Journal of the Nigerian Association of Mathematical Physics
Volume32,(*November, 2015*), pp 91 – 98

Determination of Complex Roots of Non-Linear Equations by Implementation of the Newton Method using Java

[1]Makinde V., [1]Okeyode I.C., [1.]Akinboro F.G., [2.]Coker J.O., [1.]Adesina O.S., and [1.]Alatise O.O.

[1]Department of Physics, Federal University of Agriculture, Abeokuta
[2]Department of Physics, Olabisi Onabanjo University, Ago Iwoye.

Abstract

Hitherto computational physics methods such as root bisection and regula falsi have been used to determine simple roots of non-linear equations. Advancement in programming and language development has made possible improved efficiency and accuracy in solving numerical problems and hence the numerical computation of physical problems. Furthermore, languages such as Basic, Fortran, C, among others, have commonly been employed in solving numerical problems. In this work, Java, a modern object oriented language was deployed in solving computational physics problems, specifically, determination of complex roots of non-linear equations using the Newton Method. A numerical computation of the method shows that Newton method converges faster with greater accuracy than other methods considered in this paper.

1.0 Introduction

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. Java is such a programming language, and has been used in this work to determine complex roots of non-linear equations as set out.

Java is a modern object oriented language which facilitates disciplined approach to program design [1]. It has features that make it suitable for modern day computation which include multithreading (parallel programming), object orientation, support for internet, among others.

Computational Physics seeks numerical solutions to physical problems. It involves the use of numerical analysis methods to provide approximate solutions to problems in Physics. Gerald and Wheatley [2] described numerical analysis as the development and study of procedures for solving problems with a computer. The term "algorithm", used for a systematic procedure that solves a problem, is defined as a step by step solution to a problem in terms of the actions to be taken and the order in which they are to be taken. A computational physicist or numerical analyst often is interested in determining which of several algorithms that can solve the problem is, in some sense, the most efficient. Efficiency may be measured in many ways some of which include the number of steps in the algorithm, the time taken by the computer to execute the algorithm, the amount of computer memory used, among others. A major advantage of numerical analysis is that a numerical solution can be obtained even when a problem has no analytical solution. A numerical analysis solution is always numerical; it is an approximation, whose results can be made as accurate as desired. Analytical methods usually give a result in terms of mathematical functions that can then be evaluated for specific instances.

Solving for the complex roots of non-linear equations is one of the operations that numerical analysis can do [2]. It can also be applied in solving large systems of linear equations; obtaining the solutions of a set of non-linear equations, interpolating to find intermediate values within a table of data, finding efficient and effective approximations of functions, among others.

Several authors such as [3-4-5] used Java extensively to implement computational methods in a bid to introduce students to computational physics and to show the suitability of Java to computational science. They also enumerated the numerous ways in which computational methods can be adapted to solve numerical problems.

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

i) much of the work that had been done in the field of computational physics used FORTRAN and C;

Corresponding author: Makinde V., E-mail:victor_makindeii@yahoo.com, Tel.: +2348035994001

ii) these two languages, although still powerful and efficient, tend toward becoming old languages in that they do not provide fully for the needs of the modern day computational physicists.
iii) Java is a modern object oriented language which facilitates a disciplined approach to program design.

Some of the other features of Java that make it suitable for modern day computation include multithreading (parallel programming), object orientation, support for the internet among others.

2.0 Objectives

The objectives of this work include:

i) Implementation of the Newton method using Java.
ii) Testing the implemented method with examples obtained from text books and from other sources.
iii) Evaluating the Java implementation of the computational physics methods by comparing them with similar implementations done with other programming languages in some other texts.

3.0 Determination of the Roots of Non-Linear Equations

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

However, as we move higher in the power to which the variable x is raised, finding the roots of the equation becomes more tedious.

According to [2], it has been proved 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. Usually, such polynomials are solved by successive approximations and 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.

Makinde, et al. [6] used the root bisection method, for finding a zero, and hence root of f(x). Out of the common methods, the root bisection method is about the simplest to understand and the easiest to implement.

Makinde, et al. [7] applied the Regula-Falsi method to solve basic quadratic equations.

These are briefly highlighted next.

3.1 The Root Bisection Method

3.1.1 Theory

To find a root of f(x), the root bisection method begins with two values $x = x_1$ and $x = x_2$ that enclose a root. It is known that a root is enclosed if the function changes sign at the endpoints, that is, at $f(x_1)$ and $f(x_2)$; this is true if $(f(x_1)*f(x_2)) < 0$ [8]. It is certain that there is at least one root in the interval $[x_1, x_2]$ as long as f(x) is continuous in $[x_1, x_2]$. The method then successively divides the interval in half and replaces one endpoint with the midpoint so that again the root is enclosed. Known in advance is that the error in the estimate of the root must be less than $|(x_2–x_1)*(^1/2^n)|$ where n is the number of iterations performed.

In implementing the Root Bisection Method, the pseudocode was written to set the bracket values and algorithm for implementation. The pseudocode for the Root Bisection algorithm is stated thus.

To determine a root of $f(x) = 0$ that is accurate within a specified tolerance value, given values X_1 and X_2 such that $f(X_1) * f(X_2) < 0$, we use the following algorithm:

```
REPEAT
    Set X3 = (X1 + X2) / 2

    IF (f(X3) * f(X1) < 0):

        Set X2 = X3
    ELSE

        Set X1 = X3
    END IF
UNTIL (|X1 - X2| < 2 * tolerance value) or f(X3) = 0
```

The method may give a false root if $f(x)$ is discontinuous in $[X_1, X_2]$. The final value of X_3 approximates the root within the accuracy of the specified tolerance value [2].

The main advantage of root bisection is that it is guaranteed to work if f(x) is continuous in $[x_1, x_2]$ and if the values $x = x_1$ and $x = x_2$ actually bracket a root. Another advantage is that the number of iterations required to achieve a specified accuracy

is known in advance. To find all roots, the limits are reset to new values within the expected range $x_1 < x < x_2$, or to choose a broad all enclosing limits $[X_1, X_2]$ from inception.

The major drawback of root bisection is that it is slow to converge. Other methods such as the Newton's method require fewer numbers of iterations to achieve the same level of accuracy.

In spite of arguments that other methods find roots with fewer iterations, root bisection is nevertheless an important tool in the computational physicist's arsenal. It is generally recommended that root bisection be used for finding approximate root which can then be refined by more efficient methods. The reason is that most other methods require a starting value near to a root which, if not available, may cause them to fail completely.

3.1.2 Implementation

In the implementation of the root bisection method a Java class called *RootBisection, was created*. This class consists of six private fields and fifteen public methods which includes a constructor and the corresponding *set* and *get* assessors for each of the fields. The method called *getRoot()* implements the algorithm for the root bisection method.

A driver class called *RootBisectionMethod* was created to collect the data to satisfy the preconditions of the root bisection algorithm and to execute the *getRoot()* method of the RootBisection class which is the method that implements the root bisection algorithm. The *RootBisectionMethod* class is an application class because it contains a method called *main()* which is the entry point for all Java programs. The code listing for the *getRoot()* method was also written to implement the algorithm.

3.2 The False Position (RegulaFalsi - *in Latin*) Method

3.2.1 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 the work, pseudocode for the False Position algorithm (*regulafalsi*) 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 $f(X_2)$, we use the following algorithm:

```
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
```

It was also noted that the method may give a false root if $f(x)$ is discontinuous on the interval. The final value of X_3 approximates the root within the accuracy of the specified tolerance value [2].

3.2.2 Implementation

In implementingtheFalsePositionmethod, classes *RegulaFalsi* and *RegulaFalsiMethod were* created. Class*RegulaFalsi* extends class*RootBisection [9]*. 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.

As shown by [2], in order to obtain all roots, having found one of the roots, the limits are reset to new values within the expected range x1 < x < x2, or a broad all enclosing limits [x1, x2] is chosen from inception [10]. Either of these procedures brings out clearly the other roots of the equation being solved.

4.0 Newton's Method

4.1 Theory

One of the most widely used methods for solving equations is the Newton's Method. According to [2], Sir Isaac Newton did not publish an extensive discussion of this method, but he solved a cubic polynomial in *Principia* (1687). The version implemented here is considerably improved over his original example.

The Newton's method, like the previous ones implemented, is also based on a linear approximation of the function, but it does so by using a tangent to the curve. The method starts from a single estimate X_o, which is not too far from a root, it moves along the tangent to its intersection along the *x*-axis, and takes that as the next approximation. This is continued until either successive *x*-values are sufficiently close or the value of the function is sufficiently near zero. The criterion to be used often depends on the particular physical problem to which the equation applies. Customarily, agreement of successive *x*-values to a specified tolerance is required.

The calculation scheme followed from the right angled triangle formed by the tangent with the *x*-axis, which has the angle of inclination of the tangent line to the curve at $x = X_o$ as one of its acute angles. This is given by the expression:

$$\tan\theta = f^1(x_0) = {}^{f(x_0)}\!/_{(x_0 - x_1)}\,;\; x_1 = x_0 - \left({}^{f(x_0)}\!/_{f^1(x_0)}\right)$$ The calculation scheme is contained by computing

$$x_2 = x_1 - \left({}^{f(x_0)}\!/_{f^1(x_0)}\right) \; or, in\ more\ general\ terms,\; x_{n+1} = x_n - \left({}^{f(x_n)}\!/_{f^1(x_n)}\right), n = 0, 1, 2, \ldots$$

The Newton's method is widely used because, at least near the neighbourhood of a root, it is more rapidly convergent than any of the other two methods given in [6-7]. However, offsetting this advantage is the need for two function evaluations at each step, $f(X_n)$ and $f'(X_n)$. Another problem with Newton's method is that finding $f'(X)$ may be difficult.

4.2 Implementation

In implementing the Newton method, a pseudocode for the algorithm was written.

To determine the root of $f(x) = 0$, given a value X_o reasonably close to the root, we use the following algorithm:

Compute $f(X_o)$, $f'(X_o)$
IF ($f(X_o)$ <> 0) AND ($f'(X_o)$ <> 0)
    REPEAT
        Set $X_1 = X_0$
        Set $X_o = X_o - f(X_o) / f'(X_o)$
UNTIL ($|X_o - X_1|$ < tolerance value 1) OR ($|f(Xo)|$ < tolerance value 2)

According to [2], the method may converge to a root different from the expected one or diverge if the starting value is not close enough to the root.

4.3 Determination of Real Roots

The implementation done for the Newton's method consists of two classes, the *Newton* class and the *NewtonsMethod* class. The *NewtonsMethod* class is the application class or the driver class in which was implemented the driver program that collects the required data from the user and uses the *Newton* class to compute the root. The *getRoot()* method of the *Newton* class implements the Newton's algorithm for finding real roots and is shown next:

```
1   public double getRoot() {
2       int iterate = 0;
3       double  Xi, Xii;
4       Xii = getXo();
5       setOutput("");
6       compileOutput(String.format("\n%15s%15s%15s%15s", "ITR No","Xi",
            "Xi+1", "F(Xi+1)"));
7       do {
8           iterate += 1;
9           Xi = Xii;
10          Xii = Xi - (Function.getFofX(Xi, getCoefficients()) /
            Function.getDerivateFofX(Xi, getCoefficients()));
11          compileOutput(String.format("\n%15d%15.7f%15.7f%15.7f",
            iterate, Xi, Xii, Function.getFofX(Xii,
                getCoefficients())));
```

```
12                              } while ( !((Math.abs(Xi - Xii) < getTolerance()) || (iterate >=
                    maxIteration) ) );
13                    compileOutput(String.format("\n\n%s\n\n", "Program output for Xo
                    = " + getXo() + ", tolerance = " + getTolerance()));
14                              return Xii;
15          }
```

Fig. 1: Code Listing 1 - The *getRoot()* method of the *Newton* class

Line 10 of Fig. 1 computes the next iterate by making use of methods of a class called *Function* to determine the derivative of the function *f(X)* and the value of the function at a given value of X.

4.3.1 Tests and Results

In verifying the viability of Newton's Method in the determination of real and complex roots, two sets of examples were taken from i) [6] and [2]; and ii) [2]. When the function $f(x) = x^3 + x^2 - 3x - 3 = 0$, obtained from [2] was solved using implementation of the Newton's method whose Code Listing is shown in this work, the following results were obtained.

Example 1a: Finding the real root of $f(x) = x^3 + x^2 - 3x - 3 = 0$ starting with $X_o = 1$, and tolerance of 1E-4 by the Newton's method

Table 1: Program output for X_o = 1.0, tolerance = 1.0E-4.(Approximate root found: 1.732051)

ITR No	**Xi**	**Xi+1**	**F(X $_{i+1}$)**
1	1.0000000	3.0000000	24.0000000
2	3.0000000	2.2000000	5.8880000
3	2.2000000	1.8301508	0.9890012
4	1.8301508	1.7377955	0.0545726
5	1.7377955	1.7320723	0.0002033
6	1.7320723	1.7320508	0.0000000

When started with X_o = 2.0 for the function above, Newton's method converged after just four iterations, the result of which is shown next.

Example 1b: Finding the real root of $f(x) = x^3 + x^2 - 3x - 3 = 0$ starting with $X_o = 2$, and tolerance of 1E-4 using Newton's method [7] and [2]

Table 2: Program output for X_o = 2.0, tolerance = 1.0E-4. (Approximate root found: 1.732051)

ITR No	Xi	Xi+1	F(X $_{i+1}$)
1	2.0000000	1.7692308	0.3604916
2	1.7692308	1.7329238	0.0082669
3	1.7329238	1.7320513	0.0000047
4	1.7320513	1.7320508	0.0000000

Tables 3 and 4 show the implementation of the same function using the Root Bisector method as obtained by [2] (Table 3) and [6] (Table 4).

Table 3: Finding the root of $f(x) = x^3 + x^2 - 3x - 3 = 0$ starting with $X1 = 1$, $X2 = 2$, and tolerance of 1E-4 by root bisection method (Adapted from [2]).

ITR NO	X1	X2	X3	F(X3)	MAXIMUMERROR
1	1.000000	2.000000	1.500000	-1.875000	0.500000
2	1.500000	2.000000	1.750000	0.171875	0.250000
3	1.500000	1.750000	1.625000	-0.9433594	0.125000
4	1.625000	1.750000	1.687500	-0.409424	0.062500
5	1.687500	1.750000	1.718750	-0.124786	0.031250
6	1.718750	1.750000	1.734375	0.022030	0.015625
7	1.718750	1.734375	1.726563	-0.051756	0.007813
8	1.726563	1.734375	1.730469	-0.014957	0.003906
9	1.730469	1.734375	1.732422	0.003512	0.001953
10	1.730469	1.732422	1.731445	-0.005728	0.000977
11	1.731445	1.732422	1.731934	-0.001109	0.000488

12	1.731934	1.732422	1.732178	0.001202	0.000244
13	1.731934	1.732178	1.732056	0.000045	0.000122

Tolerance met
The result obtained by the implementation of the root bisection algorithm using Java (Code Listing of [6]) is given next:

Table 4: Finding the root of $f(x) = x^3 + x^2 - 3x - 3 = 0$ starting with $X1 = 1$, $X2 = 2$, and tolerance of 1E-4 by root bisection method. (Approximate root found: 1.732056)

ITR NO	X1	X2	X3	F(X3)
1	1.0000000	2.0000000	1.5000000	-1.6750000
2	1.5000000	2.0000000	1.7500000	0.1718750
3	1.5000000	1.7500000	1.6250000	-0.9433594
4	1.6250000	1.7500000	1.6875000	-0.4094238
5	1.6875000	1.7500000	1.7187500	-0.1247864
6	1.7187500	1.7500000	1.7343750	0.0220299
7	1.7187500	1.7343750	1.7265625	-0.0517554
8	1.7165625	1.7343750	1.7304688	-0.0148572
9	1.7304688	1.7343750	1.7324219	0.0035127
10	1.7304688	1.7324219	1.7314453	-0.0057282
11	1.7314453	1.7324219	1.7319336	-0.0011092
12	1.7319336	1.7324219	1.7321777	0.0012013
13	1.7319336	1.7321777	1.7320557	0.0000460

Program Output for X1 = 1.0; X2 = 2.0; tolerance 1.0E-04
Tables 3 and 4 show that it took the root bisection method thirteen iterations to find the approximate root within the accuracy of the tolerance value; X3 is the mid-point of the interval while *f(X3)* gives the value of the function at *X3.*
It was observed in the tables that the estimate of the root may be better at an earlier iteration than at later ones.

Table 5: Finding the root of $f(x) = x^3 + x^2 - 3x - 3 = 0$ starting with $X1 = 1$, $X2 = 2$, and tolerance of 1E-4 by the method of False Position [7]

Approximate root found: 1.732051

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	-0.0393396
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 5 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 $\sqrt{3}$ = (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 [7].

4.4 Determination of Complex Roots

The *getComplexRoot()* of the *Newton* class implements the Newton's algorithm for finding complex root and is presented next.

```
1       public double[] getComplexRoot() {
2               int iterate = 0;
3               double  Zi[], Zii[];
```

```
4                Zii = getZo();
5                setOutput("");
```



```
6                compileOutput(String.format("\n%5s%20s%20s%20s", "ITR No", "Zi",
                 "Zi+1", "F(Zi+1)"));
7                do {
8                                        iterate += 1;
9                                Zi = Zii;
10                               Zii = Function.subtractComplex(Zi,
                            Function.divideComplex(Function.getFofZ(Zi,
                           getCoefficients()), Function.getDerivativeFofZ(Zi,
                           getCoefficients())));

11               compileOutput(String.format("\n%5d  %s  %s  %s", iterate,
                                    Function.displayComplex(Zi), Function.displayComplex(Zii),
                                 Function.displayComplex(Function.getFofZ(Zii,
                                 getCoefficients())));
12} while ( !(((Math.abs(Zi[0] - Zii[0]) < getTolerance()) &&
                        (Math.abs(Zi[1] - Zii[1]) < getTolerance()) ) || (iterate >=
                        maxIteration) ) );
13                                       compileOutput(String.format("\n\n%s\n\n", "Program output for Zo
                 = " + Function.displayComplex(getZo()) + ", tolerance = " +
                  getTolerance()));
14                           return Zii;
15                }
```

Fig. 2: Code Listing 2 - *The getComplexRoot()* method of the *Newton* class

4.4.1 Verification and Result

Again, when the function $f(x) = x^3 + 2x^2 - x + 5 = 0;$ from [2] was solved for complex roots, the results obtained [2] are given next.

Example 2a: Finding the complex roots of $f(x) = x^3 + 2x^2 - x + 5 = 0$, starting with $X_O = 1 + i$, and tolerance of 1E-6 using Newton's method (as implemented by [2])

ITR No	Xi	Xi+1	$F(X_{i+1})$
1	1.0000000 + 1.0000000i	0.4862385 + 1.0458716i	1.3182709 + 0.5860965i
2	0.4862385 + 1.0458716i	0.4481399 + 1.2366549i	-0.0711541 - 0.1660429i
3	3 04481399 + 1.2366549i	0.4627205 + 1.2224248i	0.0015675 - 0.0013566i
4	0.4627205 + 1.2224248i	0.4629258 + 1.2225399i	-0.0000001 + 0.0000003i
5	0.4629258 + 1.2225399i	0.4629258 + 1.2225399i	0.0000000 + 0.0000000i

The result obtained using the current implementation of the Newton's method for finding complex roots is given next.

Example 2b: Finding the root of $f(x) = x^3 + 2x^2 - x + 5 = 0;$ starting with $X_O = 1 + i$, and tolerance of 1E-6 by the Newton's method [Current implementation of Code Listing 2] (Approximate root found: 0.4629258 + 1.2225399i)

ITR No	Xi	Xi+1	$F(X_{i+1})$
1	1.0000000 + 1.0000000i	0.4862385 + 1.0458716i	1.3182709 + 0.5860965i
2	0.4862385 + 1.0458716i	0.4481399 + 1.2366549i	-0.0711541 - 0.1660429i
3	3 0.4481399 + 1.2366549i	0.4627205 + 1.2224248i	0.0015675 - 0.0013566i
4	0.4627205 + 1.2224248i	0.4629258 + 1.2225399i	-0.0000001 + 0.0000003i
5	0.4629258 + 1.2225399i	0.4629258 + 1.2225399i	0.0000000 + 0.0000000i

Program output for X_O = 1.0000000 + 1.0000000i, tolerance = 1.0E-6

5.0 Conclusion

Computational physics utilizes available programming resources in great deal. The relevance that computational physics, numerical analysis or computational science in general has today, is therefore as a result of a lot of work that had been done in the implementation of several computational methods using computer programming languages. [11] extensively implemented computational methods using FORTRAN 90/95. Dennis Ritchie in the 1960s [1] developed C, another language that has found extensive use in computational physics, most suitable for High Performance Computing (HPC) because of its speed of execution though very susceptible to errors, especially if used by a not so skillful programmer. In the present world

driven by technology, scientific computing is fast becoming the third pillar of scientific inquiry alongside the more traditional theory and experimentation pillars. Scientists today do not have to brave the risks of hazardous or dangerous chemical

experiments but rather use computational methods implemented with programming languages such as Java to simulate and model such experiments.

This work has implemented Newton's method by numerically determining real and complex roots of equations using Java. It also reviewed similar implementations of Bisector and Regula Falsi methods for real roots using Java and other techniques. Implementations of the numerical computation of the methods show that Newton method converges faster with greater accuracy than any of the other two methods. This work has thus lent its own contribution to the set of available implementations, most especially using Java.

6.0 References

[1] Deitel, P.J. and Deitel, H.M. (2007). Java: How to Program. Pearson Education Inc., New Jersey, USA.

[2] Gerald, C.F.and Wheatley, P.O. (1999).Applied Numerical Analysis. Dorling Kindersley, India.

[3] Pang, T. (2006). Introduction to Computational Physics. Cambridge University Press, New York, USA.

[4] Stroud, K.A., and Booth, D.J (2001).Engineering Mathematics. Palgrave Macmillan, New York, USA.

[5] Stroud, K.A., and Booth, D.J (2003).Advanced Engineering Mathematics. Palgrave Macmillan, New York, USA.

[6] Makinde, V.,Akinboro, F.G., Okeyode, I.C., Mustapha, A.O., Adesina, O.S., Coker, J.O. (2012). Implementation of the Root Bisection Computational Physics Method for the Determination of Roots of Non-Linear Equations using Java. *Journal of Natural Sciences, Engineering and Technology, JNSET: 11(2) 2013 ISSN: 2277-0593 (Print); 2315-7461 (Online)* pp 73–86 *Published by Federal University of Agriculture, Abeokuta, Nigeria* http://www.funaab.edu.ng; unaabasset@yahoo.com; journals@funaab.edu.ng

[7] Makinde, V.,Akinboro, F.G., Okeyode, I.C., Mustapha, A.O., Coker, J.O., Adesina, O.S. (2013). Implementation of the False Position (Regula Falsi) as a Computational Physics Method for the Determination of Roots of Non-Linear Equations using Java. *Nature and Science Nat Sci. 11(6) ISSN: 1545-0740* pp 113-119*Published by Marsland Press, University of Michigan, USA Available Online at http://www.sciencepub.net/nature.since April 2013*

[8] Kreyszig, E. (2006). Advanced Engineering Mathematics 9th Edition. John Wiley & Sons Inc., New York, USA

[9] Adesina, O.S. (2010). Implementation of Basic Computational Physics Methods using Java. Unpublished B.Sc. Project, Federal University of Agriculture, Abeokuta, Nigeria

[10] Hoffman, J.D. (2001). Numerical Methods for Engineers and Scientists Second Edition (Revised and Expanded) Marcel Dekker Inc., New York, USA

[11] Chapman, S.J (1998). FORTRAN 90/95 for Scientists and Engineers. McGraw-Hill, USA