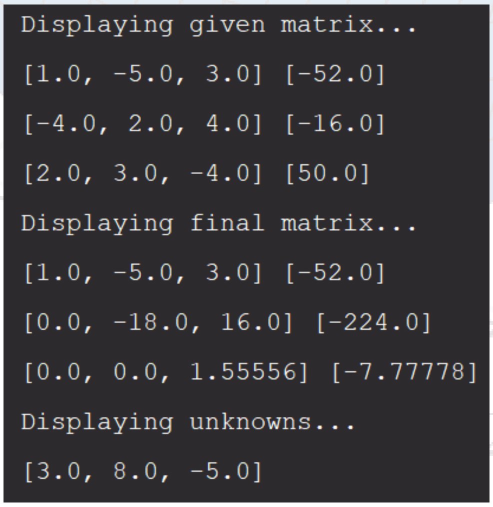
The algorithm is referred to as ***naïve*** since no modification is done to the given input matrix in order to avoid arriving at pitfall results. In the program created, it is the responsibility of the user to do the modification (pivoting) by themselves in order to get a more accurate result. Sample computation 3 shows the result of when the input matrix in sample computation 1 is pivoted.

Sample Computation 3 (with pivoting)



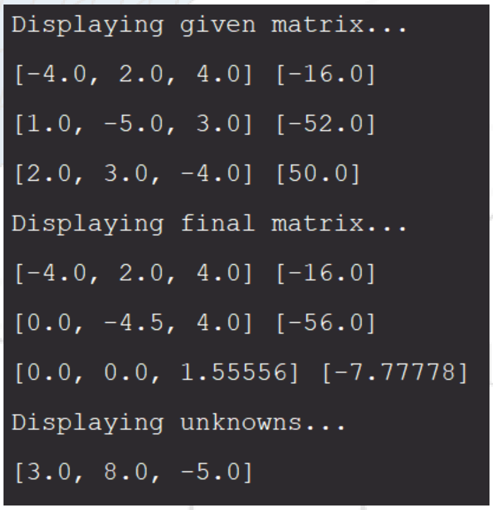
```java
double[][] input_matrix = {
                            {-4,2,4},
                            {1,-5,3},
                            {2,3,-4},
                          };
double[] input_result = {-16,-52,50};

for (double input : input_result) result.add(input);

for(int i=0; i<input_matrix.length; i++) {
    matrix = new ArrayList<>(Gaussian_Solver.generateArray(input_matrix));
}
Answer answer = Gaussian_Solver.getFinalMatrix(matrix,result);
```

Without partial pivoting



```
Displaying given matrix...
[1.0, -5.0, 3.0] [-52.0]
[-4.0, 2.0, 4.0] [-16.0]
[2.0, 3.0, -4.0] [50.0]
Displaying final matrix...
[1.0, -5.0, 3.0] [-52.0]
[0.0, -18.0, 16.0] [-224.0]
[0.0, 0.0, 1.55556] [-7.77778]
Displaying unknowns...
[3.0, 8.0, -5.0]
```

With partial pivoting



```
Displaying given matrix...
[-4.0, 2.0, 4.0] [-16.0]
[1.0, -5.0, 3.0] [-52.0]
[2.0, 3.0, -4.0] [50.0]
Displaying final matrix...
[-4.0, 2.0, 4.0] [-16.0]
[0.0, -4.5, 4.0] [-56.0]
[0.0, 0.0, 1.55556] [-7.77778]
Displaying unknowns...
[3.0, 8.0, -5.0]
```

Gauss-Seidel Method Solver

by **BAYETA IV, Reginald G.L.**

The **Gauss-Seidel Method** is a method used to solve each linear equation in the matrix [A][X] = [C] wherein [A] and [C] are given, and [X] are the unknowns. It accepts an initial guess solution array so solve for the true unknowns. This method is used to control round-off errors and to increase the iteration needed by having a good prediction in the initial guess array.

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} & \cdots & a_{1n} \\ 0 & a'_{22} & a'_{23} & \cdots & a'_{2n} \\ 0 & 0 & a''_{33} & \cdots & a''_{3n} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & 0 & a^{(n-1)}_{nn} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ \vdots \\ x_n \end{bmatrix} = \begin{bmatrix} b_1 \\ b'_2 \\ b''_3 \\ \vdots \\ b^{(n-1)}_n \end{bmatrix}$$

Diagonal values

Each row represent a linear equation
$[A_i][X_i] = [C_i]$
(i = row number)

In order to the algorithm to work, we should keep in mind the following:

- The system of linear equations should be diagonally dominant, otherwise the result will not converge.
- The diagonal values should be greater than 0.

Gauss-Seidel Method Algorithm

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} & \cdots & a_{1n} \\ 0 & a'_{22} & a'_{23} & \cdots & a'_{2n} \\ 0 & 0 & a''_{33} & \cdots & a''_{3n} \\ \vdots & \vdots & \vdots & \cdots & \vdots \\ 0 & 0 & 0 & 0 & a^{(n-1)}_{nn} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ \vdots \\ x_n \end{bmatrix} = \begin{bmatrix} b_1 \\ b'_2 \\ b''_3 \\ \vdots \\ b^{(n-1)}_n \end{bmatrix}$$

[A] [X] = [C]

Diagonal values

Each row represent a linear equation $[A_i][X_i] = [C_i]$ (i = row number)

For each row (linear equation), we solve for the unknown by using the formula:

$$x_i = \frac{c_i - \sum_{\substack{j=1 \\ j \neq i}}^{n} a_{ij} x_j}{a_{ii}}, i = 1,2,\ldots,n.$$

For each iteration, we have to use the most recent value of $\boldsymbol{x_i}$.

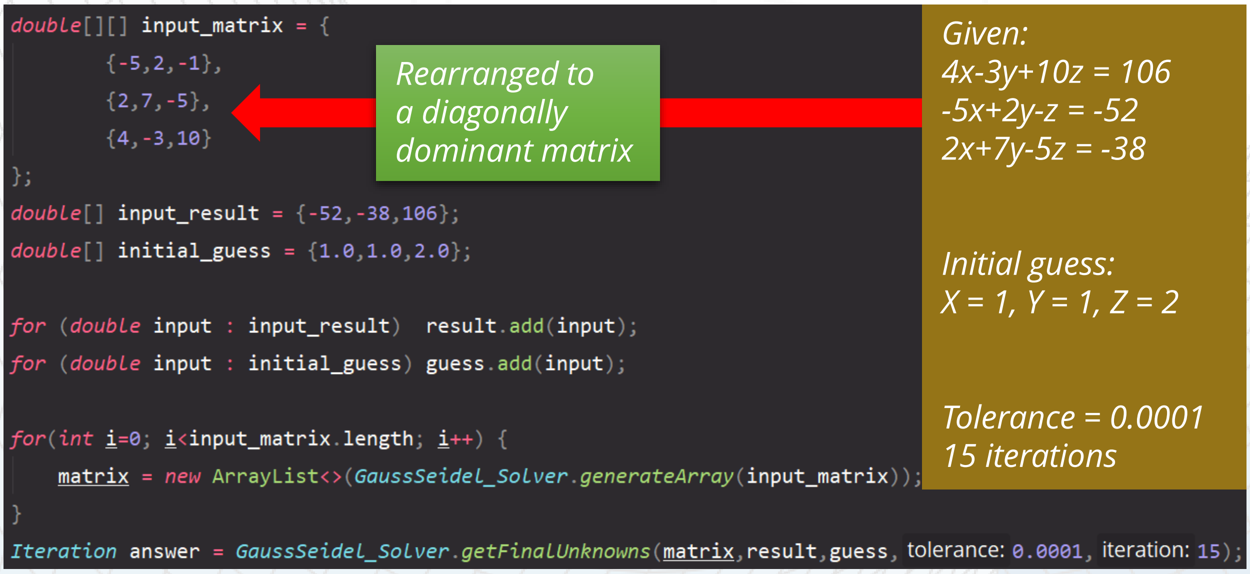
After each iteration we have to calculate the absolute relative approximate error $\boldsymbol{\epsilon_a}$ by using the formula:

$$\left|\in_a\right|_i = \left|\frac{x_i^{new} - x_i^{old}}{x_i^{new}}\right| \times 100$$

The algorithm will be stopped if and only if and only if $\boldsymbol{\epsilon_a}$ for ***all*** calculated unknowns are ***less than*** the pre-specified error tolerance $\boldsymbol{\epsilon_s}$.

Gauss-Seidel Sample Computations

Sample Computation 1

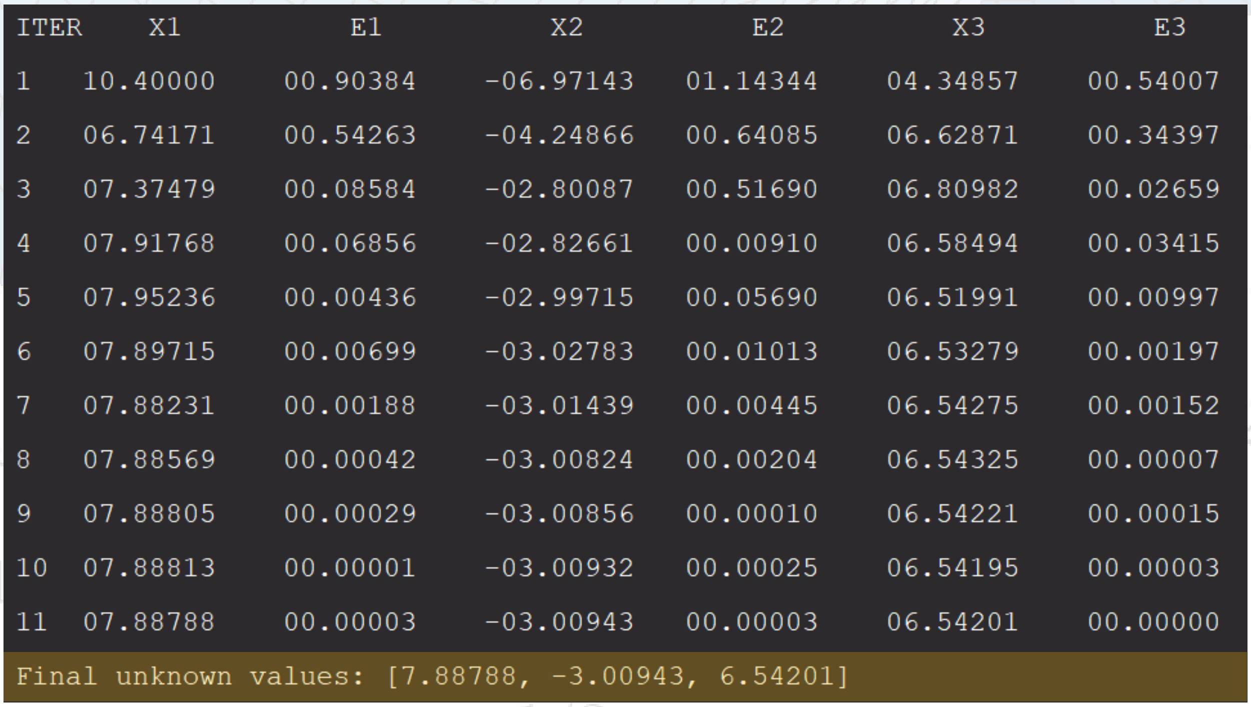


```
double[][] input_matrix = {
        {-5,2,-1},
        {2,7,-5},
        {4,-3,10}
};
double[] input_result = {-52,-38,106};
double[] initial_guess = {1.0,1.0,2.0};

for (double input : input_result)  result.add(input);
for (double input : initial_guess) guess.add(input);

for(int i=0; i<input_matrix.length; i++) {
    matrix = new ArrayList<>(GaussSeidel_Solver.generateArray(input_matrix));
}
Iteration answer = GaussSeidel_Solver.getFinalUnknowns(matrix,result,guess, tolerance: 0.0001, iteration: 15);
```

Given:
4x-3y+10z = 106
-5x+2y-z = -52
2x+7y-5z = -38

Initial guess:
X = 1, Y = 1, Z = 2

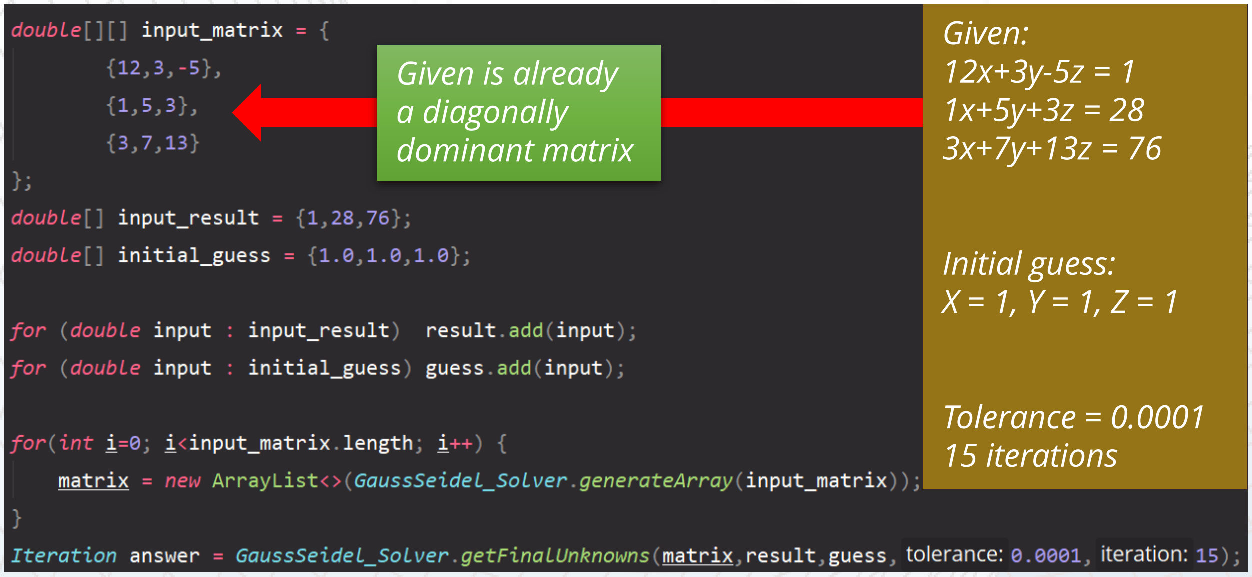
Tolerance = 0.0001
15 iterations



```
ITER    X1          E1          X2          E2          X3          E3
1    10.40000    00.90384    -06.97143    01.14344    04.34857    00.54007
2    06.74171    00.54263    -04.24866    00.64085    06.62871    00.34397
3    07.37479    00.08584    -02.80087    00.51690    06.80982    00.02659
4    07.91768    00.06856    -02.82661    00.00910    06.58494    00.03415
5    07.95236    00.00436    -02.99715    00.05690    06.51991    00.00997
6    07.89715    00.00699    -03.02783    00.01013    06.53279    00.00197
7    07.88231    00.00188    -03.01439    00.00445    06.54275    00.00152
8    07.88569    00.00042    -03.00824    00.00204    06.54325    00.00007
9    07.88805    00.00029    -03.00856    00.00010    06.54221    00.00015
10   07.88813    00.00001    -03.00932    00.00025    06.54195    00.00003
11   07.88788    00.00003    -03.00943    00.00003    06.54201    00.00000
Final unknown values: [7.88788, -3.00943, 6.54201]
```

Gauss-Seidel Sample Computations

Sample Computation 2



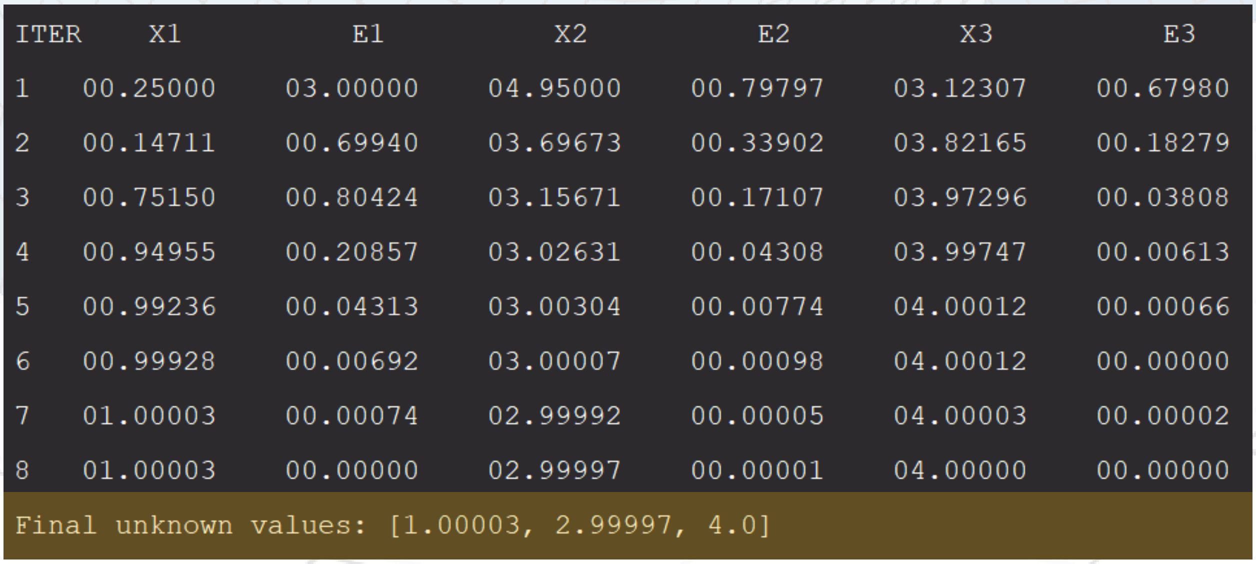
```
double[][] input_matrix = {
        {12,3,-5},
        {1,5,3},
        {3,7,13}
};
double[] input_result = {1,28,76};
double[] initial_guess = {1.0,1.0,1.0};

for (double input : input_result)  result.add(input);
for (double input : initial_guess) guess.add(input);

for(int i=0; i<input_matrix.length; i++) {
    matrix = new ArrayList<>(GaussSeidel_Solver.generateArray(input_matrix));
}
Iteration answer = GaussSeidel_Solver.getFinalUnknowns(matrix,result,guess, tolerance: 0.0001, iteration: 15);
```

Given is already a diagonally dominant matrix

Given:
12x+3y-5z = 1
1x+5y+3z = 28
3x+7y+13z = 76

Initial guess:
X = 1, Y = 1, Z = 1

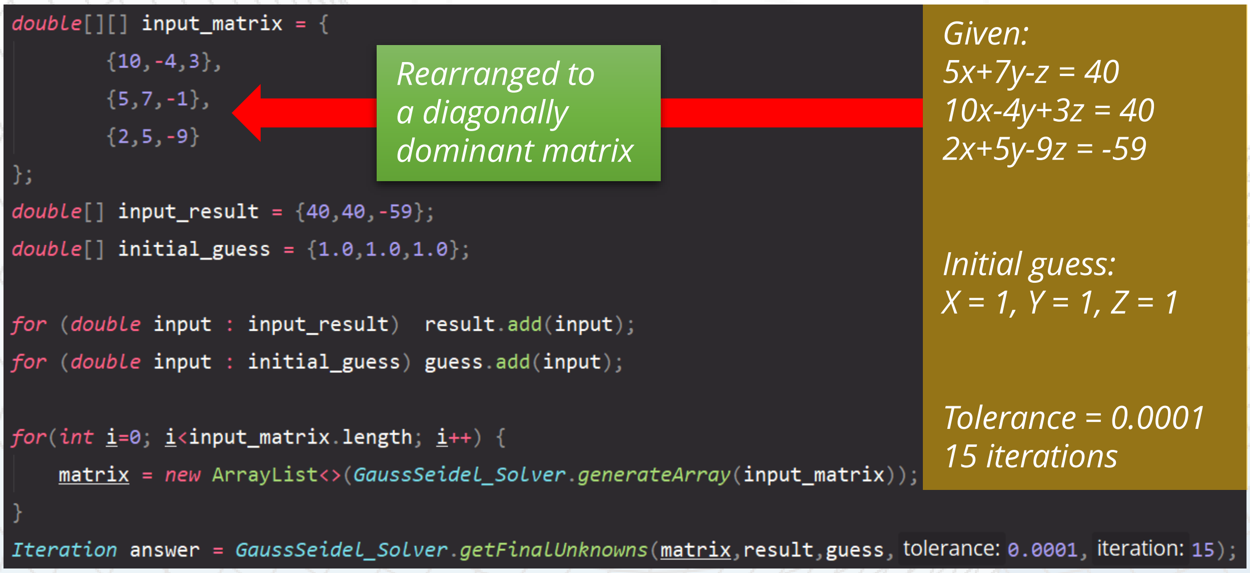
Tolerance = 0.0001
15 iterations



```
ITER    X1          E1          X2          E2          X3          E3
1     00.25000    03.00000    04.95000    00.79797    03.12307    00.67980
2     00.14711    00.69940    03.69673    00.33902    03.82165    00.18279
3     00.75150    00.80424    03.15671    00.17107    03.97296    00.03808
4     00.94955    00.20857    03.02631    00.04308    03.99747    00.00613
5     00.99236    00.04313    03.00304    00.00774    04.00012    00.00066
6     00.99928    00.00692    03.00007    00.00098    04.00012    00.00000
7     01.00003    00.00074    02.99992    00.00005    04.00003    00.00002
8     01.00003    00.00000    02.99997    00.00001    04.00000    00.00000
Final unknown values: [1.00003, 2.99997, 4.0]
```

Gauss-Seidel Sample Computations

Sample Computation 3



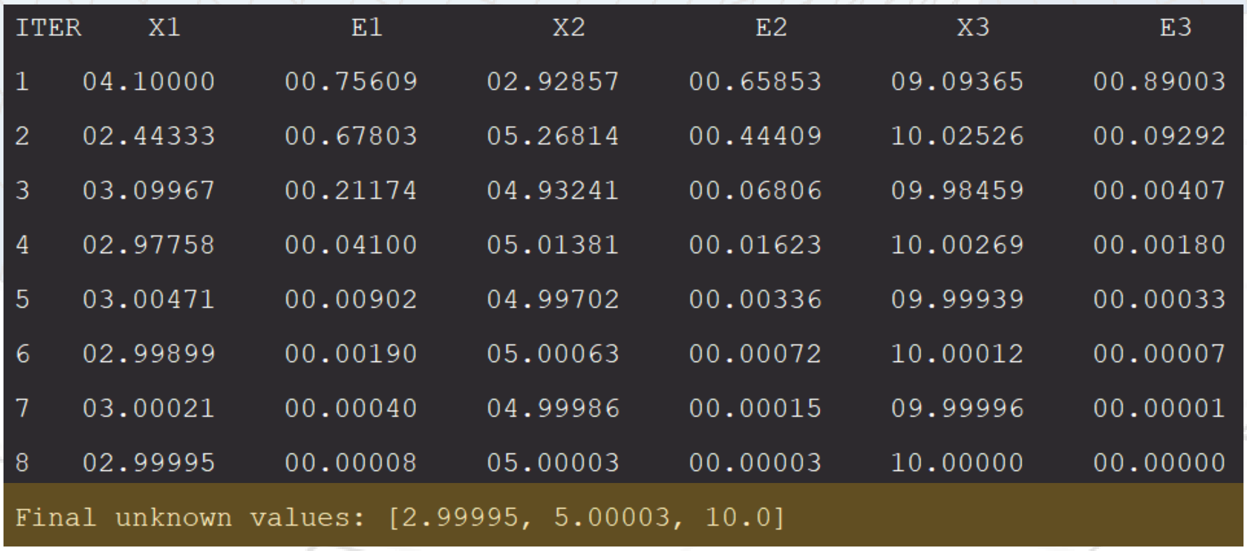
```
double[][] input_matrix = {
        {10,-4,3},
        {5,7,-1},
        {2,5,-9}
};
double[] input_result = {40,40,-59};
double[] initial_guess = {1.0,1.0,1.0};

for (double input : input_result)  result.add(input);
for (double input : initial_guess) guess.add(input);

for(int i=0; i<input_matrix.length; i++) {
    matrix = new ArrayList<>(GaussSeidel_Solver.generateArray(input_matrix));
}
Iteration answer = GaussSeidel_Solver.getFinalUnknowns(matrix,result,guess, tolerance: 0.0001, iteration: 15);
```

Rearranged to a diagonally dominant matrix

Given:
5x+7y-z = 40
10x-4y+3z = 40
2x+5y-9z = -59

Initial guess:
X = 1, Y = 1, Z = 1

Tolerance = 0.0001
15 iterations



```
ITER   X1          E1          X2          E2          X3          E3
1    04.10000    00.75609    02.92857    00.65853    09.09365    00.89003
2    02.44333    00.67803    05.26814    00.44409    10.02526    00.09292
3    03.09967    00.21174    04.93241    00.06806    09.98459    00.00407
4    02.97758    00.04100    05.01381    00.01623    10.00269    00.00180
5    03.00471    00.00902    04.99702    00.00336    09.99939    00.00033
6    02.99899    00.00190    05.00063    00.00072    10.00012    00.00007
7    03.00021    00.00040    04.99986    00.00015    09.99996    00.00001
8    02.99995    00.00008    05.00003    00.00003    10.00000    00.00000
Final unknown values: [2.99995, 5.00003, 10.0]
```