**Implementation of the False Position (Regula Falsi) as a Computational Physics Method for the Determination of Roots of Non-Linear Equations Using Java**

Makinde, V.1, .Akinboro, F.G1., Okeyode, I.C.1, Mustapha, A.O.1., Coker, J.O.2., and Adesina, O.S.1.

1.Department of Physics, Federal University of Agriculture, Abeokuta

2.Department of Physics, Lagos State Polytechnic, Ikorodu, Lagos

victor\_makindeii@yahoo.com

**Abstract:** Computational Physics cuts across all branches of Physics, Engineering, and Sciences in general. Determination of roots is one of the most common areas/topics that show up in various disciples where Computational Physics is applied or utilized. In the computation and determination of the roots of non-linear equations, various methods such as Root Bisection method, Regula Falsi method, Newton’s method among others have been implemented using FORTRAN, C, Basic, among other programming languages. This work considered the implementation of the False Position Method, otherwise known as the Regula Falsi method for the determination of roots of non-linear equations using Java. Comparison between results obtained showed that there is faster convergence and greater accuracy in the results obtained using Java than as in the results obtained using FORTRAN. Hence a good working knowledge of Java might end up being advantageous to an average physicist.

**[**Makinde, V., Akinboro, F.G., Okeyode, I.C., Mustapha, A.O., Coker, J.O., and Adesina, O.S. **Implementation of**

**the false position (regula falsi) as a computational physics method for the determination of roots of non-linear equations using Java.** *Nat Sci* 2013;11(7):118-123]. (ISSN: 1545-0740). <http://www.sciencepub.net/nature>. 20

**Keywords:** Computational Physics Methods, Regular Falsi, FORTRAN, Java.

1. **Introduction**

 Computational Physics aims at obtaining numerical solutions to physical problems in which numerical analysis methods are used to provide approximate solutions to problems in Physics (Gupta, 2010). The scale of modern day problems being solved by computational physicist requires the use of programming languages that are very easy to use; provide features which makes it possible to re-use existing codes; efficient; is capable of specifying different operations to be executed simultaneously by the computer; and that enable distributed programs to be easily developed (Arfken et. al., 2012).. Java is such a programming language.

 The relevance that computational physics, numerical analysis or computational science in general has today, is as a result of a lot of work that had been done in the implementation of several computational methods using computer programming languages. Chapman (1998) extensively implemented computational methods using FORTRAN 90/95. FORTRAN, which was developed by IBM, is essentially a computational tool; it has been used extensively to develop programs in both the defense and geophysical fields (Chapman, 1998). C, a language developed by Dennis Ritchie in the 1960s, is another language that has found extensive use in computational science. C is most suitable for High Performance Computing (HPC) because of its speed of execution but is very susceptible to errors especially if used by a not so skillful programmer.

 Java is a modern object oriented language which facilitates disciplined approach to program design (Deitel and Deitel, 2007). It has features that make it suitable for modern day computation; these include multithreading (parallel programming), object orientation, support for internet, among others.

 Pang (2006) used Java extensively to implement computational methods in his bid to show the suitability of Java to computational science as well as in introducing students to computational physics.

In this work, Java was used to implement the computational methods because

1. it is a modern object oriented language which facilitates a disciplined approach to program design.
2. it is suitable for modern day computation as it provide fully for the needs of the modern day computational physicists which include parallel programming, object orientation, support for the internet, among others.
3. FORTRAN and C have been used extensively in the implementation of computational physics problems.
4. **Objectives**

The objectives of this work include:

1. Implementation of the Regula Falsi method using Java.
2. Testing the implemented method with examples obtained from other academic sources.
3. Evaluating the Java implementation of the computational physics methods by comparing them with similar implementations done with other programming languages.
4. **Determination of the Roots of Non-Linear Equations**

Given a function f(x), if f(x) = 0, then the values of the variable x that satisfies the condition f(x) = 0 are called the *roots* of the equation. These are also known as the *zeros of f(x).*

It is quite easy to find the roots of some equations.

 For example, if the function f(x) is linear in nature may be given as f(x) = 3x - 12, then 3x - 12 = 0, is solved simply to obtain x = 4.

In a situation where f(x) is quadratic, then, there exists a standard formula, the well known quadratic formula, given by $x= \frac{-b \pm \sqrt{b^{2}- 4ac}}{2a} . $ that can be used to obtain the roots of the equation. However, as the power to which the variable x is raised increases, finding the roots of the equation becomes more difficult. It has been proven that no general formula exists for polynomials of degree greater than four meaning that there is no way to exhibit the roots in terms of "ordinary" functions (Gerald & Wheatley, 1999),. Such polynomials are usually solved by successive approximations. Some of the methods employed include: Root Bisection (or Interval Halving), Secant Method, Regula Falsi method, Fixed-Point Iteration method, Newton's method, Muller's method, among others. This work therefore focuses on solving these higher order polynomials numerically to obtain one or more of the roots of such equations.

1. **The False Position (Regula Falsi -** *in Latin***) Method**

**Theory**

 The technique employed in the False Position method is such that each next iterate is taken at an arbitrary point between the pairs of x-values that is, the two starting values rather than the midpoint as in other methods such as the root bisection method. This may result in an advantage of faster convergence than some other methods, but at the expense of a more complicated algorithm.

In achieving the goals of this work, pseudocode for the False Position algorithm (*regula falsi*) was developed and is given next:

To determine the root of *f(x) = 0,* given values *X 1* and *X 2* that bracket a root, that is, *f(X 1)* and *f(X 2)*,

 REPEAT

Set X 3 = X2 - *f(*X2) \* (X1 - X2) / (*f*(X1) - *f*(X2))

IF *f*(X3) of opposite sign to *f*(X1)

Set X2 = X3

ELSE

Set X1 = X3

END IF

 UNTIL |*f(X3)*| < tolerance value

NOTE: The method may give a false root if *f(x)* is discontinuous on the interval. The final value of X3 approximates the root within the accuracy of the specified tolerance value (Gerald & Wheatley, 1999).

**Implementation**

 In implementing the False Position method, classes *RegulaFalsi* and *RegulaFalsiMethod were* created. Class *RegulaFalsi* extends class *RootBisection* (Adesina, 2010). This feature of Java is called Inheritance and it is a technique for enhancing code reusability and for establishing what is known as a "*is-a"* relationship between the inheriting classes and the inherited class. The inheriting class is called the subclass while the inherited class is called the superclass

 By allowing *RegulaFalsi* to inherit from *RootBisection*, all the public methods of class *RootBisection* are automatically available in the *RegulaFalsi* class and can be called from within any method in *RegulaFalsi*. Class *RegulaFalsi* overrides the*getRoot()* method of class *RootBisection* from which it inherits by providing its own implementation. The term "override" in the sense that because *getRoot()* is declared and defined in *RootBisection* - the superclass, the *getRoot()*,of the *RegulaFalsi* class, that implements the False Position algorithm.

**1 public** **double** getRoot() {

2 **int** iterate = 0;

3 **double** mid, x1, x2, oppSign, fxmid;

4 x1 = getLowerLimitOfInterval();

5 x2 = getUpperLimitOfInterval();

6 setOutput("");

7 compileOutput(String.*format*("\n%15s%15s%15s%15s%15s\n", "ITR NO","X1", "X2", "X3", "F(X3)"));

8 **do** {

9 iterate += 1;

10 mid = x2 - (Function.*getFofX*(x2, getCoefficients()) \* ((x1 - x2) / Function.*getFofX*(x1, getCoefficients()) - Function.*getFofX*(x2, getCoefficients()))));

11 fxmid = Function.*getFofX*(mid, coefficients);

12 compileOutput(String.*format*("\n%15d%15.7f%15.7f%15.7f

%15.7f", iterate, x1, x2, mid, fxmid));

13 oppSign = fxmid \* Function.*getFofX*(x1, coefficients);

14 **if** ( oppSign < 0 ) {

15 x2 = mid;

16 } **else** {

17 x1 = mid;

18 }

19 } **while** ( !((Math.*abs*(x1 - x2) < getTolerance() ) || (iterate >= maxIteration)) );

20 compileOutput(String.*format*("\n\n%s\n\n", "Program output for x1 = " + getLowerLimitOfInterval() + ", x2 = " + getUpperLimitOfInterval() + ", tolerance = " + tolerance));

21 **return** mid;

22 }

Code Listing 1: The *getRoot()* method of *RegulaFalsi* class.

 Adesina (2010), implemented the Root Bisection Method using a similar algorithm. It could be observed in Code Listing 1 that the lines 10 and 19 are quite different from lines 10 and 19 of similar listing for the Root Bisection method (Adesina, 2010). Line 1 of Code Listing 1 shows how the False Position method differs from the Root Bisection method algorithm developed by Adesina (2010). Line 19 of Code Listing 1 compares the absolute value of the difference between X1 and X2 directly with the tolerance value rather than twice the tolerance value as was done in the Root Bisection method.

1. **Tests and Results**

**Illustration 1:** Gerald and Wheatley (1999) implemented the function *f(x) = x3 + x2 - 3x - 3 = 0* with FORTRAN 90/95. The result obtained is as shown in **Table 1.**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **ITR NO** | **X1** | **X2** | **X3** | **F(X3)** |
| 1 | 1.000000 | 2.000000 | 1.500000 | - 1.875000 |
| 2 | 1.500000 | 2.000000 | 1.750000 |  0.171875 |
| 3 | 1.500000 | 1.750000 | 1.625000 | - 0.943359 |
| 4 | 1.625000 | 1.750000 | 1.687500 | - 0.409424 |
| 5 | 1.687500 | 1.750000 | 1.718750 | - 0.124786 |
| 6 | 1.718750 | 1.750000 | 1.734375 |  0.022030 |
| 7 | 1.718750 | 1.734375 | 1.726563 | - 0.051756 |
| 8 | 1.726563 | 1.734375 | 1.730469 | - 0.014957 |
| 9 | 1.730469 | 1.734375 | 1.732422 |  0.003512 |
| 10 | 1.730469 | 1.732422 | 1.731445 | - 0.005728 |
| 11 | 1.731445 | 1.732422 | 1.731934 | - 0.001109 |
| 12 | 1.731934 | 1.732422 | 1.732178 |  0.001202 |
| 13 | 1.731934 | 1.732178 | 1.732056 |  0.000045 |

Table 1: Finding the root of *f(x) = x3 + x2 - 3x - 3 = 0* starting with *X1 = 1, X2 = 2,* and tolerance of 1E-4 as implemented by Gerald and Wheatley (1999) using FORTRAN 90/95

 When the function *f(x) = x3 + x2 - 3x - 3 = 0* obtained from Gerald and Wheatley (1999) was solved using the Java implementation of the method of False Position, the following results were obtained.

**Table 2:** Finding the root of *f(x) = x3 + x2 - 3x - 3 = 0* starting with *X1 = 1, X2 = 2,* and tolerance of 1E-4 by the method of False Position

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **ITR NO** | **X1** | **X2** | **X3** | **F(X3)** |
| 1. | 1.0000000 | 2.0000000 | 1.5714286 | * 1.3644315
 |
| 2. | 1.5714286 | 2.0000000 | 1.7054108 | * 0.2477451
 |
| 3. | 1.7054108 | 2.0000000 | 1.7278827 | * 0.0393306
 |
| 4. | 1.7278827 | 2.0000000 | 1.7314049 | * 0.0061107
 |
| 5. | 1.7314049 | 2.0000000 | 1.7319509 | * 0.0009459
 |
| 6. | 1.7319509 | 2.0000000 | 1.7320353 | * 0.0001463
 |
| 7. | 1.7320353 | 2.0000000 | 1.7320484 | * 0.0000226
 |
| 8. | 1.7320484 | 2.0000000 | 1.7320504 | * 0.0000035
 |
| 9. | 1.7320504 | 2.0000000 | 1.7320508 | * 0.0000005
 |
| 10. | 1.7320508 | 2.0000000 | 1.7320508 | * 0.0000001
 |
| 11. | 1.7320508 | 2.0000000 | 1.7320508 | * 0.0000000
 |
| 12. | 1.7320508 | 2.0000000 | 1.7320508 | * 0.0000000
 |
| 13. | 1.7320508 | 2.0000000 | 1.7320508 | * 0.0000000
 |

Program output for x1 = 1.0, x2 = 2.0; tolerance = 1.0E-4

Table 2 reveals that the method of False Position is faster to converge as can be seen in the values of X3; it converges at iterate 9. The values of X3 approach the true value of the root, which is √3 (1.732050808) as the number of iterations increase unlike the Root Bisection method which is irregular in that earlier estimates may be better than later ones. However, one should note that the method of False Position converges to the root from one side, which slows it down, especially if that end of the interval is farther from the root.

**Illustration 2:** The function *f(x) = x4 - 2 = 0* obtained from Gerald & Wheatley (1999), was implemented with Java for the Root Bisection Method. The following results, shown in table 3, were obtained.

**Table 3:** Finding the root of *f(x) = x4 - 2 = 0* starting with *X1 = 1, X2 = 2,* and tolerance of 1E-4

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **ITR NO** | **X1** | **X2** | **X3** | **F(X3)** |
| 1 | 1.0000000 | 2.0000000 | 1.5000000 |  3.0625000 |
| 2 | 1.0000000 | 1.5000000 | 1.2500000 |  0.4414063 |
| 3 | 1.0000000 | 1.2500000 | 1.1250000 | - 0.3981934 |
| 4 | 1.1250000 | 1.2500000 | 1.1875000 | - 0.0114594 |
| 5 | 1.1875000 | 1.2500000 | 1.2187500 |  0.2062693 |
| 6 | 1.1875000 | 1.2187500 | 1.2031250 |  0.0952845 |
| 7 | 1.1875000 | 1.2031250 | 1.1953125 |  0.0413893 |
| 8 | 1.1875000 | 1.1953125 | 1.1914063 |  0.0148350 |
| 9 | 1.1875000 | 1.1914063 | 1.1894531 |  0.0016555 |
| 10 | 1.1875000 | 1.1894531 | 1.1884766 | - 0.0049100 |
| 11 | 1.1884766 | 1.1894531 | 1.1889648 | - 0.0016293 |
| 12 | 1.1889648 | 1.1894531 | 1.1892090 |  0.0000126 |
| 13 | 1.1889648 | 1.1892090 | 1.1890869 | - 0.0008085 |

When the function *f(x) = x4 - 2 = 0* obtained from

Gerald & Wheatley (1999) was solved using Java implementation of the method of False Position, the following results were obtained.

**Table 4:** Finding the root of *f(x) = x4 - 2 = 0* starting with *X1 = 1, X2 = 2,* and tolerance of 1E-4 by the method of False Position

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **ITR NO** | **X1** | **X2** | **X3** | **F(X3)** |
| **1.** | 1.0000000 | 2.0000000 | 1.0666667 |  - 0.7054617 |
| **2.** | 1.0666667 | 2.0000000 | 1.1114413 | - 0.4740298 |
| **3.** | 1.1114413 | 2.0000000 | 1.1405419 | - 0.3078263 |
| **4.** | 1.1405419 | 2.0000000 | 1.1590327 | - 0.1953924 |
| **5.** | 1.1590327 | 2.0000000 | 1.1706082 | - 0.1222133 |
| **6.** | 1.1706082 | 2.0000000 | 1.1777857 | - 0.0757337 |
| **7.** | 1.1777857 | 2.0000000 | 1.1822096 | - 0.0466595 |
| **8.** | 1.1822096 | 2.0000000 | 1.1849261 | - 0.0286439 |
| **9.** | 1.1849261 | 2.0000000 | 1.1865903 | - 0.0175454 |
| **10.** | 1.1865903 | 2.0000000 | 1.1876085 | - 0.0107327 |
| **11.** | 1.1876085 | 2.0000000 | 1.1882308 | - 0.0065598 |
| **12.** | 1.1882308 | 2.0000000 | 1.1886110 | - 0.0040073 |
| **13.** | 1.1886110 | 2.0000000 | 1.1888432 | - 0.0024473 |
| **14.** | 1.1888432 | 2.0000000 | 1.1889849 | - 0.0014943 |
| **15.** | 1.1889849 | 2.0000000 | 1.1890715 | - 0.0009123 |
| **16.** | 1.1890715 | 2.0000000 | 1.1891243 | - 0.0005569 |
| **17.** | 1.1891243 | 2.0000000 | 1.1891566 | - 0.0003400 |
| **18.** | 1.1891566 | 2.0000000 | 1.1891763 | - 0.0002075 |
| **19.** | 1.1891763 | 2.0000000 | 1.1891883 | - 0.0001267 |
| **20.** | 1.1891883 | 2.0000000 | 1.1891956 | - 0.0000773 |

Approximate root = 1.189207115

 Program output for x1 = 1.0, x2 = 2.0; tolerance = 1.0E-4

1. **Regula Falsi Method applied to Quadratic Equations**

 The Regula Falsi has already been implemented and shown to be realizable for non-linear polynomials of order greater than 2. In order to buttress its applicability to all non-linear polynomials in general, further examples elucidating its applicability to quadratic equations are shown in the following two examples:

**Illustration 3:** f(x) = x2 – 2 = 0 (Adapted from Stroud and Booth, 2003)

 WELCOME TO THE REGULA FALSI METHOD

THIS PROGRAM IMPLEMENTATION ALLOWS ONE TO FIND THE ROOT OF A POLYNOMIAL OR NON-LINEAR EQUATION

Enter the lower limit of the interval, x1: 1

Enter the upper limit of the interval, x2: 2

Enter the degree of the polynomial: 2

Now, enter the elements of the coefficient vector one after the other.

Enter A0: -2

Enter A1: 0

Enter A2: 1

Enter the tolerance value: 0.00001

Enter the maximum number of iterations in case tolerance is not met: 20

Approximate root found: 1.414214

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **ITR NO** | **X1** | **X2** | **X3** |  **F(X3)** |
| 1 | 1.0000000 | 2.0000000 | 1.3333333 | -0.2222222 |
| 2 | 1.3333333 | 2.0000000 | 1.4000000 | -0.0400000 |
| 3 | 1.4000000 | 2.0000000 | 1.4117647 | -0.0069204 |
| 4 | 1.4117647 | 2.0000000 | 1.4137931 | -0.0011891 |
| 5 | 1.4137931 | 2.0000000 | 1.4141414 | -0.0002041 |
| 6 | 1.4141414 | 2.0000000 | 1.4142012 | -0.0000350 |
| 7 | 1.4142012 | 2.0000000 | 1.4142114 | -0..0000060 |
| 8 | 1.4142114 | 2.0000000 | 1.4142132 | -0.0000010 |
| 9 | 1.4142132 | 2.0000000 | 1.4142135 | -0.0000002 |
| 10 | 1.4142135 | 2.0000000 | 1.4142136 | -0.0000000 |
| 11 | 1.4142136 | 2.0000000 | 1.4142136 | -0.0000000 |
| 12 | 1.4142136 | 2.0000000 | 1.4142136 | -0.0000000 |
| 13 | 1.4142136 | 2.0000000 | 1.4142136 | -0.0000000 |
| 14 | 1.4142136 | 2.0000000 | 1.4142136 | -0.0000000 |
| 15 | 1.4142136 | 2.0000000 | 1.4142136 | -0.0000000 |
| 16 | 1.4142136 | 2.0000000 | 1.4142136 | -0.0000000 |
| 17 | 1.4142136 | 2.0000000 | 1.4142136 | -0.0000000 |
| 18 | 1.4142136 | 2.0000000 | 1.4142136 | -0.0000000 |
| 19 | 1.4142136 | 2.0000000 | 1.4142136 | -0.0000000 |
| 20 | 1.4142136 | 2.0000000 | 1.4142136 | -0.0000000 |

 Program output for x1 = 1.0, x2 = 2.0, tolerance = 1.0E-5

**Illustration 4:** f(x) = 2x2 – 9x + 5 = 0

 **R**esults obtained from Java implementation Regula Falsi Method for this equation are as follows:

**First Root**

Enter the lower limit of the interval, x1: 1

Enter the upper limit of the interval, x2: 4

Enter the degree of the polynomial: 2

Now, enter the elements of the coefficient vector one after the other.

Enter A0: 5

Enter A1: -9

Enter A2: 2

Enter the tolerance value: 0.00001

Enter the maximum number of iterations in case tolerance is not met: 20

Approximate root found: 3.850781

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **ITR NO** | **X1** | **X2** | **X3** | **F(X3)** |
| 1 | 1.0000000 | 4.0000000 | 3.0000000 | -4.0000000 |
| 2 | 3.0000000 | 4.0000000 | 3.8000000 | -0.3200000 |
| 3 | 3.8000000 | 4.0000000 | 3.8484848 | -0.0146924 |
| 4 | 3.8484848 | 4.0000000 | 3.8506787 | -0.0006552 |
| 5 | 3.8506787 | 4.0000000 | 3.8507765 | -0.0000292 |
| 6 | 3.8507765 | 4.0000000 | 3.8507809 | -0.0000013 |
| 7 | 3.8507809 | 4.0000000 | 3.8507811 | -0.0000001 |
| 8 | 3.8507811 | 4.0000000 | 3.8507811 | -0.0000000 |
| 9 | 3.8507811 | 4.0000000 | 3.8507811 | -0.0000000 |
| 10 | 3.8507811 | 4.0000000 | 3.8507811 | -0.0000000 |
| 11 | 3.8507811 | 4.0000000 | 3.8507811 | -0.0000000 |
| 12 | 3.8507811 | 4.0000000 | 3.8507811 | -0.0000000 |
| 13 | 3.8507811 | 4.0000000 | 3.8507811 |  0.0000000 |
| 14 | 3.8507811 | 4.0000000 | 3.8507811 |  0.0000000 |
| 15 | 3.8507811 | 4.0000000 | 3.8507811 |  0.0000000 |
| 16 | 3.8507811 | 4.0000000 | 3.8507811 |  0.0000000 |
| 17 | 3.8507811 | 4.0000000 | 3.8507811 |  0.0000000 |
| 18 | 3.8507811 | 4.0000000 | 3.8507811 |  0.0000000 |
| 19 | 3.8507811 | 4.0000000 | 3.8507811 |  0.0000000 |
| 20 | 3.8507811 | 4.0000000 | 3.8507811 |  0.0000000 |

 Program output for x1 = 1.0, x2 = 4.0, tolerance = 1.0E-5

Approximate root found: 0.649219

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **ITR NO** | **X1** | **X2** | **X3** | **F(X3)** |
| 1 | -1.0000000 | 1.0000000 | 0.7777778 | -0.7901235 |
| 2 | -1.0000000 | 0.7777778 | 0.6941176 | -0.2834602 |
| 3 | -1.0000000 | 0.6941176 | 0.6646267 | -0.0981829 |
| 4 | -1.0000000 | 0.6646267 | 0.6544741 | -0.0335943 |
| 5 | -1.0000000 | 0.6544741 | 0.6510076 | -0.0114465 |
| 6 | -1.0000000 | 0.6510076 | 0.6498273 | -0.0038946 |
| 7 | -1.0000000 | 0.6498273 | 0.6494258 | -0.0013245 |
| 8 | -1.0000000 | 0.6494258 | 0.6492893 | -0.0004503 |
| 9 | -1.0000000 | 0.6492893 | 0.6492429 | -0.0001531 |
| 10 | -1.0000000 | 0.6492429 | 0.6492271 | -0.0000521 |
| 11 | -1.0000000 | 0.6492271 | 0.6492217 | -0.0000177 |
| 12 | -1.0000000 | 0.6492217 | 0.6492199 | -0.0000060 |
| 13 | -1.0000000 | 0.6492199 | 0.6492193 | -0.0000020 |
| 14 | -1.0000000 | 0.6492193 | 0.6492190 | -0.0000007 |
| 15 | -1.0000000 | 0.6492190 | 0.6492190 | -0.0000002 |
| 16 | -1.0000000 | 0.6492190 | 0.6492190 | -0.0000001 |
| 17 | -1.0000000 | 0.6492190 | 0.6492189 | -0.0000000 |
| 18 | -1.0000000 | 0.6492189 | 0.6492189 | -0.0000000 |
| 19 | -1.0000000 | 0.6492189 | 0.6492189 | -0.0000000 |
| 20 | -1.0000000 | 0.6492189 | 0.6492189 | -0.0000000 |

 Program output for x1 = -1.0, x2 = 1.0, tolerance = 1.0E-5

**Second Root**:

Enter the lower limit of the interval, x1: -1

Enter the upper limit of the interval, x2: 1

Enter the degree of the polynomial: 2

Now, enter the elements of the coefficient vector one after the other.

Enter A0: 5

Enter A1: -9

Enter A2: 2

Enter the tolerance value: 0.00001

Enter the maximum number of iterations in case tolerance is not met: 20

1. **Conclusion**

 The scale of modern day problems being solved by computational physicist requires the use of programming languages that are very easy to use; provide features which make it possible to re-use existing codes; is capable of specifying different operations to be executed simultaneously by the computer; and that enable distributed programs to be easily developed (Dass, 2010). Java is such a programming language, and has been used in this work to determine roots of non-linear equations as set out. The relevance that computational physics, numerical analysis or computational science in general has today, is as a result of a lot of work that had been done in the implementation of several computational methods using computer programming language (Stroud and Booth, 2001).

 Implementation of the Regula Falsi method using both FORTRAN and Java implemented and compared in this work has shown that

1. Java implementation is more robust than FORTRAN implementation;
2. Java is more adaptable in shorter listings than FORTRAN;
3. Regula Falsi is fast and regular in convergence.

 It therefore means that Java, a modern object oriented language which facilitates disciplined approach to program design with features that make it suitable for modern day computation is highly effective in the implementation of basic computational physics methods in such a way that makes the realization of computational objectives easy to achieve (Chow, 2000; DeVries, 1993). It is also robust in adaptation and implementation (Kiusalaas, 2005); and therefore a veritable tool in the implementation of various computational physics methods.

**Acknowledgements:**

Appreciation goes to all who assisted in typesetting this work.

**Corresponding Author**

Victor Makinde,

Department of Physics,

Federal university of Agriculture,

P.M.B. 2240, Abeokuta, Nigeria

e-mail: victor\_makindeii@yahoo.com

**References**

1. Arfken, G.B., Weber, H.J., and Harris, F.E. 2012. *Mathematical Methods for Physicists. 7th Edition*. Associated Press. New York, U.S.A. pp 1205

2. Chapman, S.J 1998. *FORTRAN 90/95 for Scientists*

 *and Engineers*. McGraw-Hill, USA pp 431

3. Chow, T.L. 2000. *Mathematical Methods for Physicists – A Concise Introduction*. Cambridge University Press. U.S.A. pp 569

4. Dass, H.K. 2010. *Advanced Engineering Mathematics*. S Chand and Co. Publishers. New Delhi, India. pp 1358

5. Deitel, P.J. and Deitel, H.M. 2007. *Java: How to Program*. Pearson Education Inc, New Jersey, USA. pp 317

6. DeVries, P.L. 1993. *A First Course in Computational Physics*. John Wiley & Sons, New York, U.S.A. pp 435

7. Gerald, C.F. and Wheatley, P.O. 1999. *Applied Numerical Analysis*. Dorling Kindersley, India. pp 698

8. Gupta, B.D. 2010. *Mathematical Physics. 4th Edition*. Vikas Publishing House, New Delhi, India. pp 1417

9. Kiusalaas, J. (2005). Numerical Methods in Engineering *with MATLAB.* Cambridge University Press. U.S.A. pp 435

10. Pang, T. 2006. *Introduction to Computational Physics*. Cambridge University Press, New York, USA. pp 528

11. Stroud, K.A., and Booth, D.J. 2001. *Engineering Mathematics*. Palgrave Macmillan, New York, USA. pp 1236

12. Stroud, K.A., and Booth, D.J. 2003. *Advanced Engineering Mathematics*. Palgrave Macmillan, New York, USA. pp 1057

4/1/2013