KARADENIZ TECHNICAL UNIVERSITY
THE GRADUATE SCHOOL OF NATURAL AND APPLIED SCIENCE

COMPUTER ENGINEERING GRADUATE PROGRAM

DESIGN AND IMPLEMENTATION OF AN INTERPRETER FOR THE LEAST SQUARES METHOD USING SYMBOLIC APPROACHES

MASTER THESIS

Computer Eng. Nawal ABDULLAHI MOHAMED

JULY 2018
TRABZON

KARADENIZ TECHNICAL UNIVERSITY
THE GRADUATE SCHOOL OF NATURAL AND APPLIED SCIENCE

COMPUTER ENGINEERING GRADUATE PROGRAM

DESIGN AND IMPLEMENTATION OF AN INTERPRETER FOR THE LEAST SQUARES METHOD USING SYMBOLIC APPROACHES

Computer Eng. Nawal ABDULLAHI MOHAMED

This Thesis is Accepted to Give The Degree of
“ MASTER OF SCIENCE"
” By
The Graduate School of Natural and Applied Science at
Karadeniz Technical University

The date of Submission: 09.07.2018
The date of Examination: 30.07.2018

Supervisor: Asst. Prof. Dr. Hüseyin PEHLİVAN

Trabzon 2018

KARADENIZ TECHNICAL UNIVERSITY

THE GRADUATE SCHOOL OF NATURAL AND APPLIED SCIENCES

COMPUTER ENGINEERING GRADUATE PROGRAM

Nawal ABDULLAHI MOHAMED

DESIGN AND IMPLEMENTATION OF AN INTERPRETER FOR THE LEAST SQUARES METHOD 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 1761 dated 10/07/2018

Approved By

Chairman : Prof. Dr. Hasan Hüseyin BALIK ……...

Member : Assoc. Prof. Dr. Bekir DIZDAROĞLU.……...

Member : Asst. Prof. Dr. Hüseyin PEHLİVAN ...……

Prof. Dr. Sadettin KORKMAZ

Director of Graduate School

ACKNOWLEDGEMENT

This thesis is written as completion to the master of computer engineering, at Karadeniz Technical University. This thesis is focused on minimizing the square of the error for the given function using least squares method.

The successful completion of this study has been made possible by the cooperation, assistance and encouragement of many people. Sincere appreciation is expressed to my concerned master thesis supervisor, Asst. Prof. Dr. Hüseyin PEHLİVAN, for his insight, much support, objective criticism, friendly guidance and direction throughout the entire period of study. My deep gratitude also goes to other members of my thesis committee, Assoc. Prof. Dr. Bekir DIZDAROĞLU and Prof. Dr. Hasan Hüseyin BALIK for their valuable feedback and encouragement.

This acknowledgement would be incomplete without expressing my hearty thanks to my family for giving me moral support and encouragement throughout the study. My deep gratitude also goes to my father and my elder sister, for their love and support.

Nawal ABDULLAHI MOHAMED

Trabzon 2018

THESIS STATEMENT

I declare that, this Master Thesis, I have submitted with the tittle “DESIGN AND IMPLEMENTATION OF AN INTERPRETER FOR THE LEAST SQUARES METHOD 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. 30/07/2018

Nawal ABDULLAHI MOHAMED

TABLE OF CONTENTS

Page No

Master Thesis

SUMMARY

DESIGN AND IMPLEMENTATION OF AN INTERPRETER FOR THE LEAST SQUARES METHOD USING SYMBOLIC APPROACHES

Nawal ABDULLAHI MOHAMED

Karadeniz Technical University
The Graduate School of Natural and Applied Sciences
Computer Engineering Graduate Program
Supervisor: Asst. Prof. Dr. Hüseyin PEHLİVAN
2018, 59 Pages

This work describes the development of an interpreter for the least squares method which is an important technique of regression analysis that fits a mathematical or statistical model to a particular data set, using symbolic computation methods and the JavaCC code generation tool. Although the JavaCC tool is generally used when developing interpreters for programming languages, it can also be used to evaluate mathematical expressions in a similar way. The development process starts with the construction of a context free grammar that denotes the mathematical curves. Then, a parser which is generated via the JavaCC tool for this grammar is employed to represent the curves with object structures and to determine their parameters. Through these object structures, the curves are analyzed and the parameters to be computed by the least squares method are determined. For the curves with specific function components, such as exponential, logarithmic and rational functions, some symbolic computation tasks are performed, which transform those curves into polynomials.

Key words: Symbolic computation, Curve fitting, Least squares method, Context-free grammars.

Yüksek Lisans Tezi

ÖZET

SİMGESEL YAKLAŞIMLARI KULLANARAK EN KÜÇÜK KARELER YÖNTEMİ İÇİN BİR YORUMLAYICININ TASARIMI VE GERÇEKLENMESİ

Nawal ABDULLAHI MOHAMED

Karadeniz Teknik Üniversitesi
Fen Bilimleri Enstitüsü
Bilgisayar Mühendisliği Anabilim Dalı
Danışman: Dr. Öğr. Üyesi Hüseyin PEHLIVAN
2018, 59 Sayfa

Bu çalışma, sembolik hesaplama yöntemleri ve JavaCC kod oluşturma aracını kullanarak, özel bir veri kümesine uygun bir matematiksel yada istatistiksel modeli belirlemeye yönelik önemli bir regresyon analiz tekniği olan en küçük kareler yöntemi için bir yorumlayıcının geliştirilmesini göstermektedir. JavaCC aracı genellikle programlama dilleri için yorumlayıcı geliştirilirken kullanılmasına rağmen, benzer bir yol içinde matematiksel ifadeleri değerlendirmek için de kullanılabilir. Geliştirme süreci matematiksel eğrileri temsil eden bağlamdan bağımsız bir gramerin oluşturulması ile başlar. Daha sonra, bu gramere karşılık JavaCC aracıyla oluşturulan bir ayrıştırıcı, eğrileri nesne yapılarıyla temsil etmek ve parametrelerini belirlemek için kullanılır. Bu nesne yapıları içerisinden eğriler analiz edilir ve en küçük kareler yöntemi ile hesaplanacak parametreler belirlenir. Üstel, logaritmik ve rasyonel işlevler gibi belirli işlev bileşenlerine sahip eğriler için, bu eğrileri polinomlara dönüştüren bazı sembolik hesaplama işlemleri gerçekleştirilir.

Anahtar Kelimeler: Sembolik hesaplama, Eğri uydurma, En küçük kareler yöntemi, Bağlamsız gramerler.

LIST OF FIGURES

Page No

LIST OF TABLES

Page No

LIST OF ABBREVIATIONS

CAS	Computer Algebra System
AST	Abstract Syntax Tree
CFG	Context Free Grammar
BNF	Backus–Naur Form
EBNF	(Extented) Backus–Naur Form
LSM	Least Squares Method
IR	Intermediate Representation
LL Parser	Left to right Left most deriviation Parser
LR Parser	Left to right Right most deriviation Parser
LS	Least Squares

1. INTRODUCTION

In engineering applications many methods have been developed for computer environments to solve the mathematical problems which cannot be accomplished by human hand. A typical category of these methods contains symbolic computation ones that can do symbolic calculations, as well as numerical calculations. Symbolic calculation is based on finding the exact and error-free solution of mathematical problems with computer programs. In this type of calculation, mathematical equations need to be fully expressed before they can be processed, and then converted into algorithms that can be solved by computer programs [1, 2]. Today symbolic calculation systems are divided into two classes, such as general and special purpose ones. Each of these systems has a programming language in which mathematical expressions can be written. General purpose systems provide an interactive computing environment for solving problems in various branches of mathematics. Tools such as MATLAB [3], Maple [4], Macsyma [5], Mathematica [6], Axiom [7], and MuPad are examples of general purpose systems. Special purpose computing systems are equipped with a limited number of processing capacities to meet specific application requirements. The unit has developed GAP and Magma in the field of group theory, CoCoA and Macaulay for computational algebra and algebraic geometry studies, and Schoonship tools for high energy physics calculations. Each symbolic calculation system includes a specific programming language which is used to code the algorithms that will solve the equations of the mathematical models of the problems.

With the high-level programming languages that facilitate code writing processes, compilers have become indispensable tools for software developers. Compilers that implement the steps of translating code written in a high-level programming language into machine language have a complex code structure as operating systems [8,9]. Functional changes in the structure of programming languages, such as ensuring compatibility of new technologies with existing systems, are among the most important reasons for the complexity of constructing compilers. Automatic code generation tools have been developed to facilitate the analysis, interpretation, or compilation processes of formal languages. They can also be used to handle mathematical operations in symbolic calculation environments.

Programming languages are represented by context free grammar (CFG) structures that define the syntax of the source data. The representation of the syntax is usually made in the BNF notation, and there are numerous automated code generation tools that can generate parsers for them. There are many types of automatic code generation tools developed for different programming languages. There are even many tools that can generate source code in the Java language; ANTLR [10], SableCC [11], JTB [12], JavaCC [13], JLex [14] and JFlex [15]. For example, the source code to be generated by the JavaCC tool can be easily integrated with other software and serve as a handy analyzer and parser components that can process the input data.

In this thesis, we focus on the implementation of an interpreter for the least squares method which performs mathematical operations involving both numerical calculations and symbolic calculations, using the JavaCC code generation tool developed for the Java programming language. Generally, LSM needs a basic knowledge of algebra, solving sysyem of equations using matrices, and some calculus.

2. LITERATURE REVIEW

The most important use of the computer since its inception has been to do fast and errorless calculations and various systems, algorithms, techniques and methods have been developed for this purpose.

Given the developments from the past to the present day about the symbolic calculation leaving the numerical calculation to one side, the studies can be grouped into two main groups. The first one is the research of the solution methods and algorithms that can be applied by the computer. This area is studied to develop more general or faster solution methods. The other studies are the development of computer algebra system applications.

Mathematicians have developed many algorithms for math operations, which become more useful when the computer entered into human life. And today, the rapid development in science and technology has affected the lives of human beings in every field. There are many algorithms developed for the computerized solution of mathematical problems.

In the 3rd century B.C., the algorithm of finding the largest common divisors of integers founded by Euclides can be considered as the first and basic example of present symbolic computation algorithms. However, the finding of the roots of polynomial equations, the derivation, the integral and the investigation of algorithmic solution methods of differential equations have been the subject of symbolic computation.

Although symbolic computing has been used in computers since 1953, it has a long history in terms of its use in scientific development. In the United States a dynamic group of researchers stated the early basics for the area of symbolic computation from 1965 to 1980 and this brought important advances in a useful algorithms and efficeint software [16].

Leibniz worked on mathematical calculations between 1673 and 1676 in Paris, who searched for a generic symbolic language that is characteristic to the conversion of mathematical methods and expressions into algorithms and formulas.

For example; the problem of separating polynomials from multiplicities also has a long history. The first algorithm for separating univariate polynomials over integers into multipliers was found by Schubert in 1793. In 1882 this algorithm was re-invented by

Kronecker and extended to polynomials with algebraic coefficients. The specified algorithms have been fully utilized with the help of computers.

In the literature, it seems that, compared to the history of the computation with the computers, the symbolic computation may be done too early than the development of the technology. In 1953, after the invention of electronic computer, the first succesful real application was carried out as two graduate thesis. The first one was developed by J. F. Nolan from the Massachusetts Institute of Technology [17] and the other by H. G. Kahrimanian at Temple University [18].

At the end of the 1950s, list processing languages were developed. Lisp, developed by John McCarty in 1958, is the most common and longest living language [19]. It is also the oldest, second highest level programming language and plays a very important role in symbolic computation. The first program to calculate the symbolic integral was written by Slagle in 1961 with Lisp. This practice has been developed as a doctoral study [20].

It has been understood that symbolic mathematical problems can be solved using the facilities provided by Lisp language used as a programming tool. After this step, the symbolic calculation area has showed a rapid development from the 1960s.

The Schubert and Kronecker algorithms were used to divide polynomials into multipliers, but it was seen that these algorithms worked very slowly even on a computer. In 1967, Berlekamp introduced a new algorithm to more quickly divide polynomials on finite fields into multipliers [21]. In 1969, Zassenhaus showed that multipliers over the integers obtained by the Berlekamp algorithm can be obtained [22]. In 1975-76, Musser, Wang and Rothschild, working on similar methods, developed algorithms for solving multivariate and algebraic coefficient polynomials. [23, 24].

Risch developed an algorithmic solution of the indefinite integral problem for a general class of functions involving exponential, logarithmic, trigonometric and rational functions between 1968-70 [25]. Nowadays, application studies of Risch algorithm to more comprehensive function classes are going on.

There is a software package that capable of doing symbolic computation called CAS. CAS applications represent mathematical expressions symbolically and operate on these symbolical represented objects. It can be separated into general purposes which provides

computing facilities for general mathematical problems and specific purposes which gives special uses for algebraic and special mathematical areas.

At the end of the 1960s and early 1970s, the first general purpose symbolic computation systems were developed. These systems include Reduce [26] in 1967, Macsyma [27] and Reduce 2 [28] in 1971, Scratchpad [29] in 1971, and muMATH [30] in 1979.

In 1971, Macsyma based version called Maxima was developed by Paul S. Wang. This system supports many operations such as rational, logarithmic, trigonometric expressions, differentiation, integration, ordinary differential equations, linear equations, polynomials, Laplace transforms, matrices, and Taylor series [31].

Today, the major general purpose CAS systems include Mathematica [6] from wolfram research, Maple [4] from university of Waterloo, Axiom [32] from Richard Jenks, Magma [33] from university of Sydney, SageMath [34] from William A. Stein, Maxima [22] from Massachusetts Institute of Technology researchers, and Symbolic Math Toolbox (MATLAB) [35] from MathWorks.

The major special purpose CAS systems include Fermat [36] for polynomial and matrix computation, Macaulay2 [37] for algebraic geometry and commutative algebra, KANT/KASH [38] for algebraic number theory, and CoCoA-5 [39] for commutative algebra.

In 2002, Cristian Bauer developed a symbolic calculation framework called GiNac in C ++ environment to implement the symbolic computation and it was designed to handle multivariate polynomials, algebras, and other special functions [40].

In 2004, Hyungju Park pointed out that many problems in digital processing can be translated to algebraic problems and can be solved using algebraic and symbolic computation methods [41].

In 2010, Yoshinari Miyazaki developed the application of the Information Access tool for mathematical expressions with the aim of contributing to engineering education on the web. [42]. This web-based application uses MySQL as the database, Java as the programming language and Tomcat as the server. A search query was performed on the database of mathematical expressions using regular expressions.

In 2013, Yavuz TEKBAŞ presented a master thesis on derivatives of mathematical expressions [43]. His study, which was developed with the Java programming language, and

JavaCC tool, is used to decompose the expressions. According to grammar rules determined in the study, derivation, conversion and simplification are performed.

In 2015, Mir Mohammad Reza Alavi Milani conducted a general methodology study on the step-by-step evaluation of mathematical expressions in his doctoral dissertation [44]. In this work, a methodology was designed to solve mathematical expressions step-by-step and to produce new questions using template expressions. Java is used as an application development language. The parser structure generated by JavaCC is used for parsing operations.

In 2016, Baki GÖKGÖZ presented a graduate thesis for programming numerical root finding method via symbolic approaches [45]. In this work, it is described how to program numerical root-finding methods via automatic code generation tools. The programming process consists of distinct symbolic programming tasks such as differentiation, functional translation and generation of iteration expressions. A mathematical expression solved for the roots is firstly processed through some analysis operations and then represented by object structures, using the JavaCC tool.

In this work, least squares method was used to minimizes the square of the error. LSM is one of the oldest techniques of modern statistics and it was presented in 1805 by the French mathematician Legendre. In the modern statistical faramework first use of LSM can be signed to Galton (1886). He used in his project on the heritability of size which laid down the foundations of correlation and also gave the name to regression analysis [46].

Today, there are general computing software such as Mathematica (Wolfram, 1991) and (Maplesoft) Maple which symbolically calculate the solution of mathematical problems. Maple is a modern software with features such as symbolic and numerical calculation, data analysis and visualization. There is widespread use in high schools and universities for mathematics and engineering education. Likewise, Matlab is also a tool for symbolic computation.

In addition, Wolfram Alpha, a web-based search engine that can step-by-step solve mathematical problems using the Mathematica sub-structure, is one of the most recent examples in this area. It has many features such as step-by-step solution of problems, graphical representation, web based, automatic mathematical problems. Publications in the literature generally refer to the use and output of these software and do not contain information on calculation methodologies.

With the emergence of symbolic computing tools, mathematicians began using these tools to do proofs of theorems with computers. 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 on computer science related to symbolic and algebraic computation such as coding, robotic modeling, computer animations, signal / image processing have been done.

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. Research is needed to find more general solutions and to develop faster methods if necessary.

However, studies on the development of computer software to be used especially in the field of education, science and engineering are continuing by making use of the facilities of symbolic calculation. Today, the work in this area seems to focus on developing systems that work in parallel, with new languages and technologies, new user-friendly interface, ease of use, and new algorithms. In addition, systems are being developed to use numerical, geometric and symbolic calculation methods together.

3. GENERAL INFORMATION

3.1. Grammars For Formal Languages

Formal languages are generally regarded from the work of linguist Noam Chomsky in the 1950s, who tried to give a precise characterization of the structure of natural languages. His aim was to define the syntax of languages using simple and precise mathematical rules. Later it was found that the syntax of programming languages can be described using one of Chomsky's grammatical models called context-free grammars [47].

CFGs are universally used to describe the syntactic structure of programming languages, which are perfectly suited to describing recursive syntax of expressions and statements

A Context-free grammar is a 4-tuple (V, Σ, R, S) where

1. V is a finite set called the variables (non-terminals)
2. Σ is a finite set called the terminals,
3. R is a finite set of rules, where each rule maps a variable to a string $s \in (V \cup \Sigma)^*$
4. $S \in V$ is the start symbol

3.1.1. A Hierarchy of Grammars

Grammars are classified into four types by the form of their productions, which is called the Chomsky hierarchy. These classes are nested, with type 0 which is the largest and most general, and type 3 which is the smallest and most restricted.

Type- 0 Grammar (Unrestricted grammars)

Type-0 grammars generate any phrase structure grammar without any restriction. The productions can be in the form of $\alpha \rightarrow \beta$ where α is a string of terminals and non-terminals with at least one non-terminal and α cannot be null. β is a string of terminals and non-terminals.

Type- 1 Grammar (Context-sensitive grammars)

Type-1 grammars generate context-sensitive languages. The productions must be in the form $\alpha A \beta \rightarrow \alpha \gamma \beta$ where $A \in N$ (Non-terminal) and $\alpha, \beta, \gamma \in (T \cup N)^*$ (Strings of terminals and non-terminals)

The strings α and β may be empty, but γ must be non-empty.

Type- 2 Grammar (Context-free grammars)

Type-2 grammars generate context-free languages. The productions must be in the form $A \rightarrow \gamma$ where $A \in N$ (Non terminal) and $\gamma \in (T \cup N)^*$ (String of terminals and non-terminals).

Type- 3 Grammar (Right-linear or Regular grammars)

Type-3 grammars generate regular languages. This type of grammar should have a single non-terminal on both sides. The right hand side consist of a single terminal or single terminal followed by a single non-terminal. The productions must be in the form $X \rightarrow a$ or $X \rightarrow aY$ where $X, Y \in N$ (Non-terminal) and a ∈ T (Terminal).

3.1.2. Parsing Issues

3.1.2.1. Ambiguity

A terminal string $w \in L(G)$ is ambiguous if there exists two or more derivation trees for *w* or there exists two or more left most derivations of *w*.

A context free grammar is ambiguous if there exists some $w \in L(G)$, which is ambiguous.

For example, let us consider the expression "8 + 4 / 2" the derivation trees for this expression are shown below:

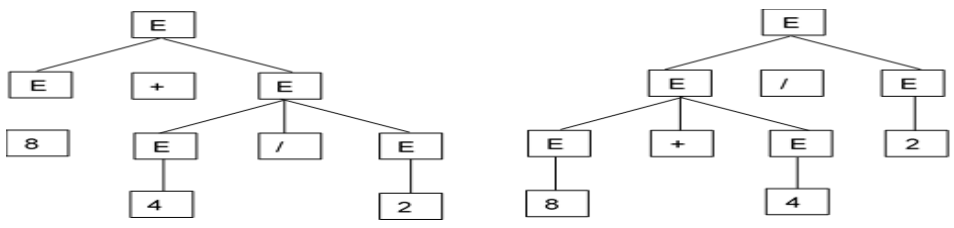

Figure 1. The different parse for expression "8 + 4 / 2"

3.1.2.2. Recursive Productions

Productions are often defined in terms of themselves. For instance a list of variables in a programming language could be indicated by this production:

```
variable_list → variable | variable_list , variable
```

The above production is refered as recursive. If the recursive nonterminal is at the left of the right-side of the production, e.g. $A \rightarrow u \mid Av$, we call the production left-recursive. Similarly, we can define a right-recursive production: $A \rightarrow u \mid vA$.

3.1.2.3. Left Factoring

Removing the common left factor that appears in two productions of the same non-terminal is called Left Factoring, the process of factoring out the common prefix of alternates. It is a useful method for manipulating grammars into a form suitable for recursive descent, it is done to avoid back-tracing by the parser.

Let:

`A→ α β | α γ` are two A-production and α ≠ null. 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 will become

`A→ α A'` `A' → β | γ`

3.1.3. Parsing Techniques

3.1.3.1. LL Parser

LL(k) parsers are one of the most basic and useful parsers of the top-down parsers. Here, the first “L” means reading the input from left to right, and the second means leftmost derivation. All LL parsers are called LL (k), and "k" denotes the number of tokens. Generally for k = 1, LL (k) may also be written as LL (1), then the grammar will be LL (1) grammar. To see whether a grammar is LL (1), first build the parse table then if one element in the

parse table contains more than one grammar rule with its right-hand side, then the grammar is not LL (1).

3.1.3.2. First and Follow Sets

The construction of a predictive parser is aided by two functions, FIRST and FOLLOW, allow us to fill in the entries of a predictive parsing table, whenever possible.

3.1.3.2.1. First Set

For a string of grammar symbols α, FIRST (α) is the set of terminals that begin all possible strings derived from α. If α → ε, then ε is also in FIRST (α).

Consider: E → T E′

E′ → + T E′ | ε

T → F T′

T′ → * F T′ | ε

F → (E) | id

Then FIRST (E) = FIRST (T) = FIRST (F) = {(, id}

FIRST (E′) = {+ , ε}

FIRST (T) = {(, id}

FIRST (T′) = {* , ε}

FIRST (F) = {(, id}

3.1.3.2.2. Follow Set

FOLLOW (A) for non-terminal A is the set of terminals that can appear immediately to the right of A in some sentential form. If A can be the last symbol in a sentential form, then $ is also in FOLLOW (A).

Consider: E → T E′

E′ → + T E′ | ε

T → F T′

T′ → * F T′ | ε

F → (E) | id

Then
```
FOLLOW (E) = { ) , $ }

FOLLOW (E') = FOLLOW (E) = { ) , $ }
FOLLOW (T) = { + , FOLLOW (E) } = { + ,  ) , $ }
FOLLOW (T') = { +,  ) , $ }
FOLLOW (F) = { * , +,  ) , $ }
```

3.1.3.3. LR Parser

It is the most popular type of bottom-up parsing technique. Here, the "L" again means reading the input from left to right, while the "R" means constructing the rightmost derivation and its grammar can describe more languages than LL grammars.

LR parser can handle a large class of context-free grammars and can detect the syntax errors as soon as they can occur. More information about LL and LR parsing algorithms can be referenced [48].

3.2. Least Squares Method

The least squares method is an important technique of regression analysis that fits a mathematical or statistical model to a particular data set [49]. Least Squares minimizes the square of the error between the original data and the values predicted by the equation. The purpose of the LSM method is to find coefficient estimates that will give these error terms as low as possible. The minimization of the error is achieved by equalizing the first derivative of the difference function to zero.

The LSM is widely used to find or estimate the numerical values of the parameters to fit a function to a set of data and to characterize the statistical properties of estimates.

The most important application of LSM is in data fitting.

3.2.1. Least Squares Method Introduction

Given data $\{(x_1, y_1), ..., (x_n, y_n)\}$, we define the error associated $p(x) = c_1 x_i + c_0$. To solve the coefficients c_1 , c_0 for error minimizing:

minimize $\mathrm{E}(c_1, c_0) = \sum_{n=1}^{N} \left(y_n - (c_1 x_n + c_0)\right)^2$.

Note that the error is a function of two variables, the unknown parameters c_1 and c_0. The goal is to find values of a and b that minimize the error. The purpose of the LSM method is to find coefficient estimates that will give these error terms as low as possible.

To minimize the error of the equation, take the derivative with respect to each coefficient c_1 , c_0

$$\frac{\partial \mathrm{E}}{\partial c_1} = \sum_{n=1}^{N} 2\left(y_n - (c_1 x_n + c_0)\right).(-x_n)$$

$$\frac{\partial \mathrm{E}}{\partial c_0} = \sum_{n=1}^{N} 2\left(y_n - (c_1 x_n + c_0)\right).(-1)$$

Setting $\frac{\partial \mathrm{E}}{\partial c_1} = \frac{\partial \mathrm{E}}{\partial c_0} = 0$ (and dividing by -2) yields

$$\sum_{n=1}^{N} \left(y_n - (c_1 x_n + c_0)\right).x_n = 0$$

$$\sum_{n=1}^{N} \left(y_n - (c_1 x_n + c_0)\right) = 0$$

Note we can divide both sides by -2 as it is just a constant; we cannot divide by x_i, as that varies with i. We may rewrite these equations as

$$\left(\sum_{n=1}^{N} x_n^2\right) c_1 + \left(\sum_{n=1}^{N} x_n\right) c_0 = \sum_{n=1}^{N} x_n y_n$$

$$\left(\sum_{n=1}^{N} x_n\right) c_1 + \left(\sum_{n=1}^{N} 1\right) c_0 = \sum_{n=1}^{N} y_n.$$

Re-write equation and put it into matrix form:

$$\begin{bmatrix} \sum_{n=1}^{N} x_n^2 & \sum_{n=1}^{N} x_n \\ \sum_{n=1}^{N} x_n & \sum_{n=1}^{N} 1 \end{bmatrix} \begin{bmatrix} c_1 \\ c_0 \end{bmatrix} = \begin{bmatrix} \sum_{n=1}^{N} x_n y_n \\ \sum_{n=1}^{N} y_n \end{bmatrix}$$

and to solve this problem, use Cramer's rule.

3.3. Language Processor

Language processor is a special type of a computer software program designed or used to perform tasks and has the capacity of translating the source code or program codes into machine codes. There are different types of language processors such as compilers,

assemblers, interpreters, preprocessors, and disassemblers. In this section, we will explain the most widely used language processors, which are compilers and interpreters.

3.3.1. Compilers

A compiler is a software that can read a program written in one language (source language) and translate it into an equivalent program in another language (target language) which can be understood by the processor. The main task of the compiler is to report if there is errors in the source language that are detected during the translation process. Examples of compiled programming languages are C and C++.

3.3.2. Interpreters

Interpreter is used a code to run on your processor, which is not the same as a compiler. An interpreter translates code like a compiler but reads the code and executes directly without previously converting them to an object code or machine code, and therefore is initially faster than a compiler. Each part of the code is interpreted and then execute separately in a sequence and an error is found in a part of the code it will stop the interpretation of the code without translating the next set of the codes. Examples of interpreted languages are Perl, Python and Matlab.

3.3.3. Differences Between Compilers and Interpreters

Both compilers and interpreters are translated the high-level language into machine language, but there are many differences between them.

The difference between an interpreter and a compiler is as follows:

- An interpreter reads one statement and translate it, after executing that statement it takes another statement in sequence. While the compiler reads the whole program and translates it in one go and then executes it.

- A compiler generates the error message only after the scanning of the whole program. Since an interpreter continues translating the program until the first error is met, and to interpret the next statement we have to fix the error.
- A compiler generates intermediate object code which needs more memory, and it will be generated every time when the program is being compiled. As an interpreter no intermediate object code is generated, it directly generates machine code.
- In analyzing and processing the source code a compiler takes larger amount of time comparatively and interpreter analyzes and processes the source code immediately.
- Besides the processing and analyzing time, programs produced by compilers run much faster than the same programs executed by an interpreter.

3.3.4. Compiler Phases

The basic compiler steps are displayed in Figure 2.

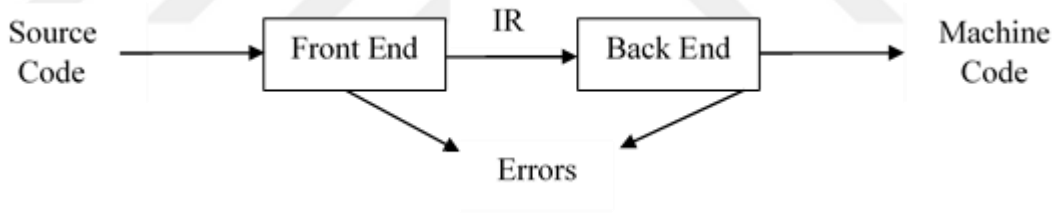

Figure 2. Structure of a Compiler

The two main components of the compiler are : the *front-end* and the *back-end.*

3.3.4.1. Front-End

The front end part consists of lexical analysis, syntactic analysis, semantic analysis, and the generation of object tree (intermediate code representation).These phases mainly depend on the source data. It is independent of the target machine.

In front-end part, code optimization can be done to reduce cost function.

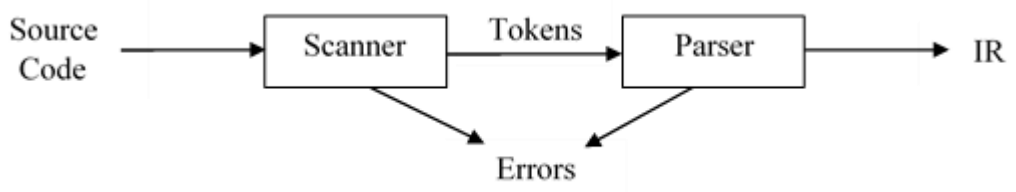

Figure 3. Structure of the Front End

3.3.4.2. Back-End

The second part of the compiler is the back-end phase which dependent on the target machine. The back-end phase include code optimization phase,the necessary error handaling, symbol table operations, and the final code generation. This phase of compiler is independent of source program.

The main task of the front-end phase is to analyze the source data and generate the object tree representataion while the back end synthesizes the target program from the object tree (intermediate code).

The generation of an intermediate code may be refered as middle end, as it depends upon source program and target machine.

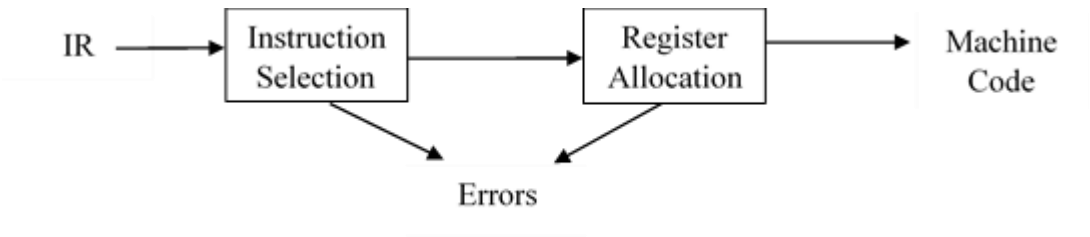

Figure 4. Structure of the Back End

3.4. Language Interpreters

In any language interpreter translating any program from one language to another, first the compiler breaks the source data to understand the meaning and the syntactic structure of the program, then it recombines in a different and meaninful way. The compiler performs

two main tasks; analysis at the front end, and does synthesis the back end. The analysis is usually broken up into: Lexical analysis, Syntax analysis and Semantic analysis.

3.4.1. Lexical Analysis (Scanning)

Lexical analysis is the first phase of a compiler. It works closely with the syntax analyzer, which reads input characters from the source program and groups them into lexemes to produce output as a sequence of tokens, by removing any whitespace or comments in the source code, and the lexical analyzer output passes to the syntax analyzer when it demands [50].

If the lexical analyzer detects an error such as invalid token, it generates an error. At this point, it may stop to process the input data or it may attempt continuing lexical analysis by skipping characters until a valid prefix is found.

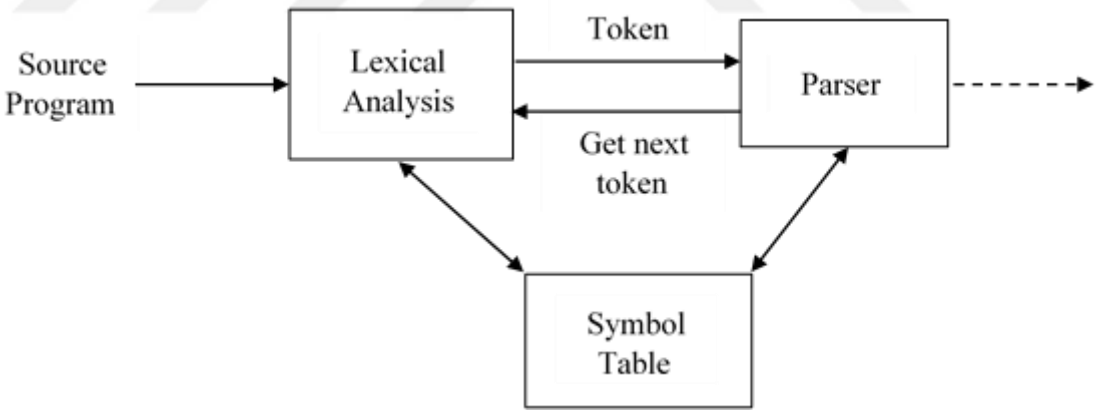

Figure 5. Interaction of lexical analyzer with parser

3.4.1.1. Regular Expressions

For lexical analysis, definitions are written and expressed using regular expressions which is an algebraic notation designed to describe sets of strings. Regular expressions are a useful tool designed for describing, matching and extracting patterns in text. Regular grammar is known as the grammar defined by regular expressions and the language defined

by regular grammar is known as regular language. Further reading for regular expressions can be referenced by Mogensen, and Torben Ægidius book [51].

The lexical analyzer needs to scan and recognize only a finite set of valid strings/tokens/lexemes that belong to the predefined language. It searches for the pattern defined by the language rules.

Every programmer should have the knowledge of implementing a tool for matching regular expressions from scratch.

Examples:

Regular Expression	Language
a∗	{ϵ, a, aa, aaa, ...}
(ab)∗	{ϵ, ab, abab, ababab, ...}
(a | b)∗	{ϵ, a, b, aa, ab, ba, bb, aaa, ...}
(a b | c)∗	{ϵ, c, ab, cc, abc, cab, ccc, abab, ...}

3.4.2. Syntax Analysis (Parsing)

Syntax analysis or parsing is the second phase of a compiler.

After lexical analysis splits the input into tokens, the goal of parsing is to recombine these tokens not into a list of characters, but into something that has meaning and reflects the structure of the text.

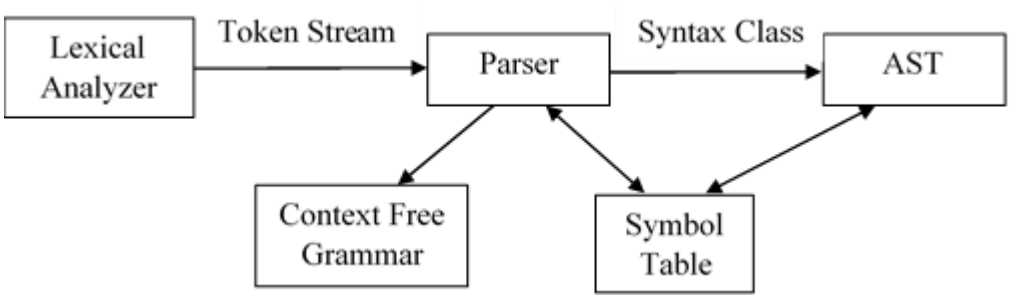

Figure 6. Syntax analyzer process

3.4.2.1. BNF

It is a standard notation developed by Noam Chomsky, John Backus, and Peter Naur for expressing syntax as a set of grammar rules. BNF is a formal mathematical procedure to identify context-free grammars. Backus-Naur Form has four symbols that has a special meaning.

< > ::= |

Given a context-free grammar (Σ, N, P, S), a non-terminal (a symbol in the alphabet N) is always enclosed in < and > (e.g. < expression>). A terminal (a symbol in the alphabet Σ) is often represented as itself, though in the context of computer languages a terminal symbol is often enclosed in single quotes. A production (non-terminal → symbols) in P is then represented as

< non-terminal > ::= symbols

The symbol | is used in BNF to combine multiple productions in P into one rule. For instance, if P := {S → A, S → B}, then P in BNF is < S > ::= A | B

- "::=" means "is defined as"
- "|" means "or"
- Angle brackets mean a nonterminal
- Symbols without angle brackets are terminals

Table 1. BNF type grammar for numerical expression

```
<expr> ::= '-' <num> | <num>

<num>  ::= <digits> | <digits> . <digits>

<digits> ::= <digit> | <digit> <digits>

<digit> ::= '0' | '1' | '2' | '3'| '4' | '5' | '6' | '7' |
'8' | '9'
```

3.4.2.2. EBNF

EBNF is the same as BNF, with additional meta-symbols:
It shows an optional opeartor. It means that the symbol (or symbols) to the left of the operator is optional (it can appear zero or one time)

• * : Which shows a repeating operator. It means that something can be repeated any number of times (and possibly be skipped altogether)

• + : Which shows the number of appearance of an elemnt. It means that something can appear one or more times

Expression of the EBNF type grammar is shorter than the expression of the BNF type grammar in Table 1, as can be seen in Table 2.

Table 2. EBNF grammar for number expression

```
<expr> ::= '-'? < digit>+ ('.' <digit>+)?

<digit> ::= '0' | '1' | '2' | '3'| '4' | '5' | '6' | '7' |
'8' | '9'
```

3.4.2.3. Syntax Classes

Syntax classes are defined to represent input data that can be generated by grammar rules as object-oriented programming constructs.

The definition of some syntactic classes is shown in Table 3

Table 3. Syntax classes

```
abstract class Exp {
public abstract double accept(Visitor v) ;
}
class Plus extends Exp {
  public Exp exp1, exp2;
  public Plus(Exp e1, Exp e2) {
    exp1 = el;
    exp2 = e2;
  }
  public double accept(Visitor V) {
    return v.visit(this);
  }
}
Class Euler extends Exp {
  public Exp exp;
  public Euler(Exp e) {
    exp = e;
  }
  public double accept(Visitor V) {
    return v.visit(this);
  }
}   …
```

3.4.2.4. Abstract Syntax Tree

An abstract syntax tree (AST) is just another representation of the source program in an intermediate code. But it is a representation that is much more submissive to analysis.

Each node in the abstract syntax tree is an object of a specific type, reflecting the underlying linguistic component or operation. After the grammar definition is done, an AST class is created for each word type [52,53]. According to JavaCC EBNF language knowledge, each element in the symbol list is placed in the object tree.

For a mathematical expression with addition, subtraction, division and multiplication only, the grammatical information to be written is as follows.

Table 4. Grammar Example for four operations

E → E + E
E → E − E
E → E * E
E → E / E
E → num

The AST classes to be created in response to this grammar are as shown in Table 5.

Table 5. AST classes for four operations

```
public abstract class Exp  { }
public class Plus extends Exp {
  public Exp el , e2;
  public Plus (Exp al, Exp a2){ el=al; e2=a2;}
}
public class Minus extends Exp {
…
}
public class Times extends Exp {
…
}
public class Divide extends Exp {
…
}
public class Num extends Exp {
public int n;
public Num (int n) { this. n = n; }
}
```

The parser grammar to produce AST from the above classes is as shown in Table 6.

Table 6. Grammar for four operations

```
Exp Start () :
{ Exp e; }
{ e=E () { return e; } }
Exp E () :
{ Exp e1, e2; }
{ e1=T () ( "+" e2=T () { e1 = new Plus (e1,e2) ;}
   |"-" e2=T () { e1 = new Minus (e1, e2); })*
{ return e1; }
Exp T() :
{ Exp e1, e2; }
e1=F() ( "*" e2=F() { e1 = new Times (e1, e2); }
   | "/" e2=F() { e1 = new Divide (e1, e2) ; })*
{ return e1; }
Exp F () :
{ Token it; }
{ t=<NUM> { return new Num (Integer.parseInt(t.image)); }
```

3.4.3. Semantics Analysis

Semantics analyzer is the third phase of the compiler and checks actual meaning of the statement parsed in a parse tree. It finds all possible remaining errors that would make program invalid, for example; a variable that is an integer cannot be directly equalized in a variable that is a string without being subject to any transformations because their token types are interpreted differently by the compiler.

3.4.4. Symbol Table

Symbol table is a data structure created and provided by compilers in order to check of identifiers used in the source program. It stores name, data type, procedure name, storage location, scope about identifiers, and other relevant information.

Symbol table is used by both the front-end and the back-end parts of a compiler.

For example, if a symbol table has to store information about the following variable declaration:

`static int interest;` then it should store the entry such as: `<interest, int, static>`. More information about symbol table is referenced [54].

3.4.5. Evaluation Methods

3.4.5.1. Instanceof Operator

In this method, the type of object that contains a node is determined by the `intanceOf` operator. After the object type is determined, the objects of the subclasses need to be derived from the upper class reference variable so that the represented process can be executed.

The related subclasses object can be derived using the cast structure.

Table 7 describes the object tree and the value of `exp` to be evaluated.

Table 7. Evaluation the `instanceof` operator

```
public static double eval (Exp exp , double x) {
if (exp instanceof Plus)
return eval(((Plus)exp).exp1, x) + eval(((Plus)exp).exp2, x);
else if (exp instanceof Minus)
return eval(((Minus)exp).exp1, x) - eval(((Minus)exp).exp2, x);
else if (exp instanceof Power)
return Math.pow(eval(((Power)exp).exp1, x), eval (((Power)exp).exp2, x));
else if (exp instanceof Euler)
return Math.pow(Math.E, eval(((Euler)exp)..exp, x));
```

3.4.5.2. Visitor Design Pattern

Visitor show off an operation to be performed on the elements of an object structure. Visitor allows you to define a new operation without affecting the classes of the elements on which it operates [55]. In this design template, a visitor class is created with a visit method for each AST object. Visitor class is an interpreter of the AST tree.
An example of visitor interface class is as in Table 8.

Table 8. Visitor interface

```
public interface IVisitor {
 public int visit (Plus e) ;
 public int visit (Minus e);
 public int visit (Times e);
 public int visit (Divide e);
 public int visit (Num e) ;
}
```

A visitor class is created by the defined interface. This class contains the necessary code for the bodies of all the methods in the interface. An interface can be used for multiple visitor classes.

A visitor class that implements the visitor interface in Table 8 is as in Table 9

Table 9. Visitor Class

```
Public class Visitor implements IVisitor {
  public int visit (Plus e) {
    return e.e1.accept(this)+e.e2.accept(this);
  }
  public int visit (Minus e) {
    return e.e1.accept(this)-e.e2.accept(this);
  }
  public int visit (Times e) {
    return e.e1.accept(this)*e.e2.accept(this);
  }
  public int visit (Divide e) {
    return e.e1.accept(this)/e.e2.accept(this);
  }
  public int visit (Num e) {
    return e. n;
  }  }
```

The visitor design template allows the source code to be more regular and readable than other methods. It also makes it easier for the developer to add or modify code. The visitor is an interpreter object with a visit method for each AST class. However, in this structure, each AST class must have an accept method that sends the task to the appropriate visit method. In this way, the control exchanges between the visitor and the AST class.

The node invoked by the visitor invokes the visit method corresponding to the class of the accept method. This object continues until the end of the tree.

In summary the visitor design template is a new evaluator.

3.5. Automatıc Code Generatıon Tools

Automatic code generation tools are symbolically used to calculate mathematical operations in a computer environment. They have been developed to facilitate the analysis, interpretation, or compilation processes of formal languages. There are many types of automatic code generation tools developed for different programming languages such as javaCC and sableCC.

3.5.1. JavaCC

JavaCC is a tool developed with Java that generates lexical analysis from regular expressions and parser from context free grammars and is known as a LL(k) parser generator.

Advantages of using JavaCC:

- Provides time saving in the production of compiler.
- Provides a standard encoding.
- Provides error-free development.
- Provides object-oriented development with the Java language.
- If the regex and grammar definitions are correct, it produces an error-free parser.

Productions of a JavaCC are the form:

```
void Assignment() : {} { Identifier() "=" Expression()
";" }
```

where `Assignment()`, `Identifier()`, and `Expression()` are nonterminal symbols; and "=" and "`;`" are terminal symbols.

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, enabling / disabling debug mode, and determining the target folder of the files to be created.

Then the body of the main class of the parser to be produced is defined. This can be done between the `PARSER BEGIN` and `PARSER END` tags to define and parse the main parser class.

The same name must follow `PARSER_BEGIN` and `PARSER_END`; This will be the name of the generated parser. 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 defined by the SKIP tag and are usually space and end-of-line characters. These can be added to other characters depending on the application.

Table 10 provides an example of a JavaCC configuration file.

Table 10. JavaCC configuration file

```
OPTIONS {
LOOKAHEAD = 1 ;
. . .
}
```

```
PARSER_BEGIN (parser_name)
. . .
class parser_name . . . {
. . .
}
. . .
PARSER_END (parser_name)
```

```
SKIP : { "  " | "\t" | "\n" }
TOKEN : {
   < NUM : ( [ "0"-"9" ] ) * >
 | < PLUS : "+" >
 | < MINUS: "-" >
 . . .
}
```

The configuration file is finally completed with the addition of grammar definitions. JavaCC uses the extended BNF to define grammar.

3.5.2. SableCC

SableCC is a parser generator object-oriented development environment working on Java, which is developed by Etienne Gagnon in 1998 as a graduate thesis. The SableCC parser generator produce syntax tree classes and using those classes it will build up syntax. To reach a shorter development cyle, SableCC isolates the machine generated code and user written code and it is LALR (1) based parser. Its productions are of the form:

```
assignment = identifier assign expression semicolon ;
```

where assignment, identifier, and expression are nonterminal symbols; and assign and semicolon are terminal symbols.

4. RESEARCH METHODOLOGY

4.1. Introduction

In engineering applications numerous methods have been developed for the computer environment to solve the mathematical problems. In general, numerical and symbolic approaches take the form of two main classes of computational methodologies.

Numerical methods involve a certain margin of error, and calculations for solving a problem are repeated until a certain error rate and/or a certain number of iterations are reached. Symbolic methods are based on finding the exact and error-free solution of mathematical problems with the help of computer environments. Manual resolution of these calculations may not be possible with long, faulty, and sometimes conventional methods.

In this study, we describe the process of developing an interpreter for the least squares method, where it first identifies the unknowns such as a and b in the input function and then takes a set of points. In some cases, the input function first needs converted into a polynomial function and then into an AST before the least squares method is applied to it.

4.2. General Structure of the Least Squares Interpreter

In the study, we describe the process of developing an interpreter for the least squares method. The interpreter language is formally defined using an Extended Backus Naur Form (EBNF). The grammatical form that is suitable for the syntactic and semantic structure of mathematical expressions has been developed and transformed into the LL (k) grammatical structure from left to right. In addition, syntax classes are defined to represent operators and functions that can be included in a mathematical expression. Before applying the method, the input function is evaluated through an AST to see whether it requires a possible conversion to a linear.

A token generator that takes the augmented generic input data and transforms it into a token sequence. It consists of a parser that generates a syntactic tree by checking the syntactic structure of the data from the token array, expression calculators that perform least squares, summation and simplification operations on the syntactic tree. The general structure of the application is shown in Figure 7.

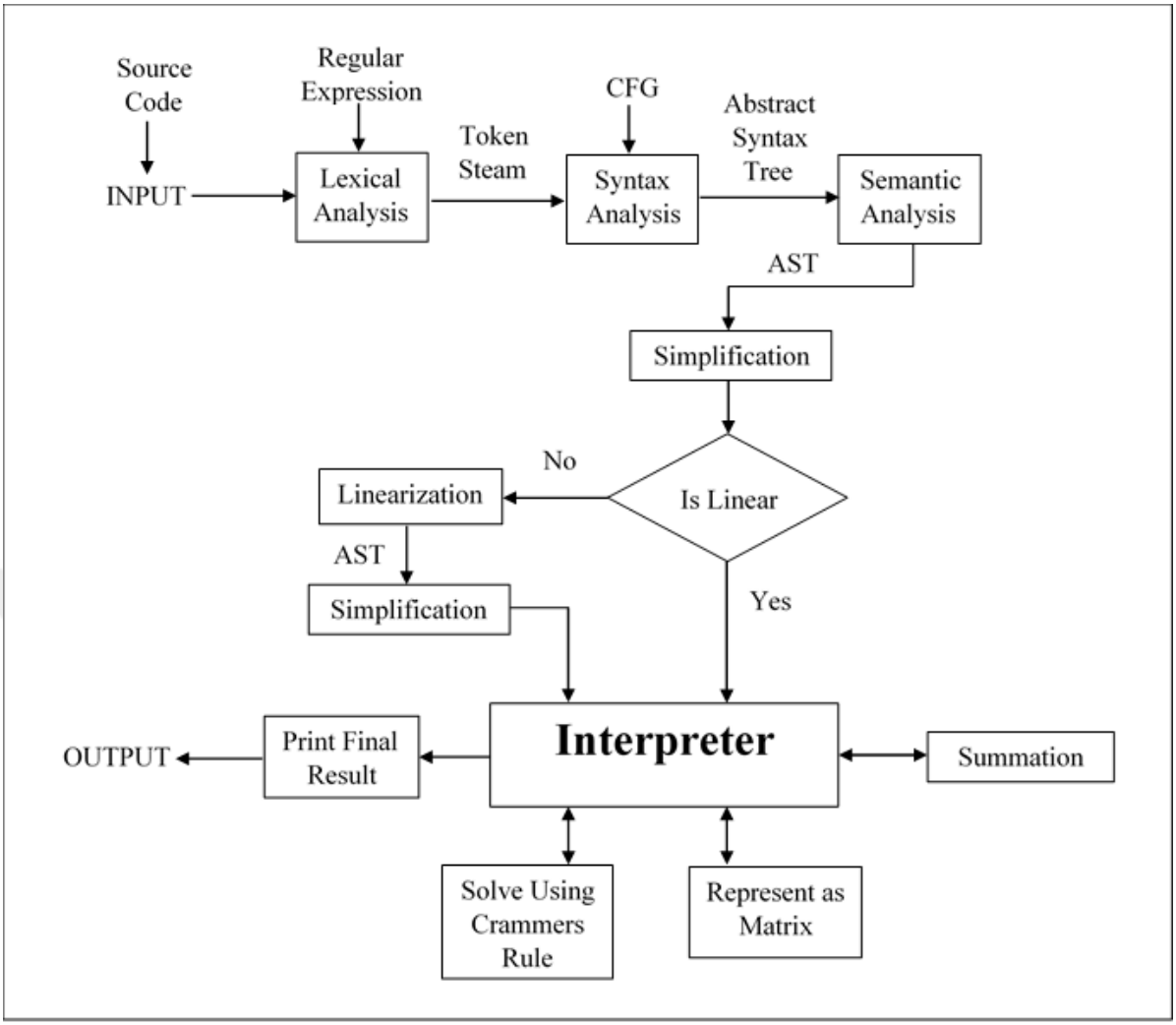

Figure 7. Architecture of the Least Squares Interpreter

4.2.1. Lexical Analysis

Lexical analysis, which is the first stage of compilation process, takes the modified source code from a system of least squares that are written in the form of sentences and breaks it into a sequence of tokens which is suitable for the syntax structure of the Java programming language, by skipping any whitespace or comments in the source code. At this point, if the application of least squares method for a given input is invalid it may stop reading the input or it may attempt continuing lexical analysis by skipping characters until a valid prefix is found.

4.2.1.1. Token Declaration

A token analyzer generates a token array by analyzing the input data in the direction of the global specifications. A token represents each word (terminal) in the input data that cannot be separated into smaller pieces.

According to the JavaCC notation, the token name and the regular expression of the represented set of words are given together in the token descriptions shown below in Table 11. Generally, each operator is represented by a different token class in the keyword.

Table 11. JavaCC token declaration for the application

```
TOKEN: {
   <PLUS: "+">
 | <MINUS: "-">
 | <TIMES: "*">
 | <DIVIDE: "/">
 | <POWER: "^">
 | <EQ: "=">
 | <COMMA: ",">
 | <LPAREN: "(">
 | <RPAREN: ")">
 | <NUM: (["0"-"9"])+("."(["0"-
                        "9"])+)?>
 | <X: "x">
 | <LN: "ln">
 | <E: "e">
 | <ID: ["a"-"z"](["a"-z|["0"-
                              "9"])*>
}
```

The mathematical expression is entered into the system in text format as follows: $a * \exp \wedge (b * x + c)$. According to the definitions in Table 11, this example is broken down into the token sequence given in Table 12

Table 12. Token Sequence

```
ID, TIMES, EXP, POWER, LPAREN, ID, TIMES, X, PLUS, ID,
LPAREN
```

The word that is not identified as token will be detected by the code generated by JavaCC and an appropriate error message will be displayed.

4.2.2. Syntax Analysis

In this stage, the source data format is defined, syntactic analysis is performed using the token array obtained from the lexical analysis. Lexical analysis splits the source input into tokens whereas the purpose of syntax analysis is to recombine these tokens using a predefined syntactic structure that is denoted by the CFG.

The syntactic rules of the Least Square Method are defined using the CFG then expressions that generate the object tree are added to these definitions. The application report syntactic errors in case of the parser detects any syntactic errors, and finally the application generates the object tee (AST) of the application. JavaCC parser generator tool is used to perform the syntax operations of the least squares equations and to generate the object tree of the least squares input data from the defined syntactic structure.

The JavaCC tool includes easy-to-use declarative constructs for describing data formats and building object trees. Syntax classes are defined for each rule of a grammar that formally represents the source data, including the token and non-terminals contained by the rule. After the grammatical structure is determined for the syntactic analysis stage, this grammatical structure must be adapted to LL(k). This is because JavaCC works with the LL(k) grammars. LL(k) grammars involves performing some necessary modifications on the CFG. Therefore, the left factoring process is performed by eliminating the recursive states from the left. After all these processes, the related grammar structure is transferred to JavaCC environment and Java code is produced which can perform the syntactical analysis requirements of the developed application.

In this section, the syntax structure developed for the least squares method and the JavaCC code structure which automatically generates the abstract tree of the syntax structure are explained in detail.

4.2.2.1. Syntax Structure of Least Squares Input Data

The JavaCC tool needs to produce LL (1) grammar by performing some operations on EBNF grammar such as eliminating the ambiguity and left factoring as shown in Table 13.

Table 13. LL (1) grammar definition for LSM

```
G={Σ, T, V, P, S}
V={Eq2,  Exp, Term, Power, Elem, Func, Num,    Id }⊆ Σ
T={log, Ln, exp, ( , ), +, -, *, /, ^, =, , , ;}⊆ Σ
Σ=T ∪ V
S={Eq}
Productions
<Eq2>     → <id>(<Exp>) = <Exp>
<Exp>     → ("+"|"-")? <term> [("+"|"-") <term> ]*
<Term>    → <power> [("*"|"/") <power>]*
<Power>   → < func > ("^"<power>)?
<Elem>    → <func> "(" <Exp> ")" | <num> | <id>
<Func>    → <num> | <id> | "log" | "ln" | "e"
<Num>     → "-"? ["0"-"9"] + ("."["0"-"9"]+)?
<Id>      →  ["a"-"w","A"-"W"](["a"-"w","A"-"W","0"-"9"])*
<Idn>     →   ["x"-"y"](["x"-"y"]|["0"-"9"])*>
```

Parsing an expression is processing the expression according to the grammar production rules. The name of methods in the JavaCC syntax description of least square expressions is determined according to the nonterminal set in Table 14, as shown in Table 13. All JavaCC methods are defined according to the grammar structure and, at the late stage, they are used to generate the related object tree, with the nodes constructed by the syntax classes, for the interpretation process.

Table 14. JavaCC Grammar Definition for LSM

```
Equation Eq2() :
{ Exp e, e1; Token t;  }
{ t = <ID> <LPAREN> e1 = E() <RPAREN>  <EQ>  e = E()
{ return new Function(t.image, e1, e ); }
}
Exp E() :
{ Exp e1, e2; Exp args5[] = new Exp[26];  }
{ e1=T()  (
      <PLUS> e2=T()  { e1 = new Plus(e1, e2); }
    | <MINUS> e2=T()  { e1 = new Minus(e1, e2); }  )*
  { return e1; }
}
Exp T() :
{ Exp e1 , e2;  }
{  e1 = P() (
      <TIMES> e2 = P()  { e1 = new Times(e1,e2); }
    | <DIVIDE> e2 = P()  { e1 = new Divide(e1,e2); } )*
  { return e1; }
}
Exp P() :
{ Exp e1, e2; }
{  e1=F() (
      <POWER> e2=P() { e1 = new Power(e1, e2); } )?
  { return e1; }
}
Exp F() :
{ Exp e; Token t;  }
{   t = <ID>  { return new Var(t.image);}
    | t = <IDN>  { return new Var(t.image);}
    | t = <NUM> { return new Num(Integer.parseInt(t.image)); }
    | <LPAREN> e = E() <RPAREN> { return e; }
    | <LN> <LPAREN> e=E() <RPAREN> { return new Ln(e); }
    | <E> <LPAREN> e=E() <RPAREN> { return new Euler(e); }
  ...
    }
```

4.2.2.2. Generating Abstract Syntax Tree

Based on the programming language used, there are many algorithms to build a parsing tree from a sequence of tokens**.** The Java language is used in this thesis as the implementation programming language, so an object-oriented parser constructed with the JavaCC generator tool generates the object tree. Each CFG rule 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 15. Abstract Syntax Tree for LSM

```
abstract class Equation {
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); }
   public double eval(double x) {
    return a.eval(x) + b.eval(x);
  } }
...
class Var extends Exp {
  String id;
  public Var(String x) { id = x; }
  public String toString( ) { return  id ; }
  public Object accept(Visitor v) { return v.visit(this); }
   public double eval(double x) { return Double.parseDouble(id);
  } }
class Num extends Exp {
  int n;
  public Num(int x) { n = x; }
  public String toString( ) { return  (String.valueOf(n));}
  public Object accept(Visitor v) {
      return v.visit(this);
   }
   public double eval(double x) {
    return n;
  }
}
```

4.2.3. Semantic Analysis

The aim of the semantic analysis is to check the actual meaning of the statement represented in the parse tree such that the token structures on the leaves of the tree start to make sense for the interpreter. Once the abstract tree is created, it is subjected to semantic analysis. The token expressions in the leaves of the abstract tree begin to make sense for the interpreter at this stage. For example; interpreter tokens are constructed as constants such as integer, variable, string, so that the corresponding token is processed according to the interpreter-loaded meaning in the program. The most important operations in this phase include type checking, scope resolution, and array bounds. As a consequence of these operations, some errors can occur semantically. Errors that may occur at this stage are called semantic errors. For example; a variable with an integer and a variable with a string cannot be directly equalized because their token types are interpreted differently by the interpreter.

In this section, semantic errors in the code structure of the source program are checked and data type information is determined for code generation operations. Type checking is the most important part of semantic analysis. The CFG used in syntactic analysis operations is combined with the semantic rules determined at this stage.

4.2.4. Simplification

When performing mathematical functions, it may be necessary to simplify the function expression. The reason for this is, during the intermediate steps of the evaluation process or before the printing process, that the tree has some parts to simplify and if the parts are not simplified, the structure of the function may contain a lot of unnecessary data which can complicate the interpretation process, reducing the legibility of the related function.

It is difficult to define a general simplification process that covers all mathematical expressions so it must be defined according to the expression or problem.

4.2.5. Transformation into polynomial

The least squares method usually uses a linear system of equations to obtain the values of unknowns in an input function. In fact, the equation system is derived from the input function, applying relative differentials on the unknowns in that function. In the case that the function is a polynomial, the resulting system will purely linear one and can be easily solved. In the other hand, when the function is nonlinear it first needs to converted into a polynomial function (by taking the log or the reciprocal of the data), and then least-squares method can be applied to the resulting linear equation.

Some particular functions that require the conversion into polynomial function before the least squares method is applied to it are exponential function, rational function and power function.

4.2.5.1. Exponential Functions

In order to apply the least squares method into an exponential function of the form $g(x) = A \,.\, e^{Bx}$, the function needs some necessary transformations of linearization. In this way, first it is converted into a polynomial of degree 1, applying the natural logarithm to both sides of the equation as follows.

$g(x) = A \,.\, e^{Bx}$

$ln\, g(x) = ln(A \,.\, e^{Bx})$

$ln\, g(x) = ln\, A + Bx \,.\, \ln e$

$\ln g(x) = \ln A + B \,.\, x$

This linearization is also applied to the related set of data points. After the data is linearized, the following substitutions are made to the equation

Let $\mathrm{z} = \ln(y)$

$a_0 = \ln(a)$, implying $a = e^{a_0}$

$a_1 = b$

The data now appears in the form of a linear model:

$z = a_0 + a_1\, x$

In the exponential model procedure, least squares linear regression method is used to solve for the a_0 and a_1 coefficients which are then used to determine the original constants of the exponential model, a and b, where $y = a\, e^{bx}$.

4.2.5.2. Power Functions

As with the exponential function, a power function of the form $g(x) = a\, x^b$ is modified via the same transformations. The application of the natural logarithm to both sides of the equation leads to the following linear function.

$g(x) = a\, x^b$

$ln\, g(x) = ln(a\, x^b)$

$ln\, g(x) = ln\, a + b \ln x$

After the linearization on data points, the following substitutions are made to the equation

Let $\mathrm{z} = \ln(y)$

$\mathrm{w} = \ln(x)$

$a_0 = \ln(a)$, implying $a = e^{a_0}$

$a_1 = b$

The linear model gets into the

$z = a_0 + a_1\, w$

In the power model procedure, least squares linear regression method is used to solve for the a_0 and a_1 coefficients which are then used to determine the original constants of the power model, a and b, where $y = a\, x^b$.

4.2.5.3. Rational Function

For rational functions to be handled by the least squares method we take the reciprocal of the function at both sides as follows:

$$y = \frac{a\, x}{b + x}$$

$$\frac{1}{y} = \frac{b}{a}\frac{1}{x} + \frac{1}{a}$$

Together with the linearization of data points, the following substitutions are made:

Let $z = \frac{1}{y}$

$q = \frac{1}{x}$

$a_0 = \frac{1}{a}$, implying $a = \frac{1}{a_0}$

$a_1 = \frac{b}{a}$, implying $b = \frac{a_1}{a_0}$

Then the data now reaches the form of a linear model:

$z = a_0 + a_1\, q$

In the saturation growth model procedure, least squares linear regression method is used to solve for the a_0 and a_1 coefficients which are then used to determine the original constants of the growth model, a and b, where $y = \frac{a\,x}{b+x}$.

4.2.6. Summation

With the least squares method, we calculate the sum of the squares of the x-values and also calculate the sum of each x-value multiplied by its corresponding y-values.

The sum of the squares of the x-values get very large according to the degree of polynomial. For example, a sum of x^4 needs for a 2nd degree polynomial (quadratic), x^6 needs for a 3rd degree polynomial and x^{16} for an 8th degree polynomial.

4.2.7. Representation by Jacobean Matrix

After the general polynomial regression model is developed using the method of the least squares, the coefficients of the polynomial model may be determined as Jacobean matrix as follows:

$$\begin{bmatrix} n & \sum_{i=1}^{n} x_i & \dots & \sum_{i=1}^{n} x_i^k \\ \sum_{i=1}^{n} x_i & \sum_{i=1}^{n} x_i^2 & \dots & \sum_{i=1}^{n} x_i^{k+1} \\ \vdots & \vdots & \vdots & \vdots \\ \sum_{i=1}^{n} {x_i}^k & \sum_{i=1}^{n} x_i^{k+1} & \dots & \sum_{i=1}^{n} x_i^{2k} \end{bmatrix} \begin{bmatrix} a_0 \\ a_1 \\ \vdots \\ a_k \end{bmatrix} = \begin{bmatrix} \sum_{i=1}^{n} f_i \\ \sum_{i=1}^{n} f_i\, x_i \\ \vdots \\ \sum_{i=1}^{n} f_i\, {x_i}^k \end{bmatrix}$$

The use of matrix representation will make things easier when it comes to applying the math to higher order polynomials.

4.2.8. Cramer's Rule

Cramer's rule allows us to solve the linear simultaneous system of equations, by finding the regression coefficients using the determinants of the square matrix. Each of the coefficients may be determined using the following equation:

$$a_k = {}^{D_k}/_{D}$$

$$\begin{bmatrix} \sum_{i=1}^{n} f_i & \sum_{i=1}^{n} x_i & \dots & \sum_{i=1}^{n} x_i^k \\ \sum_{i=1}^{n} f_i x_i & \sum_{i=1}^{n} x_i^2 & \dots & \sum_{i=1}^{n} x_i^{k+1} \\ \vdots & \vdots & \vdots & \vdots \\ \sum_{i=1}^{n} f_i {x_i}^k & \sum_{i=1}^{n} x_i^{k+1} & \dots & \sum_{i=1}^{n} x_i^{2k} \end{bmatrix}$$

The method of Cramer's rule is valid whenever the system has a unique solution (D is not equal to zero).

Table 16. Solving linear equations using Cramer's rule

```
public double[] cramers(double A[][],double B[])
  {
     double temp[][] = new double[N][N];
     double x[] = new double[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] = B[j];
             else
              temp[j][k] = A[j][k];
             }
         }
        x[i]=determinant(temp,N)/determinant(A,N);
     }
     for(int i=0;i<N;i++){
           String c = varibles.get(i).toString();
             FinalSolution.add(c);
             FinalSolution.add(" = ");
             double c1 = x[i];
                      FinalSolution.add(c1);
                      FinalSolution.add("\n");
          System.out.println(varibles.get(i).toString()+" =
"+x[i]);
     }
     return x;
      }
```

4.2.9. Printing Solution Values and Intermediate Steps

After calculating the final result, the application program will show all intermediate steps and final solution. To speed up the evaluation process of some functions, they are simplified and stored in the AST. In the printing process, a print visitor class is used, which contains the functions defined on AST to print them.

5. INTERPRETER ILLUSTRATION

In this section, a sample visual application to illustrate how the least squares interpreter works is described. The function that minimizes the sum of squared errors is entered in the user interface in the specified format. The least squares method estimates the values and displays it in the interface.

5.1. Input Data Format

First the function equation for the least squares method (mathematical expression) are entered into the interpreter as text format, then it checks whether the function input needs a transformation. If the input function is not polynomial it first converts into a polynomial function and then it identifies the unknowns such as a and b in the input function to estimate the coefficient values.

Table 17. Input data format for polynomial Least Squares Method

$p(x) = a_0 * x\text{^}0 + a_1 * x\text{^}1 + a_2 * \ x\text{^}2 + \cdots + a_n \ * x\text{^}n$
$x = \{val1, val2, val3, val4, \dots, valn\}$
$y = \{val1, val2, val3, val4, \dots, valn\}$

5.2. Step-By-Step Application of the Least Squares Method

In this section, to illustrate the methodology of the interpreter, we explain all intermediate steps needed to solve some given linear and nonlinear equations of different types in Table 18 and Table 19.

Table 18. Input data of the application (Polynomial)

$p(x) = a_0 + a_1\, x + a_2\, x^2$ $x = \{0, 0.2, 0.4, 0.7, 0.9, 1\}$ $y = \{1.016, 0.768, 0.648, 0.401, 0.272, 0.193\}$

Table 19. Input data of the application (Exponential)

$p(x) = A\,.\,e^{Bx}$ $x = \{0, 0.5, 1.25, 2, 2.7, 3, 3.5, 3.9, 4.75, 5.25\}$ $y = \{1.37, 1.48, 2.09, 2.77, 3.6, 4.1, 4.88, 6.01, 7.95, 9.9\}$

For the given input data in Table 18, we need to find or estimate the numerical values of the parameters a_0, a_1 and a_2 to fit a function to a set of data x and y and to characterize the statistical properties of estimates.

First the interpreter checks if the input equation is polynomial, then analyzes the input data format. If there is no error detected, an AST is generated as a hierarchical representation of the source data. The interpretation is carried out through the nodes of the AST. The following sections present the detailed illustration of all these phases.

5.2.1. Analysis Phase of the Interpreter

In the analysis phase the input data given in Table 18 are broken into constituent pieces of tokens according to the Token Declaration made in JavaCC for the interpreter as seen in Table 11. If there is no error in the format of the given input source data, the generated token sequence of the given input source data is given in Table 20.

Table 20. Token sequence of the given application input data

```
ID LPAREN ID RPAREN ASSIGN ID TIMES ID POWER NUM PLUS ID
                   TIMES ID POWER NUM
X ASSIGN LCB NUM COMMA NUM COMMA NUM … RCB
Y ASSIGN LCB NUM COMMA NUM COMMA NUM … RCB
```

After the sequence of tokens is generated, the syntax analyzer checks whether the token sequence it receives from the input as source data conforms to predefined syntax rules and creates an abstract syntax tree if it does not see any problem. Abstract syntax trees are often used as intermediate representations of the data between the front end and the back end. A sample token array and the intermediate code generated by the parser for this token array are shown in Table 21.

Table 21. The object tree for the input source data in Table 18

```
Eq(new Function (
            new Plus(new Times(new var (a_0),new Power(new
                                        Var(x),new Num(0))),
                     new Times(new var (a_1),new Power(new
                                       Var(x),new Num(1)))))
)
```

In the analysis phase an intermediate representation is constructed from the given source code. The remaining steps of the interpretation phase will be discussed in the next section, the following steps of the implementation of least squares method:

- Minimize the sum of the squares of the errors for the given function.
- Minimize E according to each coefficient by setting Partial Derivative
- Evaluate by dividing 2 at both sides
- Compute the sum of the squares of the x-values and also the sum of each x-value multiplied by its corresponding y-values
- Represent as Jacobian matrix form.
- Write into form of $AX = B$
- Solve using Crammer's Rule

5.2.2. Interpretation Phase

In this section, the solution using the equation AX = B which derived for the polynomials and the solution to be obtained from the equations obtained by derivation can be separated from each other.

In the interpretation phase, the final processing step of granting resources without any errors is performed here. At this stage, the source code for performing the least squares method (error estimation) starts with the function converter if it's necessary, then the partial differentiation of the given function with respect to each coefficient is calculated, the sum of the squares of the x-values and also the sum of each x-value multiplied by its corresponding y-values is computed, finally we represent as Jacobean Matrix into form of $AX = B$ solving this linear equation using Cramer's Rule.

Table 22. Least Squares Method interpretation process of object tree in Table 21

1) Minimize the sum of the squares of the errors	$E = \sum_{i=1}^{n} \varepsilon_i^2 = \sum_{i=1}^{n} [p(x_i) - f_i]^2 = \sum_{i=1}^{n} [a_0 + a_1 x + a_2 x^2 - f_i]^2$
2) Minimize E according to each coefficient by setting Partial Derivative	$\frac{\partial E}{\partial a_0} = 2 \sum_{i=1}^{n} [a_0 + a_1 x + a_1 x^2 - f_i] = 0$ $\frac{\partial E}{\partial a_1} = 2 \sum_{i=1}^{n} [a_0 + a_1 x + a_1 x^2 - f_i] . x = 0$ $\frac{\partial E}{\partial a_1} = 2 \sum_{i=1}^{n} [a_0 + a_1 x + a_1 x^2 - f_i] . x^2 = 0$
3) Evaluate by dividing 2 at both sides	$\sum_{i=1}^{n} (a_0 + a_1 x + a_1 x^2) = \sum_{i=1}^{n} f_i \ . \ 1$ $\sum_{i=1}^{n} (a_0 + a_1 x + a_1 x^2) \, x_i = \sum_{i=1}^{n} f_i \ . \ x_i$ $\sum_{i=1}^{n} (a_0 + a_1 x + a_1 x^2) \, x^2 = \sum_{i=1}^{n} f_i \ . \ x^2$
4) Compute the sum of the squares of the x-values and also the sum of each x-value multiplied by its corresponding y-values	$\sum_{i=1}^{6} x_i = 3.2$, $\sum_{i=1}^{6} f_i = 3.298$, $\sum_{i=1}^{6} {x_i}^2 = 2.5$, $\sum_{i=1}^{6} x_i f_i = 1.1313$, $\sum_{i=1}^{6} {x_i}^3 = 2.144$, $\sum_{i=1}^{6} {x_i}^4 = 1.923$, $\sum_{i=1}^{6} f_i \, {x_i}^2 = 0.744$
5) Represent as Jacobian matrix form.	$\begin{bmatrix} 6 & 3.2 & 2.5 \\ 3.2 & 2.5 & 2.144 \\ 2.5 & 2.144 & 1.923 \end{bmatrix}$
6) write into form of $AX = B$	$\begin{bmatrix} 6 & 3.2 & 2.5 \\ 3.2 & 2.5 & 2.144 \\ 2.5 & 2.144 & 1.923 \end{bmatrix} . \begin{bmatrix} a_0 \\ a_1 \\ a_2 \end{bmatrix} = \begin{bmatrix} 3.298 \\ 1.1313 \\ 0.744 \end{bmatrix}$
7) Solve using Crammer's Rule	$D = 0.2521, D_{a_0} = 0.2517, D_{a_1} = 0.2538$, $D_{a_2} = 0.0532$ $a_0 = D_{a_0} / D = 0.9986$ $a_1 = D_{a_1} / D = -1.0068$ $a_2 = D_{a_2} / D = 0.21103$

Where E denoted the sum of the squares, D is the determinant of the coefficient matrix, D_{a_0}, D_{a_1} and D_{a_2} are the determinants of the unknown vectors of a_0, a_1 *and* a_2 respectively. The given application is solved the polynomial function of curve fitting using the Least Squares Method. The final solution is:

$p(x) = 0.9986 - 1.006\,x + 0.21103\,x^2$

Table 23. Least Squares Method interpretation process which need transformation into polynomial

1) Check if it is polynomial and transform it if it's not	$p(x) = A\,.\,e^{Bx} \quad \rightarrow \quad p(x) = C + Bx$
2) Minimize the sum of the squares of the errors	$E = \sum_{i=1}^{n} \varepsilon_i^2 = \sum_{i=1}^{n} [p(x_i) - f_i]^2 = \sum_{i=1}^{n} [C + Bx - y_i]^2$
3) Minimize E according to each coefficient by setting Partial Derivative	$\frac{\partial E}{\partial C} = 2\sum_{i=1}^{n} [C + Bx - f_i] = 0$ $\frac{\partial E}{\partial B} = 2\sum_{i=1}^{n} [C + Bx - f_i]\,.\,x = 0$
4) Evaluate by dividing 2 at both sides	$\sum_{i=1}^{n} (C + Bx) = \sum_{i=1}^{n} f_i \;.\; 1$ $\sum_{i=1}^{n} (C + Bx)\,x = \sum_{i=1}^{n} f_i \;.\; x$
5) Compute the sum of the squares of the x-values and also the sum of each x-value multiplied by its corresponding y-values	$\sum_{i=1}^{10} x_i = 26.9$, $\sum_{i=1}^{10} Y_i = 12.9$, $\sum_{i=1}^{10} {x_i}^2 = 100$, $\sum_{i=1}^{10} Y_i\,x_i = 45$ $\quad Where \quad Y_i = \ln f_i$
6) Represent as Jacobian matrix form.	$\begin{bmatrix} 10 & 26.9 \\ 26.9 & 100 \end{bmatrix}$
7) write into form of $AX = B$	$\begin{bmatrix} 10 & 26.9 \\ 26.9 & 100 \end{bmatrix} . \begin{bmatrix} C \\ B \end{bmatrix} = \begin{bmatrix} 12.9 \\ 45 \end{bmatrix}$
8) Solve using Crammer's Rule	$D = 276.39, D_C = 79.5, D_B = 102.99$ $C = D_C / D = 0.2876$, $\quad B = D_B / D = 0.3726$
9) Apply with the giving function (Exponential Function)	$A = e^C = e^{0.2876} = 1.3333$ $p(x) = 1.3333\,e^{0.3726\,x}$

To apply this solution to the giving function ($p(x) = A \,.\, e^{Bx}$) we need to find A:

$A = e^{C} = e^{0.2876} = 1.3333$

and the final solution will be:

$p(x) = 1.3333\, e^{0.3726\, x}$

To simplify the usage of the program we have developed a simple interface.

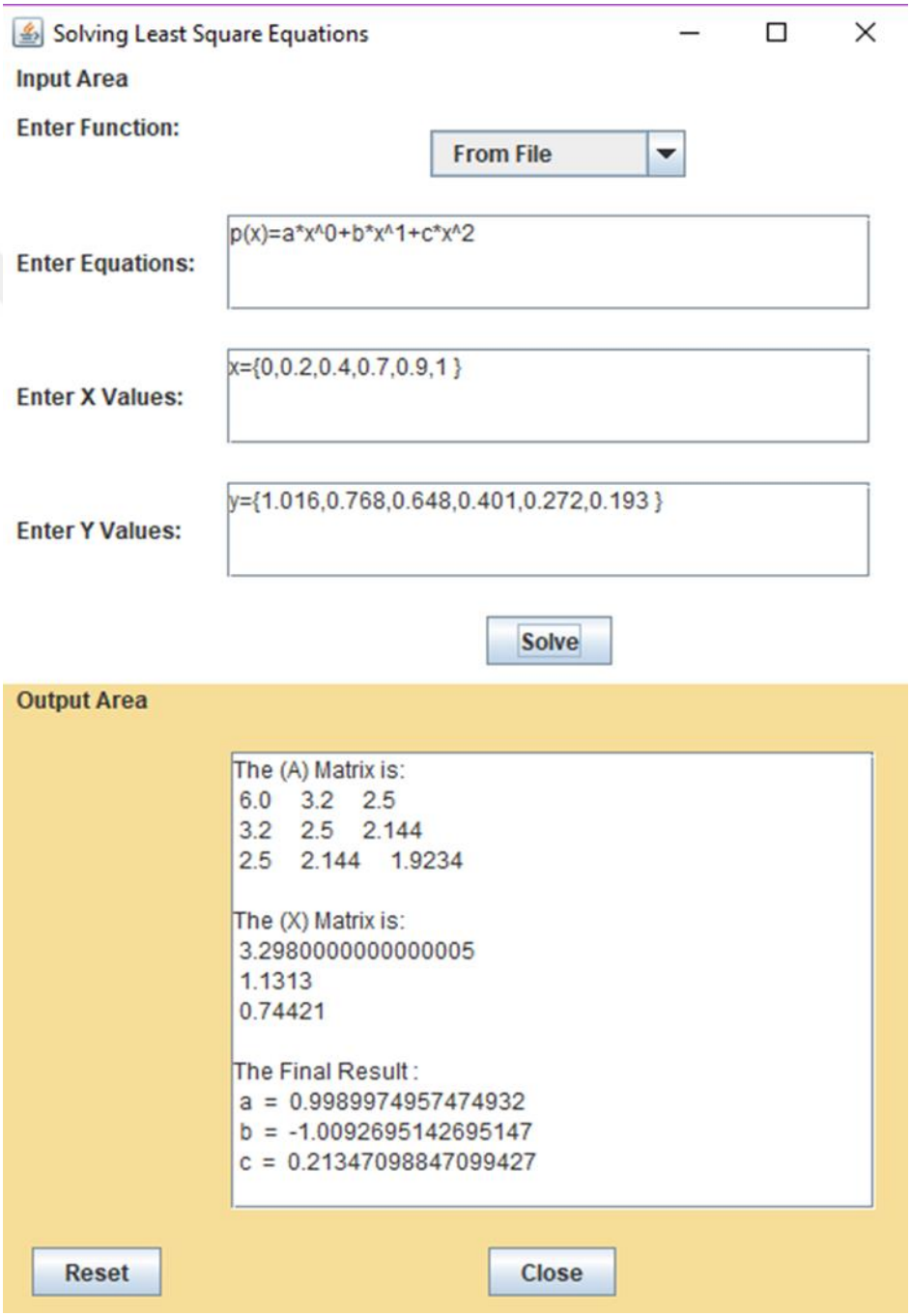

Figure 8. Application Interface

6. CONCLUSION

In this study, the development of an interpreter for the least squares method is addressed. As with the method, the input data of the interpreter are constructed by a finite set of points and an approximating function that can be represented via a formal grammar. It deals with the determination of some coefficients in the function that represents the given points with minimum error. The grammar rules of the related input language are defined in accordance with the specification requirements of the JavaCC tool, which generates code automatically from Java programming language. The development process consists of two main phases, namely data analysis and method interpretation.

The analysis phase performs two important operations such as lexical analysis and syntax analysis. The lexical analyzer (lexer) breaks the character sequence of the input data into sub-pieces called tokens and also identifies the kind of each token. Then the token sequence is examined by the syntax analyzer (parser) to check that it meets the syntax structure defined by the rules of the language grammar. On the proper sequencing of the tokens, the parser generates an intermediate code representation that has the hierarchical structure, called the abstract syntax tree (AST). This tree serves as an input to the other components of the interpreter responsible for further analysis and evaluation.

The interpretation phase performs various symbolic programming activities such as functional transformation and generation of summation expressions. More specifically, the programming process of the least squares method needs to carry out polynomial conversion, function evaluation and equation solving. Therefore, the least squares method is implemented through two basic visitor interfaces that has some evaluation methods defined on the AST produced by the parser. One visitor interface is used to transform the approximating function into a polynomial form. The other one is required to determine some parameters of the function, solving a linear system of equations derived from an input set of points.

In our task, given a set of points (x_i, y_i) and an approximating function of $(a + bx)$, we seek to find a value for a and b that minimizes the sum of the squared errors. The error

is decomposed into the difference between the observed value y_i and the predicted value $(a + bx_i)$ that we want to minimize with respect to two parameters of a and b. To carry out this task, we need a solid understanding of algebra, basic linear algebra (using matrices to solve the system of equations) and some of calculus (partial derivative and summations). The partial derivatives with respect to a and b end up with two equations with two unknowns. Then the last step is to solve the system of equations simultaneously for a and b using Cramer's rule.

The interpreter language is defined formally, using EBNF grammar, and can perform both numeric and symbolic calculations. Before applying the method, the input function is evaluated through an AST to see whether it requires a possible conversion to a polynomial (applying the natural logarithm to both sides of the equation). For such functions, the conversion allows an easy solution based on a system of linear models.

The least squares method usually uses a linear system of equations to obtain the values of unknowns in an input function. In fact, the equation system is derived from the input function, applying relative differentials on the unknowns in that function. In the case that the function is a polynomial, the resulting system will purely linear one and can be easily solved.

7. FUTURE WORKS

In the work, the least squares method is interpreted only for some types of approximating functions with a given set of data points. It can be extended to cover the another types such as trigonometric ones.

In our interpreter, to solve the linear equations we have used Cramer's rule that can solve only nonlinear equations with unique solutions. If the determinant of the coefficient matrix, denoted by D, is zero, then Cramer's Rule will not work because of divide by zero. In this case, there are obviously two scenarios that makes the system of equations inconsistent (no solution) and dependent (infinite solutions). These scenarios can also be supported with the use of a different technique of systems of equations such as Addition/Elimination or Substitution to find out and solve the type of the system.

In order to improve the usability of the interpreter, it can be moved to web environment, and visualization of operations and graphical structures, required for the minimization or representation of errors, can be obtained.

8. REFERENCES

1. J.V.Z. Gathen, J. Gerhard, Modern Computer Algebra, Cambridge University Pres, Third Edition, Cambridge, 2013.

2. W. Decker, Some introductory remarks on computer algebra, Proceedings of the third European Congress of Mathematics, Barcelona, 2000.

3. Guide, M.U.S., The MathWorks. Inc., Natick, MA 5 (1998) 333

4. 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.

5. Martin, W. A., ands Fateman, R. J., The MACSYMA system, Proceedings of the second ACM symposium on Symbolic and algebraic manipulation. ACM, (1971),59-75.Trott, M., The Mathematica guidebook for symbolics. Springer Science & Business Media, 2007.

6. Trott, M., The Mathematica guidebook for symbolics. Springer Science & Business Media, 2007.

7. http://www.axiom-developer.org/Axiom. 28-08-2017.

8. Aho A., Lam M., Sethi R., Ullman J., 2007. Compilers Principles Techniques and Tools (Second Edition). PEARSON, Boston.

9. W. Apel and J. Palsberg. Modern Compiler Implementation in Java. Cambridge University Pres, N. Delhi Revised Edition, 2007

10. T. J. Parr and R. W. Quong. ANTLR: A predicated-LL(k) parser generator. Software Practice and Experience, 25(7) (1995)789–810.

11. Gagnon, E. M., and Hendren, L. J., SableCC, An Object-Oriented Compiler Framework, Technology of Object-Oriented Language and System, 26(1998) 140-154.

12. JTB: Java Tree Builder, http://compilers.cs.ucla.edu/jtb/.

13. JavaCC, https://javacc.dev.java.net/

14. E. Berk. JLex:A lexical analyzer generator for Java(TM), http://www.cs.princeton.edu/~appel/modern/java/JLex

15. K. Gerwin. JFlex User's Manual, July 2005.

16. Boyle, A., and Caviness, B. F., Report of a Workshop on Symbolic and Algebraic Computation, Washington, DC, April 1988.

17. Nolan, J. F., Analytical Differentiation on a Digital Computer, SM Thesis, Massachusetts Institute of Technology, Massachusetts, 1953.

18. Kahrimanian, H. G., Analytical Differentiation by a Digital Computer, MA Thesis, Temple University, Philadelphia, 1953.

19. https://en.wikipedia.org/wiki/Lisp_(programming_language) Lisp (programming language). 16 June 2016.

20. Slagle, J. R., A Heuristic Program That Solves Symbolic Integration Problems in Freshman Calculus, Journal of the ACM (JACM), 10, 4 (1963) 507-520.

21. Berlekamp, E., Factoring Polynomials Over Finite Fields, Bell System Technical Journal, 46, 8 (1967) 1853-1859.

22. Zassenhaus, H., On Hensel Factorization, I., Journal of Number Theory, 1, 3 (1969) 291-311.

23. Musser, D., Multivariate Polynomial Factorization, Journal of the ACM (JACM), 22, 2 (1975) 291-308.

24. Wang, P. S., and Rothschild, L. P., Factoring Multivariate Polynomials Over the Integers Mathematics of Computation, 29, 131 (1975) 935-950.

25. Risch, R., The Problem of Integration in Finite Terms, Transactions of the American Mathematical Society, 139 (1969) 167-189.

26. 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.

27. 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.

28. 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.

29. 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.

30. 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.

31. http://maxima.sourceforge.net/SourceForge. 10-07-2018.

32. http://www.axiom-developer.org/Axiom. 10-07-2018.

33. http://magma.maths.usyd.edu.au/magma/Magma. 10-07-2018.

34. http://www.sagemath.org/SageMath. 10-07-2018.

35. https://www.mathworks.com/products/symbolic/ MATLAB 10-07-2018.

36. http://home.bway.net/lewis/. Femat. 10-07-2018.

37. http://www.math.uiuc.edu/Macaulay2/Macaulay. 10-07-2018.

38. http://page.math.tu-berlin.de/~kant/kash. Kant/Kash. 10-07-2018.

39. http://cocoa.dima.unige.it/Cocoa. 10-07-2018.

40. 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.

41. Park, H., Symbolic computation and signal processing, Journal of Symbolic Computation, 37,2 (2004) 209-226.

42. Miyazaki, Y., Iguchi, Y., and Watanabe, T., Information-Retrieval Tool for Math Expressions as Infrastructure of Training Engineers via E-Learning, 8th IFAC Symposium on Advances in Control Education, October 2009, Kumamoto City, Bildiriler Kitabı: 302-306.

43. Tekbaş, Y., *Code Production Tools Using Automatic Calculation of Derivatives and Simplification Mathematical Expressions*. Master Thesis, Karadeniz Technical University, Institute of Science and Technology, Trabzon, 2013.

44. Milani, M. M. R., *Design and Applications of Grammar-based Methodologies for Automatic Generation and Step-by-Step Solving of Mathematical Expressions*. Doctoral Thesis, Karadeniz Technical University, Institute of Science and Technology, Trabzon, 2015.

45. Gökgöz, B., *Design and Implementation of a general Interpreter for Numerical Root Finding Methods Using Symbolic Approaches*, Master Thesis, Karadeniz Technical University, Institute of Science and Technology, Trabzon, 2016.

46. Abdi, H., *Least Squares*, University of Texas at Dallas 2010.

47. Kenneth W. Regan, Department of Computer Science, State University of New York at Buffalo, Buffalo, NY 14260, USA

48. Dick, G., and Ceriel, H., *Parsing techniques*, a practical guide, Technical Report, Tech. Rep, 1990.

49. Mohamed, N., A., Pehlivan, H., *Design and Implementation of an Interpreter for the Least Squares Method Using Symbolic Approaches*, 5th International Conference on Advanced Technology & Sciences ICAT'17, May 2017, Istanbul, 129-132.

50. https://www.tutorialspoint.com/compiler_design. Parser types. 30 January 2018.

51. Mogensen, T.Æ., *Basics of compiler design*, Torben Ægidius Mogensen, 2009.

52. Appel, A. W., "*Modern Compiler Implementation in Java*", Cambridge University Press, 2002.

53. Campell, B., Lyer S. and Akbal, D. B., *"Introduction to Compiler Construction in a Java World*", CRC Press, 2013.

54. Aho, A.V., Ravi S., and Jeffrey D.U., *Compilers, Principles, Techniques*. Boston: Addison wesley, 1986.

55. Gamma E., Helm, R., Johnson, R. Ve Vlissides, J., "*Design Patterns: Elements of Reusable Object-Oriented Software*". Addison-Wesley, 1995.

.

CURRICULUM VITAE

Nawal ABDULLAHI MOHAMED was born in Jeddah, Saudi Arabia in June 1989. She graduated from Imamu Shafi'i Primary and Secondary School, Mogadishu, Somalia in 2008. She got her B.Sc. from the Computer Science department at Mogadishu University, Mogadishu, Somalia in 2012. She knows Arabic, English and Turkish languages well.