

**KARADENİZ TECHNICAL UNIVERSITY
THE GRADUATE SCHOOL OF NATURAL AND APPLIED SCIENCE**

COMPUTER ENGINEERING GRADUATE PROGRAM

**DESIGN AND IMPLEMENTATION OF A GENERAL INTERPRETER FOR
STEP-BY-STEP SOLVING OF NONLINEAR SYSTEM OF EQUATIONS USING
SYMBOLIC APPROACHES**

MASTER THESIS

Computer Eng. Mohamed Yusuf HASSAN

**SEPTEMBER 2017
TRABZON**



KARADENİZ TECHNICAL UNIVERSITY
THE GRADUATE SCHOOL OF NATURAL AND APPLIED SCIENCES

COMPUTER ENGINEERING GRADUATE PROGRAM

**DESIGN AND IMPLEMENTATION OF A GENERAL INTERPRETER FOR
STEB-BY-STEP SOLVING OF NONLINEAR SYSTEM OF EQUATIONS USING
SYMBOLIC APPROACHES**

Computer Eng. Mohamed Yusuf HASSAN

**This thesis is accepted to give the degree of
"MASTER OF SCIENCE"**

**By
The Graduate School of Natural and Applied Sciences at
Karadeniz Technical University**

The Date of Submission : 18/08 /2017

The Date of Examination : 05/09 /2017

Supervisor : Asst. Prof. Dr. Hüseyin PEHLİVAN

Trabzon 2017

KARADENİZ TECHNICAL UNIVERSITY
THE GRADUATE SCHOOL OF NATURAL AND APPLIED SCIENCES

COMPUTER ENGINEERING GRADUATE PROGRAM
Mohamed Yusuf HASSAN

**DESIGN AND IMPLEMENTATION OF A GENERAL INTERPRETER FOR
STEB-BY-STEP SOLVING OF NONLINEAR SYSTEM OF EQUATIONS USING
SYMBOLIC APPROACHES**

Has been accepted as a thesis of

MASTER OF SCIENCE

after the Examination by the Jury Assigned by the Administrative Board of
the Graduate School of Natural and Applied Sciences with the Decision Number 1716 dated
22 /08 / 2017

Approved By

Chairman : Asst. Prof. Dr. Hüseyin PEHLİVAN

.....*Hüseyin*.....

Member : Assoc. Prof. Dr. Mustafa ULUTAŞ

.....*M. Ulutaş*.....

Member : Assoc. Prof. Dr. Ahmet BARAN

.....*A. Baran*.....

Prof. Dr. Sadettin KORKMAZ
Director of Graduate School

FOREWORD

This thesis is written as completion to the master of computer engineering, at Karadeniz Technical University. The subject of this thesis is focused on finding the roots of nonlinear functions with n unknowns using a hybrid symbolic-numeric approach.

I would like to offer my heart thanks to my master thesis supervisor, Asst. Prof. Dr. Hüseyin PEHLIVAN, for much support, guidance, and patient reading of my drafts and help make it sense. I am also grateful to other members of my thesis committee, Assoc. Prof. Dr. Mustafa ULUTAŞ and Assoc. Prof. Dr. Ahmet BARAN for their valuable feedback and encouragement.

I would like to thank my parents for their love, constant encouragement and wonderful emotional support. I also like to thank my young brother who was there for me and gave me a great motivation to finish this work.

This thesis would have taken far longer to complete without the encouragement from many other friends. The days would have passed far more slowly without the support of them.

Mohamed Yusuf HASSAN

Trabzon 2017

THESIS STATEMENT

I declare that, this Master Thesis, I have submitted with the title “Design and Implementation of a General Interpreter for Step-by-Step Solving of Nonlinear System of Equations Using Symbolic Approaches” has been completed under the guidance of my Master supervisor Asst. Prof. Dr. Hüseyin PEHLİVAN. All the data used in this thesis are obtained by simulation and experimental works done as part of this work in our research labs. All referred information used in thesis has been indicated in the text and cited in reference list. I have obeyed all research and ethical rules during my research and I accept all responsibility if proven otherwise. 05/09/2017

Mohamed Yusuf HASSAN

TABLE OF CONTENTS

	<u>Page No</u>
FOREWORD.....	III
THESIS STATEMENT.....	IV
TABLE OF CONTENTS	V
SUMMARY	VIII
ÖZET	IX
LIST OF FIGURES	X
LIST OF TABLES	XI
LIST OF ABBREVIATIONS	XIII
1 GENERAL INFORMATION	1
1.1. Introduction	1
1.2. Literature Review	3
1.3. Nonlinear Equations	7
1.3.1. Nonlinear Equations Introduction	7
1.3.2. Methods for Solving Nonlinear Equations	8
1.3.3. Newton-Raphson Method.....	9
1.3.3.1. Graphical Depiction of Newton-Raphson Method.....	10
1.3.3.2. Derivative of Newton-Raphson Method.....	10
1.3.3.2.1. Derivative of Newton-Raphson Method using Graph.....	10
1.3.3.2.2. Derivative of Newton-Raphson Method of 1-Dimension System.....	11
1.3.3.2.3. Derivative of Newton-Raphson Method of N-Dimension System.....	12
1.4. Language Processors	14
1.4.1. Compilers	14
1.4.2. Interpreters.....	15
1.4.3. Mixed Compilation and Interpretation Systems	16
1.4.4. Comparison between Compilers and Interpreters	17
1.4.5. Basic Compiler Phases	17
1.4.5.1. Front-End.....	17
1.4.5.2. Back-End	18

1.5.	Mathematical Expressions Interpretation	18
1.5.1.	Lexical Analyser (Scanner)	19
1.5.2.	Syntax Analyser (Parser)	21
1.5.2.1.	Context-Free Grammar	22
1.5.2.2.	Derivation and Parsing Context-Free Grammar	23
1.5.2.3.	Parser Tree	24
1.5.2.3.1.	Ambiguity	24
1.5.2.3.2.	Left Recursion	25
1.5.2.3.3.	Left Factoring	26
1.5.3.	Semantic Analyser	27
1.5.4.	Symbol Table	27
1.6.	Parsing Techniques	28
1.6.1.	Top-Down Parsing	28
1.6.1.1.	LL Parsers	29
1.6.1.2.	First and Follow Sets	30
1.6.1.2.1.	First Sets	31
1.6.1.2.2.	Follow Sets	31
1.6.2.	Bottom-Up Parsing	33
1.6.2.1.	LR Parsers	33
1.7.	Automatic Parser Generator Tools	34
1.7.1.	JavaCC	35
1.8.	Parser Tree Evaluation	36
1.8.1.	Instance of Operator	36
1.8.2.	Interpreter () Methods	36
1.8.3.	Visitor Design Pattern	37
2.	STEP-BY-STEP SOLUTIONS FOR NONLINEAR SYSTEM OF EQUATIONS	39
2.1.	Introduction	39
2.2.	General Structure of the Implemented Mathematical Expression Interpreter ..	39
2.2.1.	Lexical Analysis	42
2.2.1.1.	Token Declaration	42
2.2.2.	Syntax Analysis	43
2.2.2.1.	Syntax Structure of Nonlinear Equations	44

2.2.2.2.	Generating Abstract Syntax Tree	46
2.2.3.	Semantic Analysis	47
2.2.4.	Nonlinear Equations Evaluation Methods	48
2.2.5.	Newton-Raphson Implementation	49
2.2.5.1.	Partial Derivative	50
2.2.5.2.	Function Evaluations	51
2.2.5.3.	Transformation of Nonlinear Equations into Linear Equations	52
2.2.5.4.	Solving Linear Equations using Cramer's Rule	52
2.2.5.5.	Newton-Raphson Iterations and Stopping Criteria	53
2.2.5.6.	Simplification	54
2.2.5.7.	Printing Solution Values and Intermediate Steps	57
3.	APPLICATION OF THE METHODOLOGY	59
3.1.	Source Data Format	59
3.2.	Step-by-Step Solving of the Given Nonlinear Equation Application	59
3.2.1.	Analysis Phase of the Application	60
3.2.2.	Interpretation Phase of the Application	62
4.	RESULTS AND DISCUSSIONS	66
5.	CONCLUSION	68
6.	FUTURE WORKS	70
7.	REFERENCES	71
CURRICULUM VITAE		

Master Thesis

SUMMARY

DESIGN AND IMPLEMENTATION OF A GENERAL INTERPRETER FOR STEP-BY-STEP SOLVING OF NONLINEAR SYSTEM OF EQUATIONS USING SYMBOLIC APPROACHES

Mohamed Yusuf HASSAN

Karadeniz Technical University
The Graduate School of Natural and Applied Sciences
Computer Engineering Graduate Program
Supervisor: Asst. Prof. Dr. Hüseyin PEHLİVAN
2017, 75 Pages

In this work, we present the design and implementation of an interpreter program for the step-by-step numerical solutions of non-linear systems of equations with multiple variables, using symbolic computation methods and automatic code generation tools. The development process starts with a representation of a nonlinear system of equations in a formal language in terms of context-free grammars then, a parser which is generated via the JavaCC tool is used to represent the nonlinear system of equations in the form of object structures.

The numerical method Newton-Raphson are employed to obtain better approximations to solutions of nonlinear systems. The interpreter can easily be extended to cover other numerical methods, only describing the related iterative computation steps. On the other hand, integrating into their own interactive development environments, researchers can input any system of nonlinear equations directly into the interpreter and get the approximating solution as an output.

Key words: Symbolic computation, system of nonlinear equations, step-by-step Solution, Newton-Raphson method, parser, context-free grammars.

Yüksek Lisans Tezi

ÖZET

SİMGESEL YAKLAŞIMLARI KULLANARAK DOĞRUSAL OLMAYAN DENKLEM
SİSTEMLERİNİN ADIM-ADIM ÇÖZÜMÜ İÇİN GENEL BİR YORUMLAYICININ
TASARIMI VE GERÇEKLENMESİ

Mohamed Yusuf HASSAN

Karadeniz Teknik Üniversitesi
Fen Bilimleri Enstitüsü
Bilgisayar Mühendisliği Anabilim Dalı
Danışman: Yrd. Doç. Dr. Hüseyin PEHLİVAN
2017, 75 Sayfa

Bu çalışmada, simgesel hesaplama yöntemleri ve otomatik kod üretme araçlarını kullanarak birden fazla değişkene sahip doğrusal olmayan denklem sistemlerinin adım-adım sayısal çözümleri için bir yorumlayıcı programının tasarımı ve gerçekleştirilmesini sunarız. Geliştirme süreci, bağlamdan bağımsız gramerleri kullanarak bir biçimsel dilde doğrusal olmayan denklem sisteminin temsil edilmesi ile başlar. Daha sonra, JavaCC aracıyla üretilen bir ayrıştırıcı, doğrusal olmayan denklem sistemini nesne yapıları formunda temsil etmek için kullanılır.

Newton-Raphson sayısal yöntemi doğrusal olmayan sistemlerin çözümlerine daha iyi yaklaşımlar sağlamak için kullanılmıştır. Yorumlayıcı, sadece gerekli olan yinelemeli hesaplama adımları tanımlanarak, diğer sayısal yöntemleri kapsayacak şekilde kolayca genişletilebilir. Diğer yandan, etkileşimli geliştirme ortamlarına entegre ederek, araştırmacılar doğrusal olmayan denklem sistemlerini yorumlayıcıya girip çıktı olarak yaklaşık çözümü elde edebilir.

Anahtar Kelimeler: Sembolik hesaplama, doğrusal olmayan denklem sistemleri, adım adım çözüm, Newton-Raphson yöntemi, ayrıştırıcı, bağlamdan bağlamdan bağımsız gramerler.

LIST OF FIGURES

	<u>Page No</u>
Figure 1. General form of multivariate nonlinear system of equations.....	8
Figure 2. Nonlinear equation solvers.....	9
Figure 3. Graphical depiction of Newton-Raphson method.....	10
Figure 4. Compilation and execution process	15
Figure 5. Interpreter process.....	16
Figure 6. Mixed systems execution architecture.	16
Figure 7. Compiler phases.....	18
Figure 8. Process of mathematical expressions interpretation	19
Figure 9. Lexical analyser process.	20
Figure 10. Syntax analyser process.	22
Figure 11. Ambiguous parser tree of expression” $x + 6 * 7/5$ ".	24
Figure 12. Unambiguous parser tree of expression “ $x + 6 * 7/5$ ”	25
Figure 13. Parsing types	28
Figure 14. Top-Down parsing types	29
Figure 15. LL (k) definition.....	29
Figure 16. LL parser process example.....	30
Figure 17. Bottom-Up parsing types	33
Figure 18. LR parser process example	34
Figure 19. JavaCC file structure	35
Figure 20. Architecture of the implemented interpreter	41
Figure 21. Application interface (1).....	64
Figure 22. Application interface (2).....	65

LIST OF TABLES

	<u>Page No</u>
Table 1. Derivation of Newton-Raphson method using graph.....	11
Table 2. Example of tokens definition.....	20
Table 3. Stream of tokens of the expression " $percentageProfit = \frac{profit}{costPrice} * 100$ "	21
Table 4. Chomsky hierarchy.....	22
Table 5. Ambiguous grammar	23
Table 6. Unambiguous grammar example.....	25
Table 7. Left recursion grammar	26
Table 8. Eliminated left recursion grammar.....	26
Table 9. An example of LL (1) grammar	30
Table 10. Algorithm to compute first sets.....	31
Table 11. Algorithm to compute follow sets	32
Table 12. First and follow sets of the grammar in Table 8.....	32
Table 13. Comparison of syntax tree evaluation approaches	36
Table 14. Syntax classes with accept () methods	38
Table 15. JavaCC token declaration for the application.....	42
Table 16. Token sequence of the expression " $f(x,y) = x^2 + y - 3 = 0$ "	43
Table 17. LL (1) grammar definition for nonlinear equations	44
Table 18. JavaCC grammar definition for nonlinear equations	45
Table 19. Abstract syntax tree for the application.....	46
Table 20. Hash table usage for Newton-Raphson initial guess values.....	47
Table 21. Adding visitor and eval methods to the syntax classes.....	48
Table 22. Parsing input source data operation.....	49
Table 23. Partial derivatives of nonlinear equations	50
Table 24. Nonlinear function evaluations.....	51
Table 25. Solving linear equations using Cramer's rule.	53
Table 26. Newton-Raphson stopping criteria.....	54
Table 27. Simplification rules for some basic transformations.....	55

Table 28. Simplification methods implementation	56
Table 29. Print derived functions	57
Table 30. Print final solution values and intermediate steps	58
Table 31. Source data format.....	59
Table 32. Input source data of the application.....	60
Table 33. Token sequence of the given application source data	60
Table 34. The object tree for the input source data in Table 32.....	61
Table 35. Newton-Raphson interpretation process of the object tree in Table 34. (Iter. 1).	62
Table 36. Newton-raphson interpretation process of the object tree in Table 34. (Iter. 2)	63
Table 37. Input data of example(2) of the application	67
Table 38. Solution of example(2) of the application.....	67

LIST OF ABBREVIATIONS

AST	Abstract Syntax Tree
BNF	Backus–Naur Form
CAS	Computer Algebra System
CFG	Context Free Grammar
EBNF	Extended Backus-Naur Form
Eq	Equation
FORMAC	FORmula Manipulation Compiler
IR	Intermediate Representation
JVM	Java Virtual Machine
LL Parser	Left to right Left most derivation Parser
LR Parser	Left to right Right most derivation Parser

1. GENERAL INFORMATION

1.1. Introduction

In the past decades, a continual improvement of programming in computer science has enabled to develop useful software tools to solve human problems. Mathematics has a vital role in human life. The mathematical operations used in engineering applications cannot be accomplished by human hands, therefore, mathematical software tools for efficiently solving mathematical problems was developed. Mathematical software or scientific software is used for mathematical modelling and statistical analysis.

Generally, scientific software can be classified into computer algebra systems that are used for the symbolic evaluation of mathematical expression (e.g. Maple [2], Mathematica [3]) and numerical computation systems (e.g. MATLAB [4], GNU Octave [5]) that are widely used for engineering applications [1]. Numerical methods find an approximate solution to mathematical problems by using direct and iterative methods. Most of the numerical methods involve in many iteration calculations. Numerical computations often propagate errors from round-off and truncation. Symbolic computation methods have been developed against this disadvantage of numerical methods. Symbolic computation is the development and manipulation of mathematical expressions. Symbolic computation or algebra computation solves mathematical problems without error and finds the exact value using computer technology [6]. In this type of calculation, mathematical equations must be fully expressed before they can be processed and then transformed into algorithms that can be solved by computer programs [7, 8]. Computer Algebra System (CAS) or symbolic computation deals with mathematical expressions symbolically rather than numerically, results are exact, that there are no numerical errors. CAS can save both time and effort in solving a wide range of mathematical problems. More details on symbolic computation and its applications can be referred to the book by Cohen [6].

In this thesis, a hybrid method, symbolic-numeric computation, that combines symbolic and numeric methods, an interpreter software tool is implemented for step-by-step solving of a multivariate nonlinear system of equations. Symbolic approaches are used to analyse the source data and represent it in an intermediate structure for later evaluation.

Numerical computation method, Newton-Raphson iteration is employed to obtain better approximations to solutions of nonlinear systems.

With the rapid development of high-level programming languages that facilitate code writing processes, compilers have become indispensable tools for software developers. In computer programming, the translation and execution of programs are carried out in different ways. The first way is to use a compiler to read a source program written in a particular language (source code) and translate it into another equivalent language (machine code), reporting possible errors detected during the translation process. The output of the compiler is a machine code that can be run by the user with an input to produce an output. An interpreter is another language processor, which directly runs the source code instructions, instead of producing machine code (target language). The interpreter translates the source code statement by statement that makes easy to detect errors. Compilers are faster than interpreters because every time the program starts running, all the lines should be translated again in the interpreter. Automatic code generation tools have been developed to facilitate the steps of analysis, and parsing of source data generated by programming languages. There are a lot of compilers and interpreters developed with these tools called compiler compilers, some that can only generate code for source code in the Java language; ANTLR [9], SableCC [10], JTB [11], JavaCC [12], JLex [13], and JFlex [14]. For example, the source code to be generated by the JavaCC tool can be easily integrated with other software and serve as a handy analyser and parser components that can process the input data

The proposed system parses the source data, converts it into a tree data structure using a Context Free Grammar (CFG) and then interprets the intermediate code representation in the form of a tree structure into a final result. The methodology consists of three phases which are source data analysis to represent the source data with a tree data structure, numerical programming to simplify and solve expressions using symbolic approaches, and finally the evaluation of the expressions using the Newton-Raphson numerical method to get and print the final result of the equations.

In this thesis, solving of multivariate nonlinear equations is discussed. The thesis is organized as follows. In Section 1. Literature review, nonlinear equations, compilers, interpreters, grammars, parsing techniques, mathematical expression interpretation phases, and evaluation of expressions is discussed. In Section 2, Step-by-Step solving of nonlinear equations, the architecture of the system, and implementing Newton-Raphson methods are

presented. Application of the methodology are discussed in Section 3, results and discussions are presented in Section 4. In Section 5, the conclusion is presented and finally, in Section 6, future research and open research topics are presented.

1.2. Literature Review

In many scientific disciplines where mathematical problems need to be solved; the development of computer hardware and software that produce very fast and error-free solutions for the complex problems encountered in physics, computer science, mathematics, chemistry, engineering, astronomy, and biology had to wait. The lack of advanced software, many difficult problems remained unsolved [15].

The rapid development of science and technology has affected the lives of human beings in every field. In the 1950s the development of digital computers has enabled easy, fast and successful computations of different problems in many areas such as applied science and engineering. The most important use of the computer has been to do fast and errorless calculations to solve complex problems in different scientific disciplines, the availability of computers have been enabled to develop various systems, algorithms, techniques, and methods for this purpose. The impact of rapid development and high speed computers has increased the use of computer systems in mathematics field enabling the computation of mathematical problems both in numerically and symbolically; this rapid growth of computer systems in mathematics make possible to do many operations such as mathematics e-learning, comparing the efficiency between two algorithms, and developing new algorithms for automatic mathematical solution.

The state of art in this work, symbolic computation and root value finding operations are discussed. Although symbolic computing has been used in computers since 1953, it has a long history in terms of its use in scientific development [16]. We have come a long way since then many scientific software tools to computer mathematical problems have been developed. We will focus on the most recent developments in symbolic and root finding methods, the beginning, and the early developments in this field.

The technology is changing the way in which secondary schools and university education are delivered. Many e-learning systems to teach and support mathematical learning have been created, Coursera [18] and Edventure [19] provided for automatic

assessment, intelligent mathematical learning systems such as algebra problem generation [20], and automatic solution assessment [21] has also been developed [17].

There are many algorithms developed for the computerized solution of mathematical problems. Euclid's algorithm is the most common algorithms for finding the greatest common divisor of integers found by Euclid in the 3rd century and this algorithm is among the most basic algorithms of the symbolic computation systems used today in generalizations [22]. However, the finding of the roots of polynomial equations, the investigation of algorithmic solution methods of derivative, integral and differential equations has been the subject of symbolic computation.

Decomposing factorization of polynomials into a product of irreducible factors has a long history. The first algorithm for decomposing univariate polynomials over integers into products was found by Schubert in 1793. In 1882 this algorithm was extended to polynomials with algebraic coefficients by Kronecker. The Schubert and Kronecker algorithms were very slow even for computers, In 1967 Berlekamp developed a fast algorithm for factorization of polynomials over the finite fields into products [23], the Berlekamp algorithm is an important factor (Factorization of Polynomials). As a result of his work on the Berlekamp algorithm, Zassenhaus showed in 1969, products over the integers obtained by this algorithm can be used to obtain products over the integers [24]. In 1975-76, Musser [25] Wang and Rothschild [26] developed similar methods for multivariate and algebraic coefficient polynomials.

Symbolic computation systems began in the early 1960's, it has attracted the attention of many mathematicians and developers to design and develop new systems in the future in this field, Association for computing Machinery (ACM) special interest group on symbolic and algebra manipulation (SIGSAM) was formed in 1965 to bring come together different researchers to publish the recent development in algebraic algorithms and their applications, in the next ten years the old systems were revised and new systems were developed, these systems have added a new ability to scientific computing by providing exact mathematical computation without error and manipulation of mathematical expressions [27].

There are various applications and technologies developed in the literature related to computation on the computer. In the beginning of 1950s, after the invention of the electronic computer, there were some challenging factors to design symbolic computation systems such as slow speed and small storage computers. J.F. Nolan from the

Massachusetts Institute of Technology [28] and the other by H. G. Kahrmanian at Temple University [29] has developed automatic computation applications for analytical differentiation in their graduate thesis.

In 1958, Lisp programming language was invented by John McCarthy and he published its design in the paper [30]. Lisp is the most common and longest living language. Lisp is also the second oldest high-level programming language after FORTRAN that is older than one year. Lisp enables to make many operations such as computing with symbolic expressions rather than numbers and symbolic expressions representation as list structures in the memory of the computer. Lisp Language has played a very important role in the growth of symbolic computation. Symbolic automatic INTEGRATION (SANIT) was the first program to calculate the symbolic integration problems in calculus and it was written by Slagle in 1961 with Lisp language, this substance was developed as a doctoral dissertation at the Massachusetts Institute of Technology developed as a doctoral study [31].

After Lisp language invention, it was understood that mathematical problems can be solved using personal computers with symbolic computation, after the recovery of this advantage from Lisp language, in the 1960s, the symbolic computations has shown a rapid growth development.

In 1963 Jean E. Sammet developed by FORMAC, FORMula MANipulation Compiler (FORMAC) was an early experimental programming system that had the capability of handling formal of mathematical expressions such as computation, manipulation, and use of symbolic expressions on the computer and it was built in FORTRAN language [32].

CAS software is a software package that capable of doing the symbolic computation. CAS applications represent mathematical expressions symbolically and operate on these symbolically represented objects. CAS can be separated into general purposes which provide computing facilities for general mathematical problems and specific purposes which give special uses for algebraic and special mathematical areas.

The first general purpose symbolic computation systems were developed at the end of the 1960s and early 1970s. These systems are, Reduce [33] in 1967, Macsyma [34] and Reduce 2 [35] in 1971, Scratchpad [36] in 1971, and muMATH [37] in 1979.

Macsyma based version called Maxima was developed in 1971 by Paul S. Wang. This system supports many operations such as differentiation, integration, ordinary

differential equations, rational, logarithmic, trigonometric expressions, linear equations, polynomials, Laplace transforms, matrices, and Taylor series [38].

Today, the major general purpose CAS systems include Maple [2] from University of Waterloo, Mathematica [3] from Wolfram research, SageMath [39] from William A. Stein, Axiom [40] from Richard Jenks, Maxima [24] from Massachusetts Institute of Technology researchers, Magma [41] from University of Sydney, and Symbolic Math Toolbox (MATLAB) [42] from Mathworks.

The major special purpose CAS systems include CoCoA-5 [43] for commutative algebra, Fermat [44] for polynomial and matrix computation, KANT/KASH [45] for algebraic number theory, and Macaulay2 [46] for algebraic geometry and commutative algebra.

In 2002, GiNac a special purpose system in C ++ environment was developed by Cristian Bauer to implement the symbolic computation and it was designed to handle multivariate polynomials, algebras, and other special functions [47].

In 2004, Hyungju Park defined that many problems in digital processing can be converted to algebraic problems and can be solved using algebraic and symbolic computation methods [48].

In 2013, Yavuz TEKBAŞ presented the graduate thesis entitled "CODE PRODUCTION TOOLS USING AUTOMATIC CALCULATION OF DERIVATIVES AND SIMPLIFICATION MATHEMATICAL EXPRESSIONS" [49]. In this work, A CFG is developed for syntactic and semantic structures of mathematical expressions, JavaCC an automatic code generation tool was used to generate abstract syntax tree (AST) as an object tree, and finally evaluating object tree was handled to simplify and derive the expressions.

In 2015, Mir Mohammad Reza Alavi Milani conducted in his doctoral dissertation, grammar based methodologies for automatic generation and step-by-step solving of mathematical expressions [17]. In this work, CAS like system, grammar-based methodologies that were organized into two parts was developed. The first part was designed a methodology that solves mathematical expressions step-by-step, and in the second part was designed for the production of new questions using template expressions.

In 2016, Baki GÖKGÖZ presented a graduate thesis entitled as "DESIGN AND IMPLEMENTATION OF A GENERAL INTERPRETER FOR NUMERICAL ROOT FINDING METHODS USING SYMBOLIC APPROACHES" [50]. In this work, A CFG

is developed for syntactic and semantic structures of mathematical expressions, an automatic code generation tool of JavaCC was used to generate AST as the object tree, and finally the object tree was simplified before its evaluation and then the root finding methods were used to find roots of equations. In this thesis, the designed application can find roots for only equations that have only one variable.

In this thesis, Newton-Raphson method was used for finding multiple roots of multivariate nonlinear equations [51]. There are many papers devoted to iterative methods for root solver, the literature for previously implemented algorithms see, e.g., [52–62].

As a result, many problems have been solved throughout the history of symbolic computation and it has been proved that some problems cannot be resolved algorithmically and some problems need further research to get better performance and result.

With the emergence and growth of symbolic computing tools, mathematicians began using these tools to do proofs of theorems with computers and save time for mathematical operation by hand. In later periods, these tools began to be used in high schools and universities in support of mathematics education. Apart from the studies given here, many studies have been done on computer science related to symbolic and algebraic computations such as coding, modelling, computer animations, signal or image processing.

1.3. Nonlinear Equations

Nonlinear Equations are very important in science and engineering fields. They have many real world applications. The solution of nonlinear equations is one of the most difficult problems in scientific computation [63].

1.3.1. Nonlinear Equations Introduction

An equation related to a straight line is called linear equation, for example

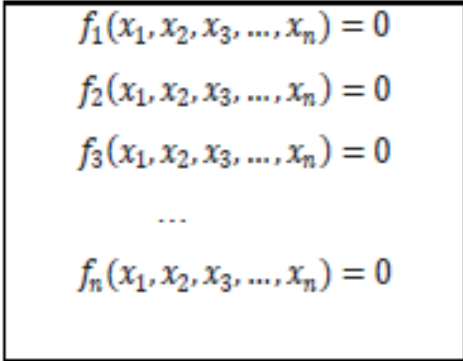
$$f(x) = mx + c \tag{1}$$

The equation (Eq. 1) describes a straight line with slope m and the linear equation $f(x) = 0$, involving such an f , is easily solved to give $x = -c/m$ ($m \neq 0$). If the function $f(x) = 0$ is not a linear equation (do not relate a straight line) then it's called

nonlinear equation. Algebraic equations and transcendental equations are nonlinear equations.

Given a continuous function $f(x)$, finding the value of x_1 such that $f(x_1) = 0$ is called root finding problems, if x_1 satisfies the equation of $f(x_1) = 0$ then x_1 is the root of the function $f(x_1) = 0$ or we can say x_1 is a zero of the function of f .

The System of nonlinear equations is a set of simultaneous equations with multi variable unknowns. Nonlinear equations may have just one solution, no solutions, or many solutions.



$$\begin{aligned} f_1(x_1, x_2, x_3, \dots, x_n) &= 0 \\ f_2(x_1, x_2, x_3, \dots, x_n) &= 0 \\ f_3(x_1, x_2, x_3, \dots, x_n) &= 0 \\ &\dots \\ f_n(x_1, x_2, x_3, \dots, x_n) &= 0 \end{aligned}$$

Figure 1. General form of multivariate nonlinear system of equations

1.3.2. Methods for Solving Nonlinear Equations

Finding an exact solution to nonlinear equations is very difficult because the change of input is not proportional to the change of output [64]. Nonlinear systems may appear chaotic, unpredictable or counter-intuitive.

Several ways are possible to solve System of nonlinear equations. We can divide them into three main methods which are an analytical method, graphic method, and numerical methods.

Numerical methods can be further classified into bracketing and open methods, many methods are available to solve nonlinear equations such as Bisection method, Newton's method, secant method, fixed point iterations, and Muller's method. In this thesis, we will use Newton's method.

In bracketing methods, the method starts with an interval that contains the root and the solution is obtained in the smaller interval containing the root.

In the open methods, the method starts with one or more good initial guess points. In each iteration, a new guess of the root is obtained.

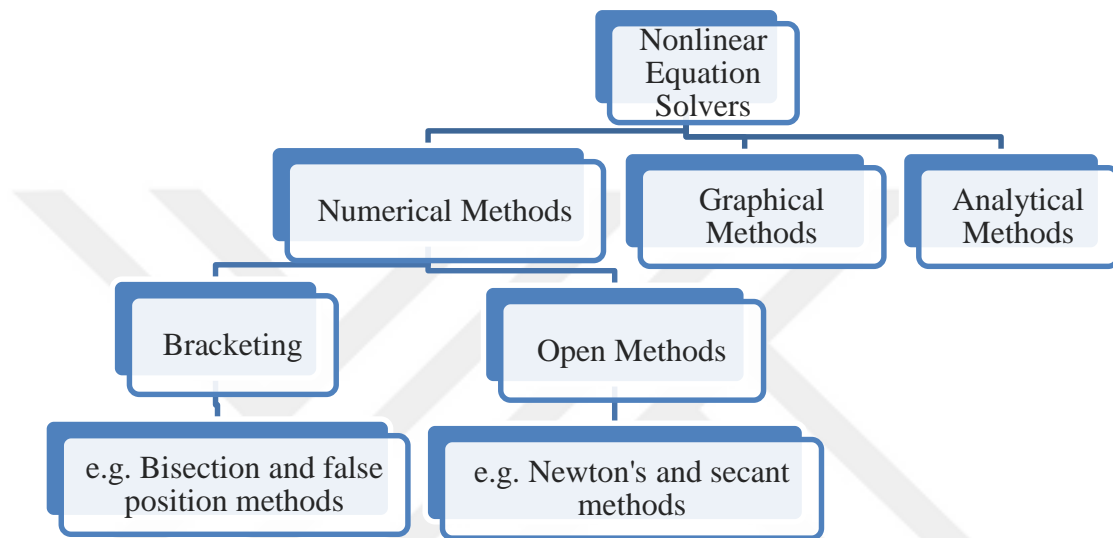


Figure 2. Nonlinear equation solvers

1.3.3. Newton-Raphson Method

Newton-Raphson method also known as Newton's method is the most widely used method to solve a nonlinear equation. It is based on Taylor series expansion. Given an initial guess of the roots of (x_0, \dots, x_n) Newton Raphson method uses information of the given function and its derivative at that point to find better guess of the root. The Newton Raphson formula is showed in Eq. (2) where k, i is the number of iteration is.

$$x_i^{k+1} = x_i^k - \frac{f(x_i^k)}{f'(x_i^k)} \quad (2)$$

1.3.3.1. Graphical Depiction of Newton-Raphson Method

Assumptions:

- Given $f(x)$ is continuous and the first derivative of f is known.
- Given an initial guess of x_0 such that $f'(x_0) \neq 0$. Where f' denotes the first derivative of f .

Let's assume at x_i is the initial guess, then a tangent to the function of x_i that is $f'(x_i)$ is extrapolate down to the x -axis to provide an estimate of the root x_{i+1} .

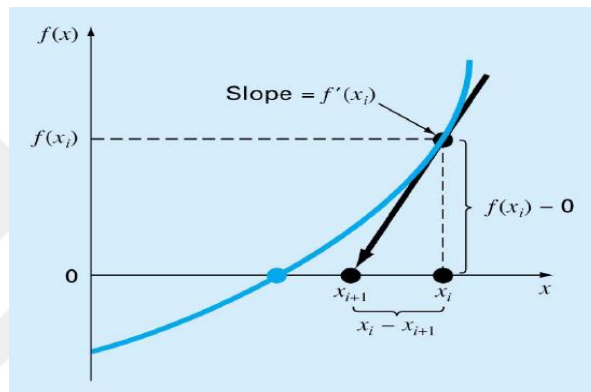


Figure 3. Graphical depiction of Newton-Raphson method

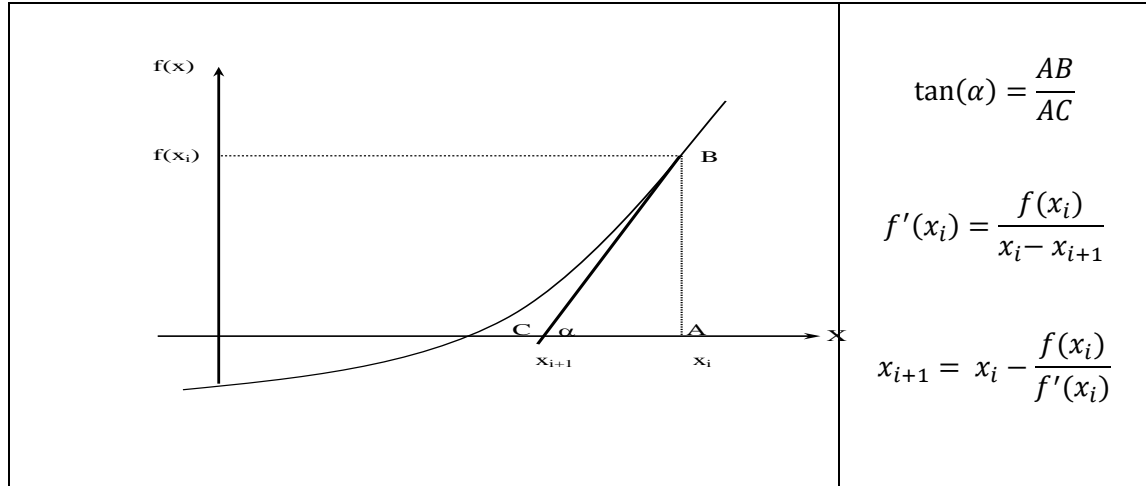
1.3.3.2. Derivation of Newton-Raphson Method

1.3.3.2.1. Derivation of Newton-Raphson Method Using Graph

Assumptions:

- Given $f(x)$ is continuous and the first derivative f' is known.
- Given an initial guess of x_0 such that $f'(x_0) \neq 0$. Where f' denotes the first derivative of f .

Table 1. Derivation of Newton-Raphson method using graph



1.3.3.2.2. Derivation of Newton-Raphson Method of 1-Dimension System

Consider Taylor series expansion of $f(x)$ at the value $x = x_0$.

$$f(x) = f(x_0) + f'(x_0)(x - x_0) + \frac{f''(x_0)}{2!}(x - x_0)^2 + \dots \quad (3)$$

First approximation to the root of the equation $f(x) = 0$ can be found using only the two first terms of the expansion in Eq. (3), obtaining the Eq. (4).

$$f(x) = 0 \approx f(x_0) + f'(x_0)(x_1 - x_0) \quad (4)$$

First approximation is given by:

$$x_1 = x_0 - f(x_0)/f'(x_0).$$

Second approximation is given by:

$$x_2 = x_1 - f(x_1)/f'(x_1),$$

And third approximation is given by:

$$x_3 = x_2 - f(x_2)/f'(x_2),$$

This iteration procedure can be generalized by writing the below equation, where i is the iteration number.

$$x_{i+1} = x_i - f(x_i)/f'(x_i).$$

We start to check the Taylor series after each iteration, the program should check to see if the convergence condition is satisfied.

1.3.3.2.3. Derivation of Newton-Raphson Method of N-Dimension System

We start the Taylor series expansion of n variables to obtain a form of Newton Raphson method. E.g., we start two-dimensional system to obtain values of x_1 and x_2 . Given the below equations:

$$f_1(x_1, x_2) = 0 \quad (5)$$

$$f_2(x_1, x_2) = 0$$

We extend f_1 and f_2 equations into Taylor series into two dimension with respect to iteration k .

$$f_1(x_1^{k+1}, x_2^{k+1}) = f_1(x_1^k, x_2^k) + \frac{\partial f_1(x_1^k, x_2^k)}{\partial x_1}(x_1^{k+1} - x_1^k) + \frac{\partial f_1(x_1^k, x_2^k)}{\partial x_2}(x_2^{k+1} - x_2^k) + \dots \quad (6)$$

$$f_2(x_1^{k+1}, x_2^{k+1}) = f_2(x_1^k, x_2^k) + \frac{\partial f_2(x_1^k, x_2^k)}{\partial x_1}(x_1^{k+1} - x_1^k) + \frac{\partial f_2(x_1^k, x_2^k)}{\partial x_2}(x_2^{k+1} - x_2^k) +$$

After the first derivative terms, resulting equation expressed in terms of Newton correction.

$$\Delta x_1^k = (x_1^{k+1} - x_1^k) \quad (7)$$

$$\Delta x_2^k = (x_2^{k+1} - x_2^k)$$

LHS of equation (6) are zero, gives

$$\frac{\partial f_1(x_1^k, x_2^k)}{\partial x_1} \Delta x_1^k + \frac{\partial f_1(x_1^k, x_2^k)}{\partial x_2} \Delta x_2^k = -f_1(x_1^k, x_2^k) \quad (8)$$

$$\frac{\partial f_2(x_1^k, x_2^k)}{\partial x_1} \Delta x_1^k + \frac{\partial f_2(x_1^k, x_2^k)}{\partial x_2} \Delta x_2^k = -f_2(x_1^k, x_2^k)$$

Eq. (8) can be generalized in $n \times n$ case in Jacobian matrix form.

$$J\Delta x = -f \quad (9)$$

Where J is the Jacobian matrix and it can be generalized as:

$$J_{ij} = \frac{\partial f_i}{\partial x_j}$$

$$\Delta x = -J^{-1}f$$

Eq. (9) is Newton's Method for $n \times n$ system, can be written in this form

$$\begin{bmatrix} \frac{\partial f_1(x_1^k, x_2^k, \dots, x_n^k)}{\partial x_1} & \frac{\partial f_1(x_1^k, x_2^k, \dots, x_n^k)}{\partial x_2} & \dots & \frac{\partial f_1(x_1^k, x_2^k, \dots, x_n^k)}{\partial x_n} \\ \frac{\partial f_2(x_1^k, x_2^k, \dots, x_n^k)}{\partial x_1} & \frac{\partial f_2(x_1^k, x_2^k, \dots, x_n^k)}{\partial x_2} & \dots & \frac{\partial f_2(x_1^k, x_2^k, \dots, x_n^k)}{\partial x_n} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial f_n(x_1^k, x_2^k, \dots, x_n^k)}{\partial x_1} & \frac{\partial f_n(x_1^k, x_2^k, \dots, x_n^k)}{\partial x_2} & \dots & \frac{\partial f_n(x_1^k, x_2^k, \dots, x_n^k)}{\partial x_n} \end{bmatrix} \begin{bmatrix} \Delta x_1^k \\ \Delta x_2^k \\ \vdots \\ \Delta x_n^k \end{bmatrix} = - \begin{bmatrix} \frac{\partial f_1(x_1^k, x_2^k, \dots, x_n^k)}{\partial x} \\ \frac{\partial f_2(x_1^k, x_2^k, \dots, x_n^k)}{\partial x} \\ \vdots \\ \frac{\partial f_n(x_1^k, x_2^k, \dots, x_n^k)}{\partial x} \end{bmatrix} \quad (10)$$

$$x_n^{k+1} = \Delta x_n^k + x_n^k \quad (11)$$

Where k is the iteration number and $k = 0, 1, 2, \dots, n$

After each iteration, the program should check to see if the convergence condition is satisfied. A stopping criteria is required for the iterations. E.g.

$$\Delta x_i^k < \varepsilon_i, i = 1, 2, \dots, n$$

$$\Delta x_i^{k+1} - \Delta x_i^k < \varepsilon_i, i = 1, 2, \dots, n$$

$$f_i(x_i^k) < \varepsilon_i, i = 1, 2, \dots, n$$

$$f_i(x_i^{k+1} - x_i^k) < \varepsilon_i, i = 1, 2, \dots, n$$

Where ε is a pre-specified error tolerance.

Comparing to other numerical methods, Newton-Raphson method is a faster and better method to find roots of nonlinear equations with a condition that is required to fulfil; a good initial guess near to the root is given to converge to the root point. If the initial

guess is not a good guess or close to the inflection point of the function, Newton-Raphson method diverges away from the root. When the initial guess is close enough to a simple root of the function then Newton's method is guaranteed to converge quadratically. Quadratic convergence means that the number of correct digits is nearly doubled at each iteration.

1.4. Language Processors

The Computer is capable of executing instructions of object code. An object code is a string of binary bits (0, 1) stored in computer's memory. A human can't understand low-level languages. They write source program in one of the high-level languages. Therefore, it's necessary to find a tool that maps between high-level languages and low-level languages. Language processor is a software program that maps an input of a source language into an output a target language. There are different types of language processors such as assemblers, compilers, pre-processors, interpreters, and disassemblers. In this part, we will explain the most widely used language processors which are compilers and interpreters.

1.4.1. Compilers

The Compiler is a type of language processor that translates a source code of high-level language into a machine code of a low-level language to create an executable program. Delphi, Visual, C/C++, COBOL, and Java languages are examples of such language processor.

Modern compilers are used to generate codes that are platform dependent from any source code. In the process of creating these codes, the compilers create intermediate codes that dependent on environment.

In compilers, the amount of memory used by the underlying code is small and the fast operation is considered as compilers do variable allocation without need variable lookup at runtime. Therefore, compilers perform code optimization in order to get the best efficiency from the generated code and exploit hardware features. In case of errors in the source program, compilers catch and report these errors in the compilation process.

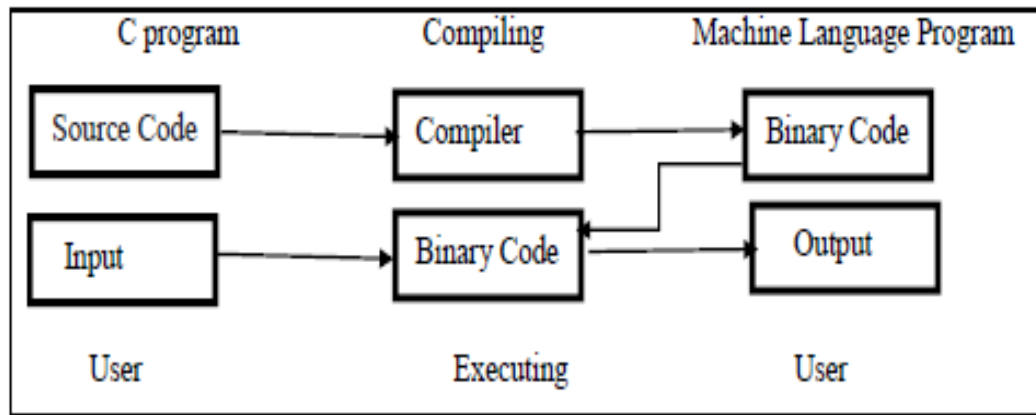


Figure 4. Compilation and execution process

1.4.2. Interpreters

Interpreters are another type of language processor, such as compilers, but the goal of interpreters are different from compilers. Interpreters, instead of translating the source program into a machine code (target program), it interprets each command in user's program into a result. No target program is saved, instead, it analyse the user commands and produces a binary code for the voltages carrying out operation of the computer hardware component. Interpreters; run source code in a programming language, translate and represent it into intermediate code, and execute and interpret the compiled code when it arrives. Examples of interpreter languages include Python, Ruby, Lisp, and Pascal.

Interpreters use one of three strategies for program execution [65].

- 1) Parsing the source code and performs directly its behaviour. E.g., Lisp Programming
- 2) Translate the source program into an intermediate representation (IR) then execute this IR. E.g., Python, MATLAB, PHP, and JavaScript.
- 3) Mixed type, the compiler is a part of the interpreter system, and the interpreter execute precompiled source code made by the compiler. UCSD Pascal is an example of this type.

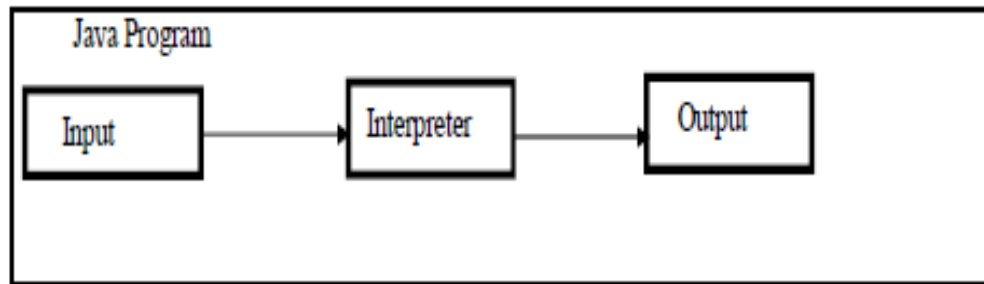


Figure 5. Interpreter process

1.4.3. Mixed Compilation and Interpretation Systems

There are some mixed systems which combines both compilers and interpreters. Compiler translates source program in a high-level language into an intermediate code, then the interpreter executes the intermediate code to low-level language. Java language which combines the two strategies of compilation and interpretation is an example of such system. Source files in Java languages compiles into Java Virtual machine (JVM) bytecodes, then these JVM bytecodes can be interpreted over the JVM on the hosting computer. Frequently executed code is loaded by Just-in-time compiler which enables to optimize the code and then runs that code to increase the interpretation speed.

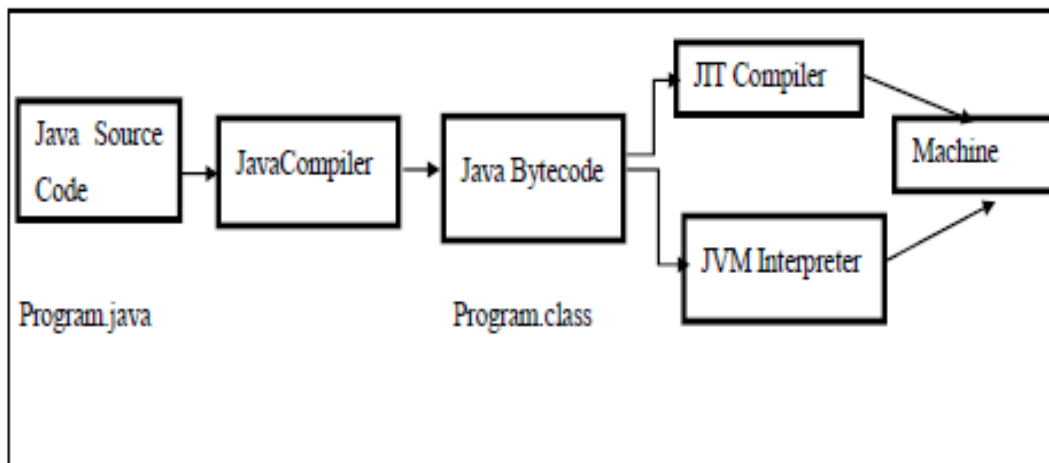


Figure 6. Mixed systems execution architecture

1.4.4. Comparison between Compilers and Interpreters

Compilers and interpreters do the same job in general, translating source program into target program but there are many difference between them.

- Compilers creates machine code for later evaluation but interpreters don't, instead they directly maps a source program into a result.
- Compilers translates all source code at once no need for re-compilation unless we change the source code while interpreters are very slow because every time the programs starts running, all the source code lines should be translated again.
- Every time the source program is starting, the interpreter needs to be loaded into memory, this causes the source code to allocate less space in memory. In the compiler, more space is allocated to the source code in memory, the compiler loads only the source code at compile time, and it is not loaded at the next run time.
- Running programs translated into machine language faster than programs interpreted.
- Interpreters perform better error detection than compilers, the reason for this is that the compiler shows a list of many errors in the whole program while interpreter shows errors in each instruction. In compilers, some mistakes that cannot be detected at compile time can be detected at the time of interpretation.
- Making changes to the code of compiled programs is slower and more difficult than modifying the code of interpreted programs.

1.4.5. Basic Compiler Phases

Compilation process can be grouped into two phases, front-end and back-end.

1.4.5.1. Front-End

Front-end reads the source program and analysis it. It is primarily dependent on the source language. The front end phase consists of lexical analysis, syntax analysis and semantic analysis, and intermediate code representation, sometimes some of the code optimization can be done in the front end.

1.4.5.2. Back-End

Back-end generates target program. It is independent of the source code and is dependent on the target code. The back end, however, includes code generation and code optimization.

Symbol table and error handler is part of the compilation process.

Compilers and interpreters can share the same front-end but they have different back-end as the output the two is different.

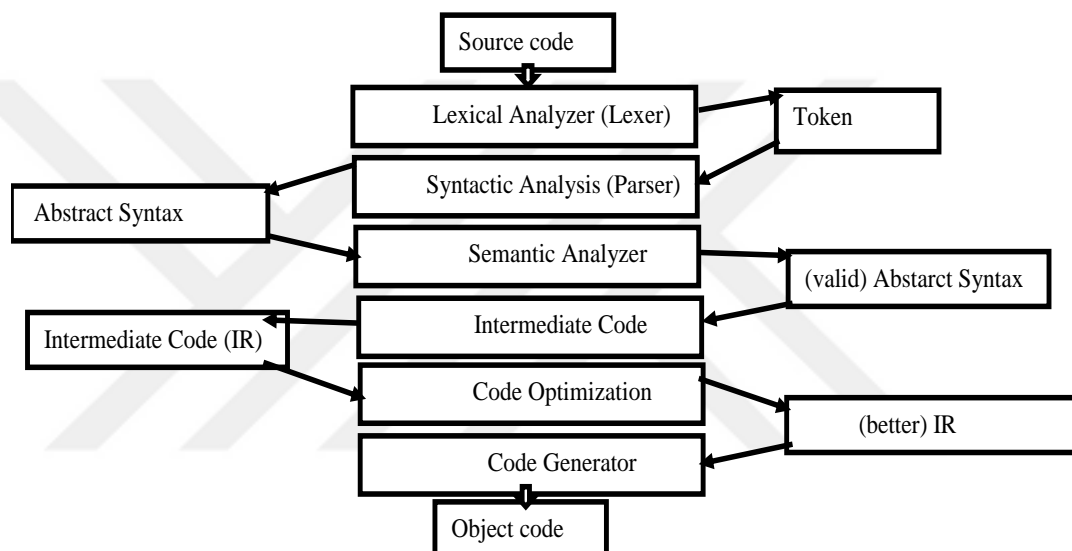


Figure 7. Compiler phases

1.5. Mathematical Expressions Interpretation.

Mathematical expressions contain all mathematical symbols such as numbers, operators (like add and subtract), functions (like sin and cos), constants, and variables (like x and y) in algebraic operations. Mathematical expressions computation in the computer is needed for efficiency computation and time-saving. Mathematical expressions are given based on the structure and rules of the interpreters as input data to the interpret software, the output of this expression is the result of the mathematical expression.

Mathematical expressions interpretation is the process of analysing the source program of mathematical expressions to represent it in an intermediate code and evaluating

the IR of the mathematical expressions to print a final. The interpretation contains two phases which are analysis and execution phases. The analysis phase consists lexical analysis, syntax analysis, and semantic analysis. The analysis phase produces an intermediate code representation as an output in the form of tree data structure. The next phase is the execution of the IR code into a final result, this phase translates the IR code by evaluating the data tree structure and then print the result. The general process of interpreting mathematical expressions is given in Figure 8.

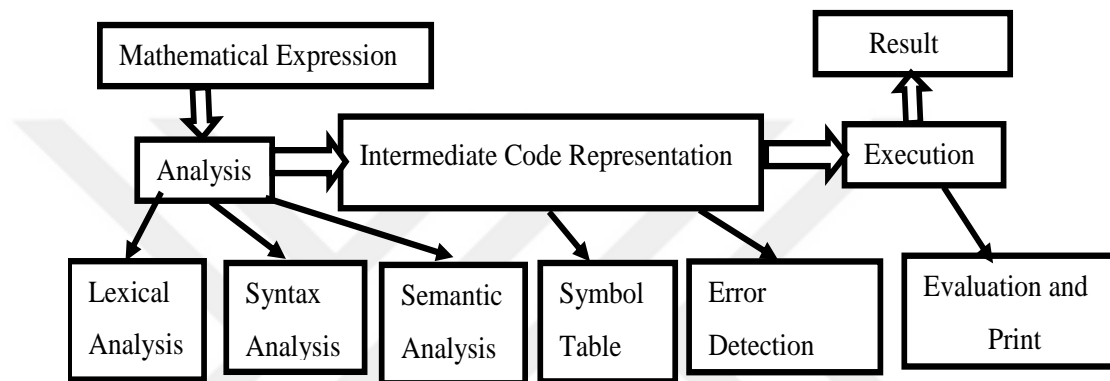


Figure 8. Process of mathematical expressions interpretation

1.5.1. Lexical Analyser (Scanner)

The task of lexical analyser also called lexer or scanner is to read a stream of characters from a source program, converts it into series of tokens by grouping the characters into a meaningful sequences called lexemes and removing any white space or comments in the source program [66]. Lexical analyser breaks up the program into a sequence of pieces called tokens in accordance with the word structure of the relevant programming language. Tokens, which is a stream of characters, are traditionally written using regular expressions. The pattern in Table 3, is an example of predefined regular expression rules that identify the lexeme to be a valid token or not. In programming languages, integral literals, string literals, keywords, identifiers, symbols, punctuations, and operators are types of tokens. Lexical analysis reports error if the token is invalid. Lexical errors include misspellings of identifiers, operators, or keywords. The output of lexical analyser is a stream of tokens that are passed to the parser for syntax analyser. Lexical

analyser output is a token of the form (token name, attribute-value), the first component token-name is an abstract symbol that is used during syntax analysis, and the second component attribute-value points to an entry in the symbol table for this token. Information from the symbol-table entry is needed for semantic analysis and code generation [66]. To produce lexical analysers from a regular expression description of the tokens of a language a software tool that is called scanner generator is used.

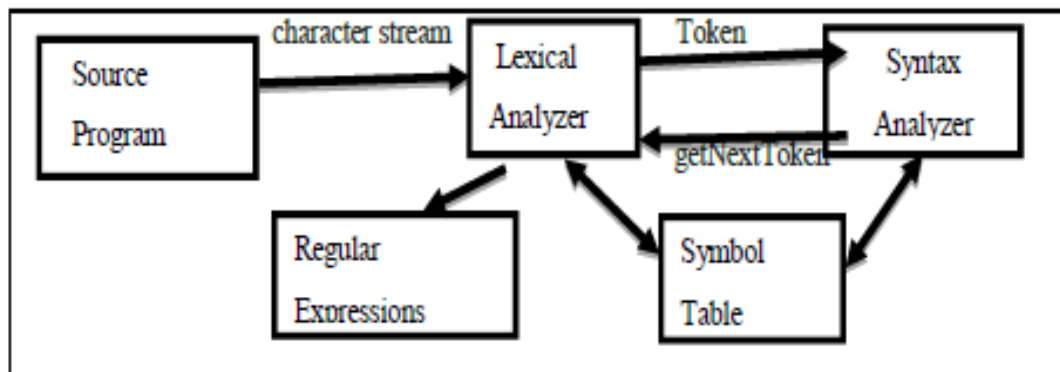


Figure 9. Lexical analyser process

There are several types of token such as variables, numbers, and keywords, each of these expressions is defined by its own regular expression. Further reading for regular expressions can be referenced by Mogensen, and Torben Ægidius book [67].

Table 2. Example of tokens definition

TOKEN	REGULAR EXPRESSION OF TOKEN DEFINITION
ID	<code>["a" - "z"] (["a" - "z"] ["0" - "9"]) *</code>
NUMBER	<code>(["0" - "9"] + ("." ["0" - "9"] +) ? ></code>
ASSIGN	<code>"="</code>
TIMES	<code>"*"</code>
DIVIDE	<code>"/"</code>

For example, we have the below equation as source program, the lexical analyser will read the source program as a stream of characters and breaks up into pieces of a stream of tokens as in Table 3.

$$\text{percentageProfit} = \frac{\text{profit}}{\text{costPrice}} * 100 \quad (12)$$

Table 3. Stream of tokens of Eq. (12)

ID, ASSIGN, ID DIVIDE, ID, TIMES, NUMBER
--

The identifier profit is a lexeme that would be mapped into a token (ID, 2), where ID is an abstract symbol standing for identifier and 1 points to the symbol table entry for profit. The symbol table entry for an identifier holds information about the identifier, such as its name and type. The assignment symbol = is a lexeme that is mapped into the token (ASSIGN). We have omitted the second component since this token needs no attribute-value. The remaining tokens follow the same logic of the previous two tokens.

1.5.2. Syntax Analyser (Parser)

The next phase is syntax analysis (also called parser). The parser has two main tasks; first parser checks the grammatical structure of the stream of tokens produced by the lexical analyser and then generates a data structure called parser tree or syntax tree. The output of syntax analyser is an intermediate representation also called AST. The interior nodes of this tree represent an operation and the children of the nodes represent the arguments of the operation produced by the lexical analyser.

Formal grammars are used to specify the syntax of the representative language of equations. The Backus Norm Form (BNF) presents a special notation for CFG describing the syntax of formal languages. A formal language is a set of strings over a finite set of symbols in programming languages and natural languages, which can be defined by a context-free grammar given according to the specifications of parser generator tool [68]. A software tool that is called parser generator is used to automatically produce syntax analysers from a grammatical description of a programming language. JavaCC tool is an example of parser generators and it will be explained in the end of this section. In addition

according to the specified syntax rules, the syntax analysis must also reject invalid tokens by reporting syntax errors.

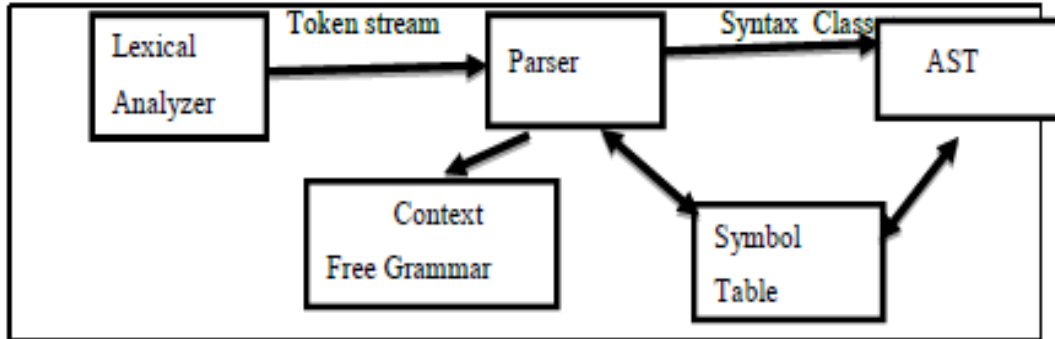


Figure 10. Syntax analyzer process

1.5.2.1. Context-Free Grammar

The hierarchical structure of programming languages is defined by the combination of its syntax and semantics and it is described by grammars. Semantics add meaning and interpretation to the syntax of programming language structure.

In 1956, Noam Chomsky formalized generative grammars and classified them into four types that are called Chomsky hierarchy [69]. In general, CFG is a more powerful notation than regular expression. Every regular expression is context free grammar and every construct that can be described by a regular expression can be described by a grammar, but not vice-versa. In Table 4, we defined the four types of Chomsky hierarchy. CFG is type-2 grammar that is used to describe the syntax structure of most programming languages. Syntactic categories are statements, expressions, and declarations. These are defined by rules of the form $A \rightarrow \gamma$. A is a nonterminal and γ is a string of terminals and nonterminal (it can be empty). CFG is a set of production rules that is used to generate the parse tree.

Table 4. Chomsky hierarchy

Type-0	Recursively enumerable grammars
Type-1	Context-sensitive grammars
Type-2	Context-free grammars
Type-3	Regular grammars

CFG are made up of finite set of grammar rules and G it is a 4-tuple (N, T, P, S) which are nonterminal (syntactic categories of sets sentences), terminals (the basic symbols from which sentences are formed), production rules (rules specifying how the terminals and nonterminal combine to form sentence) , and starting symbol respectively.

- N is a finite set of non-terminal symbols.
- T is a finite set of terminals where $N \cap T = \text{NULL}$.
- P is a set of rules, $P: N \rightarrow (N \cup T)^*$, i.e., the asterisk represents the Kleene star operation.
- S is the start symbol, it is used to represent the whole sentence. It must be an element of N .

In Table 5, E , and F , are nonterminal. Terminals are `id`, `num`, and `()`. Start symbol is E . One Production rule is $E \rightarrow E + E$.

Table 5. Ambiguous grammar

$E \rightarrow E + E$	$E \rightarrow F$
$E \rightarrow E - E$	$F \rightarrow \text{num}$
$E \rightarrow E * E$	$F \rightarrow \text{id}$
$E \rightarrow E / E$	$F \rightarrow (E)$

1.5.2.2. Derivation and Parsing Context-Free Grammar

Derivation is carried out starting with the start symbol, and repeatedly replacing any non-terminal on the right-hand side. There are two types of derivation methods. The first method is the leftmost derivation, in each step of derivation, apply production to leftmost nonterminal. The second derivation is the right most derivation and in each step of derivation, apply production to rightmost nonterminal. For every leftmost derivation, there is a right most derivation, and vice versa.

1.5.2.3. Parser Tree

A parser serves as a tree constructor that transforms the token sequence in the source data into an object tree in accordance with the syntactic structure [70]. CFG can be represented using a parser tree. Each internal node is labelled by a nonterminal. Each leaf is terminal symbol. The construction of a parse tree can be made by rewriting the production rules. Beginning with the start symbol, each rewriting step replaces a nonterminal by the body of one of its Productions.

There are some issues that we have to deal with when parsing CFG to generate parse tree or AST such as ambiguity, recursive rules, and left factoring.

1.5.2.3.1. Ambiguity

Ambiguity is happened when it is possible to derive several parse trees for the same string. Ideally there should be only one parse tree for each string (unambiguous). A grammar is said to be ambiguous if it generates an ambiguous string.

Example: if we have a source input of mathematical expression

$$x + 6 * 7/5 \quad (13)$$

The grammar sentence of expression (13) according straight line programming grammar rules is as following. $id + num * num/num$.

The parser tree of expression (13) has different parse tree in the leftmost derivation and in the rightmost derivation therefore; it is an ambiguous as shown Figure 11. The problem of ambiguity is that there is no high operator precedence in mathematical expressions.

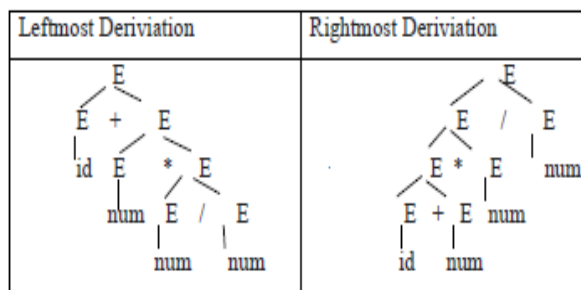


Figure 11. Ambiguous parser tree of expression" $x + 6 * 7/5$ "

We can rewrite ambiguous grammar in Table 5 to remove ambiguity as in Table 6:

Table 6. Unambiguous grammar example

$E \rightarrow E + T$	$T \rightarrow T * F$	$F \rightarrow \text{num}$
$E \rightarrow E - T$	$T \rightarrow T / F$	$F \rightarrow \text{id}$
$E \rightarrow T$	$T \rightarrow F$	$F \rightarrow (E)$

Now if we try to derive expression (13), it will have one parse tree both in leftmost derivation and in rightmost derivation.

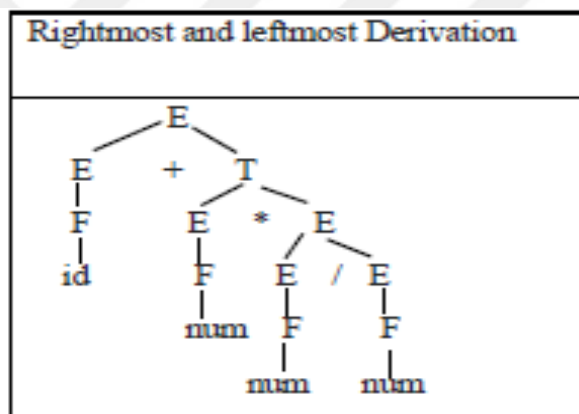


Figure 12. Unambiguous parser tree of expression "x + 6 * 7 / 5"

1.5.2.3.2. Left Recursion

In general, when the function calls its self is said to be recursion. A grammar is said to have direct left recursion if the leftmost symbol on the right side of a production rule is the same as the non-terminal on its left side, for example, $A \rightarrow A\alpha$ or indirect left recursion if it can be made itself some sequence of substitutions, for example $A \rightarrow \beta A\alpha$ where β can be yielded in an empty string. Left recursion is problem because parser cannot make decision which rule should be applied as rules have common symbol. Top-down parsers can go infinite recursion in case of left recursion. Example, In Table 7, the grammar has left recursive because expression E has three different production rules that

start the same symbol of T and the parser can't make an immediate decision about which rule has to be read.

Table 7. Left recursion grammar

$E \rightarrow E + T$		$E + T$		T
$E \rightarrow T * F$		$T * F$		F
$F \rightarrow (E) \mid \text{id} \mid \text{num.}$				

We can eliminate the left recursion by re-writing production rules without changing the syntax structure of the grammar. As shown in Table 8. Sometimes we may have hidden left-factors when a rule may not appear to have left factor.

Table 8. Eliminated left recursion grammar

$E \rightarrow TE'$
$E' \rightarrow + TE' \mid -TE' \mid \epsilon$
$T \rightarrow FT'$
$T' \rightarrow *FT' \mid /FT' \mid \epsilon$
$F \rightarrow (E) \mid \text{id} \mid \text{num}$

1.5.2.3.3. Left Factoring.

Sometimes two productions have the same start symbol. Production rules have common left factor that define in terms of themselves, removing the common left factor that appears in two productions of the same nonterminal is called left factoring.

Example

$$A \rightarrow fB \mid fC \quad (14)$$

Where A, B, C are non-terminals and f is a terminal. In this case, the parser will be confused as to which of the two productions to choose and it might have to back-trace. After left factoring, the grammar is left factored

$$A \rightarrow fL \quad (15)$$

$$L \rightarrow B \mid C$$

1.5.3. Semantic Analyser.

After the construction of a parser tree, the interpreter uses semantic analyser to cover the meaning of a program by analysing its parse tree. Semantic analysis checks whether the syntax structure constructed in the source program derives any meaning or not. Semantic analysis is performed during run time of the program.

In semantic analysis, many operations such as type checking, scope resolution, array-bound checking, and subroutines arguments calls is performed. Semantic errors include type mismatch, undeclared variable, arithmetic errors, and multiple declaration of variable. At the time these errors is detected an exception about the error is raised. Example `int a = "5";` cannot be detected during lexical and syntax analysis.

1.5.4. Symbol Table

Symbol table is a data structure used to record information about the identifiers used in the program and various attributes of identifiers. It stores type, scope, storage location, procedure name, return type, and other relevant information. Symbol tables information is collected and created in the analysis phase and later it used by the evaluation phase (synthesis phase) of the interpreter (compiler). Example the token x , in symbol table its stored information such as $(x,1)$ where x is the token-name and 1 is the attribute value that point the entry of this token in the symbol table. Detailed information about symbols is referenced [66].

1.6. Parsing Techniques

Syntax analysers follow the syntax structure defined by means of a context-free grammar. The way the structure syntax is implemented (derivation) divides parsing into two types: top-down parsing and bottom-up parsing [71].

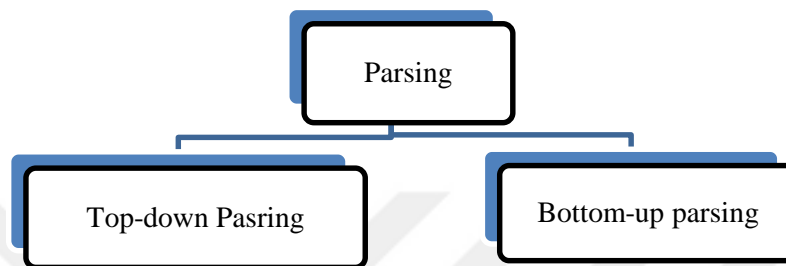


Figure 13. Parsing types

1.6.1. Top-Down Parsing

Top-Down Parsing constructs parse tree for input sentence by parsing from the start symbol using leftmost derivation to the input sentence. The key objective of Top-Down parsing is determining which production rule to be applied for a nonterminal. Recursive parsing is a top-down parser that uses recursive procedures to process the input sentence and determine the production rule to be applied. Recursive descent may require back-tracing or may not require. Predictive parsing is a recursive descent parser that doesn't require back-tracing, the problem is a recursive parser which suffers from back-tracing, means a bad pack, the production rule is not matched and fails, it starts again to process the input using different rules of the same production. In general, Top-Down parsers can't handle left recursion and left factoring, therefore; we have to eliminate the left recursion and left factoring before the parsing begins.

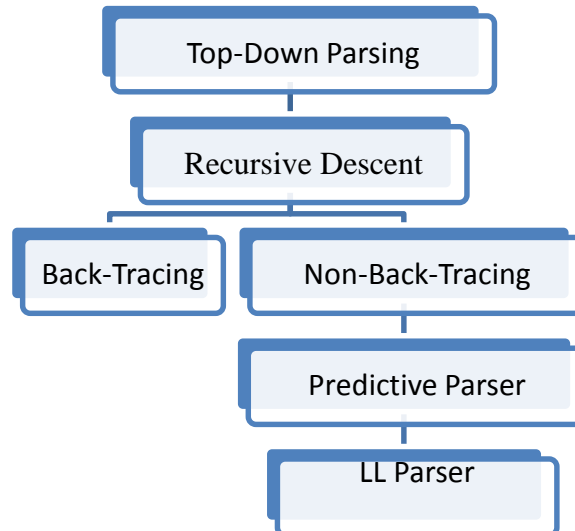
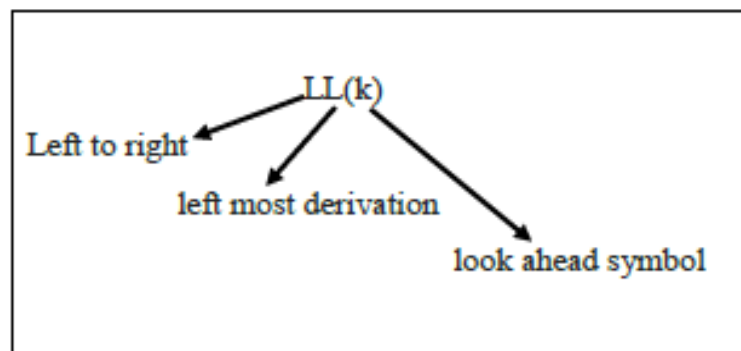


Figure 14. Top-Down parsing types

1.6.1.1. LL Parsers

LL (k) parsers analyse the input from left to right performing left most derivation, where k is the number of tokens when parser parses a sentence. When $k = 1$ the grammar is called LL (1) grammar. LL (1) grammar will not work with left recursion and left-factoring grammars because the grammar can read one symbol at one time.

Figure 15. LL (k) definition

The grammar in Table 9 is LL (1) grammar, the parser can read enough information from right-hand side symbols to choose which production rule is to be applied.

Table 9. An example of LL (1) grammar

$S \rightarrow T$
$S \rightarrow (S * T)$
$S \rightarrow \text{number}$

Given Grammar		
•	$S \rightarrow E$	
•	$E \rightarrow T + E$	
•	$E \rightarrow T$	
•	$T \rightarrow \text{number}$	
LL(2) parser for input "num+num+num"		
Production	Input	Action
S	num + num + num	Predict $S \rightarrow E$
E	num + num + num	Predict $E \rightarrow T + E$
T + E	num + num + num	Predict $T \rightarrow \text{num}$
num + E	num + num + num	Match num
+ E	+ num + num	Match +
E	num + num	Predict $E \rightarrow T + E$
T + E	num + num	Predict $T \rightarrow \text{num}$
num + E	num + num	Match num
+ E	+ num	Match +
E	num	Predict $E \rightarrow T$
T	num	Predict $T \rightarrow \text{num}$
num	num	Match num
		Accept

Figure 16. LL parser process example

1.6.1.2. First and Follow Sets

Predictive parsing requires the grammar should have left recursion or left factoring to enable that the first terminal symbol of each production right-hand side provides enough

information to choose a production, therefore; predictive parsing requires two functions to aid getting this information.

1.6.1.2.1. First Sets

$FIRST(\alpha)$

If α is a string of non-terminals and terminals then $FIRST(\alpha)$ is the all terminal symbols that begin any string derived from α . If $\alpha \rightarrow \epsilon$ the ϵ is included $FIRST(\alpha)$.

For $X\alpha_1$ and $X\alpha_2$, $FIRST(\alpha_1)$ and $FIRST(\alpha_2)$ cannot be overlapped.

Table 10. Algorithm to compute first sets

<ol style="list-style-type: none"> 1. If A is a terminal then First(A) is A! 2. If $A \rightarrow \epsilon$ is a production then add ϵ to first(A) 3. If A is nonterminal and there is a Production $A \rightarrow B_1B_2, \dots, B_k$ then add First($B_1B_2.. B_k$) to first(A) 4. First($B_1B_2.. B_k$) is one of the following cases. <ol style="list-style-type: none"> 1. First(B_1) if First(B_1) doesn't contain ϵ 2. If First(B_1) does contain ϵ then First($B_1B_2.. B_k$) is everything in First(B_1) except for ϵ as well as everything in First($B_2.. B_k$)
--

1.6.1.2.2. Follow Sets

$FOLLOW(A)$

For nonterminal A, if A can derive empty string $FOLLOW(A)$ is defined to be the set of terminals that can appear immediately to the right of A.

$A \rightarrow Ab\beta\alpha$. Here b is $FOLLOW(A)$

$A \rightarrow A\beta ab$. Here if α and β are produced an empty string then b is $FOLLOW(A)$

\$ representing the input right end marker is in $FOLLOW(A)$

Table 12. First and follow sets of the grammar in Table 8

	Null?	First Sets	Follow Sets
	no	{ (id, num) }	{ \$,) }
E'	yes	{ +, -, ϵ }	{ \$,) }
T	no	{ (id, num }	{ +, -, \$,) }
T'	Yes	{ *, /, ϵ }	{ +, -, \$,) }
F	no	{ (, id, num }	{ *, /, +, -, \$,) }

Table 11. Algorithm to compute follow sets

1. First, put \$ (the end of input marker) in Follow(S) (S is the start symbol)
2. If there is a production $A \rightarrow DBF$, **then** everything in FIRST(F) except for ϵ is placed in FOLLOW(B).
3. If there is a production $A \rightarrow DB$, **then** everything in FOLLOW(A) is in FOLLOW(B).
4. If there is a production $A \rightarrow DBF$, where FIRST(F) contains ϵ , **then** everything in FOLLOW(A) is in FOLLOW(B).

1.6.2. Bottom-Up Parsing

Bottom-up construct parse tree for input sentence by parsing from the input using leftmost derivation to the start symbol. Shift-Reduce parsing is a technique used Bottom-UP parsers making a shift and reduce action. At the beginning of parsing, the stack is empty and the parsing process finishes with success if (EOF) the end of file marker is shifted. Stack action pushes the first input token top onto the stack and the reduce action is when the grammar right-hand side ($A \rightarrow BCD$) replaces left-hand side grammar using pop function (pop B, C, and D from top of the stack).

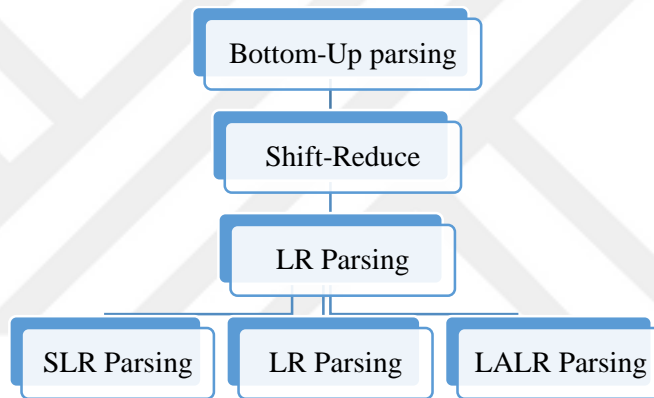


Figure 17. Bottom-Up parsing types

1.6.2.1. LR Parsers

LR also called LR(k) parsers analyse the input from left to right performing right most derivation, where k is the number of look ahead tokens symbols. LR parser is the most general bottom-up parsing methods. LR parsing starts with an empty stack and ends with the root of non-terminal on the stack.

Three algorithms for LR are LR (1) parsing, SLR (1) parsing, and LALR (1) parsing.

Simple LR (SLR) parsing is a simple that works on small size grammars and makes a fast construction. Look Ahead LR (LALR) works on intermediate size grammars. More information about LL and LR parsing algorithms can be referenced [72].

Given Grammar		
• $S \rightarrow E$		
• $E \rightarrow T + E$		
• $E \rightarrow T$		
• $T \rightarrow \text{number}$		
LR(1) parser for input "num+num+num"		
Workspace	Input	Action
-----	-----	-----
	num + num + num	Shift
num	+ num + num	Reduce $T \rightarrow \text{num}$
T	+ num + num	Shift
T +	num + num	Shift
T + num	+ num	Reduce $T \rightarrow \text{num}$
T + T	+ num	Shift
T + T +	num	Shift
T + T + num		Reduce $T \rightarrow \text{num}$
T + T + T		Reduce $E \rightarrow T$
T + T + E		Reduce $E \rightarrow T + E$
T + E		Reduce $E \rightarrow T + E$
E		Reduce $S \rightarrow E$
S		Accept

Figure18. LR parser process example

1.7. Automatic Parser Generator Tools

In this section, automatic parser generator tools are presented. A compiler-compiler generates scanner and parser for particular language from its grammar. Compiler-compiler takes grammars as input and produces a compiler as an output. As we mentioned before scanner and parser analysers use grammars to analyse the input data, these grammars should follow the compiler-compiler rules. There are many automatic code generation tools that are used to day. These tools are different depending the programming language code to be generated. Some of these tools are JavaCC [12], javaCup [74] for object oriented languages, and YACC [75], and bison [76] for imperative languages. In this thesis, we are using Java programming language so we will discuss automatic code generation tool for java Language and some other important tools.

1.7.1. JavaCC

JavaCC is a java-based parser generator and lexical analyser generator from context-free grammar and regular expressions respectively.

JavaCC uses a configuration file with the extension ".jj". This file starts with setting options. Among these options are the number of tokens to be looked at when making predictive production, whether to activate or deactivate the debug mode, and to specify the target folder of the files to be created.

Next, the body of the parser to be produced is defined. This can be done between the *PARSER_BEGIN* and *PARSER_END* tags by defining the main parser class. Any code to be added to this field will be recognized exactly as the class will be created by JavaCC.

As a third step, the skip character and the token list required for lexical analysis are defined using regular expressions. The token list is identified using the regular expressions, if necessary, under the *TOKEN* tag. Skipped characters are identified by the *SKIP* tag, and are usually space and end-of-line characters. These can be added to other characters according to the application.

```

options {
  DEBUG_PARSER = false;           // options included here
  LOOKAHEAD = 1;
}
PARSER_BEGIN(Parser_name)         // parser begin tag
public class Parser_name {        // classes
  -----                          // code input here
}
PARSER_END(Parser_name)           // parser end tag

TOKEN: { ----- }               // lexical rules is defined in terms of regular expressions
SKIP: { ----- }                 // skip words, spaces, and symbols defined
Grammar Rules                      // syntax structure is defined in terms of CFG

```

Figure 19. Javacc file structure

1.8. Parse Tree Evaluation Approaches

Syntax classes are implemented using the object-oriented concepts of Java in case of java programming language. As a result of syntactic analysis of the source data, syntax classes are used to generate object trees via the JavaCC generator tool. Each grammar rule is generally defined by a syntax class which is then used to evaluate mathematical expressions. The AST structure holds the essential sections of syntax classes in the form of a tree. An object tree can be evaluated using one of three approaches from the inner most nodes towards the root node. The three approaches are instanceof and type cast, dedicated methods, and visitor design pattern, In Table 13, comparison about each one's advantage and disadvantage are presented.

Table 13. Comparison of syntax tree evaluation approaches

Method	Object derivations	Class Compilation
Instanceof Operator	Yes	No
Interpreter Methods	No	Yes
Visitor Design Pattern	No	No

1.8.1. Instanceof Operator

In this method, the type (class or subclass or interface) of an object (instance) belonging to a node can be determined using the instanceof operator. In order to be able to perform the represented operation after determining the object type of the node, it is necessary to derive the subclass object from the super class. In this type, the disadvantages are that code constantly uses type cast and instanceof to determine the type of an object.

1.8.2. Interpreter () Methods

In this method, an interpreter () method is added to each class of the syntax, which can perform the operation represented by the class. To evaluate an object tree node, it is sufficient to call the eval () method of the object that contains the node. Therefore, it is not

necessary to determine the node object type to be evaluated. The disadvantages of this method are that for each new operation on objects, new dedicated methods have to be added and all classes must be recompiled.

1.8.3. Visitor Design Pattern

We can see from the comparison in Table 7 that visitor pattern is a good approach to evaluate the syntax tree compared to other parser tree evaluation approaches. Visitor Design Pattern is defined to operate on the object tree structure. Visitor pattern comes under behaviour pattern category, we use a visitor interface and a visitor class which changes the executing algorithm of an element class. When the visitor algorithm varies, the execution algorithm of the element also vary. As per the syntax class, element object has to accept the visitor object so that visitor object handles the operation on the element object. A visit () and accept () method is added to each syntax classes. To evaluate nodes of syntax tree visit () method is used. In this way, visit () and accept () methods call each other until all the nodes of the object tree are visited. The Visitor class serves as an interface with a visit () method declaration for each syntactic class. The definition of the visit () methods is done in a class that implements the Visitor interface. A different evaluation of syntactic class objects requires the definition of another Visitor interface.

In visitor design, we can add new operations on an object without touching the other code, visitor pattern also gathers same operations More detailed information about visitor design pattern is referenced by the book “Design Patterns: Elements of Reusable Object-Oriented Software” [73].

Table 14. Syntax classes with accept () methods

<pre>abstract class Exp { public abstract Object accept(Visitor v); } class Plus extends Exp { Exp a, b; public Plus(Exp x, Exp y) { a = x; b = y; } public Object accept(Visitor v) { return v.visit(this); } }</pre>	<pre>class Minus extends Exp { Exp a, b; public Minus(Exp x, Exp y) { a = x; b = y; } public Object accept(Visitor v) { return v.visit(this); } }</pre>
--	---

2. STEP-BY-STEP SOLUTIONS FOR NONLINEAR SYSTEM OF EQUATIONS.

2.1. Introduction

There is a need for the use of different computational methodologies and programming methods to solve mathematical problems with computer programs. In general, there are two main computational methods which are numerical and symbolic approaches.

Numerical methods find an approximate solution to mathematical problems. Numerical computations often propagate of errors from round-off and truncation. Symbolic computation methods have been developed against this disadvantage of numerical methods. Symbolic computation is the development and manipulation of mathematical expressions. Symbolic computation or algebra computation solves mathematical problems without error and finds the exact value using computer technology.

In this thesis, using a hybrid method with symbolic-numeric computation which combines symbolic and numeric methods, an interpreter software tool is implemented for step-by- step solving of a multivariate nonlinear system of equations. Symbolic approaches are used to analyse the source data and represent it in an intermediate structure for later evaluation. Numerical computation method, i.e., Newton-Raphson iteration is employed to obtain better approximations to solutions of nonlinear systems.

2.2. General Structure of the Implemented Mathematical Expression Interpreter

In this study, solving nonlinear system of equations undergoes two main phases, source data analysis, and interpreter phases. In the source data analysis, lexical analysis, syntax analysis, and semantic analysis are carried out. In case of any error in any of these stages, the expression which is the source of the error is reported without passing to the next stage. JavaCC tool is used for analysing the source data, which generates automatic source code in the Java programming language. A token generator takes the general source expression, analyses the source data using regular expressions, and transforms it into a token sequence. In the syntax analysis, JavaCC parser generates a syntax tree from the

token array according to an Extended Backus Naur Form (EBNF) grammatical form that is suitable for the syntactic and semantic structure of mathematical expressions and transformed into the LL (k) grammar, which is a left-to-right grammatical structure. In addition, syntax classes in the form of syntax tree are defined to represent operators and functions that can be included in a mathematical expression. The analysis phase produces an intermediate code representation as an output in the form of tree data structure.

The source expression is evaluated with the help of an interpreter developed based on the syntax tree (object trees). In the other stage, the evaluation process is started and the object tree is interpreted with the evaluation method of visitor design pattern. In the interpreter phase, many intermediate operations such as derivation, function transformations, matrix calculations, and simplification for the solution of a system of mathematical equations is done. The interpretation phase produces a final result by evaluating the object trees using visitor design pattern and shows all intermediate steps carried out to solve these expressions. Newton-Raphson method is implemented to obtain better approximations to solutions of nonlinear systems. In the Newton-Raphson method, functions are transformed into linear equations in the form of matrices equations. Cramer's rule is employed to solve these matrices and find new solutions. In the Newton-Raphson method, the root is not bracketed. In fact, some initial guesses of the root are needed to get the iterative processes started to find the roots of a system of equations. Convergence in open methods is not guaranteed but if the given initial guesses are good enough Newton-Raphson does convergence, faster compared to other numerical methods. All the programming processes from the source expression analysis and the interpretation of this source expression to the production of the final result are presented in detail. In the following sections, the General structure of solving nonlinear system of equations is shown in Figure 20.

To display the solution steps of a nonlinear system of equations, the expressions on the object tree after each evaluation is printed. The main components of the expression analyser and expression interpreter can be listed as follows:

- Lexical Analysis
- Syntax Analysis
- Semantic Analysis
- Nonlinear Equations Evaluation methods
- Implementing of Newton Raphson Method

- Partial Derivatives
- Function Evaluations
- Transformation of Nonlinear Equations into linear equations
- Solving linear Equations using Cramer's Rule
- Newton Raphson Iterations and Stopping criteria
- Simplification
- Printing Solution Values and Intermediate Steps

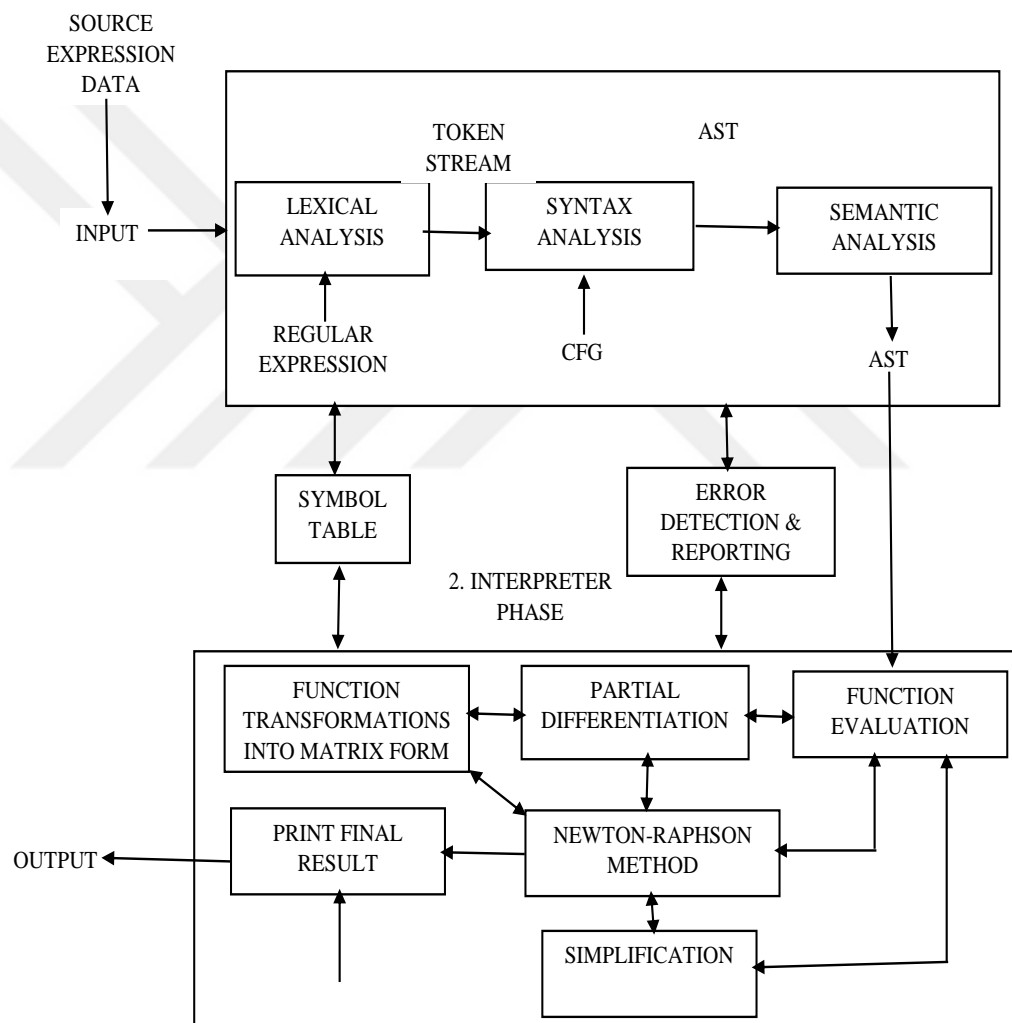


Figure 20. Architecture of the Implemented Interpreter

2.2.1. Lexical Analysis

The first step in the analysis process of the source program, which is taken a system of nonlinear equations, initial guess values and absolute error as the input data and the nonlinear equations are subjected to the lexical analysis. In this section, the source program is divided into a sequence of pieces called token which is suitable for the syntax structure of the Java programming language which is the coding language used in the application. Each possible token is defined by means of regular expressions. The compiler compiler JavaCC tool uses regular expressions for token definitions. The definitions are declared into the JavaCC tool in a file with the extension .jj.

2.2.1.1. Token Declaration

As we mentioned earlier, the JavaCC file consists of several subdivisions, such as options, the main parser body, token list in terms of regular expressions, skipped tokens such as white spaces, and syntax related methods in terms of CFG according to JavaCC rules. In this section, the token definitions for nonlinear equations are presented.

Table 15. JavaCC token declaration for the application

TOKEN: {			
<PLUS: "+">	<MINUS: "-">	<TIMES: "*">	<DIVIDE: "/">
<POWER: "^">	<ASSIGN: "=">	<COMMA: ",">	<SEMI: ";">
<LPAREN: "(">	<RPAREN: ")">	<SIN: "sin">	<COS: "cos">
<TAN: "tan">	<LOG: "log">	<LN: "ln">	<EXP: "exp">
<ID: (["a"- "z", "A"- "Z"](["a"- "z", "A"- "Z", "0"- "9"])*>			
<NUM :(["0"- "9"]+("."(["0"- "9"]+)?>			
}			
SKIP: { " " "\n" "\t" "\r" "\r\n" }			

Example: Given input source data (Nonlinear equations):

$$f(x,y) = x^2 + y - 3 = 0 ; \quad (16)$$

The characters could be grouped into the lexemes which are then mapped into a sequence of tokens passed on to the syntax analyser, as shown in Table 16.

Table 16. Token sequence of Eq. (16)

ID LPAREN ID COMMA ID RPAREN ASSIGN ID POWER NUM PLUS ID MINUS NUM ASSIGN NUM <SEMI>

Scanner reports errors in the case that invalid token is detected. Lexical errors include misspellings of identifiers, operators, or keywords. The output of scanner is passed to the parser for syntax analysis. Lexical analyser output is a sequence of pairs of the form (token name, attribute-value), the token-name is an abstract symbol that is used during syntax analysis, and the attribute-value points to an entry in the symbol table for this token. Information from the symbol-table entry is needed for semantic analysis.

2.2.2. Syntax Analysis

In general, every programming language needs predefined elements in a proper way and has a defined sequence of the language component combinations. The output of Lexical analyser is a sequence of predefined tokens (words, operators, symbols, functions) but it does not have a defined format. In this section, the syntax structure (format) of the tokens is declared using CFG.

The JavaCC parser, which performs syntax operations, has two main tasks; to perform the formal check of the syntax structure of the source input data and to carry out the production process of the object tree. For formal check, grammar types such as BNF and CFG, which define the syntax structure of the data, are used, and expressions that generate the object tree are added to these definitions. Using the JavaCC tool, it quite easy to define the syntax structure of nonlinear equations and generate abstract object trees for nonlinear equations.

Syntax classes and methods are defined for each rule of a grammar that represents the source data. After the grammatical structure is determined for the syntax analysis stage, this grammatical structure must be adapted to LL (k) because JavaCC works with the LL (k) algorithm. Therefore, in order to be able to comply with the LL (k) algorithm of the grammar structure, left factoring has been performed by eliminating the recursive states

from the left. After all these processes, the related grammatical structure was transferred to JavaCC environment and Java code is produced, which can perform syntax analysis processes for the developed application.

In this section, the grammar structure developed for the application and the JavaCC code structure, which generates automatic code in the Java programming language of this grammar are discussed.

2.2.2.1. Syntax Structure of Nonlinear Equations

In order to use the compiler compiler tool JavaCC, some operations have to be made on EBNF grammar such as eliminating the ambiguity and left factoring to produce an LL (1) grammar as shown in Table 17.

Table 17. LL (1) grammar definition for nonlinear equations

$G = \{ \Sigma, T, V, P, S \}$
$V = \{ \text{Eq}, \text{Eq2}, \text{ArgL}, \text{Exp}, \text{Term}, \text{Power}, \text{Elem}, \text{Func}, \text{Num}, \text{Id} \} \subseteq \Sigma$
$T = \{ \text{sin}, \text{cos}, \text{tan}, \text{log}, \text{Ln}, \text{exp}, (,), +, -, *, /, ^, =, , , ; \} \subseteq \Sigma$
$\Sigma = T \cup V$
$S = \{ \text{Eq} \}$
Productions
$\langle \text{Eq} \rangle \rightarrow \langle \text{Eq2} \rangle (; \langle \text{Eq} \rangle) ?$
$\langle \text{Eq2} \rangle \rightarrow \langle \text{id} \rangle (\langle \text{ArgL} \rangle) = \langle \text{Exp} \rangle = \langle \text{num} \rangle$
$\langle \text{ArgL} \rangle \rightarrow \langle \text{id} \rangle (, \langle \text{ArgL} \rangle) ?$
$\langle \text{Exp} \rangle \rightarrow ("+" "-") ? \langle \text{term} \rangle [("+" "-") \langle \text{term} \rangle] ^*$
$\langle \text{Term} \rangle \rightarrow \langle \text{power} \rangle [("*" "/") \langle \text{power} \rangle] ^*$
$\langle \text{Power} \rangle \rightarrow \langle \text{element} \rangle ("^" \langle \text{power} \rangle) ?$
$\langle \text{Elem} \rangle \rightarrow \langle \text{func} \rangle (" (\langle \text{Exp} \rangle)" \langle \text{num} \rangle \langle \text{id} \rangle$
$\langle \text{Func} \rangle \rightarrow \text{"Sin" "Cos" "Tan" "log" "ln" "e"}$
$\langle \text{Num} \rangle \rightarrow "-" ? ["0" - "9"] + ("." ["0" - "9"] +) ?$
$\langle \text{Id} \rangle \rightarrow ["a" - "z", "A" - "Z"] (["a" - "z", "A" - "Z", "0" - "9"]) ^*$

Parsing an expression is processing the expression according to the grammar production rules. The name of methods in JavaCC syntax description of nonlinear expressions is determined according to the nonterminal set in Table 17 as shown in Table 18. All JavaCC methods are defined according to the grammar structure, then the JavaCC parser generator tool is used to generate syntax classes from the JavaCC methods in the form of object trees for interpreter process.

Table 18. JavaCC grammar definition for nonlinear equations

```

Equation Prog() :{ Equation eq; }{
    eq = Eq() <EOF> { return eq; }
}
Equation Eq() :{ Equation eq1, eq2; }{
    eq1=Eq2 () (<SEMI> eq2=Eq () {eq1 = new CompoundEquation
        (eq1, eq2) ;})? { return eq1; }
}
Equation Eq2() :
{ Token t1,t2; Exp e; Exp args[] = new Exp[26]; }
{
    t1 = <ID> <LPAREN>( ArgL(args, 0)) <RPAREN> <ASSIGN> e=E()
    <ASSIGN> <NUM>
    { return new Function(t1.image,args,e); }
}
...
Exp E() :{ Exp e1, e2; int n=1; }{
    (<PLUS> | <MINUS> {n=-1 ;})? e1=T() { if (n<0) e1=new
    Times(new Num(-1), e1); } ( <PLUS> e2=T()
    { e1 = new Plus(e1, e2); }
    | <MINUS> e2=T() { e1 = new Minus(e1, e2); }
    )* { return e1; }
}
...
Exp F() :{ Token t1,t2; Exp e; }{
    t1=<ID> { return new Var(t1.image); }
    | t2=<NUM> { return new Num(Double.parseDouble(t2.image)); }
    | <LPAREN> e=E() <RPAREN> { return e; }
    | <SIN> <LPAREN> e=E() <RPAREN> { return new Sin(e); }
    ...
}

```

2.2.2.2. Generating Abstract Syntax Tree

There are many parser generating tools. In this thesis, Object-Oriented based JavaCC parser generator tool is used to create the object trees. Each rule of a CFG grammar is represented as a syntax class. The formation of the hierarchical structure of the grammatical object tree depends on the execution of the grammar rules used to form the source data. A syntax tree (object tree) consists of several nodes linked together in a hierarchical structure. From these nodes on the object tree, each node is derived from syntax classes and contains an object that represents a process or data.

Table 19. Abstract syntax tree for the application

```

abstract class Equation {
}
class CompoundEquation extends Equation {
    Equation eq1, eq2;
    public CompoundEquation(Equation a, Equation b) {
        eq1 = a;
        eq2 = b;
    }
}
...
abstract class Exp {
    public abstract Object accept(Visitor v);
    public abstract double eval(double x);
}
class Plus extends Exp {
    Exp a, b;
    public Plus(Exp x, Exp y) {
        a = x;
        b = y;
    }
    public double eval(double x) {
        return a.eval(x) + b.eval(x);
    }
}
class Sin extends Exp {
    Exp a;
    public Sin(Exp x) {
        a = x;
    }
    public double eval(double x) {
        return Math.sin(a.eval(x));
    }
}
...

```

2.2.3. Semantic Analysis

In the analysis phase, the syntax tree created in the syntax analysis phase is subjected to the semantic analysis process. The semantic process works on the token structures on the leaves of the tree to make sense for the interpreter. For example; a double variable is a double token, and the string variable is a string variable. As mentioned earlier, the most important operations in this phase include type checking, scope resolution, and array-bound. Faults that may occur at this stage are called semantic errors. For example; an integer variable and a string variable cannot be directly equalized because their token types are different. In this phase, semantic errors in the source code are checked and data type information is specified for the interpreter. The creation of the symbol table and the implementation of the type check are the most important parts of the semantic analysis. The semantic information is not displayed in the context-free language. The CFG used in the syntactic analysis is combined with semantic rules.

In this thesis, the symbol table is created to record the information of some identifiers to retrieve later in the evaluation process. For example; the initial guesses values of the Newton-Raphson method and the absolute error data are recorded in the symbol table (hash table) and the interpreter uses the table to look up easily these values in the evaluation process. Symbol tables are mostly implemented as hash tables, where the source code symbol itself is treated as a key for the hash function and the return value is the information about the symbol.

Table 20. Hash table usage for Newton-Raphson initial guess values

```

--Hash tables for initial guess values in the form of
(var_name,var_value)
class Table {
    Hashtable table = new Hashtable();
    public void put(String id, String d) {
        table.put(id, new String(d));
    }
    public String get(String id) {
        return table.get(id).toString();
    }
}

```

2.2.4. Nonlinear Equations Evaluation Methods

The output of the analysis phase is a meaningful and error free syntax classes in the form of object trees. These syntax classes represent grammar rules in the JavaCC grammar methods. The next phase is to interpret the nodes (process or data) on the object tree according to the implemented interpreter algorithm. The interpreter phase uses Newton-Raphson algorithm to interpret the nodes on the object trees. Visitor design pattern and interpreter methods are used to operating on the object tree, each of which is explained in Section 1.8. Visitor design pattern is a useful pattern that enables to add new operations to the system without changing other classes or methods. The eval () method is also added to each syntax class, which can perform the operation represented by the class. To evaluate an object tree node, it is sufficient to call the eval () method of the object that contains the node.

Table 21. Adding visitor and eval methods to the syntax classes

```

abstract class Equation {
public abstract Object accept(Visitor v);
}
class CompoundEquation extends Equation {
    Equation eq1, eq2;
    public CompoundEquation(Equation a, Equation b) {
        eq1 = a;
        eq2 = b;
    }
    public Object accept(Visitor v) {
        return v.visit(this);
    }
}
...
abstract class Exp {
public abstract Object accept(Visitor v);
public abstract double eval(double x);
}
class Plus extends Exp {
    Exp a, b;
    public Plus(Exp x, Exp y) {
        a = x;
        b = y;
    }

    public Object accept(Visitor v) {
        return v.visit(this);
    }
}
    public double eval(double x) {
        return a.eval(x) + b.eval(x);
    }
}...

```

2.2.5. Newton-Raphson Implementation

In the interpreter phase, we implement the Newton-Raphson algorithm to solve the nonlinear expressions and obtain better approximation solutions. We have the input data in the form of object trees. The interpretation process includes partial derivatives, function evaluations, function transformations into linear equations in matrix form, solving the linear equations using Cramer's rule, Newton-Raphson iterations and stopping criteria, simplifications, controlling of step-by-step solutions, and finally printing the result with all intermediate solution steps.

First, the NLEParser class (Nonlinear Equations Parser) is called to read and parse the input source data and represent it in the form of object tree in the case that no error is detected during the analysis phase as shown in Table 22.

Table 22. Parsing input source data operation

```

public class EQSolver {
    static Equation eq = null;
    static Function f = null;
    static boolean endEq = false;

    public static void solveEquations(){
        try{
            eq = new NLEParser(System.in).Eq();
            ...
            while (!endEq)
            {

                if (eq instanceof CompoundEquation)
                {
                    f = (Function)((CompoundEquation)eq).eq1;
                    eq = ((CompoundEquation)eq).eq2;
                }
                else
                {
                    f = (Function)eq;
                    endEq = true;
                }
            }
            catch(ParseException ex) {
                System.out.println("Fail!\n" + ex.getMessage());
            }
            ...
        }
    }
}

```

2.2.5.1. Partial Derivative

Given a system of nonlinear equations, initial guess values, and absolute error tolerance as input data, as mentioned in Section 1.3, the first step of Newton-Raphson method is to find the derivative of the given functions with respect to specific variables (partial derivative). For a given function f_1 with x_n variables, the partial derivative of this function is shown below and the code is given in Table 23. Derivative processing is done by evaluating the AST tree. For this, a class doing partial derivatives of nonlinear functions is created with visitor design template and it is used general derivative rules.

$$f_1(x_1^{k+1}, x_2^{k+1}) = f_1(x_1^k, x_2^k) + \frac{\partial f_1(x_1^k, x_2^k)}{\partial x_1} (x_1^{k+1} - x_1^k) + \frac{\partial f_1(x_1^k, x_2^k)}{\partial x_2} (x_2^{k+1} - x_2^k) + \dots$$

Table 23. Partial derivative of nonlinear equations

```

class EQSolver {
    ...
    for (int i=0; i<f.args.length; i++)
    {
        if (f.args[i]==null)
            break;
        var = f.args[i].toString();
        derives{row}[column] = (Exp) (new EQDerive().visit(f.e));
    }
    ...
}
class EQDerive implements Visitor {

    public Object visit(Equation equ){
        equ.accept(this);
        return null;
    }
    ...
    public Object visit(Plus e) {
        Exp a = (Exp) (e.a.accept(this));
        Exp b = (Exp) (e.b.accept(this));
        return new Plus(a, b);
    }
    ...
    public Object visit(Sin e) {
        Exp a = (Exp) (e.a.accept(this));
        return new Times(a, new Cos(e.a));
    }
    public Object visit(Var e) {
        if(EQSolver.var.equals(e.id) )
            return new Num(1);
        else
            return new Num(0);
    }
    public Object visit(Num e) {
        return new Num(0);}
}

```


2.2.5.2. Function Evaluations

The next step after partial derivative is to evaluate the derived functions and the normal functions using the initial guesses of the Newton-Raphson method as shown in Table 24. The implemented application can evaluate every nonlinear function that with n variables, as we mentioned earlier, we look up the initial values from the hash table and update them after each iteration.

Table 24. Nonlinear function evaluations

```

public class EQEval implements Visitor {
    public Object visit(Equation equ) {
        equ.accept(this);
        return null;
    }...
    public Object visit(Function eq) {
        eq.e.accept(this);
        return null;
    }
    public Object visit(Exp e) {
        return e.accept(this);
    }
    public Object visit(Plus e) {
        double a = ((Double) (e.a.accept(this))).doubleValue();
        double b = ((Double) (e.b.accept(this))).doubleValue();
        return new Double(a+b);
    }...
    public Object visit(Sin e) {
        double a = ((Double) (e.a.accept(this))).doubleValue();
        return new Double(Math.sin(a));
    }...
    public Object visit(Var e) {
        String id = e.id;
        return Double.parseDouble(EQSolver.t.get(id));
    }
    public Object visit(Num e) {
        return new Double(e.n);
    }
}

```

2.2.5.3. Transformation of Nonlinear Equations into Linear Equations

The next step is to represent the derived functions in the form of a Jacobian matrix and evaluate them, then the unknowns is represented as x_n vector and the functions is represented in the another vector as f_n as shown in the next page.

$$\begin{bmatrix} \frac{\partial f_1(x_1^k, x_2^k, \dots, x_n^k)}{\partial x_1} & \frac{\partial f_1(x_1^k, x_2^k, \dots, x_n^k)}{\partial x_2} & \dots & \frac{\partial f_1(x_1^k, x_2^k, \dots, x_n^k)}{\partial x_n} \\ \frac{\partial f_2(x_1^k, x_2^k, \dots, x_n^k)}{\partial x_1} & \frac{\partial f_2(x_1^k, x_2^k, \dots, x_n^k)}{\partial x_2} & \dots & \frac{\partial f_2(x_1^k, x_2^k, \dots, x_n^k)}{\partial x_n} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial f_n(x_1^k, x_2^k, \dots, x_n^k)}{\partial x_1} & \frac{\partial f_n(x_1^k, x_2^k, \dots, x_n^k)}{\partial x_2} & \dots & \frac{\partial f_n(x_1^k, x_2^k, \dots, x_n^k)}{\partial x_n} \end{bmatrix} \cdot \begin{bmatrix} \Delta x_1^k \\ \Delta x_2^k \\ \vdots \\ \Delta x_n^k \end{bmatrix} = - \begin{bmatrix} \frac{\partial f_1(x_1^k, x_2^k, \dots, x_n^k)}{\partial x} \\ \frac{\partial f_2(x_1^k, x_2^k, \dots, x_n^k)}{\partial x} \\ \vdots \\ \frac{\partial f_n(x_1^k, x_2^k, \dots, x_n^k)}{\partial x} \end{bmatrix}$$

In Newton Raphson method, nonlinear Equations is simplified as linear equations in the form of a matrix as $AX = B$, where A is the Jacobean matrix, X is the unknown vector, and B is the functions vector. The linear equations in the form of matrix and their solution is explained in the next section

2.2.5.4. Solving Linear Equations Using Cramer's Rule

Cramer's rule is used to solve a system of linear equations with n unknowns, It uses to find the solution by finding the determinants of the square coefficient matrix and it a way to solve for just one variable at each time by replacing the corresponding column on the left hand side by the variable vector of right hand side of the equations. This method is valid whenever the system has a unique solution ($D \neq 0$).The general form of n linear equations is defined as below:

$$\begin{bmatrix} a_{11} & a_{12} & \dots & a_{1n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & \dots & a_{nn} \end{bmatrix} \begin{bmatrix} x_1 \\ \vdots \\ x_n \end{bmatrix} = \begin{bmatrix} d_1 \\ \vdots \\ d_n \end{bmatrix},$$

First, we have to calculate the determinant of the coefficient matrix and it is represented as D

$$D \equiv \begin{vmatrix} a_{11} & a_{12} & \dots & a_{1n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & \dots & a_{nn} \end{vmatrix}.$$

Second, we calculate the determinant of each column in the coefficient matrix

$$D_k \equiv \begin{vmatrix} a_{11} & \cdots & a_{1(k-1)} & d_1 & a_{1(k+1)} & \cdots & a_{1n} \\ \vdots & \ddots & \vdots & \vdots & \vdots & \ddots & \vdots \\ a_{n1} & \cdots & a_{n(k-1)} & d_n & a_{n(k+1)} & \cdots & a_{nn} \end{vmatrix}.$$

Finally, the solution of each unknown is computed as:

$$x_{k=D_k/D}$$

Table 25. Solving linear equations using Cramer's rule

```

public void cramers(double lhsEvaluation[][],double rhs[])
{
    double temp[][] = new double[N][N];
    for(int i=0;i<N;i++){
        for(int j=0;j<N;j++){
            for(int k=0;k<N;k++){
                if(k == i)
                    temp[j][k] = (-1)*rhs[j];
                else
                    temp[j][k] = lhsEvaluation[j][k];
            }
        }
    }
    deltaValues[i]=determinant(temp,N)/determinant(lhsEvaluation,N);
    String s = t2.get(i);
    solutionValues[i] = Double.parseDouble(t.get(s)) +
    deltaValues[i];
}

```

2.2.5.5. Newton-Raphson Iterations and Stopping Criteria

Comparing to other numerical methods, Newton-Raphson method is a faster to converge to the solution (root point) with a condition that is a good initial guess near to the root. Newton-Raphson method diverges away from the root if the initial guess is not a good guess. When the initial guess is close enough to a simple root of the function then

Newton's method is guaranteed to converge quadratically (the number of correct digits is nearly doubled at each iteration).

Newton Raphson Method Iterations needs stopping criteria to terminate the iterations for one of two results; better approximation (success) or the solution diverges for the root point (failure). There are four stopping criteria as we mentioned in Section 1.3.3. In our system, we used a stopping criteria to check if we meet a given condition (absolute error).

First, we compare the value of each Δx_n with a given \mathcal{E} (absolute error), if all the values of Δx_n are less than the absolute error, then the iterations are terminated and the final solutions are calculated. Sometimes, the given initial guess may not good enough to converge to the root point and in other cases, recursive iterations may happen. To handle the above and similar situations, we have used to check the iteration number (25 maximum). For example, More than 25 iterations, the iterations are terminated with failure.

Table 26. Newton-Raphson stopping criteria

```
public static boolean controlDeltaValues( ) {
    for(int k = 0; k < N; k++){
        if (deltaValues[k] > errorValue)
            return false;
    }
    return true;
}
```

2.2.5.6. Simplification

It may be necessary to simplify the expression during the intermediate steps of the evaluation process or before the printing process. There is no a common definition of simplicity for every situation, the simplification must be defined according to the expression or problem.

The simplification is normally done by rewriting the rules. To this end, there are a lot of things to consider and a lot of rules that need to be rewritten. In simplest terms, simplification rules should be arranged to reduce the size of the expression or algebraic and trigonometric transformations is applied to an expression as part of the evaluation process.

In our system, after derivation of mathematical functions, it may be necessary to simplify the function expression. The reason for this is that the generated function after the

derivation process has some expressions necessary to simplify because the structure of the function may contain a lot of data which is complicated and unnecessary and this makes the readability of the related function less. To avoid possible problems, the derived expression is subjected to simplification.

In this phase, some basic transformations for simplifications such as numerical, distributive, associative, commutative transformations are performed. In the evaluation process, some special cases needed to be simplified as shown in Table 27. Simplification operations can be performed according to the syntax class of the node accessed by these methods. More details about these transformations can be referenced in [17].

Example. Some cases that need to be simplified are showed below:

$$0 + a \rightarrow a$$

$$1 * a \rightarrow a$$

$$0 - a \rightarrow -a$$

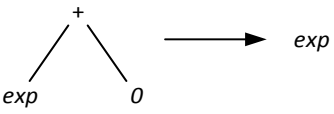
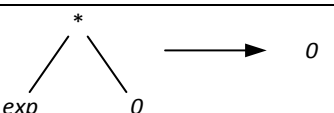
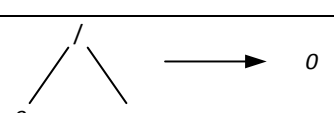
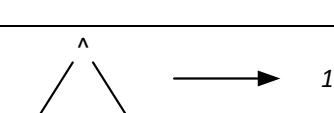
$$a + (-1) \rightarrow a - 1$$

$$2 + a + 5 \rightarrow a + 7$$

$$a^0 \rightarrow 1$$

$$a^1 \rightarrow a$$

Table 27. Simplification rules for some basic transformations

$exp + 0$		Plus(exp, Num(0))
$exp * 0$		Times(exp, Num(0))
$0/exp$		Divide(Num(0), exp)
exp^0		Power(exp, Num(0),)

In our system, after derivation of mathematical functions, some function expressions is simplified. The reason for this is that some expressions don't need read from the AST to calculate their values. As shown in Table 27, any number multiplied by zero is equal to zero so we automatically simplify like these expressions to zero.

Table 28. Simplification methods implementation

```

public class EquationSimplify implements Visitor {
    public Object visit(CompoundEquation equ){
        equ.eq1.accept(this);
        equ.eq2.accept(this);
        return null;
    }
    public Object visit(Plus e) {
        Exp a = (Exp) (e.a.accept(this));
        Exp b = (Exp) (e.b.accept(this));
        if (a instanceof Num && ((Num)a).n == 0.0)
            a = null;
        if (b instanceof Num && ((Num)b).n == 0.0)
            b = null;
        if (a == null && b == null)
            return new Num(0);
        if (a == null)
            return b;
        if (b == null)
            return a;
        return new Plus(a, b);
    }
    public Object visit(Sin e) {
        Exp a = (Exp) (e.a.accept(this));
        if (a instanceof Num && ((Num)a).n == 0.0)
            return new Num(0);
        return e;
    }...
}

```

2.2.5.7. Printing Solution Values and Intermediate Steps

One of the major goals of this work is to help students and users to show all intermediate steps and final solution. During the calculations and evaluation process of Newton-Raphson Method, all intermediate steps are stored in an Array list. For example, derived functions are simplified and stored in the AST for speed up the evaluation process. In the print process, print class looks up the derived functions from AST to print them as shown in Table 29. Other intermediate solution steps are stored in Java arrays and update in each iteration.

Table 29. Print derived functions

```

public class EquationPrint implements Visitor {
    public Object visit(Equation equ){
        equ.accept(this);
        return null;
    }
    public Object visit(Times e) {
        String a;
        if (e.a instanceof Num && ((Num) (e.a)).n== -1)
            a = "-";
        else
            a = (String) (e.a.accept(this));
        String b = (String) (e.b.accept(this));
        if (e.a instanceof Plus || e.a instanceof Minus)
            a = "(" + a + ")";
        if (e.b instanceof Plus || e.b instanceof Minus)
            b = "(" + b + ")";
        Return new String (a + (a.equals ("-")? "" : "*") + b);
    }

    public Object visit(Sin e) {
        String a = (String) (e.a.accept(this));
        return new String("sin(" + a + ")");
    }
    ...

```

After the final solution is obtained. All intermediate and final solution is presented as shown in Table 30.

Table 30. Print final solution values and intermediate steps

```
Public class EQSolver {
    public static void main(String[] args) {
    ...
    while(proceed){
    ...
    flag = controlDeltaValues();
    if(flag){
        for(int i=0;i<N;i++){
            System.out.println("FinalDelta("+ t2.get(i) + ")=" +
                deltaValues[i]);
        }
        System.out.println("all Deltavalue is less than the given
            absolute error ");
        for(int i=0;i<N;i++){
            System.out.println("Final solution value of("+ t2.get(i) +
                ")=" +solutionValues[i]);
        }
        proceed=false;
    }
    else {
        for(int i=0;i<N;i++){
            System.out.println("Delta("+ t2.get(i) + ")=" +
                deltaValues[i]);
        }
        System.out.println("all Deltavalue is not less than the given
            absolute error ");
    }
    ...
    }
```


3. APPLICATION OF THE METHODOLOGY

In this section, a sample application for the Newton-Raphson method is illustrated. The functions to perform the root calculation and other Source data are entered in the interface in the specified format. The Newton-Raphson numerical method is used to calculate the root values of the respective function and display it in the interface.

3.1. Source Data Format

The format of the source data is illustrated the LL (1) Grammar Definition for Nonlinear Equations in Table 17. First, in brackets, all initial values such as Newton-Raphson initial guesses, and absolute error value are entered by the user. Second, the nonlinear functions to calculate the root of the application are entered separated by semi colon.

General structure of source data format is shown in Table 31:

Table 31. Source data format

$(initVar_1 = value, initVar_2 = value, \dots, initVar_n = value);$ $absErrorValue;$ $f_1(x_1, x_2, x_3, \dots, x_n) = 0;$ $f_2(x_1, x_2, x_3, \dots, x_n) = 0;$ $f_3(x_1, x_2, x_3, \dots, x_n) = 0;$... $f_n(x_1, x_2, x_3, \dots, x_n) = 0$

3.2. Step-by-Step Solving of a Given Nonlinear Equation Application

In this section, to illustrate the methodology of our system, we explain all intermediate steps needed to solve a given nonlinear equations in Table 32.

Table 32. Input source data of the application

$(x_0 = 0.6, \quad y_0 = 1.5);$ $0.08;$ $f_1(x, y) = x^2 + y - 3 = 0;$ $f_2(x, y) = y^2 + x - 5 = 0$

The given input source data in Table 32, the initial Newton-Raphson guess of x_0 and y_0 is 0.6 and 1.5 respectively. The absolute error is 0.08. We have two given functions f_1 and f_2 . Using Newton-Raphson iterations, we solve these equations to get a better approximation to the roots of the given functions. First, the system analyzes the given input source data format using the analysis phase, in case of no error is detected, the source data is generated as AST. Secondly, the generated AST nodes (data or operator) is interpreted using the interpretation phase. In this section, the methodology of the given application is illustrated.

3.2.1. Analysis Phase of the Application

The given source data (nonlinear equations) in Table 32 undergoes lexical analysis according to the JavaCC Token Declaration for the Application in Table 15. There is no error in the format of the given input source data, therefore; the generated token sequence the given input source data is given in Table 33.

Table 33. Token sequence of the given application source data

ID LPAREN ID COMMA ID RPAREN ASSIGN ID POWER NUM PLUS ID MINUS NUM ASSIGN NUM <SEMI> ID LPAREN ID COMMA ID RPAREN ASSIGN ID POWER NUM PLUS ID MINUS NUM ASSIGN NUM

After the sequence of tokens is generated, the syntax analyser first task is to control the token component combinations according to the LL (1) grammar defined in Table 17. The given input source data structure is correct according to the defined LL (1) grammar.

The next task is to generate the AST of the given source data as illustrated in Table 34. Hash table is used to store the information of the given initial values in the form of $(varName, varValue)$ so that in our application the hash table is stored as $(x = 0.6, y = 1.5)$. The last step of the analysis phase after the syntax analysis is the semantic analysis to make type checking, scope resolution, and array-bound that we have used in the JavaCC file.

Table 34. The object tree for the input source data (nonlinear equations) in Table 32.

```
Eq(new CompundEq(
    new Function(
        new Minus(new Power(new Var(x), new Num(2)
            ), new Minus
                (new Var(y), new Num(3))))),
    new Function( new Minus(new Power(new Var(y), new Num(2)),
        new Minus(new Var(x), new Num(5))))
)
```

In the analysis phase, an intermediate representation object tree (AST) in Table 34 was created. The next interpreting phase, the AST nodes are interpreted using the implemented Newton-Raphson algorithm. As we explained in Section 2.2.5. The implemented Newton-Raphson undergoes the below steps and the output is shown in Table 35.

- 1) Calculate the partial derivative of the given function.
- 2) Evaluate the derived functions using the given initial guess values.
- 3) Represent (2) result in Jacobin matrix form.
- 4) Evaluate the original functions using the given initial values.
- 5) Convert the nonlinear equations into linear equations in the form of $AX = B$ where A is the Jacobean matrix, X is the unknowns vector, and B is the evaluated original functions.
- 6) Solve the linear equations using Cramer's rule.
- 7) Control the stopping criteria, if it is met calculate the final solution else go back step (2).

3.2.2. Interpretation Phase of the Application

In the interpretation phase, several operations are done according to the Newton-Raphson steps illustrated in Section 3.2.1. The process starts with the partial differentiation of the given function, then evaluating the derived function and to represent it in the Jacobian matrix form, the normal functions are evaluated according to the initial guess value, the process converts into linear equations in the form of $AX = B$, we solve this linear equations using Cramer's rule and finally, Newton-Raphson iterations are carried out until the stopping criteria is met with success finding the solution or failure with exceeding the iteration number.

Table 35. Newton-Raphson interpretation process of object tree in Table 34. (Iter. 1)

1) Partial Derivative of f_1 and f_2	With Respect to x $2x$ 1 With Respect to y $2y$ 1
2) Evaluating Results in Step (1) using initial values(0.6,1.5)	1.2 1 3 1
3) Evaluating f_1 and f_2	-1.1400 and -2.15 respectively
4) Represent the result of (2) in Jacobian matrix form.	$\begin{bmatrix} 1.2 & 1 \\ 1 & 3 \end{bmatrix}$
5) Convert into $AX = B$	$\begin{bmatrix} 1.2 & 1 \\ 1 & 3 \end{bmatrix} \cdot \begin{bmatrix} \Delta x \\ \Delta y \end{bmatrix} = \begin{bmatrix} -1.1400 \\ -2.15 \end{bmatrix}$
6) Solving the linear equations using Cramer's rule	$D = 2.6, Dx = 1.27, Dy = 1.44$ $\Delta x = Dx/D = 0.488$ $\Delta y = Dy/D = 0.554$
7) Both Δx and Δy is greater than the absolute error 0.08. Update initial value and go (2)	$x_1 = x_0 + \Delta x = 1.088$ $y_1 = y_0 + \Delta y = 2.054$

Table 36. Newton-Raphson interpretation process of object tree in Table 34. (Iter. 2)

1) Partial Derivative of f_1 and f_2	With Respect to x $2x$ 1 With Respect to y $2y$ 1
2) Evaluating Results in Step (1) using updated initial values(1.088,2.054)	2.177 1 4.108 1
3) Evaluating f_1 and f_2	0.239 and 0.307 respectively
4) Represent the result of (2) in Jacobian matrix form.	$\begin{bmatrix} 1.2 & 1 \\ 1 & 3 \end{bmatrix}$
5) Convert into $AX = B$	$\begin{bmatrix} 2.177 & 1 \\ 1 & 4.108 \end{bmatrix} \cdot \begin{bmatrix} \Delta x \\ \Delta y \end{bmatrix} = \begin{bmatrix} 0.239 \\ 0.307 \end{bmatrix}$
6) Solving the linear equations using Cramer's rule	$D = 7.94,$ $Dx = -0.673,$ $Dy = -0.429.$ $\Delta x = -0.085$ $\Delta y = -0.054$
7) Both Δx and Δy is less than the absolute error 0.08. The stopping criteria was met.	$x_2 = x_1 + \Delta x = 1.003$ $y_2 = y_1 + \Delta y = 2.000$

Where D is the determinant of the coefficient matrix, Dx and Dy are the determinants of the unknowns vectors of x and y respectively. The given application is solved in two iterations using the implemented Newton-Raphson method. The final solution is:

$$x_2 = x_1 + \Delta x = 1.003$$

$$y_2 = y_1 + \Delta y = 2.000$$

The exact root of the given functions are (1, 2), we can clearly see that Newton-Raphson is a good method to solve nonlinear equations.

We have developed an interface for simplification of the usage of the program. It's simple interface to use.

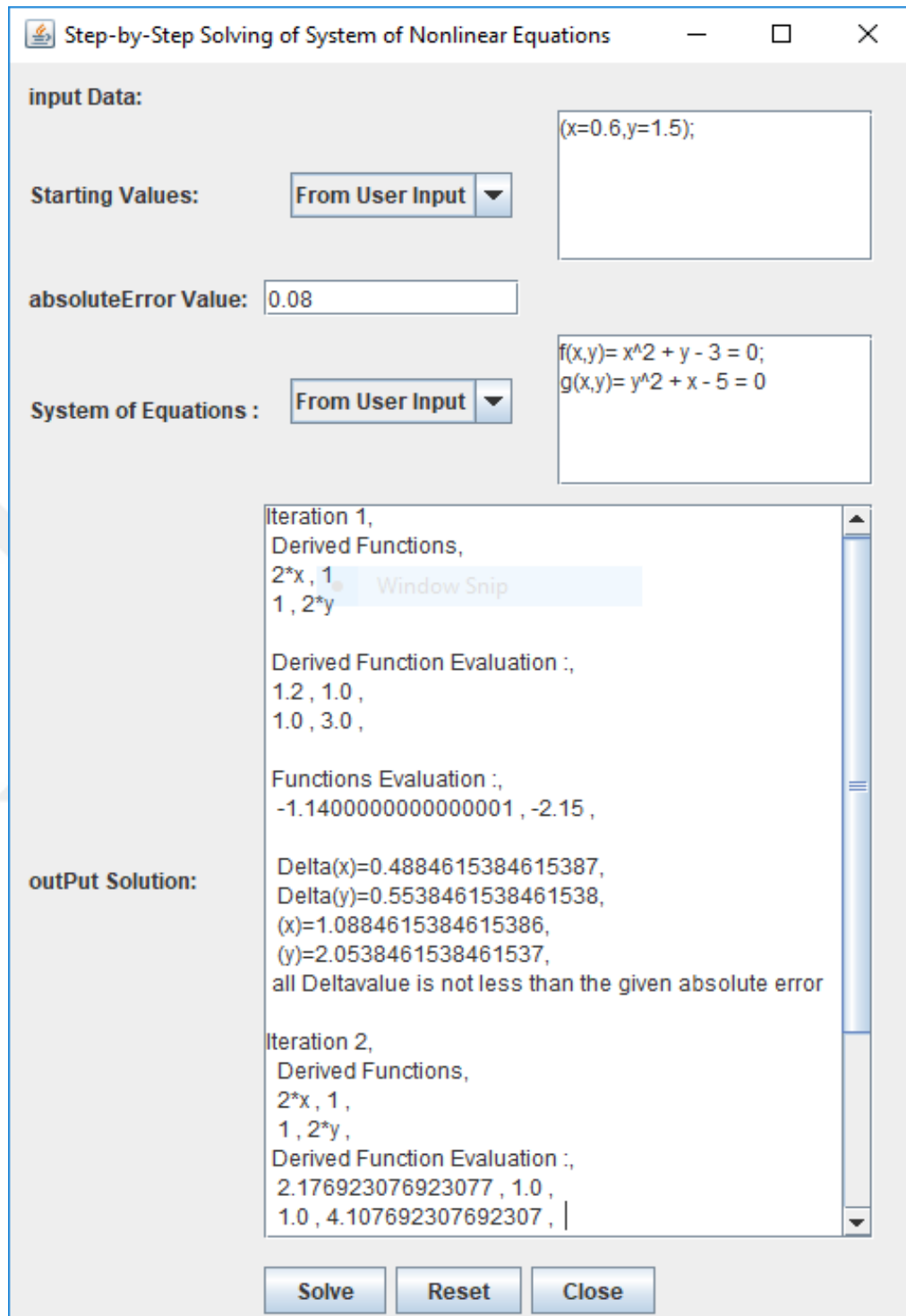


Figure 21. Application Interface (1)

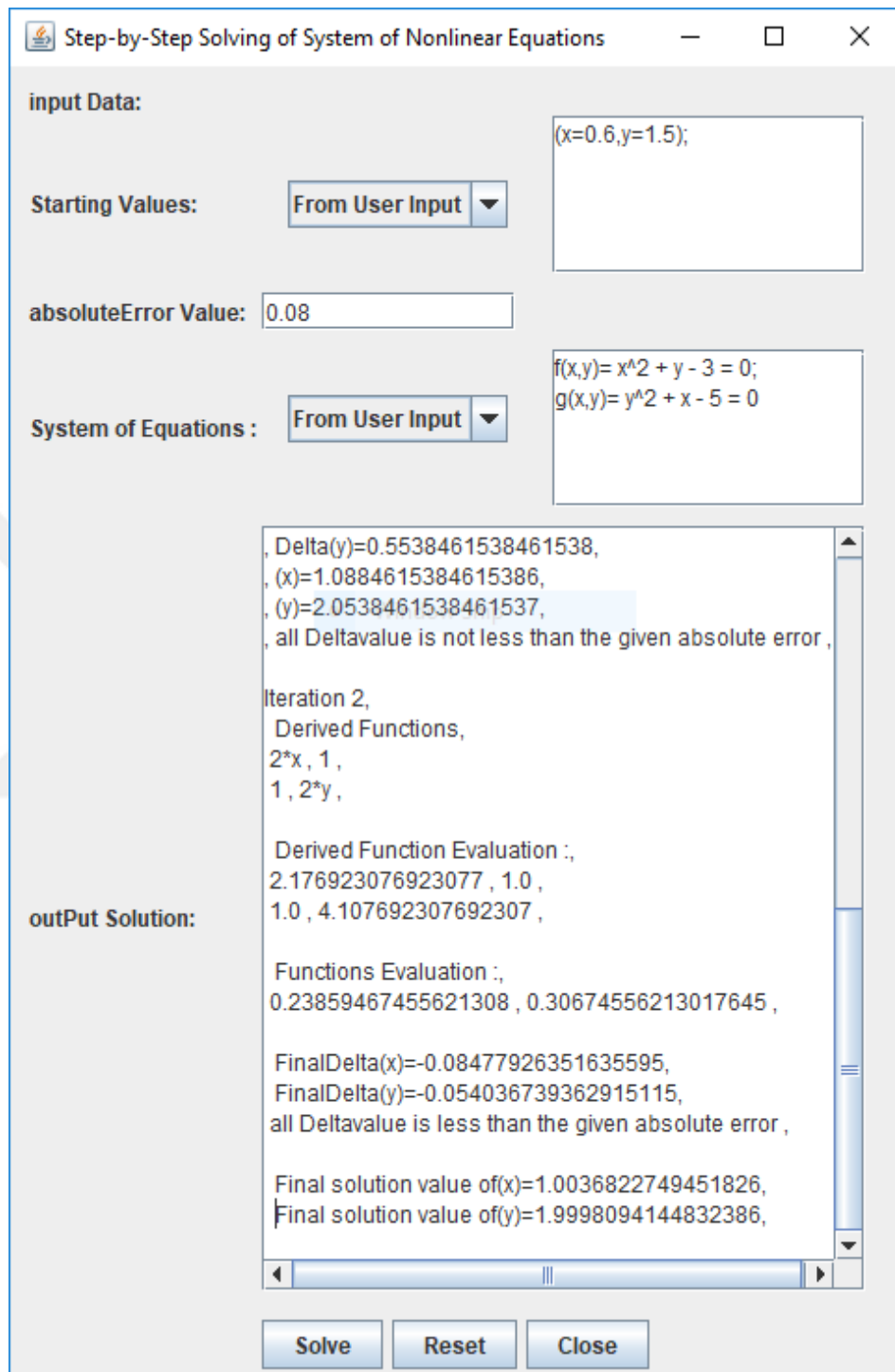


Figure 22. Application Interface (2)

4. RESULTS AND DISCUSSIONS

Our work can produce solutions for $n \times n$ nonlinear equations, Although Newton-Raphson method is the most useful method for finding function roots and it is the fastest convergence method compared to other numerical methods with a condition of a good initial guess values. Initial values near a root, the number of significant digits approximately doubles with each step (converges on the root quadratically), there are some situations need to discuss and solve.

The Newton-Raphson iteration may have a unique solution, no solution or infinite solutions. In our system, we are using Cramer's rule to solve the Newton-Raphson iterations and this method is valid whenever there is a unique solution ($D \neq 0$), When ($D = 0$) means there is no a unique solution and there are two possible situations.

- The system may be inconsistent (no solution at all) if $D = 0$ and at least one of $D_k \neq 0$.
- The system may be dependent (an infinite solutions) if $D = 0$ and all of $D_k = 0$.

In case of there is no unique solution, Cramer's rule is not valid and other methods can be used to solve the linear equations (infinite solutions) such as matrix row operations.

The Newton-Raphson iteration may sometimes go recursive solution or the solution may diverge from the root points, in such cases, we defined maximum iteration number (maximum iteration number = 25), the programs automatically terminate the iterations if the program iterations exceed the defined maximum iterations.

In general, the features of the system developed are as follows:

- Nonlinear equations can be parsed.
- Performs step-by-step solution of a nonlinear system of equations by applying Newton-Raphson Method.
- It can solve linear equations by applying Cramer's Rule.
- It can calculate the derivatives of mathematical expressions.
- It can calculate the partial derivatives of mathematical expressions.
- It can evaluate mathematical functions.
- It can do mathematical simplifications.

- It can display mathematical expressions.
- It may be integrated into other projects as a framework.

Nonlinear equations are one of the most complicated subjects in science and engineering. It's very difficult to solve these equations by hand. As a result, symbolic programming systems have contributed significantly to computer aided education as well as providing considerable convenience in research because long and complicated calculations can produce definite results. In this respect, it is obvious that the field of usage will increase gradually.

In our work, the implemented system is an example of a symbolic computation work on the computer aided step-by-step solution of a system of nonlinear equations using automatic code generation tools. For these reasons, our study has been considered to contribute to further research and to the field of computer-aided education.

As an example, we will solve a system of nonlinear equations as shown in Table 37 and Table 38.

Table 37. Input data of example (2) of the application.

Input Data of Example (2)	
$(x_0 = 1.6, \quad y_0 = 3.5);$	
0.01;	
$f_1(x, y) = x^2 + x * y - 10 = 0;$	
$f_2(x, y) = y^2 + 3 * x * y^2 - 57 = 0$	

Table 38. Solution of example (2) of the application

$x^{(k)}$	$y^{(k)}$	$x^{(k+1)}$	$y^{(k+1)}$	$ (x^{(k+1)} - x^{(k)}) $ and $ (y^{(k+1)} - y^{(k)}) < \epsilon ?$
1.6	3.5	2.016	2.904	0.416 and 0.596 > ϵ
2.016	2.904	1.999	3.001	0.017 and 0.097 > ϵ
1.999	3.001	1.999	2.999	0.000 and 0.002 < ϵ

Where k is the number of iterations.

5. CONCLUSION

In this study, a hybrid symbolic-numeric approach for step-by-step solving of system of nonlinear equations was implemented. Formal grammar rules of the related language were determined by using JavaCC, which automatically generates code from Java programming language, so that the developed application, numerical methods are calculated symbolically. In the developed application, there are two main phases that the system follows for finding the roots of nonlinear equations. First the analysis phase, a grammar structure was prepared in the EBNF notation for expressions of mathematical functions, and this structure was used to define according the JavaCC structure. The analysis phase consists of lexical analysis, syntax analysis, and semantic analysis. Lexical analysis is to read the source input as a stream of characters representing them as token sequences according regular expression and JavaCC rules. The combinations of the token sequence is controlled by the syntax structure according to the defined CFG. The syntax analyser (parser) also generates an intermediate code representation. JavaCC parser was generated the object tree (Abstract Syntax Tree). An object tree that can represent all the mathematical expressions was generated with the aid of the CFG grammar. This object tree consists of token structures (data or operator) that are understood by the defined grammar. Using these node structures, the next phase of the system that is the interpretation phase is performed operations such as partial derivation, simplification, printing expression, and root computation operations.

In this work, we show how to solve system of nonlinear equations with all intermediate steps using Newton-Raphson method with automatic code generation tools. The Newton-Raphson method uses visitor design pattern technique to operate on the object trees. The programming process of Newton-Raphson consists of various symbolic programming activities such as partial derivation, function evaluations, nonlinear transformations into linear equations, solving linear equations using Cramer's rule, generation of iteration solutions, and stopping criteria the iterations. An input mathematical expression to be performed on the root calculation is first passed through several analysis processes using the JavaCC tool, which generates automatic code in the Java programming language according the predefined grammar rules, and is then each grammar rule is

Represented by syntax class in the form of object structures. All computations required by the Newton-Raphson method for solving the problem are carried out through these object structures.

Visitor Design pattern and interpreter methods are added into the syntax classes to operate and evaluate the token node structures on the object tree. The visitor design pattern simplifies to add new operations to the system without changing the other operations.

In our system, we have some issues need to be considered when solving nonlinear system of equations. Nonlinear equations may have a unique solution, no solution or infinite solutions. In our system, we are using Cramer's rule to solve the Newton-Raphson iterations and this method is valid whenever there is a unique solution ($D \neq 0$), When ($D = 0$) means there is no a unique solution and there are two possible situations. The first possible situation is that the system may be inconsistent (no solution at all) if $D = 0$ and at least one of $D_k \neq 0$. The second possible situation is that the system may be dependent (an infinite solutions) if $D = 0$ and all of $D_k = 0$. In case of there is no a unique solution, Cramer's rule is not valid and other methods can be used to solve the linear equations (infinite solutions) such as matrix row operations. The Newton-Raphson iteration may sometimes go recursive solution or the solution may diverge from the root points, in such cases we defined maximum iteration number, the programs automatically terminates the iterations if the program iterations exceeds the defined maximum iterations.

The developed interpreter can easily be extended to cover other numerical methods, only describing the related iterative computation steps. On the other hand, integrating into their own interactive development environments, researchers can input any system of non-linear equations directly into the interpreter and get the approximating solution as an output. Generally, many common and special purpose symbolic systems commonly used today don't show intermediate steps but show only the final result of the process. With our system to help students and users, symbolic analysis processes of various nonlinear equations systems, such as differential equations, function evaluations, solving of linear equations can be realized in areas where all engineering and scientific calculations used in mathematical operations are made. In addition, each calculation step leading to the solution of the problem can be shown.

6. FUTURE WORKS

Other Numerical methods for step-by-step solving nonlinear equations can be integrated to our system.

In our application, we have used Cramer's rule to solve the linear equations but it can solve only nonlinear equations with unique solutions. It can be implement matrix row operations technique to find other possible solutions (no solution and infinity solution) rather than unique solution.



7. REFERENCES

1. Papadimitriou, S., Scientific programming with Java classes supported with a scripting interpreter, IET software, 1, 2 (2007) 48-56.
2. Monagan, M. B., Geddes, K. O., Heal, K. M., Labahn, G., Vorkoetter, S. M., McCarron, J., and DeMarco, P., Maple 10 Introductory (Advanced) Programming Guide, Waterloo Maple, Waterloo, 2005.
3. Trott, M., The Mathematica guidebook for symbolics. Springer Science & Business Media, 2007.
4. Guide, M.U.S., The mathworks. Inc., Natick, MA 5 (1998) 333.
5. Eaton, John W., GNU Octave Manual, Network Theory Limited., URL: <http://www.Octave.Org> (2002).
6. Cohen, Joel S., Computer algebra and symbolic computation: Mathematical methods. Universities Press, 2003.
7. Von Zur Gathen, J., and Gerhard, J. Modern computer algebra, Cambridge University press, 2013.
8. Buchberger, B., Collins, G. E., and Loos, R., Computer algebra: symbolic and algebraic computation, 1985.
9. Parr, T., and Fisher K., LL (*): The foundation of the ANTLR parser generator, ACM SIGPLAN Notices, 46, 6 (2011) 425-436.
10. Gagnon, E., Menking, B., Nowostawski, M., Agbakpem, K. K., and Gergely, K., SableCC. An Object-Oriented Compiler Framework, Master of Science, School of Computer Science, McGill University, Montreal, 2002.
11. Tao, K., Wang, W., and Palsberg, J., Java Tree Builder JTB, 2000.
12. Kodaganallur, V., Incorporating language processing into java applications: A javacc tutorial, IEEE software, 21, 4 (2004) 70-77.
13. Berk, E., and Ananian, C.S., JLex: A lexical analyser generator for Java (TM), Department of Computer Science, Princeton University. Version 1, 2005.
14. Gerwin, K., JFlex User's Manual, (2005) 12-42.
15. Petković, I., and Herceg, Đ., Symbolic computation and computer graphics as tools for developing and studying new root-finding methods, Applied Mathematics and Computation, 295 (2017) 95-113.

16. Boyle, A., and Caviness, B.F., Future directions for research in symbolic computation, Report of a Workshop on Symbolic and Algebraic Computation, 1990.
17. Milani, M., Design and Applications of Grammar-based Methodologies for Automatic Generation and Step-by-step Solving of Mathematical Expressions, Doctoral dissertation, Karadeniz Technical University, The graduate school of Natural and applied science, Trabzon, 2015.
18. <https://www.coursera.org/Coursea>. 27 May 2017.
19. <https://ntulearn.ntu.edu.sg/Edventure>. 27 May 2017.
20. Singh, R., Gulwani, S., and Rajamani, S. K., Automatically generating algebra problems, Twenty-Sixth AAAI Conference on Artificial Intelligence, 2012.
21. Rasila, A., Harjula, M., and Zenger, K., Automatic assessment of mathematics exercises: Experiences and future prospects, ReflekTori 2007, (2007) 70-80.
22. Motzkin, T., The euclidean algorithm, Bull. Amer. Math Soc, 55, 12 (1949) 1142-1146.
23. Berlekamp, E. R., Factoring polynomials over finite fields, Bell System Technical Journal, 46, 8 (1967) 1853-1859.
24. Zassenhaus, H., On Hensel Factorization, I., Journal of Number Theory, 1,3(1969), 291-311.
25. Musser, D., Multivariate Polynomial Factorization, Journal of the ACM (JACM), 22,2 (1975), 291-308.
26. Wang, P. S., Automatic Computation of Limits, Proceedings of the Second ACM Symposium on Symbolic and Algebraic Manipulation. Los Angeles, Mart 1971 Bildiriler Kitabı, 128-133.
27. Griesmer, J.H., Symbolic mathematical computation: a survey, ACM SIGSAM Bulletin, 10, 2 (1976): 30-32.
28. Nolan, J.F., Analytical differentiation on a digital computer, Diss. Massachusetts Institute of Technology, 1953.
29. Kahrimanian, H. G., Analytical Differentiation by a Digital Computer, MA Thesis, Temple University, May 1953.
30. McCarthy, J., Recursive functions of symbolic expressions and their computation by machine, Part I., Communications of the ACM, 3,4 (1960) 184-195.

31. Slagle, J.R., A heuristic program that solves symbolic integration problems in freshman calculus, Journal of the ACM (JACM), 10, 4 (1963) 507-520.
32. Sammet, J.E., and Bond, E. R., Introduction to FORMAC, IEEE Transactions on Electronic Computers, 4 (1964) 386-394.
33. Hearn, A.C., REDUCE: A user-oriented interactive system for algebraic simplification, Symposium on Interactive Systems for Experimental Applied Mathematics: Proceedings of the Association for Computing Machinery Inc. Symposium. ACM, (1967), 79-90.
34. Martin, W. A., and Fateman, R. J., The MACSYMA system, Proceedings of the second ACM symposium on Symbolic and algebraic manipulation. ACM, (1971), 59-75.
35. Hearn, A.C., REDUCE 2: A system and language for algebraic manipulation, Proceedings of the second ACM symposium on Symbolic and algebraic manipulation ACM, (1971), 128-133.
36. Blair, F., Griesmar, J. H., and Jenks, R. D., SCRATCHPAD/1: An interactive facility for symbolic mathematics, Proceedings of the second ACM symposium on Symbolic and algebraic manipulation, Los Angeles, 1971.
37. Rich, A. D., and Stoutemyer, D. R., Capabilities of the muMATH-79 computer algebra system for the INTEL-8080 microprocessor, Symbolic and Algebraic Computation, Springer, Berlin, Heidelberg, (1979) 241-248.
38. <http://maxima.sourceforge.net/SourceForge>. 28-08-2017.
39. <http://www.sagemath.org/SageMath>. 28-08-2017.
40. <http://www.axiom-developer.org/Axiom>. 28-08-2017.
41. <http://magma.maths.usyd.edu.au/magma/Magma>. 28-08-2017.
42. <https://www.mathworks.com/products/symbolic/> MATLAB 28-08-2017.
43. <http://cocoa.dima.unige.it/Cocoa>. 28-08-2017.
44. <http://home.bway.net/lewis/>. Femat. 28-08-2017.
45. <http://page.math.tu-berlin.de/~kant/kash>. Kant/Kash. 28-08-2017
46. <http://www.math.uiuc.edu/Macaulay2/Macaulay>. 28-08-2017
47. Bauer, C., Frink, A., and Kreckel, R., Introduction to the GiNaC framework for symbolic computation within the C++ programming language, Journal of Symbolic Computation, 33, 1 (2002) 1-12.

48. Park, H., Symbolic computation and signal processing, *Journal of Symbolic Computation*, 37, 2 (2004) 209-226.
49. Tekbaş, Y., Code Production Tools Using Automatic Calculation of Derivatives and Simplification of Mathematical Expressions, Master thesis, Karadeniz Technical University, The graduate school of Natural and applied science, Trabzon, 2013.
50. GÖKGÖZ, B., Design and Implementation of a General Interpreter for Numerical Root Finding Methods Using symbolic Approaches, Master Thesis, Karadeniz Technical University, The graduate school of Natural and applied science, Trabzon, 2016.
51. HASSAN, M.Y. and PEHLIVAN, H., Design and Implementation of a General Interpreter for Automatic Generation and Step-by-Step solving of Nonlinear System of Equations, International Symposium of mathematical methods in engineering, April 2017 Ankara, Proceeding Book, 78.
52. Chun, C., and Neta, B., A third-order modification of Newton's method for multiple roots, *Applied Mathematics and Computation*, 211, 2 (2009) 474-479.
53. Li, S. G., Cheng, L. Z., and Neta, B., Some fourth-order nonlinear solvers with closed formulae for multiple roots, *Computers & Mathematics with Applications*, 59,1 (2010) 126-135.
54. Shengguo, L., Xiangke, L., and Lizhi, C., A new fourth-order iterative method for finding multiple roots of nonlinear equations, *Applied Mathematics and Computation*, 215,3 (2009) 1288-1292.
55. Neta, B., and Johnson, A. N., High-order nonlinear solver for multiple roots, *Computers & Mathematics with Applications*, 55, 9 (2008) 2012-2017.
56. Sharma, J. R., and Sharma, R., Modified Jarratt method for computing multiple roots, *Applied Mathematics and Computation*, 217,2 (2010) 878-881.
57. Zhou, X., Chen, X., & Song, Y., Constructing higher-order methods for obtaining the multiple roots of nonlinear equations, *Journal of Computational and Applied Mathematics*, 235,14 (2011) 4199-4206.
58. Petković, I., and Herceg, Đ., Symbolic computation and computer graphics as tools for developing and studying new root-finding methods, *Applied Mathematics and Computation*, 295 (2017) 95-113.
59. Yun, Beong In. "A non-iterative method for solving non-linear equations." *Applied Mathematics and Computation* 198, 2 (2008) 691-699.
60. Yun, B.I., Transformation methods for finding multiple roots of nonlinear equations, *Applied Mathematics and Computation*, 217, 2 (2010) 599-606.

61. Neta, B., and Changbum C., On a family of Laguerre methods to find multiple roots of nonlinear equations, Applied Mathematics and Computation, 219,23 (2013) 10987-11004.
62. Li, S. G., Cheng, L. Z., and Neta, B., Some fourth-order nonlinear solvers with closed formulae for multiple roots, Computers & Mathematics with Applications, 59,1 (2010) 126-135.
63. Gallopoulos, E., Elias H., and John R. R., Computer as thinker/doer: Problem-solving environments for computational science, IEEE Computational Science and Engineering, 1, 2 (1994) 11-23.
64. Boeing, G., Visual Analysis of Nonlinear Dynamical Systems: Chaos, Fractals, Self-Similarity and the Limits of Prediction, Systems, 4, 4 (2016) 37.
65. <https://en.wikipedia.org/wiki/Interprete>. 31 May 2017.
66. Aho, A.V., Ravi S., and Jeffrey D.U., Compilers, Principles, Techniques. Boston: Addison Wesley, 1986.
67. Mogensen, T.Æ., Basics of compiler design, Torben Ægidius Mogensen, 2009.
68. Earley, J., An efficient context-free parsing algorithm, Communications of the ACM, 13, 2 (1970) 94-102.
69. Chomsky, N., Three models for the description of language, IRE Transactions on information theory, 2, 3 (1956: 113-124.
70. Appel, Modern Compiler Implementation in Java, Cambridge University Press, Revised Edition, New Delhi, 2007.
71. https://www.tutorialspoint.com/compiler_design. Parser types. 31 May 2017.
72. Dick, G., and Cerial, H., Parsing techniques, a practical guide, Technical Report, Tech. Rep, 1990.
73. Gamma E., Helm, R., Johnson, R. and Vlissides, J., Design Patterns: Elements of Reusable Object-Oriented Software, Addison-Wesley, 1995.
74. Hudson, S., Flannery, F., Ananian, S., Wang, D., and Appel, A., JavaCup User's Manual, 1998.
75. Johnson, S.C., Yacc: Yet another compiler-compiler. Vol. 32, Murray Hill, NJ: Bell Laboratories, 1975.
76. Aaby, A.A., Compiler construction using flex and bison, *Walla Walla College*, 2003.

CURRICULUM VIATE

Mohamed Yusuf HASSAN was born in Mogadishu in December 1989. He graduated from Jabir Bin Hayan Primary and Secondary School, Mogadishu, Somalia in 2008. He got his B.Sc. from the computer science department at Hadhramout University of Science and technology, Mukalla, Yemen in 2013. He knows Arabic, English, and Turkish languages well.

