Modelling of Process Systems with Genetic Programming

Marco Lotz

Thesis submitted in partial fulfilment of the requirements for the Degree

MASTER OF SCIENCE IN ENGINEERING

(CHEMICAL ENGINEERING)

In the Department of Process Engineering at the University of Stellenbosch

Supervised by: Professor C. Aldrich

December 2006

Declaration:

I, the undersigned, hereby declare that the work contained in this thesis is my own original work and I have not previously in its entirety or in part submitted it at any university for a degree.

…….. ……..

Marco Lotz

Abstract

Genetic programming (GP) is a methodology that imitates genetic algorithms, which uses mutation and replication to produce algorithms or model structures based on Darwinian survival-of-the-fittest principles. Despite its obvious potential in process systems engineering, GP does not appear to have gained large-scale acceptance in process engineering applications. In this thesis, therefore, the following hypothesis was considered:

> *Genetic programming offers a competitive approach towards the automatic generation of process models from data.*

This was done by comparing three different GP algorithms to classification and regression trees (CART) as benchmark. Although these models could be assessed on the basis of several different criteria, the assessment was limited to the *predictive power* and *interpretability* of the models. The reason for using CART as a benchmark, was that it is well-established as a nonlinear approach to modelling, and more importantly, it can generate interpretable models in the form of IF-THEN rules.

Six case studies were considered. Two of these were based on simulated data (a regression and a classification problem), while the other four were based on real-world data obtained from the process industries (three classification problems and one regression problem).

In the two simulated case studies, the CART models outperformed the GP models both in terms of predictive power and interpretability. In the four real word case studies, two of the GP algorithms and CART performed equally in terms of predictive power. Mixed results were obtained as far as the interpretability of the models was concerned. The CART models always produced sets of IF-THEN rules that were in principle easy to interpret. However, when many of these rules are needed to represent the system (large trees), the tree models lose their interpretability – as was indeed the case in the majority of the case studies considered.

Nonetheless, the CART models produced more interpretable structures in almost all the case studies. The exception was a case study related to the classification of hot rolled steel plates (which could have surface defects or not). In this case, the one of the GP models produced a singularly simple model, with the same predictive power as that of the classification tree. Although GP models and their construction were generally more complex than classification/regression models and did not appear to afford any particular advantages in predictive power over the classification/regression trees, they

could therefore provide more concise, interpretable models than CART. For this reason, the hypothesis of the thesis should arguably be accepted, especially if a high premium is placed on the development of interpretable models.

Opsomming

As metode boots genetiese programmering (GP) genetiese algoritmes na, wat mutasie en reproduksie gebruik om algoritmes of model-strukture, gebaseer op Darwin se oorlewing van die sterkste konsep, te skep. Literatuur bewys dat, ten spyte van die voor die handliggende voordele van GP as metode in stelsel-ingenieurswese, dit nog nie op groot skaal in prosesingenieurswese toegepas word nie. Dus word die volgende hipotese in die tesis geëvalueer:

> *Genetiese programmering bied 'n kompeterende benadering tot die outomatiese generering van prosesmodelle van data.*

Dit is bereik deur drie verskillende GP algoritmes te vergelyk met klassifikasie-en regressie-bome (classification and regression trees, CART) as maatstaf en verwysing. Alhoewel die modelle op verskillende kriteria beoordeel kon word, was die beoordeling beperk tot die voorspelbaarheidsvermoë en interpreteerbaarheid van die modelle. Ses gevalle-studies was bestudeer. Twee gevalle-studies was gebaseer op gesimuleerde data ('n regressie-en 'n klassifikasie-probleem), terwyl die ander 4 gebaseer was op egte data verkry van prosesindustrieë (drie klassifikasie-probleme en een regressie-probleem).

In die twee gesimuleerde gevalle-studies het die klassifikasie-en regressie-modelle (CART) beter gevaar as die GP modelle in beide die kriteria van voorspelbaarheidsvermoë en interpreteerbaarheid. In die vier gevalle-studies gebaseer op egte industrie data het die GP modelle (uitgesluit GP250) ongeveer so goed gevaar as die CART modelle aangaande voorspelbaarheidsvermoë. Wisselende resultate is verkry aangaande die interpreteerbaarheid van die modelle. Die CART modelle het altyd AS-DAN (IF-THEN) reëls opgelewer wat in prinsiep maklik interpreteerbaar is. Dit gesê, as daar baie van die reëls (groot boomstrukture) is, dan verloor die modelle hul interpreteerbaarheid – in die meeste gevalle-studies was dit die geval.

Nietemin, die CART modelle het in omtrent al die gevalle-studies modelle geproduseer wat meer interpreteerbaar is. Die uitsondering was die gevalle-studie aangaande die klassifikasie van staalplate (wat foute op die oppervlakte het al dan nie). In die gevalle-studie het een van die GP modelle 'n enkele eenvoudige reël opgelewer, met dieselfde voorspelbaarheidsvermoë as die klassifikasie-boom.

Alhoewel GP modelle en hulle konstruksie oor die algemeen meer kompleks was as regressie/klassifikasie modelle, sonder om enige noemenswaardige verbetering in voorspelbaarheidsvermoë te lewer, kon hulle wel meer

kompakte, interpreteerbare modelle lewer in vergelyking met die CART modelle. Vir die rede word die hipotese van die tesis met voorbehoud aanvaar.

Acknowledgements and Dedications

To God, the Holy Trinity, for not being as impatient with me as I am with Him.

To my parents, Christo and Nita, who, in the words of Koza (1992), were “best of generation individuals…”

To Christo, my older brother, for showing me that in life your opponent might be bigger, stronger and faster, but that is no reason to come second. To Sanet for adding a dimension to our family life that can not be accomplished by brothers alone. To their unborn child who will enter this world with so much love waiting.

To Professor Chris Aldrich for his guidance. I can only hope that my professional career can be as fulfilling as the later part of my academic career. In no small part can this be attributed to Professor Chris Aldrich.

To Juliana Steyl and Mr M.O. Pienaar for their administrative excellence.

To Dr Sara Silva and Dean Barker for guidance during the coding process. Also a special word of thanks to Dean for his friendship.

To Thys Lourens who lived a full life in 24 years and 10 months, those who had to witness his passing and the rest of us who still finds him in our dreams. We are eternally less without you. The late nights were for you my friend.

With regards to this research:

I never did a day's work in my life. It was all fun.

- Thomas A. Edison

With regards to writing the thesis:

We have a habit in writing articles published in scientific journals to make the work as finished as possible, to cover up all the tracks, to not worry about the blind alleys or describe how you had the wrong idea at first, and so on. So there isn't any place to publish, in a dignified manner, what you actually did in order to get to do the work.

- Richard Feynman

Table of Contents:

ACKNOWLEDGEMENTS AND DEDICATIONS 7

1. INTRODUCTION 12

2. BASIC CONCEPTS AND APPLICATIONS OF GENETIC PROGRAMMING IN PROCESS ENGINEERING 16

2.1 The Genetic Programming algorithm 16

2.2 Creation of generation 1 21

2.2.1 Cloning or Copying 21

2.2.2 Crossover or sexual recombination 21

2.2.3 Mutation 23

2.3 Feasibility and Applicability of GP 25

2.4 Current Applications of GP in Process Engineering 27

2.4.1 Modelling two continuous stirred tank reactors in series (McKay et al., 1996) 27

2.4.2 Binary vacuum distillation (Willis et al., 1997) 28

2.4.3 Screw cooking extruder (Willis et al., 1997) 29

2.4.4 Acid pressure leaching of nickeliferous chromites (Greeff and Aldrich, 1997) 29

2.4.5 Uranium and radium liberation models (Greeff and Aldrich, 1997) 30

2.4.8 Simultaneous zinc and lead Imperial smelting process (Chen et al., 2003) 33

2.5 Conclusions from the literature review 34

3. GENETIC PROGRAMMING SOFTWARE 37

3.1 The use of GP in the current research 37

3.2 GP algorithms used in this study 38

3.2.1 GPLAB Toolbox (Silva, 2004) 38

3.2.2 GP250 (Swart and Aldrich, 2001) 39

3.2.3 Discipulus™ 40

4. CASE STUDY 1: ORDINARY LEAST SQUARES REGRESSION 43

4.1 Regression by means of MATLAB's Statistics Toolbox 43

4.2 Regression by means of Silva's Genetic Programming Toolbox 45

4.3 Regression by means of GP250 Toolbox 48

4.4 Regression by means of Discipulus 49

4.5 Summary of investigation of OLS Regression 51

5. CASE STUDY 2: CHESS BOARD PATTERNS 53

5.1 Regression by means of MATLAB's Statistics Toolbox 53

5.2 Tree – development by means of Silva's Genetic Programming Toolbox 55

5.3 Tree – development by means of GP250.m employing Genetic Programming 58

5.4 Regression by means of Discipulus. 59

5.5 Summary of investigation of Chess Board classification example 60

6. CASE STUDY 3:PRECIPITATION OF ZINC IN AMMONIACAL SOLUTIONS (*HYDROLOGY* DATA SET) 63

6.1 Non-Linear Tree Regression Analysis 63

6.2 Tree – development by means of Silva's Genetic Programming Toolbox 65

6.3 Tree – development by means of GP250.m employing Genetic Programming: 69

6.4 Regression model development by means of Discipulus 70

6.5 Summary of investigation of Hydrology classification example 72

7. CASE STUDY 4: PLATINUM FROTH FLOTATION SYSTEM 75

7.1 Introduction 75

7.2 Discussion of Data 75

7.3 Defining the Benchmark approach 76

7.4 Generating classification rules with GP using GPLAB 78

7.5 Interpreting GP produced classification rules 79

7.6 Test set results 87

7.7 Generating classification rules with GP using GP250.m 88

7.8 Generating a classification model using Discipulus 90

7.9 Test set results 91

7.10 Summary of investigation of Flotation Froth classification example 92

8. CASE STUDY 5: VISCOSITY INDEX ON AN INDUSTRIAL LIQUID-LIQUID EXTRACTION PLANT 95

8.1 Regression by means of statistical tree regression 96

8.2 Regression by means of Silva's Genetic Programming Toolbox 97

8.2 Regression by means of GP250 99

8.3 Regression by means of Discipulus 100

8.4 Summary of investigation of Viscosity Index regression example 100

9. CASE STUDY 6: CLASSIFICATION OF SURFACE DEFECTS IN STEEL PLATES 103

9.1 Discussion of data and benchmark method 103

9.2 Regression by means of Silva's Genetic Programming Toolbox 105

9.2 Regression by means of GP250 107

9.3 Regression by means of Discipulus 108

9.4 Summary of investigation of Steel Plate surface defect classification 108

10. DISCUSSION OF RESULTS AND CONCLUSIONS 111

10.1 Robustness of GP models 111

10.3 Conclusions and general remarks 116

10.4 Objectives revisited 118

11. FUTURE WORK 120

ADDENDUMS 122

Addendum A: MATLAB conversion of C/C++ code of case study 1 122

Addendum B: C/C++ code of case study 2 126

Addendum C: Typical mathematical expression of case study 2 129

Addendum D: C/C++ code of case study 3 134

Addendum E: C/C++ code of case study 4 137

Addendum F: C/C++ code of case study 5 140

Addendum G: C/C++ code of case study 6 143

REFERENCES 146

1. Introduction

> The best way to become acquainted with a subject is to write a book about it.
>
> - Benjamin Disraeli

Modelling of any chemical or mineral engineering process is extremely important, because it enables the process engineer to gain insight into the behaviour of a specific system (Zquez-Roman and King, 1996). This insight through modelling can be used to control a process. Even if the model cannot be interpreted, such as in the case of neural networks, a reliable model can still be very useful for process control or other forms of decision support.

A model aims to approximate reality. Many chemical engineering processes are complex and depend on many variables. Deriving an exact model can sometimes not be achieved due to the non-deterministic nature of the problem and the large number of variables concerned. An example of this is the rolling system used in steel manufacturing (Oduguwa, 2005a, 2005b). If an exact model could indeed be derived, it might be too complex to be useful. The question then becomes when is a model a "good enough" approximation? There is no simple answer to this question. Some guidelines are given by Marlin (2000) when he states that a model exerts sufficient control if it can, within limits, guarantee the following:

- Safety – No price can be placed on human lives and any model should be able to guarantee the safe operability of a process. Safety is also of legislative concern (Occupational Health and Safety Act, Act 85 of 1993).

- Environmental protection – Increased legal constraints and good engineering practice requires that a model should ensure environmental protection (National Water Act, Act 36 of 1998, National Air Act, Act 39 of 2004, Environment Conservation Act 73 of 1989).

- Smooth plant operation and production rate – Marlin (2000) argues that a model should be able to limit unwanted fluctuations in a process. These fluctuations could damage equipment.

- Product quality – A model should be sufficiently accurate to ensure that the product quality remains acceptable.

- Profit optimization – Without compromising on safety a model should be able to optimize the attained profit by manipulating the other discussed

model criteria above. These criteria include pollution production, product production rate and product quality.

- Monitoring and diagnosis – A model should also be able to provide process specific information necessary for solving problems experienced.

Data-driven models are often used when dealing with complex process systems, since they allow the rapid development of useful models, particularly where first-principles approaches would be intractable or costly to develop. Decision trees (Utgoff, 1999) are often used for this purpose and Androulakis (2004) argues that the three main reasons for using these models are:

- The resulting model is easy to understand and provides insight into the process for humans.

- These models are suited for exploratory modelling because of the lack of a priori knowledge of the system.

- The scalability of the algorithm provides gradual model predictive degradation.

The application of decision trees was illustrated by Corma (2005) by using high-throughput characterization in combinatorial heterogeneous catalysis.

Decision trees are an example of an explanatory modelling technique. Explanatory models aim to produce transparent models. The transparency of explanatory models is that the explicit structure of the model is given (Herrmann, 1997). This enables the user to investigate and interpret the relative importance of each variable. Regression and classification trees can successfully produce transparent explanatory models in many cases. Unfortunately regression and classification trees sometimes produce large numbers of rules. If two decision trees (classification or regression) offer the same accuracy the tree with the fewer discreet rules is preferred by the machine learning community (Androulakis, 2004). According to Androulakis the idea that the accuracy of a model is associated with its complexity dates back at least as far as William of Ockam's razor: "one should not increase, beyond what is necessary, the number of entities required to explain anything."

Evolutionary computing offers an alternative method for producing transparent explanatory models. In the late 1950's researchers began to realize the potential to utilize the ability of repetitive calculations performed by computers to evolve better strategies as computer programs (Holland et al., 1962). As with many inventions mimicking nature seemed to provide an elegant solution.

Evolutionary computing harnesses the power of natural selection to turn computers into optimisation tools (Odugawa, 2005 (B)). Evolutionary programming was based on the controversial theory of evolution by Darwin (1859). Darwin (1859) based his theory on observing nature while on his voyages. Only in the early 1990's has the understanding and implementation of new strategies, combined with the required computational power, allowed researchers to evaluate the usefulness of evolutionary computing for realistic problems (Koza, 1992).

The evolutionary programming strategy employed in this thesis is that of genetic programming. Genetic programming is a specific type of evolutionary programming. The basis of the genetic programming paradigm is that entities that perform tasks crucial to their survival better than other entities will survive and reproduce at a higher rate than the poor performing entities. The genetic programming (GP) algorithm treats individual computer programs as genetic individuals potentially capable of recombining or changing to form new individuals. GP will be discussed in more detail in the following chapter.

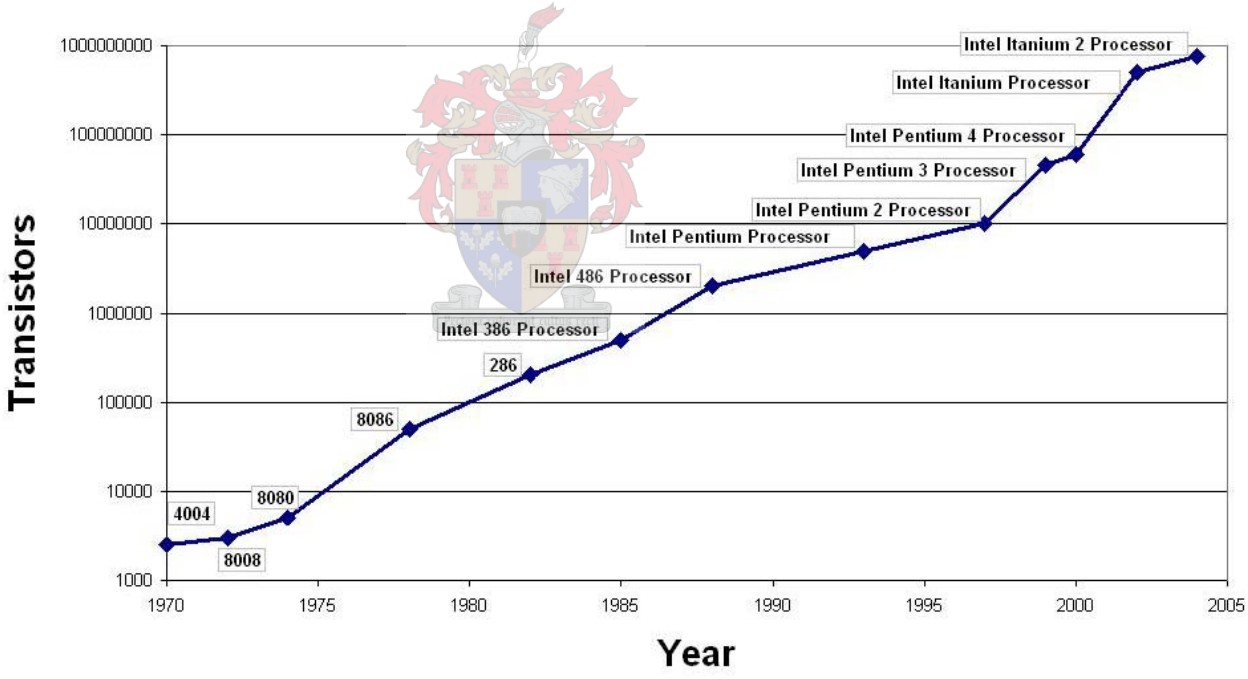


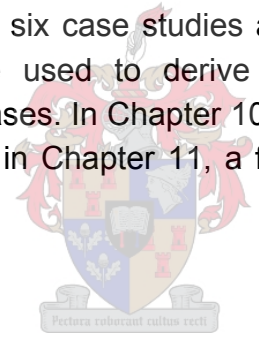
Figure 1.1: The dramatic increase in the numbers of transistor on electronic circuits in accordance with Moore's law (Source: Intel).

Historically GP was not widely applicable due to computing power limitations. However, these limitations are constantly receding, as computers become ever more powerful. In 1965 Gordon Moore, co-founder of Intel, stated that the number of transistors per square inch on integrated circuits had doubled every year since the integrated circuit was invented. This statement became known as Moore's law. Currently the prediction is that transistor density would

double every 18 months. The increase in transistors and transistor density leads to a direct increase in computing power. Figure 1.1 illustrates the increase in transistors of Intel central processing units. Owing to this increase in computing power GP is emerging as a comparable input-output modeling strategy. The best prediction currently is that Moore's law will hold for at least a further 20 years (Intel). With a further increase in computing power the feasibility and application of GP can only increase.

Despite the exponential growth in computer power since the general introduction of GP in the early 1990s (Koza, 1992), the methodology does not appear to have gained commensurately in popularity, if the few available commercial software packages or few papers published in chemical engineering journals are anything to go by.

In this thesis, the use of GP as a feasible approach to the development of process engineering models will be considered. The layout of the thesis is as follows. In the next chapter, GP methodology is explained, followed in the same chapter by a literature review of GP applications in chemical engineering. In chapters 4-9, six case studies are considered in which three different GP algorithms are used to derive models. These models are compared with CART in all cases. In Chapter 10, the conclusions of the thesis are summarized, and finally, in Chapter 11, a few suggestions are made on future work in the area.



2. Basic Concepts and Applications of Genetic Programming in Process Engineering

When you steal from one author, it's
plagiarism; if you steal from many, it's research.
- Wilson Mizner

2.1 The Genetic Programming algorithm

Genetic Programming (GP) is the application of the Genetic Algorithm (GA) on non-fixed string individuals. GP in essence contain simple rules that imitate biological evolution (Grosman and Lewin, 2004).

The basis of the Genetic Programming algorithm is that entities that perform tasks crucial to their survival better than other entities will survive and reproduce at a higher rate than the poor performing entities. This is of course the Darwinian "Survival of the Fittest" concept. The GP algorithm breeds individual computer programs and not fixed string individuals.

Koza (1992) encapsulates the GP algorithm as the following iterative procedure that should be conducted on the population of programs until the termination criteria have been satisfied:

- Generate an initial population of random computer programs consisting of the declared functions and terminals of the problem.
- Execute each program in the population and evaluate the fitness of each individual program.
- Generate a new population of programs by applying the primary operations of reproduction and crossover. Reproduction occurs by copying programs with high fitness measures to the new population. Crossover occurs when at least two new computer programs are generated by genetically recombining two current generation programs. The parts of the program used for genetic recombination should be chosen randomly.

The operators are applied with a probability based on fitness. The operators are applied to each individual program chosen. Secondary operations exist that could be applied in series or parallel. These secondary operations consist mainly of the mutation function. The result (exact or approximate solution) for

the GP paradigm is the best individual program(s) that appeared in any generation for the specific run. Figure 2.1 is a flowchart of the GP algorithm: ("i" refers to an individual in a population of size "M")

The algorithm will be illustrated by evaluating a simple example. For a more in depth discussion of the algorithm see Koza (1992, 2004). Assume that the following simple quadratic equation is under investigation:

$$y = m^2 + 2m + 5 \qquad (2.1)$$

Input data will be generated for integer values of 'm' from 1 to 10. The corresponding y values can thus be determined. This will form the input-output data necessary to derive a model.

The terminal and function set should be defined. For a simple symbolic regression problem the function set would typically be F = {+, -, ×, /, sin, cos}. It is important to note that some mathematical operators need multiple input arguments and some only need one. (The +, -, × and / operators need two input arguments and the trigonometric sin and cos operators need only a single input argument.) The division operator will normally be a restricted operator as not to divide by zero. Any mathematical operator or expression could be included in the function set as long as it provides a closed expression. Since only a single input exists only 'm' needs to be defined as a terminal. In some GP application some constants are defined as auxiliary terminals so that the evolution of constants is helped. If only 'm' is defined as a terminal than the integer '1' is evolved by the restricted division of $\frac{m}{m}$. Clearly this will be more tedious than defining '1' simply as an auxiliary terminal. For this example the terminal set will consist of only 'm.'

Typically a regression program will begin with mathematical operator taking two arguments. Let us assume that the '+' operator was chosen. This produces a simple tree structure that looks like the structure shown in Fig. 2.1.

Another level could be added to this simple tree by adding more functions listed in the function set. For illustrative purposes let us assume the multiplication and 'sin' functions were added. It is important to note that the multiplication operator takes two inputs and the 'sin' function takes only one input. The adapted tree illustrated now looks like:

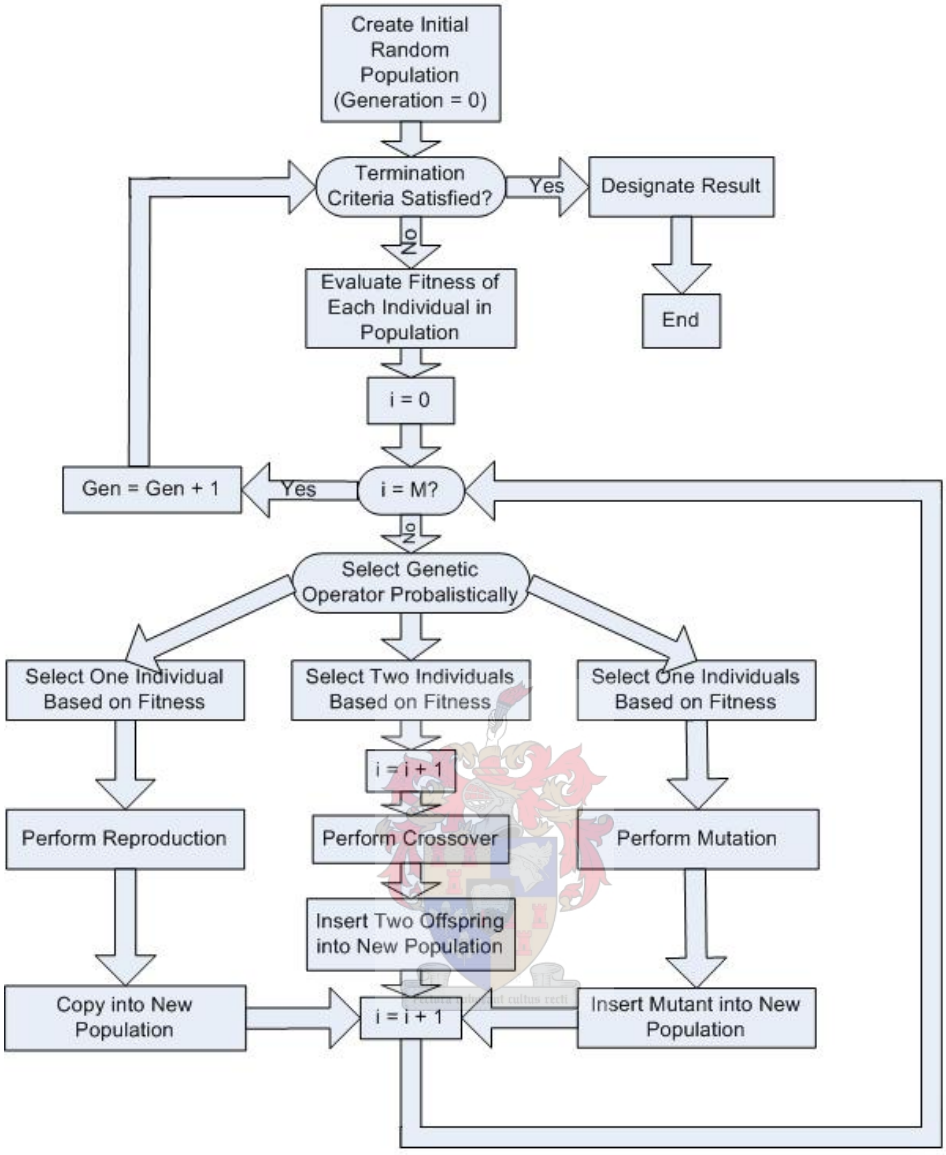


Figure 2.1: Adaptation of Koza's (1992) flowchart for Genetic Programming algorithm

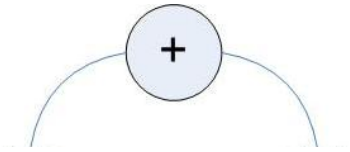


Figure 2.2: Origin of a simple tree structure

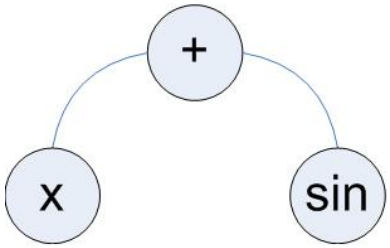


Figure 2.3: Adding a second layer to the tree structure program

The tree can be completed by applying the defined terminal set:

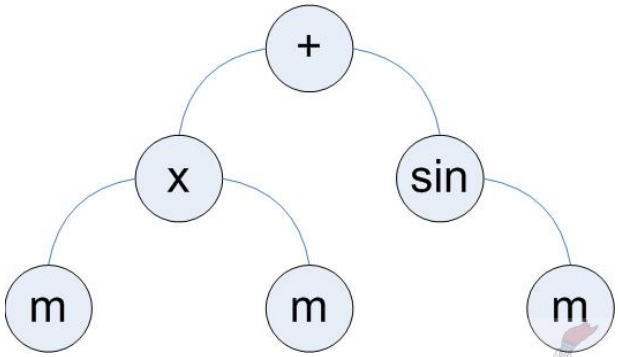


Figure 2.4: Individual 1 of random generation 0

This equation computes to:

$$y = m \times m + \sin(m) \tag{2.2}$$

Since it is a syntactically valid expression or computer program it can be evaluated for the defined values for 'm.' For this illustrative example we can choose the number of individuals in the random generation as four. The three other random generated computer programs could for example be:

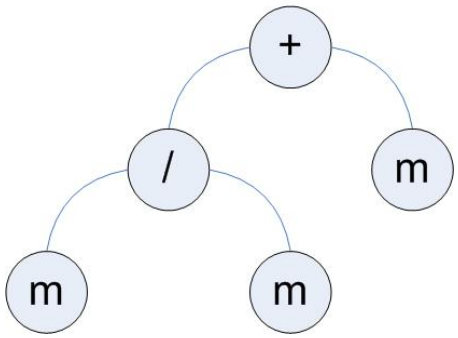


Figure 2.5: Individual 2 of random generation 0

This equation computes to:

$$y = m \div m + m \tag{2.3}$$

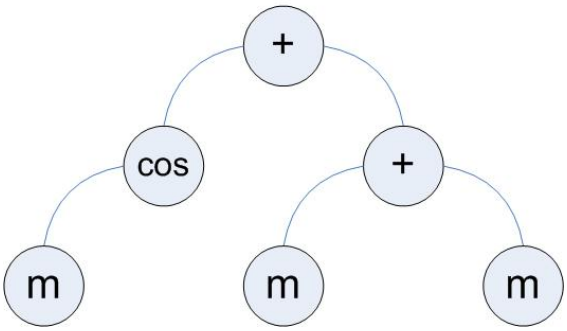


Figure 2.6: Individual 3 of random generation 0

This equation computes to: $y = \cos(m) + m + m$ (2.4)

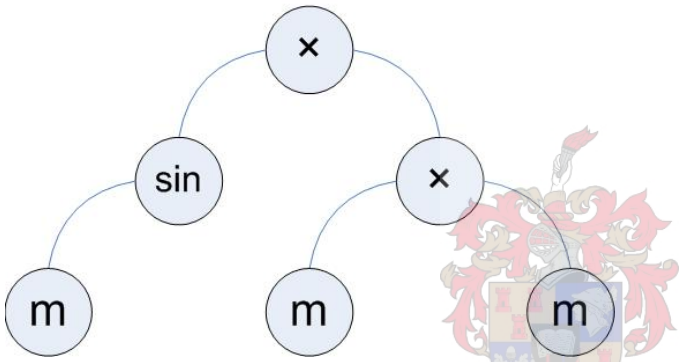


Figure 2.7: Individual 4 of random generation 0

This equation computes to:

$$y = \sin(m) \times (m \times m) \tag{2.5}$$

The fitness of the individual computer programs must be determined to assign a fitness value to each of them. Many different ways exist to do this. For this example the fitness will be determined by calculating the average absolute error for ‘y’ if integer values of 1 to 10 are assigned to ‘m.’ In this fitness calculation the “lower is better” principle applies where a smaller fitness value indicates a more accurate model. Table 2.1 illustrates the results:

Table 2.1: Average absolute error results for generation 0

Random generation 0	
Individual	**Average absolute error**
1	15.9
2	48.0
3	43.6
4	51.1

2.2 Creation of generation 1

The main genetic operations applied in GP will be illustrated by the formation of generation 1. For this example we will assume that the number of individuals per generation stays the same.

2.2.1 Cloning or Copying

This is the simplest of all genetic operations. The best individual program(s) are simply copied to the next generation. Since this example has only four individuals we will only copy the best individual from generation 0 to generation 1. Individual 1 of generation 0 now becomes individual 1 of generation 1. Cloning or copying ensures that the worst possible solution of generation (n+1) is the best solution produced by generation (n). In vast populations cloning is done probabilistically based on fitness.

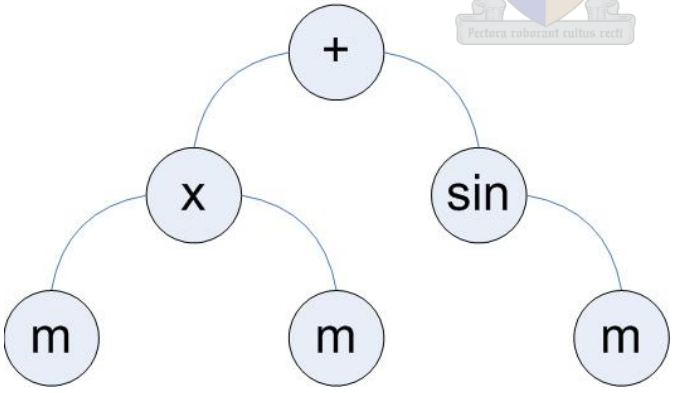


Figure 2.8: Individual 1 of generation 1

2.2.2 Crossover or sexual recombination

The crossover operation needs two parent programs and will produce two children programs. The two parent programs are picked probabilistically based on fitness. For this example we will chose the two programs with the 2nd and 3rd best fitness measure, i.e. the 2nd and 3rd individuals of generation

0. The crossover operation is executed at random nodes of the tree structure of the parent programs. Figure 2.8 illustrates the crossover operation:

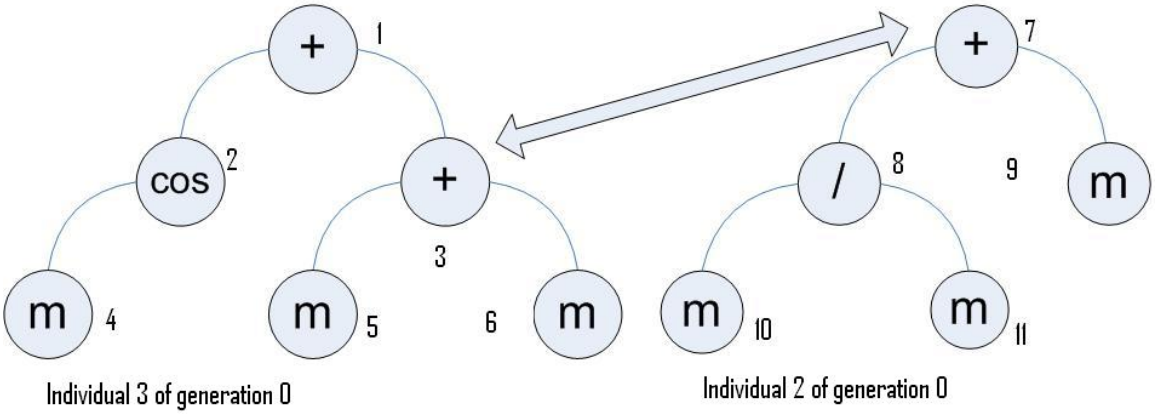


Figure 2.9: Illustration of crossover operation

In this example the random nodes chosen for crossover were nodes 3 and 7 as indicated by the arrow. After crossover the two new children programs are formed. If crossover is performed correctly both children programs will be syntactically executable computer programs since both parent programs were syntactical correct computer programs. The children produced look as follows:

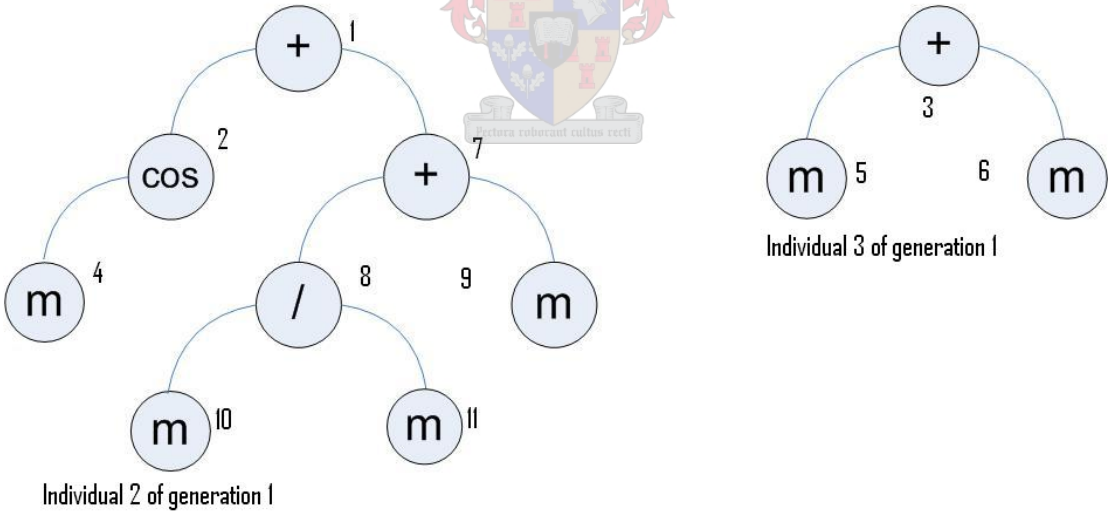


Figure 2.10: Result of crossover operation

These tree structures represent computer programs simplified as:

Individual 2 of generation 1: $$y = \cos(m) + \frac{m}{m} + m \qquad (2.6)$$

Individual 3 of generation 1: $$y = m + m \qquad (2.7)$$

2.2.3 Mutation

The final genetic operation discussed is that of mutation. During mutation only one parent is required to produce one child. Normally mutation will be applied probabilistically to the parent generation. In this example mutation will be performed on the remaining unchanged individual 4 of generation 0. A node is simply chosen randomly and the operator changed with a random operator from the function set.

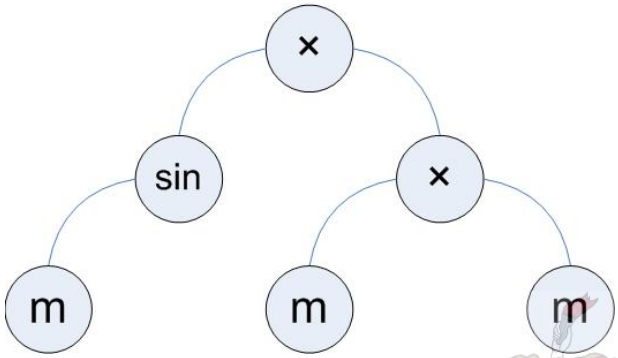


Figure 2.11: Individual 4 of random generation 0

This equation computes to $y = \sin(m) \times (m \times m)$. (2.8)

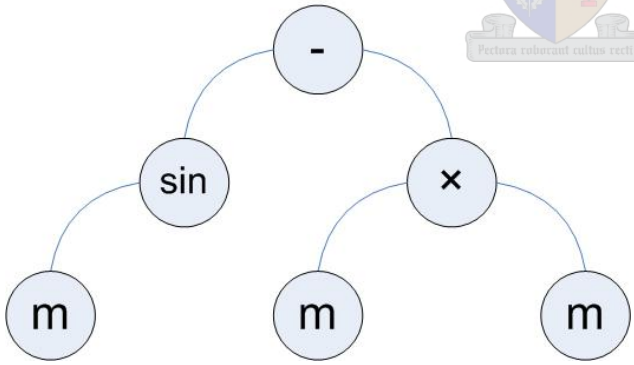


Figure 2.12: Mutant of individual 4 of random generation 0

This is now individual 4 of generation 1. This equation computes to:

$$y = \sin(m) - (m \times m) . \quad (2.9)$$

The purpose of mutation is to ensure diversity remains in a population. This diversity aims to prevent premature model convergence. Premature model convergence often results in models trapped in local minima or maxima and global mina or maxima results are never produced. Mutation rates are always

much smaller than the rates at which crossover is performed. Koza (1992) argues that diversity in a population will remain even with limited or no mutation since the chance that crossover will produce a generation similar to the parent generation is very small. The mutation operations is of much greater importance in classic genetic algorithm (GA) applications where the crossover of binary strings could produce children very similar to their parents.

The fitness values will now be determined for generation 1. The process then repeats itself until a set termination criteria is met. The termination criteria could be a value for which the absolute error value of the model would be tolerable or the termination criteria could also be that after a certain number of generations the algorithm should stop.

Boolean operators like ‘and’, ‘or’ and ‘not’ and Boolean functions like ‘if then else’ could also be included in the function set. The GP algorithm should then be capable of deciding when to use the Boolean operators and functions to ensure that meaningful or executable rules are still produced. An example of an un-executable program will be:

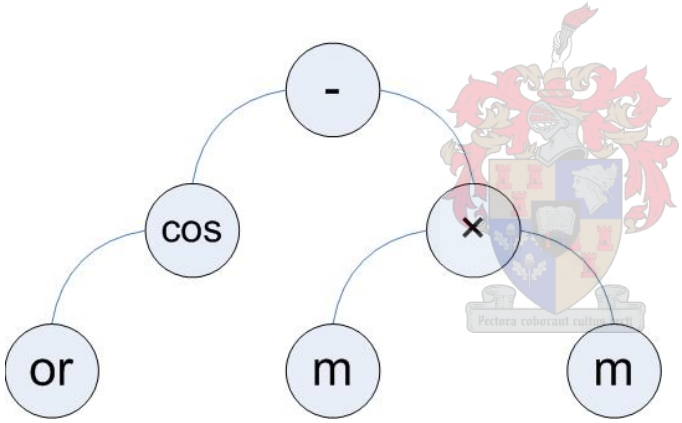


Figure 2.13: Meaningless or impossible rule produced by GP.

In the above program the ‘cos(or)’ operation has no meaning. The type of GP implementation capable of simultaneously using mathematical operators and functions and Boolean operators and functions is known as a strongly typed GP application.

2.3 Feasibility and Applicability of GP

Grosman and Lewin (2004) describe the ability of GP to vary the structural complexity of the tree building blocks as a great advantage. In classical symbolic regression examples this will mean that the order of the function to be fitted to the data does not have to be known before hand. In more complex and diverse examples the depth and relative complexity of empirical tree like models need not be fixed at the start of an investigation.

Grosman and Lewin state that empirical models in process systems engineering either have to start with a predefined structure or undetermined black box type structures. (Neural Networks would be an example of an undetermined black box type structure.) In principle, GP overcomes both above mentioned problems, since it varies its complexity to find the best solution and ends with an explicit model.

There are two schools of thinking when it comes to the applicability of GP in chemical engineering. Some researchers use the standard GP algorithm as is, while other researchers spent much time and effort to overcome what they view as pitfalls of the GP algorithm. Some of these adaptations of the GP algorithm include:

2.3.1 Improvements by Grosman and Lewin (2004) of the GP algorithm:

- In what is referred to as elitism, the fittest individual(s) are allowed to survive in their current form until a better solution is found. This is in contrast to what Grosman and Lewin viewed as the common practice where only a percentage of the previous generation survives with the remainder made up of new genetically-generated solutions.

- Grosman and Lewin furthermore explicitly ensured that the number of variables in models matched the degrees of freedom implied by the model structure.

2.3.2 According to Yean et al. (1999) standard GP techniques have inadequate estimation techniques for numerical parameters of the produced functional tree. Similar problems were encountered by Greeff and Aldrich (1997). This limits the application of GP to solve engineering type problems. According to Yean et al. (1999) GP also has the tendency to get stuck on local minima or maxima resulting in an inadequate final tree. Improvements by Yean et al. (1999) include:

- Devising a group of additive genetic programming trees. (GAGPT) These trees consisted of a primary tree and a set of auxiliary trees. To generate the combined output of these trees the GAGPT output was summed.

- Introduction of weights to the nodes of the GP tree. They were not the first to do this, but instead of dealing with whole weights of the functional tree the Yean et al. (1999) proposed that only a small portion of the weights must be chosen and estimated. This process is then repeated on another small selection of weights until the value of all weights are found. By doing this, the authors constructed a linear association matrix or small linear associative memory called LAM.

2.3.3 Csukás and Balogh (1998) proposed a methodology that combines structural modelling with genetic programming. This example combines known conservational law knowledge with the GP algorithm. The result is a system that takes advantage of domain specific information and the optimizing capabilities of GP. The evolutionary model developed by Csukás and Balogh (1998) uses three kinds of knowledge:

- Exact knowledge obtained from the detailed dynamic simulation. This domain specific input is provided by the expert human user.

- Heuristic knowledge provided to the program. These heuristics are also domain specific.

- And the uncertain genetic knowledge originating from evolution.

2.3.4 Cao et al. (1999) focused on producing a hybrid evolutionary modelling algorithm (HEMA) for kinetic models for chemical reactions. These kinetic models consisted of ordinary differential equations. Cao et al. (1999) embedded a genetic algorithm (GA) into genetic programming (GP). The GP focused on optimizing the structure of the model and the GA was used to optimize its parameters. The researchers tested the HEMA on two chemical reaction systems. The HEMA generated a model that had satisfactory predictive power.

2.4 Current Applications of GP in Process Engineering

To date, few industrial applications of the GP algorithm have been reported in the chemical engineering literature. Applications that have been found are discussed below

2.4.1 Modelling two continuous stirred tank reactors in series (McKay et al., 1996)

McKay et al. (1996) modelled a chemical reaction in a series two continuous stirred tank reactors (CSTR). In this reaction, reagent A is converted to a desired product B, which is in turn converted to an undesired product C, according to eq. 2.10.

$$A \rightarrow B \rightarrow C \qquad (2.10)$$

The reaction is exothermic and heat is removed from both reactors. By controlling the temperature of the reactors the formation of undesired species C could be averted.

The optimal temperature could vary owing to stock batch variations. The goal of this study was to investigate whether GP could be used to develop a non-linear, steady-state, process model. This model should be able to determine the best temperature set points to maximize the production of B. Firstly input-output data had to be generated. A mechanistic model could be derived to provide the required data. The training set consisted of 300 data entries and the validation set also consisted of 300 data entries. The inputs measured were the temperatures in both reactors, the inlet concentration of species A and the amount of cooling provided to each reactor. The output was the single measurement / calculation of the outlet concentration of species B exiting the second reactor. The inlet concentration of A was an unnecessary input according to the researchers who investigated whether GP could separate useful inputs from irrelevant inputs. The researchers decided to scale the input-output data because the magnitudes of the inputs and outputs were different. This scaling was apparently not deemed necessary by other authors, since it is not frequently performed.

The GP function set consisted of F = {+, - , /, ×, ^, sqrt, square, log, exp}. The following GP settings were used: Mutation probability 20%, crossover probability 80%. Cloning was also implemented at a probability of 10%, but these individuals were then subjected to mutation and crossover. A tree size penalty was incorporated to ensure that the tree models did not overfit the data. \50 individual runs were performed to assess the robustness of this application of GP. The researchers achieved a model accuracy ranging from

66 – 90%. A successful model was defined as a model that had an RMS error below a tolerable value set at 0.06. The best model had an RMS error of 0.028 on the validation set.

McKay et al. (1996) investigated all the successful models to establish what input variables were used most frequently. The relative importance of each input could be established in this way. By doing this it was illustrated that the reactor temperatures and molar concentration of A in the feed were the most important input variables to control the molar concentration of B. Not only was model parsimonious, but some of the phenomenological information of the system could be deduced from investigating the frequency of inputs used. The authors concluded by arguing that for GP to become a modelling tool of greater engineering use it must be able to develop dynamic input-output models.

2.4.2 Binary vacuum distillation (Willis et al., 1997)

Willis et al. (1997) were some of the first researchers to show that GP was able to evolve empirical models for chemical process engineering. They have used GP to derive symbolic expressions, but regression constants were determined with standard methods. The researchers felt that the standard GP could not adequately derive constant parameters.

The first application was that of steady state binary distillation. A vacuum column was equipped with 48 trays, reboiler (steam heated) and a total condenser. The GP model was constructed so that the bottom product composition could be inferred from temperature and pressure measurements. Measurements were only available from trays 12, 27 and 42. They could also derive a model from first principles that consisted of several hundred differential and algebraic equations and used this model to generate data for the GP model.

The GP algorithm used 25 individuals per generation and 50 generations per run. The objective was to minimize the RMS error of the training data. The training data consisted of 150 simulated measurements of steady state compositions at the mentioned trays, together with the corresponding bottom product composition. A set of 50 measurements were used as validation set to prevent overfitting. The best model achieved by GP had an RMS error on the validation data set of 0.0189. A linear model achieved by using a batch least squares approach had an RMS error on the same validation set of 0.33.

2.4.3 Screw cooking extruder (Willis et al., 1997)

The second piece of equipment investigated by Willis et al. (1997) was a twin screw cooking extruder. Although such an extruder is not typically considered to fall in the realm of process engineering, it performs in the same manner as more conventional process equipment. Extruders are used in the food industry, because they allow for a short residence time and high temperature cooking. This has nutritional and cost implications. A typical extruder consists of a shell housing one or more helical screws. The screws rotate to convey the feed material. As the food progresses down the shell an increase in pressure and temperature occurs.

Plant data were obtained from a pilot scale APV Baker MPF 40 cooking extruder operated by the CSIRO Division of Food Science, Australia. The extruder processed corn flour. The modelling inputs were the screw speed, feed rate, feed moisture content and the feed temperature. The degree of gelatinisation of the starch product was chosen as a single measure of product quality.

The data set consisted of 450 training set records and 150 validation set records. As a benchmark, an ANN with a single hidden layer containing four neurons was trained using a Levenberg-Marquardt optimisation algorithm. The best ANN model achieved an RMS error of 0.095 on the validation data set. A GP run consisted of 30 individuals per generation executed for 100 generations. The best GP model achieved an RMS error of 0.045 on the validation data set. Willis et al. (1997) argued that a greater advantage than the RMS error decreasing is the ability of GP to automatically eliminated irrelevant model inputs. This offers a parsimony that the ANN could not match.

2.4.4 Acid pressure leaching of nickeliferous chromites (Greeff and Aldrich, 1997)

Greeff and Aldrich (1997) investigated the evolution of empirical regression models using GP for the acid pressure leaching of nickeliferous chromites. The samples investigated were beneficiated lateritic chromite overburden samples at 250 - 260°C with 0.3 - 0.4 g H_2SO_4 in the presence of additives. The measured inputs were temperature (°C) as X1, ammonium sulphate concentration (mol/l) as X2 and acid concentration (mol/l) as X3. The output was the single dissolution prediction of the relevant metal element. The benchmark strategy against which the GP models were to be compared was a quadratic regression model derived by Das et al. (1997), as discussed by Greeff and Aldrich (1997). A model was generated for the dissolution of

cobalt, nickel and iron. These three models predicted the amount of metal dissolved after 1h, 2h and 3h.

Unlike the smaller populations used by Willis et al. (1997), a GP run consisted of 2000 individuals per generation and 100 generations. The function set was: F = {+, -, ×, /} and the terminal set T = {X1, X2, X3, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9}. A second investigation included the time (t) in the terminal set: T = {t, X1, X2, X3, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9}. The integers in the terminal set were auxiliary terminals as discussed earlier. The cloning probability was set to 10% and the crossover probability to 90%. The researchers did not deem it necessary to apply the mutation genetic operator. A successful hit was defined for the nickel and cobalt leaching prediction as an error of less than 1%. A successful hit defined for the iron leaching was even stricter with a tolerable error of 0.5%.

The GP produced model for the extraction of nickel had an RMS value of 3.14 compared to the benchmark's RMS value of 2.52. For the extraction of iron the GP produced model had an RMS value of 0.264 compared to the benchmark's RMS value of 0.893. In the case of cobalt the GP produced model had an RMS value of 2.15 compared to the benchmark's RMS value of 2.58. The cobalt and nickel models produce comparable results, but GP outperformed quadratic regression with regards to the iron model.

The researchers concluded that the GP models were too complex to interpret. Furthermore, evolving the model structure was much easier than estimating the associated parameters afterwards.

2.4.5 Uranium and radium liberation models (Greeff and Aldrich, 1997)

A second case study by Greeff and Aldrich (1997) investigated the evolution of empirical regression models using GP for the leaching of uranium and radium. Quadratic models were derived with statistical methods by previous researchers (Kondos and Demopoulos, 1993). These models were derived for the leaching of high grade arseniferous uranium ore to optimize the co-extraction of uranium and radium. The inputs to the model were the pulp density, concentration of the hydrochloric leaching acid, the concentration of the calcium chloride additive and time. The output was the single predicted percentage extraction of the metal.

The GP runs consisted of 2000, 4000 and 6000 individuals per generation and 100 generations. The function set was: F = {+, -, ×, /, log, exp} and the terminal set T = {X1, X2, X3, X4, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9}. The cloning probability was set to 10% and the crossover probability to 90%. Aldrich and

Greeff (1997) did not deem it necessary to apply the mutation genetic operator. A successful hit was defined as having a predicted percentage extraction error of 2% or less.

In this case study the GP model for the extraction of radium had an RMS value of 14.14 compared to the benchmark's RMS value of 10.41. For the extraction of uranium the GP produced model had an RMS value of 1.94 compared to the benchmark's RMS value of 6.07.

As before, Aldrich and Greeff (1997) concluded that the GP models were too complex to interpret. Furthermore, evolving the model structure was much easier than estimating the associated parameters. The researchers argued that the model structure evolution must be carried out by GP and the parameter identification by more standard techniques like gradient descent, etc., as was found by Willis et al. (1997).

2.4.6 Modelling of a mixing tank (Grosman et al., 2002)

More recently, Grosman et al., 2002) used GP to produce non-linear dynamic models from data. These authors used GP to estimate the structure of a model and did parameter estimation with a non-linear least squares algorithm. The model was tuned by minimizing the quadratic error.

The first case study was a mixing tank consisting of a fresh water feed and saturated salt water feed. Both the fluid level in the tank and the mixed effluent should be maintained at specific set points. Mathematical modelling proceeded with the assumptions that the exiting effluent could be modelled as a sudden expansion and that perfect mixing occurred in the tank. Two differential equations were derived to model the system. Decentralized proportional and integral (PI) control was used as the benchmark control strategy. PI control is often applied as control strategy on plants for its ease of implementation.

To evolve the GP models delayed input-output data were provided. Two sets of data were derived so that one set acted as training data and the other as validation data. According to Grosman et al. (2002), since trajectory matching was used, sampling time was not a critical factor. A sample rate of 0.15 time units was used. Two linear models were derived by GP to control the tank level and effluent concentration.

The performance of the PI control versus the GP evolved model was tested for two situations. The first was a pulse of 40% in the influent salt water flow when the mixing process was at steady state. The GP model control exhibited exceptional disturbance control resulting in very little error. The PI controller

exhibited oscillatory control with measurements fluctuating greatly. Detuning the P&I controller decreased the oscillations, but increased the settling time.

The second test situation induced was a set point change of both the fluid level in the tank and the effluent salt concentration. The GP model achieved rapid acquisition of the set points without violating the hard constraints of the process. (A typical hard constraint is that the tank may not overflow during a set point change.) The P&I controller again had non-ideal oscillatory control, but much worse it violated the hard constraints placed on the system. In this case the P&I controller would have caused the tank to overflow.

2.4.7 Modelling of a Karr liquid-liquid extraction column (Grosman and Lewin, 2004)

A second investigation of Grosman and Lewin (2004) focused on the Karr liquid-liquid extraction column. A Karr column consists of a bank of perforated plates in a column. The column is attached to a mechanism that lets it oscillate at a set frequency. The conservation equations and discretization of the partial differential equations of the system was used to generate the necessary input-output data for GP models. Three manipulated variables were identified, namely the continuous phase flow rate, the dispersed phase flow rate, and the oscillating frequency of the column plates. Limits were placed on these variables. The output variables were the dispersed phase effluent concentration and the column hold-up. Random magnitude step changes were initiated to capture the dynamics of the system. A total of 160 samples were taken with 10s intervals. As before, crossvalidation via a training set and validation set was used to combat model overfitting. The benchmark control strategy was a linear multivariable controller that involved two single input single output loops and a dynamic decoupler compensating for strong input parameter interaction. In contrast the GP derived model was non-linear.

A set point change of ±30% in the desired disperse phase concentration was initiated. This offered insight into the control achieved by the two model strategies. During the +30% step, Grosman and Lewin (2004) commented that near perfect decoupling was achieved by a benchmark model in approximately 600 s. The GP model performed slightly worse, but settled much faster in about 200 s. During the -30% step change, excessive oscillatory behaviour was exhibited by the benchmark model compared to the GP produced model.

2.4.8 Simultaneous zinc and lead Imperial smelting process (Chen et al., 2003)

Chen et al. (2003) applied GP as an optimizing technique in the Imperial smelting process (ISP). An ISP is a typical complex metallurgical process in which lead and zinc are simultaneously and continuously smelted in a closed furnace. Multiple complex chemical reactions are involved for which little process details are known. The ISPs in question were at the Shaoguan Smelting Plant in Guangdong Province, China.

The goal was to identify the optimal control strategy. As in most chemical process engineering applications multiple objectives had to be satisfied. These objectives included the maximal production rate, process stability, quality and minimum cost. The control strategy of the ISP was evaluated as a multi-step transition control program with each step corresponding to a configuration of the vector of control parameters. The optimized solution could be indicated as by the precedence of operations. Moreover, the ISP is slow enough to allow human intervention. For this reason all models produced for optimizing could be accepted or declined by the operator of the ISP.

In this research the nodes of the GP model were control vectors and the link element indicated the sequence of operations. Limitations were placed on the length of the evolved individuals because of the lag of the process. A further limitation placed on the GP application was that chains could be developed only. This implies that a single control vector had one control vector input and one output. The branch structure as illustrated previously was not allowed. In doing this the researchers actually applied the genetic algorithm on non-fixed length individuals.

A typical run consisted of 200 individuals and 500 generations. The genetic operator probabilities were fixed at a probability for cloning of 10%, mutation probability of 15%, crossover probability of 60% and inversion probability of 15%. Inversion is a special case of mutation in which a single execution is inverted. (In the illustrative example this would imply that a ‘+’ would change to an ‘-‘, for instance.)

2.4.9 Identification of nonlinear systems (Madár et al., 2005)

Madár et al. (2005) have considered the use of genetic programming to identify nonlinear input-output models from data based on three case studies involving simulated systems. The main idea of their paper was to use of orthogonal least squares methods to estimate the contribution of the branches of the tree-structures generated by genetic programming. In this way the

structure and order of the models could be estimated with genetic programming, while the model parameters were estimated in a more efficient manner.

2.4.10 Identification of response surface models (Lew et al., 2006)

Lew et al. (2006) made use of genetic programming or symbolic regression as they have also referred to it, to extend the class of possible models that could be fitted to response surfaces. Although not strictly speaking an application in chemical engineering, the example is sufficiently relevant to be considered as such. They have considered both univariate and multivariate data, as well as simulated (structural modulus and Poisson's ratio for honeycomb structures) and real data (9 degrees-of-freedom mass spring system). These authors have found that genetic programming models could outperform artificial neural networks in fitting nonlinear response surfaces to their data.

Interestingly, they considered one of the limitations of genetic programming to be the fact that model parameters or constants had to be evolved randomly and suggested that in future work some optimization routine, such as simulated annealing could be combined with genetic programming for more efficient fitting of models.

2.4.11 Heat transfer correlations with symbolic regression (Cai et al., 2006)

Cai et al. (2006) used genetic programming to generate heat transfer correlations to data and found that the new correlations fitted the data better than their published counterparts. They made use of penalty parameters that ensured that the correlations would not be too complicated. The parameters of the correlations were determined with a complementary Nelder-Mead algorithm.

2.5 Conclusions from the literature review

From the above review of the available literature, the following conclusions can be made:

- First of all, genetic programming is not as yet a well-established self-contained methodology for the development of models in process engineering, as is evident from the few papers that could be found in the open literature.

- The tedious evolution of constant parameters, as discussed in section 2.1, is considered as a limitation according to some authors, notably Greeff and Aldrich (1997), Willis et al. (1997), Madár et al.

(2005), Lew et al. (2006) and Cai et al. (2006). Other authors, such as McKay et al. (1996), Grossman et al., (2002), Lewin and Grossman (2004) and Chen et al. (2003) did not report any problems as far as the simultaneous generation of models and associated parameters were concerned. With the exception of Lew et al. (2006) none of the authors commented on the computational expense incurred by using genetic programming. None of the authors (Lew et al., 2006, included) considered computation as a serious problem, possibly because none of them considered realistically large systems (many tens of variables and many thousands of records).

- In principle, one of the major advantages of using genetic programming is that interpretable models can be developed. Cai et al. (2006) and Madár et al. (2005) reported the successful development of parsimonious, interpretable models. In contrast, Greeff and Aldrich (1997) mentioned this as an objective that they failed to achieve.

- As a final remark, no models designed to classify data could be found, despite the importance of classification problems in process engineering and the obvious capability of genetic programming to be able to do so. This is particularly relevant in the design of knowledge based systems.

In the light of these conclusions from the literature review, the general hypothesis of this thesis is stated as follows:

> *Genetic programming offers a competitive approach towards the automatic generation of process models from data.*

The results reported in the literature presented above suggest that this hypothesis is valid, but the small number of case studies themselves is less convincing, given the bias towards publication of successful results and non-publication of unsuccessful results.

Apart from the literature review, the objective of this thesis will be to undertake various simulation experiments to compare the modelling capability of GP with classification and regression trees. The reason why classification and regression trees are selected over many other candidate families of models, such as neural networks, kernel-based modes, splines, etc. is that they are particularly relevant in the generation of *interpretable* models. In the process industries in particular, where the use of expert systems are well-established,

these classification and regression trees are sometimes used to generate IF-THEN rules that can be incorporated into the knowledge-bases of these expert systems. Therefore,

i. Although models can be assessed on the basis of several different criteria, the assessment will be limited to the *predictive power* and *interpretability* of the models.

ii. In the experiments on simulated and real-world data, both classification and regression problems will be considered

3. Genetic programming software

Everything should be made as simple
as possible, but not simpler.

- Albert Einstein

3.1 The use of GP in the current research

Firstly a benchmark approach was established to have some measure of the results obtained by various GP applications. The measure of success had to measure the accuracy and complexity of the derived model. The benchmark strategy was that of non linear tree regression as employed in the 'treefit'-function in MATLAB's StatsToolbox.

Various software packages employing GP was used in this study. All three GP packages had specific advantages, as will be discussed later on. A general approach was followed for all six case studies evaluated. This general approach consisted of the following procedures:

- Generate a training set of data. Usually this consisted of 80% of the total amount of data available. The data was initially randomized.
- Generate a test data set. The remaining 20% usually produced the test data set.
- Evolve a GP strategy on the training data set for a specified number of individuals and generations.
- Evaluate the accuracy of the evolved rules on the unseen test data set.
- Simplify the evolved rules for human interpretability. Evaluate the complexity of the derived model.

Strategies with a result differing less than 5% in accuracy were deemed to be a non-significant difference. The measure of complexity will be discussed in each specific case study.

3.2 GP algorithms used in this study

3.2.1 GPLAB Toolbox (Silva, 2004)

Firstly the GP algorithm was employed by means of Silva's (Silva, 2004) GPLAB toolbox. It consisted of 123 MATLAB files and was run as a MATLAB application. This toolbox was designed with a highly modular approach. This implies that the specific functions and properties needed could be customized for each case study individually.

The GP algorithm was executed on high level data structures. Examples of these high level data structures could be arrays or matrices like in MATLAB. Applying GP on high level data structures were extremely resource intensive. Typical runs employed 200 – 500 individuals and lasted for typically 200 – 500 generations. (The total amount of individuals produced was roughly 40 000 – 250 000.) Of course the number of individuals or generations used does not act as an indication of success. The duration of these runs varied greatly in accordance to the complexity of the problem and the allocated generations and individuals.

GPLAB can generate models by using mathematical operators or Boolean functions and operators. Since it is not a strongly typed GP application, GPLAB does not exert any control on the placement of possible operators. Non-executable combinations of mathematical and Boolean operators will lead to non-executable or meaningless models. GPLAB could use any custom MATLAB function that verifies closure to produce its trees. Additionally GPLAB can also incorporate the additional functions listed in table 3.1:

Table 3.1: Additional functions available in GPLAB

Additional Functions	
and	equal
or	greater than
not	Less than or equal
not and	If-then-else
not or	

The produced models were in the form a tree structure. This structure still needed to be simplified to be human interpretable. These tree structures and the simplification there of will be discussed in detail during the case studies.

Regression models were mainly produced by using mathematical operators and functions. The operators and functions used could be altered between runs or between different case studies. Regression models produced numeric

results of many significant numbers. Post model output evaluation rounded these outputs or results to the same number of significant numbers as the input values.

Classification models were produced by the additional functions as stated in table 3.1. The result was a model that looked similar to the result produced by an expert system. The classification models also resembled the benchmark statistical tree regression model in structure. A fundamental difference between the GPLAB classification model and the benchmark statistical regression model was that the benchmark method always compared a parameter to a numeric value whereas GPLAB compared a parameter to another parameter, function of parameter or a numeric value. Another fundamental difference of GPLAB derived classification models was that the output produced was an integer value denoting a class or induced class. By induced class it is meant that the integer predicted does not correspond to any integer associated with a class. This will be discussed during the case studies applicable. These types of classification models will be called pure classification models since no rounding was performed to predict the class associated integer.

As stated in table 3.1, GPLAB could use the 'or'-operator in producing models. The 'or'-operator produced disjunctive rules if used. The implication of this will be discussed in the later sections. Only GPLAB could produce disjunctive (IF-THEN, OR) rules.

3.2.2 GP250 (Swart and Aldrich, 2001)

The second application of the GP algorithm was GP250 developed by Swart and Aldrich (2001). It consisted of four MATLAB files and was run as a MATLAB application. This toolbox was designed with a highly modular approach. GP250 was still a work in progress.

As with GPLAB the GP algorithm was executed on high level data structures. Examples of these high level data structures could be arrays or matrices like in MATLAB. Applying GP on high level data structures were extremely resource intensive. GP250 needed less computational time compared to GPLAB because GP250 did not offer any real time graphical outputs as GPLAB. Typical runs employed 100 – 400 individuals and lasted for typically 100 – 400 generations. (The total amount of individuals produced was roughly 10 000 – 160 000.) Again it should be stated that the number of individuals or generations used does not act as an indication of success.

GP250 could only generate models by using mathematical operators. The question of strongly typed GP applications was thus not an issue as with GPLAB. GP250 generated its models by using the mathematical operators of table 3.2:

Table 3.2: Available mathematical operators of GP250

Index:	Function:	Index:	Function:
1	+	9	cos
2	-	10	tan
3	×	11	sinh
4	÷	12	cosh
5	log	13	sig
6	exp	14	power
7	log10	15	asin
8	sin	16	asinh

As stated earlier GP250 was still a work in progress. The output of the produced model was in a crude format. The model output was given in reverse Polish notation with integers assigned to the level of the operator or parameter. This output still needed to be simplified to be humanly interpretable. Due to the time consuming simplification most runs were limited in the maximum allowed complexity. The level at which the complexity was limited to was deduced from the best achieved runs by GPLAB.

GP250 produced regression models by using mathematical operators and functions. The operators and functions used could be altered between runs or between different case studies. Regression models produced numeric results of many significant numbers. Post model output evaluation rounded these outputs or results to the same number of significant numbers as the input values precisely as was done with GPLAB.

GP250 produced classification models fundamentally different from GPLAB. GP250 produced classification models by rounding the result of a regression model to the nearest integer associated with a class. This had to be done manually since GP250 did not determine or apply the threshold value for rounding automatically. The threshold value was simply estimated as 0.5 as would be the case in normal mathematical rounding.

3.2.3 Discipulus™

The third and final implementation of GP algorithms was through Discipulus. Discipulus is a commercially available GP package produced by the AIM

Learning Company. Discipulus is a standalone Windows application. In this study the entry level Discipulus Lite edition was used.

Discipulus differs from GPLAB and GP250 since it executes the GP algorithm on low level binary code. Applying GP on low level data structures enabled Discipulus to evaluate vast numbers of models compared to the previous GP applications. Typical runs employed 500 individuals and lasted for typically 50 000 generations. (The total number of individuals produced was roughly 25 000 000 - 26 000 000.) These runs lasted for 1 - 4 hours. As earlier, it should be stated that the number of individuals or generations used does not act as an indication of success. Discipulus could only generate models by using mathematical operators and predefined functions. The issue of strongly typed GP applications was thus not an issue as with GPLAB. Discipulus used basically the same mathematical operators as stated in table 3.2.

The produced model was available for further investigation as C/C++ code, Assembler or a mathematical equation. The C/C++ code was imported into MATLAB and investigated further. Normally the C/C++ code was between 110 – 200 lines. It consisted of sequential steps that should be executed on an initial value or parameter. This code was difficult to interpret without importing it to MATLAB and investigating the result. The mathematical equation equivalent of the produced model consisted of a single formula normally of at least 4 pages. This was even less humanly interpretable than the corresponding C/C++ code. The Assembler code was not investigated further.

Discipulus produced regression models by using mathematical operators and functions. The operators and functions used could be altered between runs or between different case studies. Regression models produced numeric results of many significant numbers. Post model output evaluation rounded these outputs or results to the same number of significant numbers as the input values precisely as was done with GPLAB and GP250. A useful feature of Discipulus was that training could be stopped and resumed at anytime.

Discipulus produced classification models in the same way as GP250. It produced classification models by rounding the result of a regression model to the nearest integer associated with a class. Discipulus did this automatically for binary classification. The default threshold value was simply set as 0.5. This value could be changed, but the manual strongly advised against it. Discipulus was only capable of binary classification. Higher level classification was forced by handling the classification problem as a regression problem. The 0.5 threshold value was then manually forced.

Table 3.3 acts as a summary of the three GP applications used in this research. Various criteria discussed above are indicated in the table:

Table 3.3: Summary of the GP applications used in this research

		GPLAB	GP250	Discipulus
Type of application:		MATLAB	MATLAB	Standalone Windows application
GP applied to:		high level data structures	high level data structures	low level binary code
Typical values:	**Computational time per run**	4 - 28 hr	2 - 8 hr	1 - 4 hr
	Individuals	200 - 500	100 - 400	500
	Generations	200 - 500	100 - 400	50000
Function set:	**Mathematical operators**	Yes	Yes	Yes
	Mathematical functions	Yes	Yes	Yes
	Boolean operators	Yes	No	No
	Boolean functions	Yes	No	No
Capable of producing disjunctive rules:		Yes	No	No
	User defined operators	Yes	No	No
	User defined functions	Yes	No	No
Structure of classification model:		pure classification model	rounded regression model	rounded regression model
Structure of regression model:		pure regression model	pure regression model	pure regression model
Structure of produced model:		tree	brute model output	C/C++ or Assembler or closed expression
Available graphical output:		extensive	poor	extensive

4. Case Study 1: Ordinary Least Squares Regression

> Celestial navigation is based on the premise that the Earth is the centre of the universe. The premise is wrong, but the navigation works. An incorrect model can be a useful tool.
>
> - Kelvin Throop III

4.1 Regression by means of MATLAB's Statistics Toolbox

The first case study, an ordinary least squares regression problem was investigated. It consisted of two input variables and the resulting output. The two input variables were x_1, x_2 elements of [-4, 3]. The output was z so that

$$z = 1.21 \times (1 + 2x_1 - 2{x_1}^2) \times (1 + 2x_2 - 2{x_2}^2) \times \exp(-\frac{{x_1}^2}{2} - \frac{{x_2}^2}{2}) \qquad (4.1)$$

Figure 4.1 represents a surface plot of the data generated:

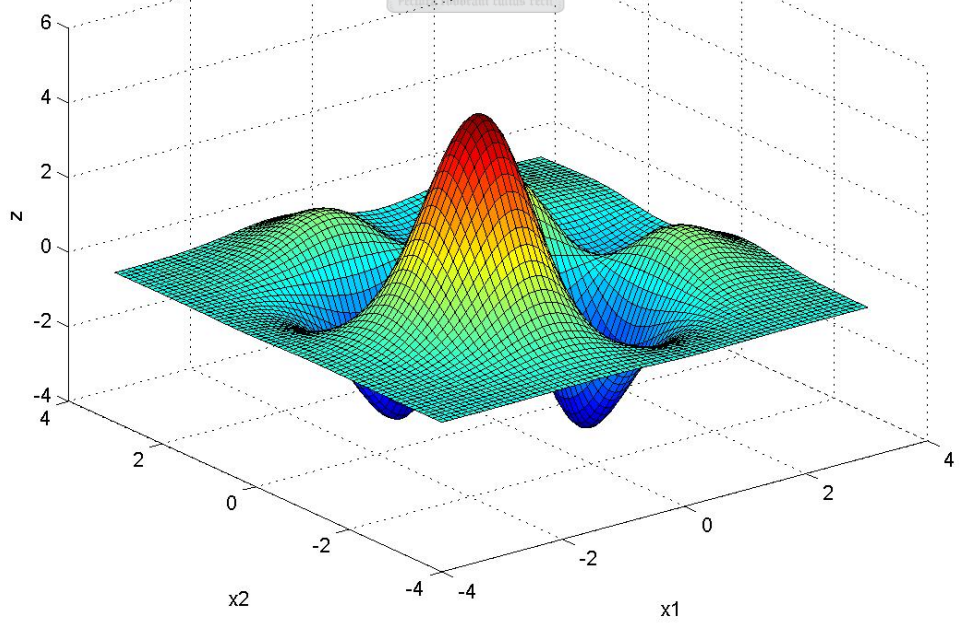


Figure 4.1: Surface plot of z-expression

Data was generated for x_1 and x_2 for the interval stated above. Increments of 0.025 were used. This produced 281 data points for x_1 and 281 data points for x_2. (A mesh of 281 × 281 produced a total of 78 961 data points.) The corresponding z – values were determined.

The MATLAB function 'treefit' was used to fit a tree structure to the data. This was of course the benchmark strategy. The accuracy of the tree was determined on the same set of x_1 and x_2 values known to the tree as R^2 = 0.99. The determined tree contained 519 terminal nodes. Some simplification could be done on this complex tree. The tree was pruned and the optimum number of terminal nodes was determined as 287. This implied a great simplification in the structure of the tree. This simplification can only be warranted if it does not affect the accuracy of the tree. The R^2 = 0.98 of the pruned tree indicated that almost halving the amount of terminal nodes had negligible influence on accuracy. Again it should be noted that the tested x_1 and x_2 values were those used initially in determining the original tree.

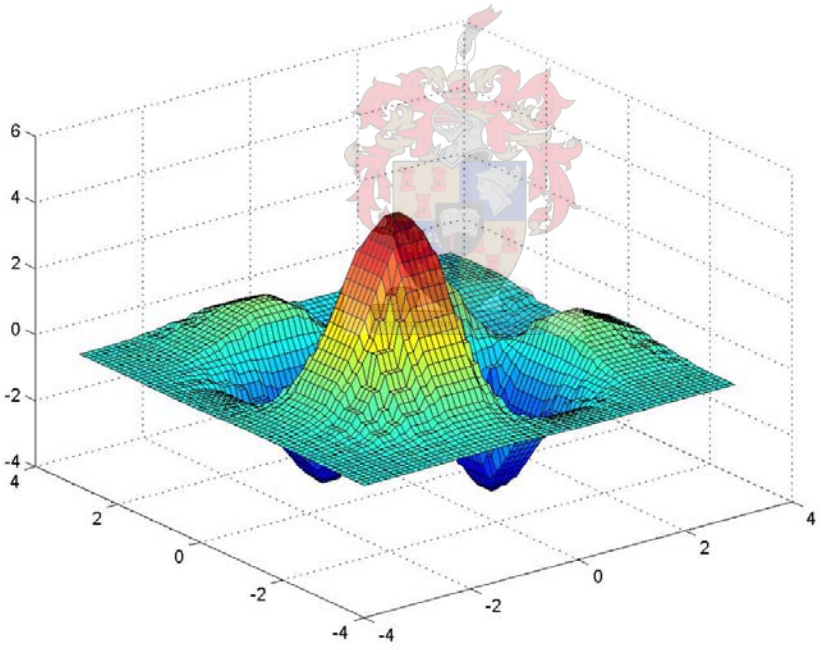


Figure 4.4: Surface plot of z-expression for pruned regression tree on unseen data (287 terminal nodes).

The true accuracy of the original and pruned trees was tested by evaluating unseen data. This unseen data was still in the stated intervals of x_1 and x_2, but now had increments of $\pi/120$. (= 0.02618.) The original and pruned trees both had R^2-values of 0.98. It is difficult to give a text representation, even of the pruned tree, because it still consisted of 287 terminal nodes. This implies that

a result of R^2 = 0.98 was achieved by using 287 individual rules. Further examples will graphically illustrate the trees produced by the treefit function. Figure 4.4 will however present a surface plot of the data generated by the pruned tree on unseen data. (R^2 = 0.98) The result of the incremental approximation of the treefit function can clearly be seen on figure 4.4:

It should be stated that these tree structure functions gave reproducible results. This would not have been the case if an evolutionary algorithm was used to determine the tree structure.

4.2 Regression by means of Silva's Genetic Programming Toolbox

Exactly the same z-expression (eq. 4.1) was investigated. This time genetic programming was used to solve this regression problem. The toolbox used was that of Sara Silva entitled GPLAB. The same 281 data points per input were used to train on as in the previous discussion.

The aim of this investigation was to produce a set of disjunctive rules. Disjunctive rules are formed by implementing the 'or'-statement. Unfortunately the current form of GPLAB does not permit disjunctive rule formation in symbolic regression problems. The reason for this is that GPLAB does not exert any control on the placement of the 'or'-statement. The result is that non executable tree structures are formed. For this reason the regression problem again focussed on conjunctive rules as produced by the treefit function previously used. To illustrate this:

$\sin(\cos(x_1 + x_2))$ allowable part of tree structure (4.2)

$\sin(OR(x_1 + x_2))$ non-executable part of tree structure (4.3)

It was found that a delicate balance between the number of individuals used and the number of generations run existed. Too few individuals caused a lack of diversity even with high mutation rates. Too many individuals caused excessively long runs or even total collapse of the run due to memory shortages. Too few generations didn't allow for this evolutionary programming paradigm to evolve significantly. Again, too many generations made runs excessively long. These long runs of up to 30 hours did not produce significant improvement but increased the complexity greatly. The phenomenon of increased model complexity without comparable model accuracy improvement is called model bloat (Koza, 2004). In the end a run with 400 individuals and 20 generations produced the best result.

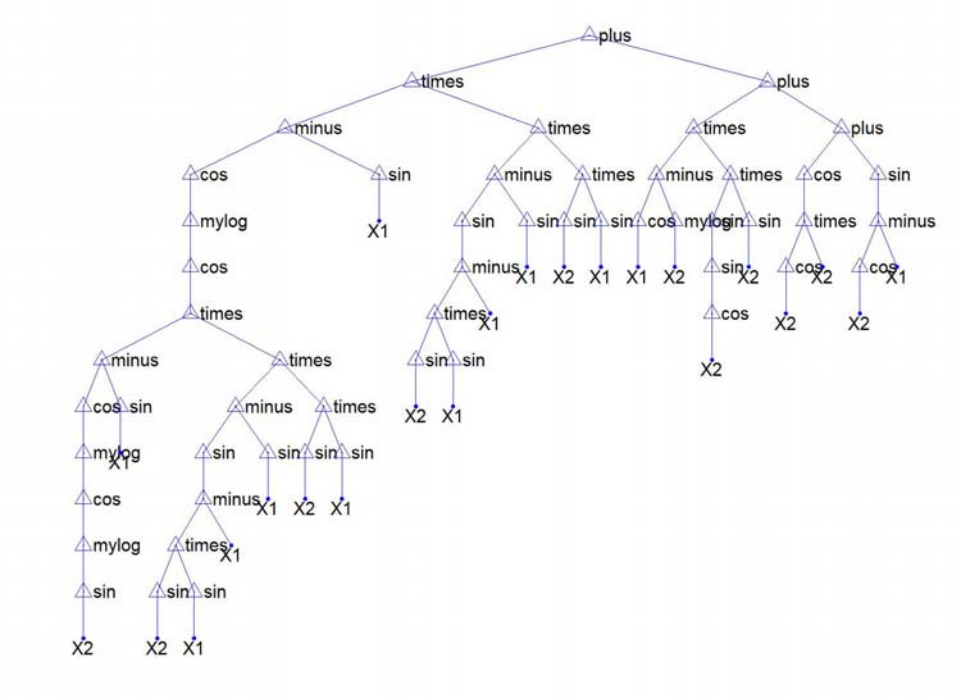


Figure 4.5: Tree structure produced by Best Individual (Silva's GPLAB).

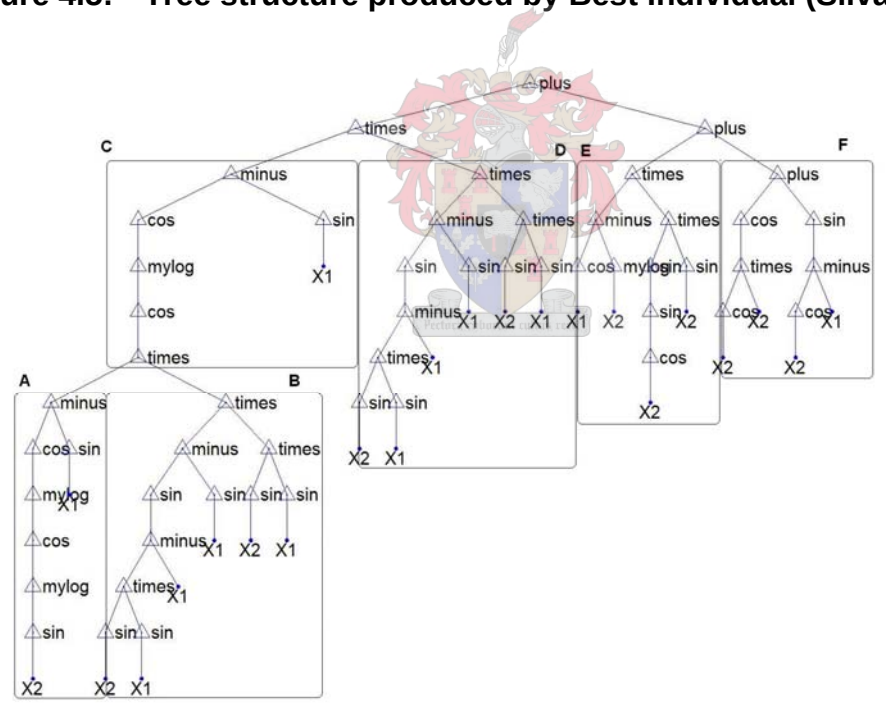


Figure 4.6: Subdivided tree structure of Best Individual (Silva's GPLAB).

Figure 4.5 illustrates the tree structure produced: (Note that this example was not case sensitive.). The structure also represents a single mathematical equation that can be evaluated by simplifying each branch. Figure 4.5 was

subdivided into five arbitrary sections, labelled A to F, to assist in the simplification. The equation derived from Figure 4.6 then becomes:

$$Answer = C \times D + E + F \tag{4.4}$$

Where

$$C = \cos(\log(\cos(A \times B))) - \sin(x_1) \tag{4.5}$$
$$A = \cos(\log(\cos(\log(\sin(x_2))))) - \sin(x_1) \tag{4.6}$$
$$B = (\sin(\sin(x_2) \times \sin(x_1) - x_1) - \sin(x_1)) \times (\sin(x_2) \times \sin(x_1)) \tag{4.7}$$
$$D = (\sin(\sin(x_2) \times \sin(x_1) - x_1) - \sin(x_1)) \times \sin(x_2) \times \sin(x_1) \tag{4.8}$$
$$E = (\cos(x_1) - \log(x_2)) \times (\sin(\sin(\cos(x_2))) \times \sin(x_2)) \tag{4.9}$$
$$F = \cos(\cos(x_2) \times x_2) + \sin(\cos(x_2) - x_1) \tag{4.10}$$

Clearly combining the fragments of equation 4.4 will add no further insight into this empirical model. For this reason the model was not recombined as a single expression.

No pruning was done, since this is not a function supported by GPLAB. The true accuracy was again tested by evaluating the same unseen data as before. (x_1 and x_2, but now had increments of $\pi/120$. (0.02618)) The Best Individual tree had an R^2 -value of R^2 = 0.91. This, of course, was a worse result than achieved with the treefit benchmark strategy.

Figure 4.7 represents a surface plot of the data generated by the Best Individual tree on the unseen data. (R^2 = 0.91)

In conclusion GPLAB was unable to produce disjunctive rules in this symbolic regression problem. Furthermore the best result achieved was less accurate than that of the treefit benchmark strategy. It should be stated that the GPLAB toolbox gave non reproducible results. This was expected since it is based on evolutionary programming.

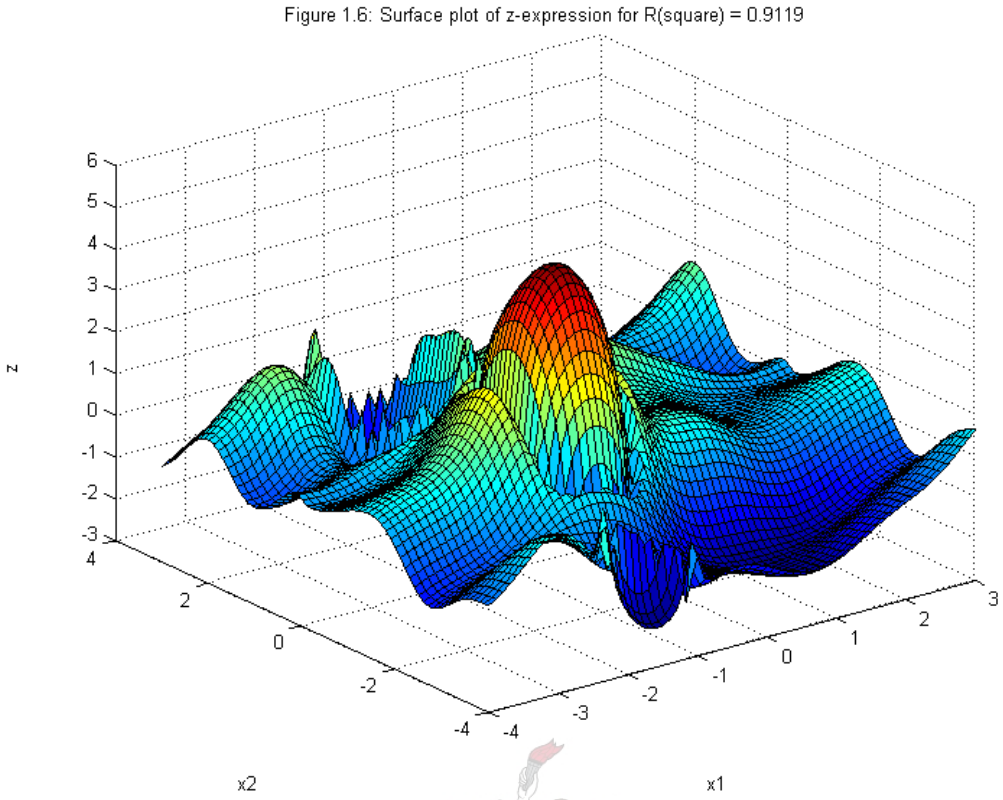


Figure 4.7: Surface plot of z-expression for Best Individual tree on unseen data (Silva's GP).

4.3 Regression by means of GP250 Toolbox

Data were again generated for x_1 and x_2 for the interval stated above. This was the same as used previously in the GPLAB investigation and Statistics Toolbox investigation. The investigation now focussed on the application of GP250.m. Various runs were undertaken differing in number of individuals used and number of generations evolved. Finally it was decided to again make use of 400 Individuals and 20 Generations as was the most successful case found for Silva's GPLAB. The runs took about 2 minutes which was considerably shorter than Silva's approximately 20 minute runs. GPLAB produced more visual outputs than GP250.m**.** The calculations necessary in these operations can account for some of the time needed by GPLAB during each run as compared with GP250.m.

Typical correlation coefficients ranged from 0.80 - 0.90. These correlation coefficients were all produced for the training data. Due to the evolutionary nature of the Genetic Programming employed by GP250.m reproducible results were not obtained. GP250.m also does not automatically recombine the output to form a visual representation of the obtained tree. Manually

recombining the mathematical operators and parameters using the indicated levelling produced:

$$z = \sin(\sin(\log_{10}(\sin(\exp(\sin(x1)))) + \log_{10}(\exp(\exp(\sin(x1))))) - \sin((\sin(x2) + (x2 + x1)))) \quad (4.11)$$

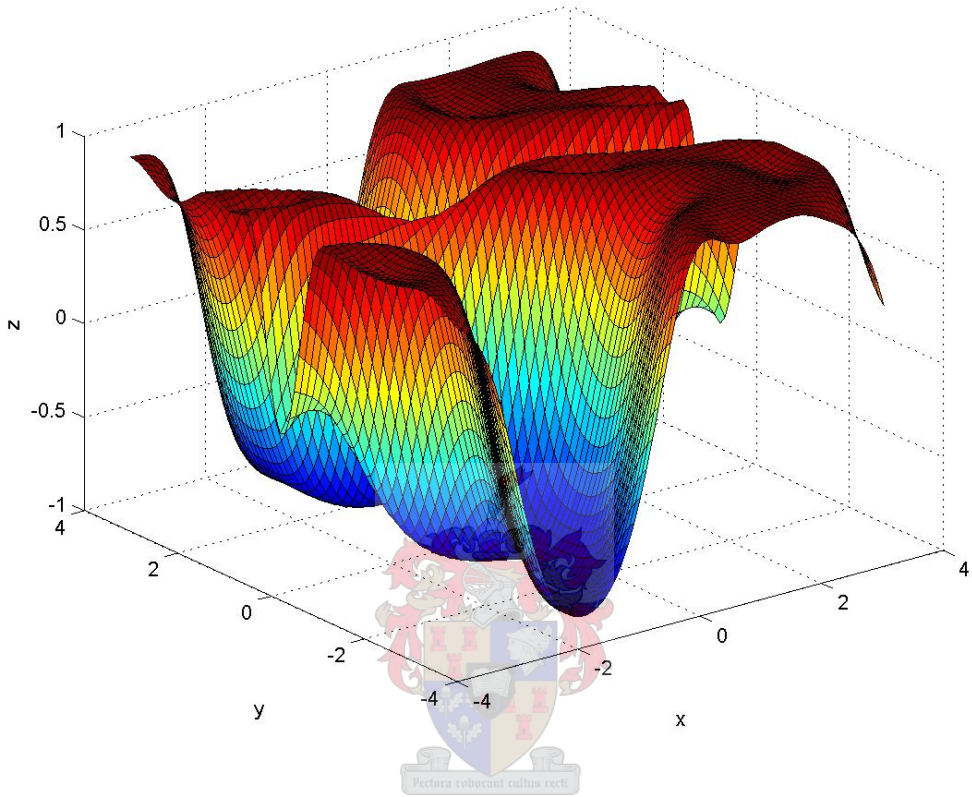


Figure 4.8: Surface plot of best fit z-expression produced by GP250.m

Clearly figure 4.8 deviates greatly from figure 4.1. The R^2 = 0.16 also indicates the poor fit on unseen data. The predicted model produced by GPLAB achieved a much better result than GP250.m

4.4 Regression by means of Discipulus

This time the GP algorithm, as applied by Discipulus, were used to investigate the regression problem. The training data set previously used was too big for Discipulus to train on. The training data set used now had increments of 0.05 and not 0.025. A finer increment of up to 0.39 was possible. The finer data set did not lead to better regression models. For this reason an increment of 0.05 was deemed sufficient. A total of 141 increments were made that produced a mesh consisting of 19 881 data entries. This was much smaller than the previous training data set of 281 data entries. (281 data entries produced a

mesh of consisting of 78 961 data entries.) The test data were generated by $\pi/80$.

Discipulus only makes use of mathematical operators to produce functions. The possibility of GP to produce disjunctive rules was thus not an option while evaluating this implementation of GP. The default Discipulus settings were found to be adequate. The default settings were 500 individual that ran for approximately 50 000 generations. A total of 25 – 26 million individuals were produced in total. These runs took about 4 – 5 hours. The produced C/C++ code was not as graphically illustrative as was the result produced by GPLAB. Still the code could be exported to MATLAB for further investigation. Figure 4.9 illustrates the best individual found:

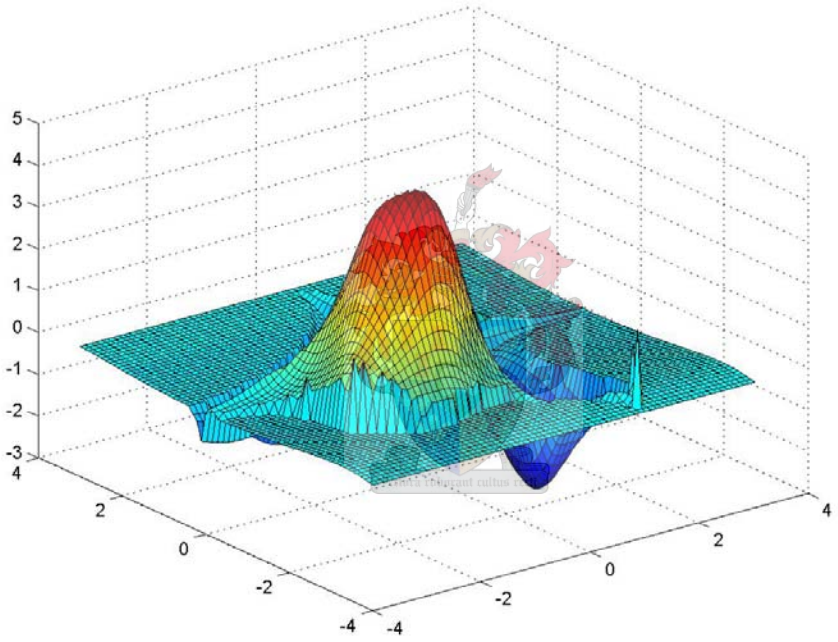


Figure 4.9: Surface plot of z-expression for Best Individual tree on unseen data. (R^2 = 0.72)

The derived regression model had a R^2 = 0.72. The C/C++ code could not be converted to a single expression by Discipulus. Transforming C/C++ code to a single closed expression is a supported feature of Discipulus, but in this case the expression was too long to be viewed.

In conclusion, Discipulus was able to produce a symbolic regression model with a R^2 = 0.72. This was not the best prediction produced by GP. Moreover, many of the runs made with Discipulus gave non-reproducible results, but these runs were fewer than with the other algorithms.

4.5 Summary of investigation of OLS Regression

The benchmark CART model (MATLAB's function *treefit*) achieved an accuracy of R^2 = 0.98. The final pruned tree consisted of 287 individual rules which were simpler than the original tree consisting of 519 derived rules. A large training set of 78 961 entries was used. It was now possible to evaluate GP evolved models, since the 287 individual rules was the best possible solution obtainable by statistical regression

Originally it was planned to derive models consisting of a disjunctive rule set by means of GP. In the end this was not possible, since only GPLAB could use Boolean function and operators and mathematical operators and functions, but this GP application was not a strongly typed application. This led to the development of non-executable rules, as was shown earlier. In the end all GP applications considered this case study as a pure regression problem in which models were derived using only mathematical operators and functions. The same training data set was used for GPLAB as was used in the benchmark strategy. The best GPLAB model had an accuracy of R^2 = 0.91 on the same test set used previously.

The same training data set was used for GP250 as was used in previous investigations. During training promising models with a R^2 = 0.80 - 0.90 were evolved. When the test set was evaluated the models failed in achieving the same results. The best GP250 model had an accuracy of R^2 = 0.16 on the same test set used previously.

The training data set previously used was too big for the investigation employing Discipulus. The final training data set used had a total of 141 increments producing a mesh of 19 881 data entries. The test data were generated by $\pi/80$. The best model had a R^2 = 0.72. The C/C++ code could not be converted by Discipulus to a single expression. Transforming C/C++ code to a single closed expression is a supported feature of Discipulus, but in this case the expression was too long to be viewed.

As can be seen from Table 4.1, the benchmark strategy outperformed all other GP evolved models. GPLAB performed the best of the GP models and the achieved R^2 = 0.91 is at least comparable to that of statistical tree regression:

Table 4.1: Summary of all methods employed in OLS regression case study

Method:	Input Variables used:	Accuracy: (R^2 on unseen data)	Model description:	Training set size:
Statistical tree regression	x_1, x_2	0.98	287 simple individual rules	78961
GPLAB	x_1, x_2	0.91	1 complex empirical model	78961
GP250	x_1, x_2	0.16	1 complex empirical model	78961
Discipulus	x_1, x_2	0.72	1 complex empirical model	19881

5. Case study 2: Chess Board Patterns

5.1 Regression by means of MATLAB's Statistics Toolbox

A set of binary data had to be classified with a classification tree. The binary data resembled a chess board. The complete set of data is represented as Figure 5.1:

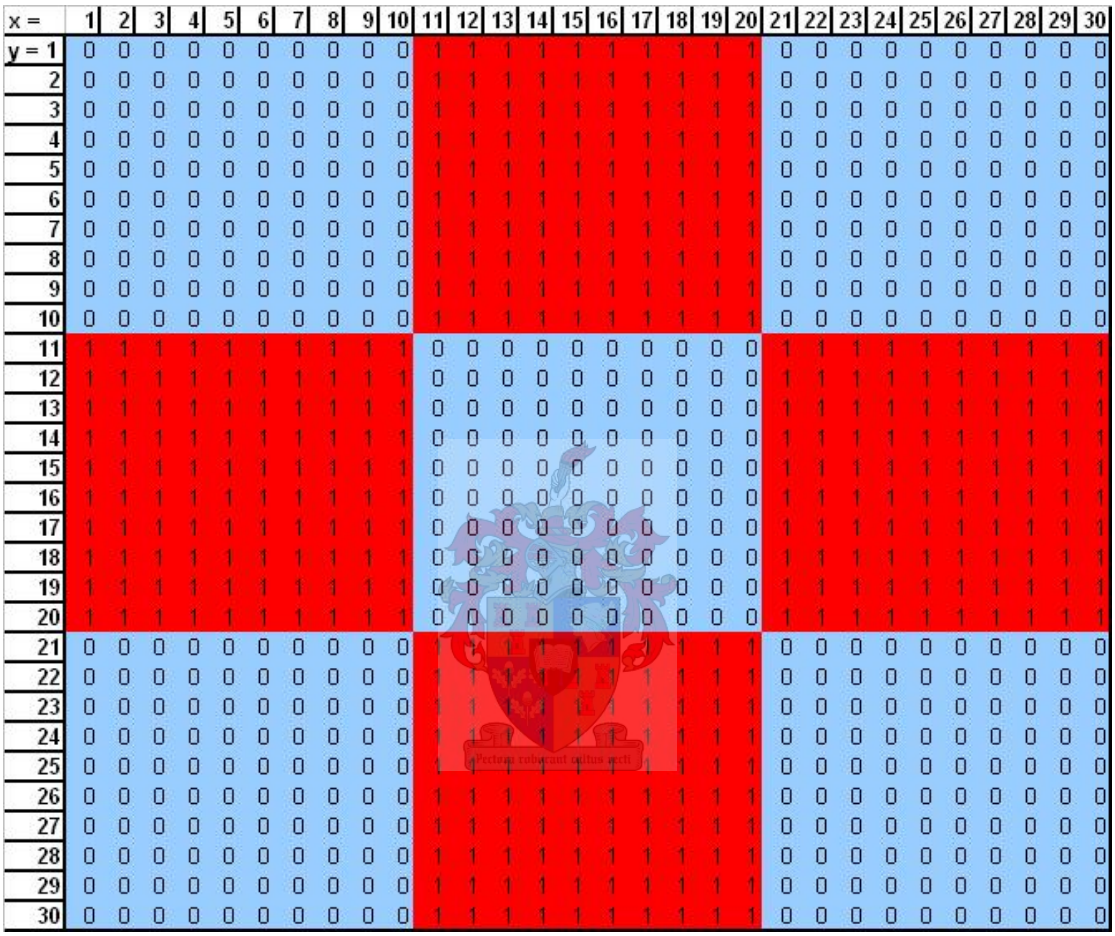


Figure 5.1: Graphic representation of the Chessboard data set.

A subsampled data set was constructed out of the complete binary data set. A tree structure was fitted to this subsampled data set using MATLAB's Stats toolbox and the treefit function. This acted as the benchmark strategy. Through trial and error it was discovered that the subsampled data set consisting of only a specific set of 51 data points was sufficient to produce a tree structure that could classify all 900 original data points correctly. Executing this tree structure produced an exact replica of Figure 5.1. The subsampled data set is represented as Figure 5.2:

x =	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30
y = 1	0	0	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0
2	0	0	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0
3	0	0	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0
4	0	0	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0
5	0	0	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0
6	0	0	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0
7	0	0	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0
8	0	0	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0
9	0	0	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0
10	0	0	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0
11	1	1	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1	1	1
12	1	1	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1	1	1
13	1	1	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1	1	1
14	1	1	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1	1	1
15	1	1	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1	1	1
16	1	1	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1	1	1
17	1	1	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1	1	1
18	1	1	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1	1	1
19	1	1	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1	1	1
20	1	1	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1	1	1
21	0	0	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0
22	0	0	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0
23	0	0	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0
24	0	0	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0
25	0	0	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0
26	0	0	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0
27	0	0	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0
28	0	0	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0
29	0	0	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0
30	0	0	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0

Figure 5.2: Subsampling of the Chessborad data (51 data points). The two classes are denoted by the different colours.

It was found that each of the 9 sub blocks in the original data set had to be sampled between 5 – 7 times. It should be stated that not any 51, or even more, data points would necessarily be the correct data points to sufficiently predict all 900 original data points. Furthermore it would be possible that by correctly choosing other data points a tree structure predicting all 900 original data points could be produced. This new selection of sampling data points could even possibly have less than 51 data points. It was decided that a 100% correct predictive tree with less than 6% (51 data points) sampling of the original data produced a good enough result.

As could be expected a tree with 9 terminal nodes produced the correct and most efficient classifier. It should be noted that all 9 terminal nodes had the binary values of 1 or 0. In the tree structure x is denoted as x1 and y as x2. The tree is represented as Figure 5.3:

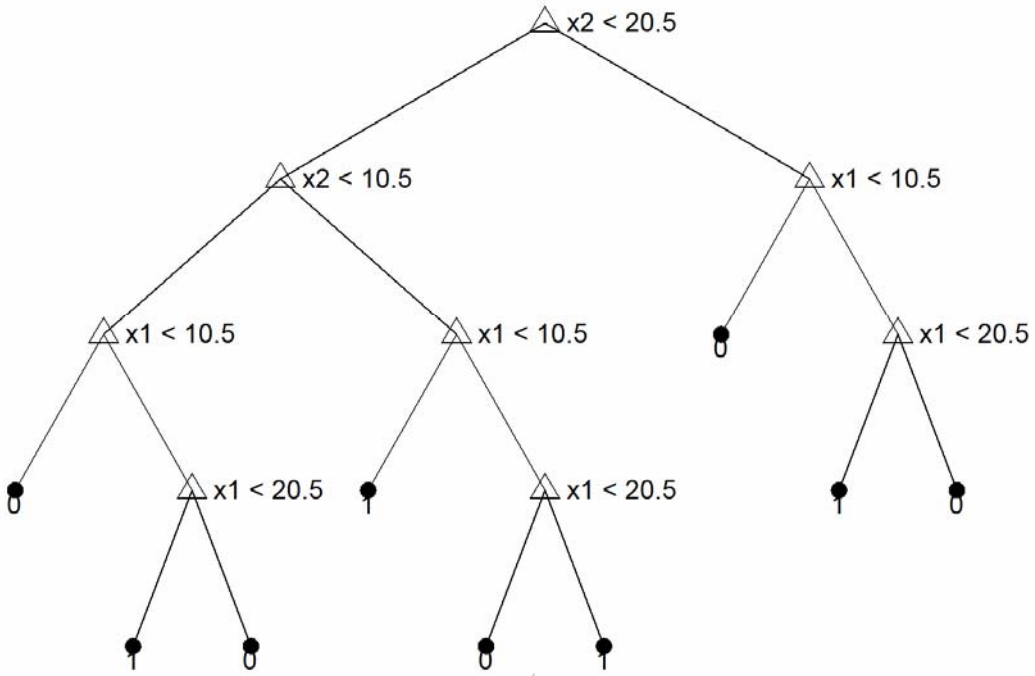


Figure 5. 3: Representation of a correct tree and the most efficient tree.

The classification tree had reproducible results. This would not have been the case if an evolutionary classification algorithm was used.

5.2 Tree – development by means of Silva's Genetic Programming Toolbox

The same problem was attempted again but by using Silva's GPLAB toolbox. The aim of this investigation was to produce a set of disjunctive rules. Disjunctive rules are formed by implementing the 'OR' – statement. In this example it was possible to produce disjunctive rules. Clearly the 51 points chose previously to train on contained sufficient information to correctly predict all 900 points. The same 51 points were initially used to produce a tree structure using genetic programming.

Because of available computing power and on the recommendation found in Koza (1992) it was decided to make use of 500 Individuals. (Koza (1992) actually recommended using 400 individuals.) Various runs were executed varying the amount of generations. The number of generations varied from 10 to 100. Unfortunately a good result was never achieved. The R^2 – values ranged between $R^2 = 0.025$ and $R^2 = 0.030$. Between 70% - 80% of the diversity were lost within 5 – 15 generations even with mutation rates in excess of 50%. A typical poorly performing disjunctive tree is shown below.

This specific tree was run for 100 Generations using 500 Individuals and had a $R^2 = 0.0276$.

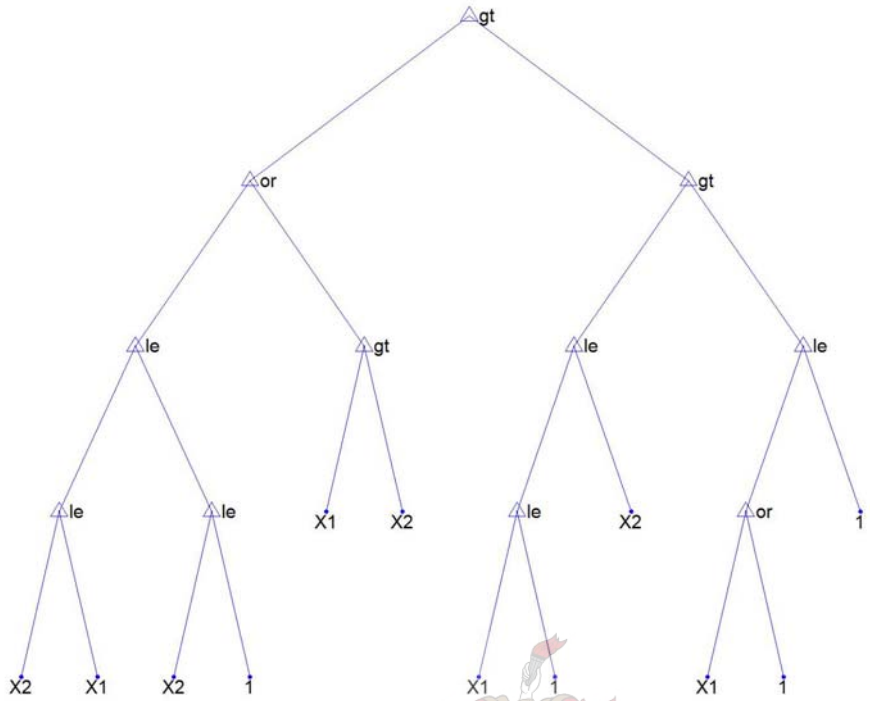


Figure 5.4: Typical poor performing disjunctive tree. ($R^2 = 0.0276$)

The training set was changed to see if other sets did not produce better results. This was indeed the case. It was found that training GPLAB on all 900 points produced the best result. Unfortunately the result was still poor. As final result GPLAB consistently produced a tree that produced only zeros for any coordinate pair shown on Figure 5.1. Again various runs were performed using 500 Individuals and 10 to 100 Generations.

One of these 'zero' – producing disjunctive trees are shown below. This specific tree was run for 100 Generations using 500 Individuals.

The correctly predicted values achieved by the above displayed tree will be all the 'zero'- entries shown in light blue below. All of the 'one'- entries were wrongly predicted and left with no fill colour in Figure 5.6.

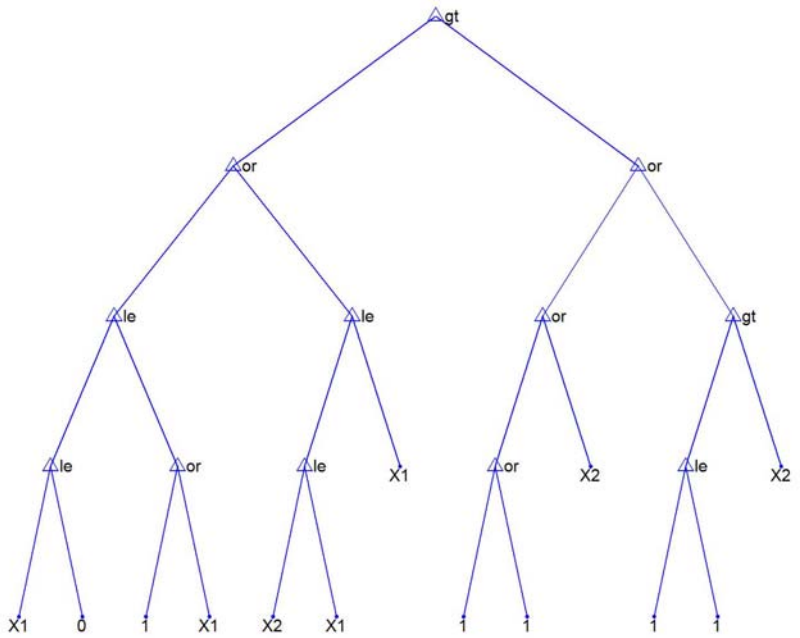


Figure 5.5: Typical 'zero' – producing disjunctive tree.

x =	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30
y = 1	0	0	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0
2	0	0	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0
3	0	0	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0
4	0	0	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0
5	0	0	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0
6	0	0	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0
7	0	0	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0
8	0	0	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0
9	0	0	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0
10	0	0	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0
11	1	1	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1	1	1
12	1	1	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1	1	1
13	1	1	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1	1	1
14	1	1	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1	1	1
15	1	1	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1	1	1
16	1	1	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1	1	1
17	1	1	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1	1	1
18	1	1	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1	1	1
19	1	1	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1	1	1
20	1	1	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1	1	1
21	0	0	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0
22	0	0	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0
23	0	0	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0
24	0	0	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0
25	0	0	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0
26	0	0	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0
27	0	0	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0
28	0	0	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0
29	0	0	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0
30	0	0	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0

Figure 5.6: Correct predictions by typical 'zero' – producing disjunctive tree shown in colour.

Lastly the influence of the ‘or’- operator was investigated. Various runs were conducted under the same conditions as previously stated, but excluded the option of using the ‘or’- operator. (500 Individuals at 10 – 100 Generations training on all 900 points.) Although various trees were produced all had the same net effect of being ‘zero’- producing conjunctive trees.

One of these ‘zero’- producing conjunctive trees are shown below. This specific tree was run for 100 Generations using 500 Individuals.

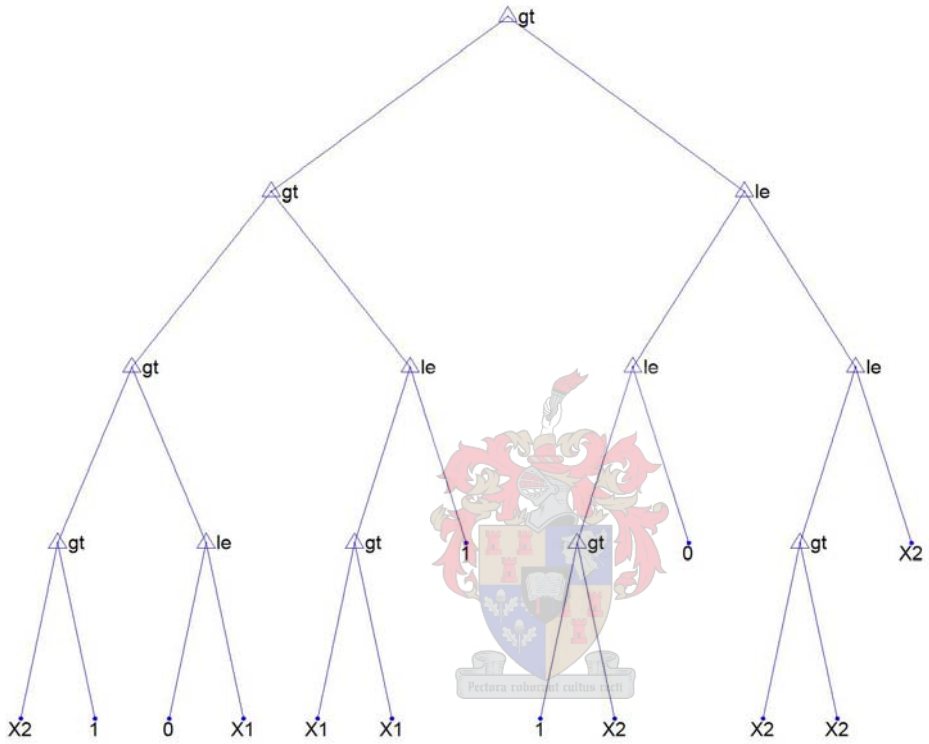


Figure 5.7: Typical ‘zero’ – producing conjunctive tree.

Clearly the ‘or’- operator had little or no influence. As conclusion it can be stated that the way in which GPLAB is setup does not allow for the efficient classification of binary data if presented in such a chess like block format. The Statistics Toolbox of MATLAB did achieve much better results.

5.3 Tree development by means of GP250.m employing Genetic Programming

The problem was attempted a third time in total and a second time using genetic programming. As stated in the title GP250.m was used in this investigation. The aim of this investigation was investigate whether the failure

of GPLAB to correctly classify the data was due to programming limitations in GPLAB or due to limitations to genetic programming as an algorithm.

The same 51 points chose previously to train on was used again as training set. Various runs were undertaken differing in number of generations employed and number of individuals per generation. The individuals ranged from 50 – 400 and the generations from 20 – 1000. In total 20 – 30 runs were undertaken. The best model produced by GP250.m had a best correlation value of 0.588. It consisted of 42 mathematical operators, x1 and x2. The model had a depth of 7 levels as proposed by the output of GP250.m The model was reconstructed manually and was found to be:

$$Class = (\cos(\exp(\frac{x_1}{x_2})) + \cos(\cos(\exp(\frac{x_1}{x_2})))) \times ((\cos(\tan(x_1 - x_2)) + \cos(\exp(x_2 + x_1))) + ((\cos(\tan(x_1 \times x_2)) + \cos(\tan(x_1 - x_2)))) \quad (5.1)$$

The model was evaluated on all 900 data points. Unfortunately the model produced a narrow band of values ranging from 1.5937 – 1.5958. GP250.m does not automatically determine the threshold values necessary for classification. All the values were rounded to the nearest class. For all 900 data points this implied being classified as class 1. In essence the same result was obtained from GP250.m as with GPLAB in that genetic programming failed to produce a model capable of binary classification.

5.4 Regression by means of Discipulus.

As final investigation the same training data subset used previously was provided to the commercial GP package Discipulus to train on. Because of application of the GP algorithm it was able to execute a vast number of individuals per generation. Typically a run consisted of 500 individuals that evolved for 50 000 generations. A typical run then consisted of 25 000 000 total individuals. Such a run took about 60 – 90 minutes on the computer in use.

Discipulus does not produce the same type of decision / classification tree as GPLAB, but the resulting C/C++ code can still be evaluated. Instead Discipulus produced a regression model. From the C/C++ code it was clear that both x and y coordinates were used in evolving the regression model. The C/C++ code will not be shown in text. The result of the model will rather be shown shortly. Values of 0.5 and bigger were interpreted as ‘1’ and values of smaller than 0.5 were interpreted as ‘0.’ The best individual prediction produced was evaluated and the resulted is illustrated below:

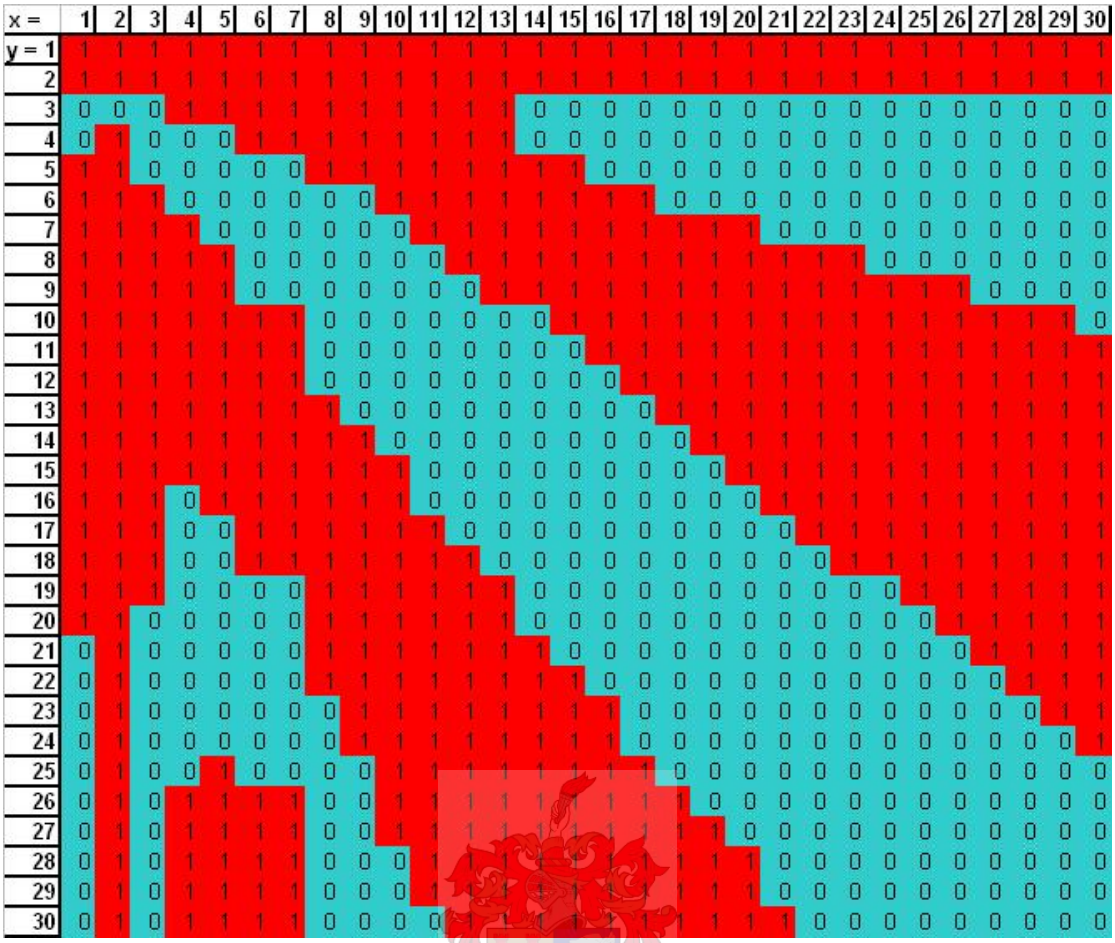


Figure 5.8: Best prediction by Discipulus

The Discipulus model had an accuracy of 70.7%. This was the only of the GP codes that produced a result other than a trivial result. It should be remembered that the trivial zero prediction still had an accuracy of 55%. The complex evolved Discipulus model effectively outperformed the trivial results by 15%. Some of the 51 provided training data points were misclassified in the final prediction.

5.5 Summary of investigation of Chess Board classification example

The training set consisted of 51 data points as illustrated previously. The test set was the prediction of all 900 data entries including the 51 which were available for training. The training data set and test set was the same for all the investigations.

The benchmark statistical tree regression (MATLAB's function treefit) achieved an accuracy of 100%. The tree model consisted of 9 terminal nodes indicating that 9 rules were derived for perfect classification. It was now possible to evaluate GP evolved models since the derived 9 individual rules was the best possible solution obtainable by statistical regression.

In this case study it was possible to derive disjunctive rules using GPLAB. This was the case since Boolean operators and functions could be used exclusively to evolve a binary classification model. Unfortunately a good model was never evolved. Even when the training set was expanded the best model evolved was a trivial zero producing tree. This tree predicted all 900 entries to be zeros under all possible scenarios. By doing this no classification was performed and an accuracy of 500 out of 900 entries was achieved.

During the GP250 investigation GP was used to derive a regression model that when rounded to the nearest class associated integer would produce a classification model. Only mathematical operators and functions were used in evolving the model. By employing GP in this fashion it was investigated whether the failure of GPLAB to correctly classify the data was due to programming limitations in GPLAB or due to limitations to genetic programming as an algorithm. Unfortunately the evolved model produced the same trivial solution in not classifying the data as was the case with GPLAB.

As final investigation the commercial GP package Discipulus was used to evolve a model. Discipulus also derived a classification model by rounding a regression model to the nearest class associated integer as was the case with GP250. Again only mathematical operators and functions were used to evolve these models. Because of application of the GP algorithm as a stand alone application Discipulus was able to execute a vast number of individuals per run. The best Discipulus model had an accuracy of 70.7%. This was the only of the GP codes that produced a result other than a trivial result. It should be remembered that the trivial zero prediction still had an accuracy of 55%. The complex evolved Discipulus model effectively outperformed the trivial results by 15%.

Table 5.1: Summary of all methods employed in Chess Board case study

Method:	Input Variables used:	Accuracy: (% on total data set)	Model description:	Training set size:
Statistical tree regression	x, y	100	9 simple individual rules	51
GPLAB	x, y	56	1 trivial classification model	51
GP250	x, y	44	1 trivial empirical model	51
Discipulus	x, y	71	1 complex empirical model	51

From the above table it is clear that, as in the previous case study, the benchmark strategy performed the best. GP did succeed in using far less individual number of rules, but at the expense of accuracy in classification.

6. Case Study 3:Precipitation of Zinc in Ammoniacal Solutions (*Hydrology* data set)

6.1 Non-Linear Tree Regression Analysis

Ammonium chloride based hydrometallurgical processes have generated considerable interest in recent years, since aqueous solutions of high concentrations of ammonium chloride are especially appropriate for the treatment of complex raw materials of both oxide and sulphide types (Figueiredo *et al.*, 1993; Limpo *et al.*, 1992). A detailed knowledge of the solubility of metal chlorides as a function of temperature and composition of solution is necessary for process development and in this case study data generated by Limpo *et al.* (1995) are examined by means of biplots.

This data set consisted of 108 exemplars obtained from an experiment where zinc chloride is hydrolysed in a watery ammoniacal-ammonium chloride solution. Four variables were measured, namely the temperature of the solution (°C), the concentration of chloride anions (Cl^-), the concentration of zinc cations (Zn^{2+}) and the ammonia concentration (NH_3). The temperature ranged from 30-50°C, and the concentrations were all less than 5 M. Under these conditions, three phases of zinc chloride can occur, *viz.* $Zn(NH_3)_2Cl_2$, $Zn(OH)_2$ and $Zn(OH)_{1.6}Cl_{0.4}$. Of the 108 observations, 43 were associated with the precipitation of $Zn(NH_3)_2Cl_2$, $Zn(OH)_2$ and $Zn(OH)_{1.6}Cl_{0.4}$, while six observations consisted of mixed precipitates, *i.e.* three observations associated with $Zn(NH_3)_2Cl_2$-$Zn(OH)_{1.6}Cl_{0.4}$ and three observations associated with $Zn(OH)_2$-$Zn(OH)_{1.6}Cl_{0.4}$.

In this case study four input variables were thus associated with one target variable or class. The inputs were: TEMP (Temperature (deg C)), CL (Chloride concentration (M)), ZN (Zinc concentration (M)) and NH3 (Ammonia concentration (M)). The output target was PHASE (Type of precipitate))

A tree structure was fitted to the data using MATLAB's *treefit* function. This was again the benchmark strategy. The tree was trained on 82 data points out of a possible 102 acceptable data points. It should be stated that the original data set consisted of 108 data points. Six of these data points had dual target values. These six questionable data points were omitted from the data set used. The phases were originally represented by integers as shown in Table 6.1:

Table 6.1: Integer representation of different phases.

Phase:	Corresponding Integer:
A	1
B	2
C	3

The inputs were represented by x – values as illustrated in Table 6.2:

Table 6.2: x – value representation of different inputs.

Input:		Represented as:
TEMP	Temperature (deg C)	x1
CL	Chloride concentration (M)	x2
ZN	Zinc concentration (M)	x3
NH3	Ammonia concentration (M)	x4

Because of the relative simplicity of the produced tree structure and its imperfect fit pruning was not investigated. The new tree structure obtained is illustrated below as Figure 6.1:

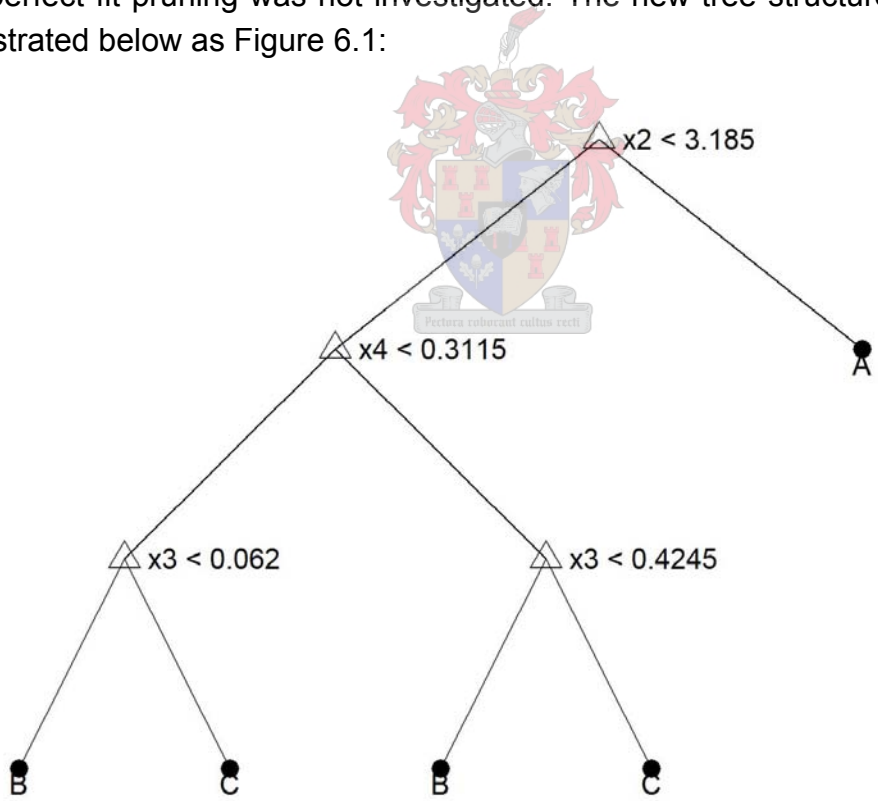


Figure 6.1: Fitted tree structure.

This tree was evaluated on the 20 data points not previously seen. The tree achieved 90% accuracy on the unseen data. (18 of 20 cases correctly

predicted.) The results are shown as Tables 6.3 (the incorrect predictions are indicated in red.) and are comparable to what could be achieved with a linear discriminate analysis model.

Table 6.3: Input data, Actual type of precipitate and Predicted type of precipitate for the various temperature classes.

Temperature (deg C)	Chloride (M)	Zinc (M)	Ammonia (M)	Actual type of precipitate	Predicted type of precipitate
50	4.57	0.834	1.093	A	A
50	2.95	0.466	0.940	B	C
50	2.97	0.657	0.360	C	C
50	2.31	0.416	0.221	C	C
50	1.78	0.402	0.101	C	C
50	1.77	0.209	0.621	B	B
50	1.15	0.135	0.068	C	C
40	4.19	0.520	1.521	A	A
40	3.21	0.470	0.874	A	A
40	2.92	0.737	0.272	C	C
40	2.30	0.387	0.174	C	C
40	1.74	0.144	0.145	C	C
40	1.14	0.072	0.063	C	C
30	3.93	0.410	0.430	A	A
30	2.88	0.303	0.521	A	B
30	2.94	0.336	0.498	B	B
30	2.28	0.500	0.137	C	C
30	1.73	0.128	0.128	C	C
30	1.15	0.128	0.479	B	B
30	1.12	0.143	0.044	C	C

It should be stated that these tree structure functions gave reproducible results. This would not have been the case if an evolutionary algorithm was used to determine the tree.

6.2 Tree – development by means of Silva's Genetic Programming Toolbox

The same case study was attempted again but by using Silva's GPLAB toolbox. The aim was now to produce a set of disjunctive rules. Disjunctive rules are formed by implementing the 'or'- statement. In this example it was possible to produce disjunctive rules. In this case study the 'or'- statement and the 'if – then – else' – statement was included. (The 'if – then – else' statement was denoted by 'myif.')

The GPLAB program trained on exactly the same data as what the ‘treefit’ – function trained on. Numerous runs were attempted varying in the number of individuals and generations used. The best run was achieved with 200 individuals and 100 generations. In many runs the quick disappearance of diversity caused premature convergence even with high mutation rates. Normally good runs had accuracies in the region of 50% - 60%. The best run had an astounding accuracy of 90%. This Individual’s tree is illustrated below. Certain sectors of the tree were as labelled as shown:

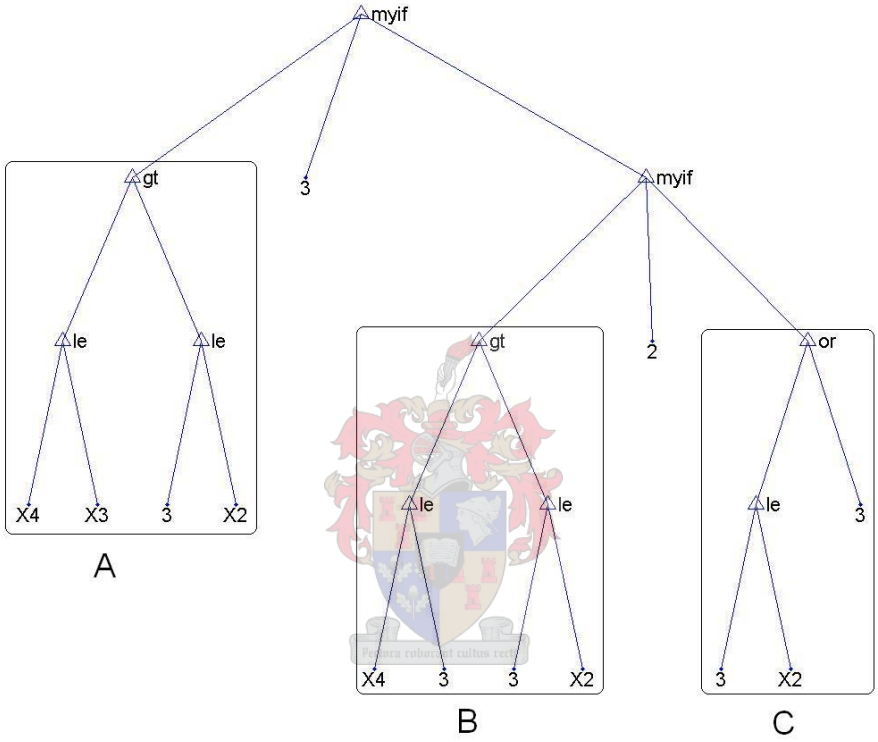


Figure 6.2: Best Individual tree structure. (90% accuracy)

Further investigation revealed that the sector labelled ‘C’ could only yield the binary variable ’1.’ A high level approach would point out the irrelevance of the ‘or’-statement. The implementation of the ‘or’-statement produced a binary variable of ‘1’ or ‘True’ if one or more of the branches of the ‘or’-statement produced a numeric value of 1 or greater. The ‘or’-statement in sector ‘C’ had a trivial answer of ‘1’ since one of the branches consisted of the terminal value of 3.

The left side of sector ‘B’ poses the following condition:

IF X4 ≤ 3 THEN 1

An investigation of the complete data set used shows that this is never the case. This 'IF'-statement in sector 'B' yields a trivial result. Taking this into account and simplifying the complete sector 'B' simplifies to:

IF 3 > X2 THEN 1

Sector 'A' has no trivial sub branches and can not be simplified further. Sector 'A' holds the decisive criteria for deciding whether classification will be done by the first 'if – then – else' – statement or by the second nested 'if – then – else' – statement. With these simplifications taken into account the fitted tree structure can be simplified as follows:

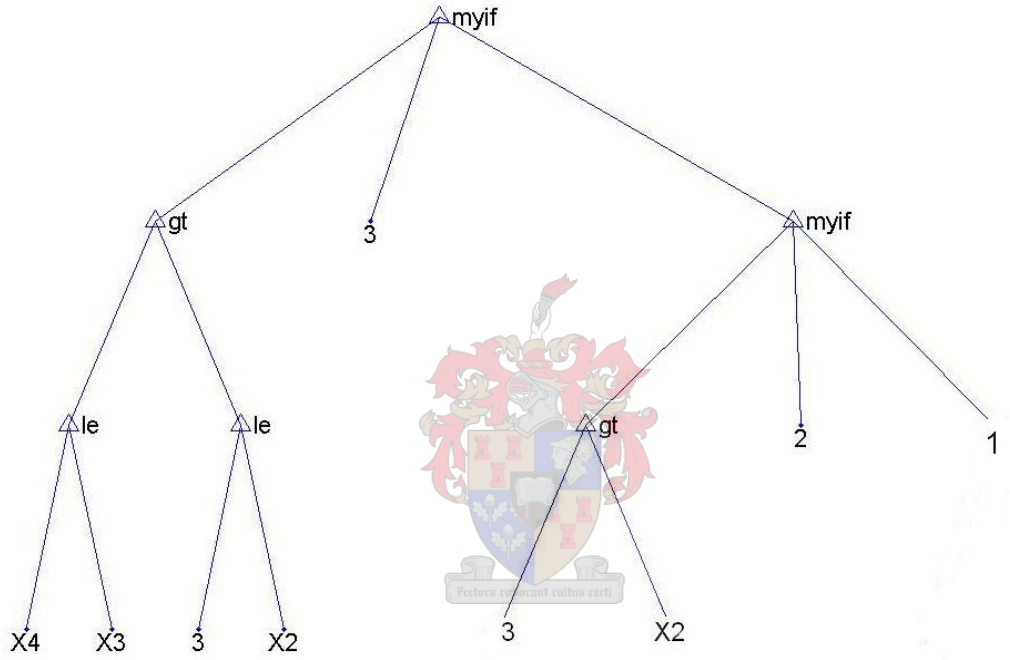


Figure 6.3: Simplified best Individual tree structure. (90% accuracy)

The final rule set consisted of 3 rules. The rules produced by figure 6.3 can be summarized as:

Rule 1:

IF $X4 \leq X3$ AND $3 > X2$
THEN Class 3/C

Rule 2:

IF $X4 \leq X3$ AND $3 \leq X2$ OR
IF $X4 > X3$ AND $3 \leq X2$ OR
IF $X4 > X3$ AND $3 > X2$

AND IF 3 > X2
THEN Class 2/B

Rule 3:

IF X4 ≤ X3 AND 3 ≤ X2 OR
IF X4 > X3 AND 3 ≤ X2 OR
IF X4 > X3 AND 3 > X2
AND IF 3 ≤ X2
THEN Class 1/A

These three rules were used to classify the same test data set used to evaluate the 'treefit' – function. The results of the classification are shown in Table 6.4

Table 6.4: Input data, Actual type of precipitate and Predicted type of precipitate for the various temperature classes and both classification techniques.

Temperature (deg C)	Chloride (M)	Zinc (M)	Ammonia (M)	Actual type of precipitate	Prediction by "treefit"-function	Prediction by GPLAB
50	4.57	0.834	1.093	A	A	A
50	2.95	0.466	0.940	B	C	B
50	2.97	0.657	0.360	C	C	C
50	2.31	0.416	0.221	C	C	C
50	1.78	0.402	0.101	C	C	C
50	1.77	0.209	0.621	B	B	B
50	1.15	0.135	0.068	C	C	C
40	4.19	0.520	1.521	A	A	A
40	3.21	0.470	0.874	A	A	A
40	2.92	0.737	0.272	C	C	C
40	2.30	0.387	0.174	C	C	C
40	1.74	0.144	0.145	C	C	B
40	1.14	0.072	0.063	C	C	C
30	3.93	0.410	0.430	A	A	A
30	2.88	0.303	0.521	A	B	B
30	2.94	0.336	0.498	B	B	B
30	2.28	0.500	0.137	C	C	C
30	1.73	0.128	0.128	C	C	C
30	1.15	0.128	0.479	B	B	B
30	1.12	0.143	0.044	C	C	C

As can be seen from Table 6.4 the GPLAB produced classification also misclassified only two test entries as did the 'treefit' – function. (Both

classification techniques were 90% accurate.) The main difference between the rules produced by the two techniques is that the 'treefit' – function generated five simple rules whereas GPLAB achieved the same accuracy in classification with three more complex rules.

6.3 Tree – development by means of GP250.m employing Genetic Programming:

The hydrology problem was attempted a third time using GP250.m. The same training set of data was used than previously. Numerous runs were attempted varying in the number of individuals and generations. The individuals ranged from 20 – 500 and the generations from 20 – 1000. In total 20 – 30 runs were undertaken.

The best model produced by GP250.m had a best correlation value of 0.957. It consisted of 46 mathematical operators and the variables x1 to x4. The model had a depth of 7 levels as proposed by the output of GP250.m The model was reconstructed manually and was found to be:

$$Class = \sin\left(\frac{\sin(\sin(x_2))}{\log_{10}(x_4)} \times \frac{\log_{10}(\sin(x_4)) - \frac{x_2}{x_4}}{\log_{10}\left(\frac{\sin(x_4)}{\sin(x_3)}\right)} - \frac{\sin(\sin(x_2))}{\log_{10}(\sin(x_4))} \times \sin\left(\log_{10}\left(\frac{x_2}{x_3}\right)\right) + \log_{10}\left(\frac{\exp(x_3)}{x_3} + \sin(\sin(x_2))\right)\right) \quad (6.1)$$

The model was evaluated on the same test data set as was the case in the previous two investigations. GP250.m does not automatically determine the threshold values necessary for classification. The threshold values were approximated as follows:

Table 6.5: Approximation of threshold values:

Range	Integer	Class
< 1.51	1	A
1.51 - 2.50	2	B
> 2.50	3	C

Unfortunately the best model produced by GP250.m did not classify the 20 test cases well at all. Only 5 of the 20 test data cases were classified correctly. Table 6.6 represents the result obtained from classification of the GP250.m model and compares it with the previous strategies:

Table 6.6: Results obtained from various classification methods on test data:

Test Case	Actual type of precipitate	Prediction by "treefit"-function	Prediction by GPLAB	Prediction by GP250.m
1	A	A	A	C
2	B	C	B	C
3	C	C	C	A
4	C	C	C	A
5	C	C	C	A
6	B	B	B	A
7	C	C	C	A
8	A	A	A	A
9	A	A	A	A
10	C	C	C	A
11	C	C	C	A
12	C	C	B	C
13	C	C	C	A
14	A	A	A	A
15	A	B	B	C
16	B	B	B	C
17	C	C	C	A
18	C	C	C	C
19	B	B	B	C
20	C	C	C	A

GP250.m was the only classification strategy that failed to produce an accuracy of 90% on the test data set. It should be stated that GP250.m only used mathematical operators to produce its classification model. GPLAB used Boolean operators and Boolean functions to achieve its classification model.

6.4 Regression model development by means of Discipulus

As a final investigation Discipulus was used to develop a regression model. The values produced by the regression model were rounded to the nearest integer assigned to a class. In doing this the regression model acted as a classification model. In this sense Discipulus and GP250 apply the GP algorithm in the same way. Silva's GPLAB toolbox differs by producing a classification model that resembles an expert system's separate rules.

Numerous runs were attempted. Again the default values of Discipulus produced the best results. The best run was achieved with 500 individuals and

50 000 generations. These runs took about 2 hours on the pc used. Normally good runs had accuracies in the region of 70% - 80%. The best run had an accuracy of 90%. As mentioned previously the derived model is presented to the user in C/C++ code. This does not lend itself to be represented in text. The model was exported to MATLAB and investigated further. The evolved model was tested on the same test set used previously. The table below illustrates the results of the rounding performed to produce the classification model. The misclassified entries are in red:

Table 6.7: Results obtained from various classification methods on test data:

Actual type of precipitate	Regression Prediction	Rounding: (Errors in red)	Class Representation:
1	1.067	1	A
2	2.045	2	B
3	2.556	3	C
3	2.787	3	C
3	2.870	3	C
2	2.032	2	B
3	2.905	3	C
1	1.021	1	A
1	3.000	3	C
3	2.800	3	C
3	2.964	3	C
3	2.722	3	C
3	2.898	3	C
1	1.166	1	A
1	2.170	2	B
2	2.272	2	B
3	3.034	3	C
3	2.795	3	C
2	1.745	2	B
3	3.073	3	C

As can be seen from Table 6.7 Discipulus was able to produce a classification model capable of correctly classifying 18 out of the 20 test entries. A direct comparison between all the results of the various classification models are represented in Table 6.8:

Although the implementation of GP in Discipulus and GPLAB differs greatly both managed to achieve 90% prediction accuracy. The GP code GP250 still produced the worst results. The benchmark approach also produced an accuracy of 90%.

Table 6.8: Actual type of precipitate and Predicted type of precipitate for the various classification techniques.

Actual type of precipitate	Prediction by treefit-function	Prediction by GPLAB	Prediction by GP250.m	Prediction by Discipulus:
A	A	A	C	A
B	C	B	C	B
C	C	C	A	C
C	C	C	A	C
C	C	C	A	C
B	B	B	A	B
C	C	C	A	C
A	A	A	A	A
A	A	A	A	C
C	C	C	A	C
C	C	C	A	C
C	C	B	C	C
C	C	C	A	C
A	A	A	A	A
A	B	B	C	B
B	B	B	C	B
C	C	C	A	C
C	C	C	C	C
B	B	B	C	B
C	C	C	A	C

6.5 Summary of investigation of Hydrology classification example

The training set consisted of 82 data points and the test set consisted of the remaining 20 data entries. The training data set and test set was the same for all the investigations.

The benchmark statistical tree regression (MATLAB's function treefit) achieved an accuracy of 90%. The tree model consisted of 5 terminal nodes indicating that 5 rules were derived for classification. It was now possible to evaluate GP evolved models since the derived 5 individual rules was the best possible solution obtainable by statistical regression

As in the previous case study it was again possible to derive disjunctive rules using GPLAB. This was the case since Boolean operators and functions could be used exclusively to evolve a binary classification model. The GPLAB model had an accuracy of 90%.This case study differs from the previous one in that GPLAB did succeed in evolving a good model. Further investigation into the evolved model showed that the disjunctive 'or'-statement was indeed a trivial part of the tree that could be simplified. One other trivial subbranch was also identified and removed. The simplified model consisted of a 'if – then – else' –

statement with a nested second 'if – then – else' – statement. After simplification the evolved model consisted of 3 individual classification rules.

GP250 was again used to investigate whether GP could derive a regression model that when rounded to the nearest class associated integer would produce a classification model. Only mathematical operators and functions were used in evolving the model. GP250.m does not automatically determine the threshold values necessary for classification. The threshold values were approximated as showed earlier. Unfortunately the best model produced by GP250.m had a classification accuracy of only 25%.

As final investigation the commercial GP package Discipulus was used to evolve a model. Discipulus also derived a classification model by rounding a regression model to the nearest class associated integer as was the case with GP250. Again only mathematical operators and functions were used to evolve these models. Because of application of the GP algorithm as a stand alone application Discipulus was able to execute a vast number of individuals per run as in the previous case study. The best Discipulus model had an accuracy of 90%.

Table 6.9: Summary of all methods employed in Hydrology classification case study

Method:	Input Variables used in model:	Accuracy: (% on total data set)	Model description:	Training set size:
Statistical tree regression	x_2, x_3, x_4	90	5 simple individual rules	82
GPLAB	x_2, x_3, x_4	90	3 complex classification rules	82
GP250	x_2, x_3, x_4	25	1 complex empirical model	82
Discipulus	x_1, x_2, x_3, x_4	90	1 complex empirical model	82

The implementation of the GP algorithm differs greatly between Discipulus and GPLAB. Both implementations produced predictive model accuracies comparable to the benchmark strategy. The importance of this is that GP produced pure classification models, consisting of Boolean operators and functions, and GP produced regression models, consisting of mathematical functions and operators, rounded to the nearest class associated integer could both produce successful classification models. It is important to note

that only Discipulus needed the temperature input, input x_1, to develop its model. Furthermore the rules produced by the GPLAB model lends itself much more to understanding the system compared to the information that could be deduced from the Discipulus empirical model.

7. Case Study 4: Platinum Froth Flotation System

7.1 Introduction

The goals of the investigation were:

- To investigate whether GP could produce better classification compared to standard approaches.

- To investigate whether GP could generate fewer classification rules and achieve comparable classification results compared to standard approaches. (These fewer rules would also be investigated for complexity.)

7.2 Discussion of Data

Froth flotation is an economically important process and an efficient way of treating large ore tonnage. The process is complex, and in practice the control of industrial flotation plants is often based on the visual appearance of the froth phase, which depends on the the experience and ability of a human operator. Control is typically comprised of fixing of initial set points, a settling period to stabilize the process, a period of measurement and evaluation, and a final estimate of appropriate set points. Optimal settings depend on the judgment of the operator and as a result optimal control is not usually maintained, especially where incipient erratic behaviour in the plant is difficult to detect.

This problem can be alleviated by automated decision support for operators, based on a machine vision system that can capture images of the froth and a model that can extract features from the digitized images and classify them into predefined fault states, each of which would be indicative of some process conditions that can be rectified by the operator.

In this case study, 297 records of five image features or variables (x_1, x_2 .. x_5) were considered (Moolman et al., 1995). Each record represent one of three possible process conditions labelled A, B and C.

These classes were determined by a plant expert, with B the desired class representing conditions of optimal stability. Class A represents too stable a froth preventing optimal recovery of the concentrate, while too unstable a froth having a similar net effect is denoted by class C. Figure 7.1. is a principal component score plot (biplot) that gives an impression of the distribution of the froth conditions. The variables (x_1, x_2, .. x_5) are superimposed on the score plot, as explained in more detail in Aldrich et al. (2004) and show fairly strong correlation between variables x_4 and x_5 and x_1, x_2 and x_3 (variables x_1 and x_3

are almost perfectly correlated). As indicated by the figure, classes A and B appear to overlap strongly in the score space, while class C would be fairly easy to distinguish from the other two classes.

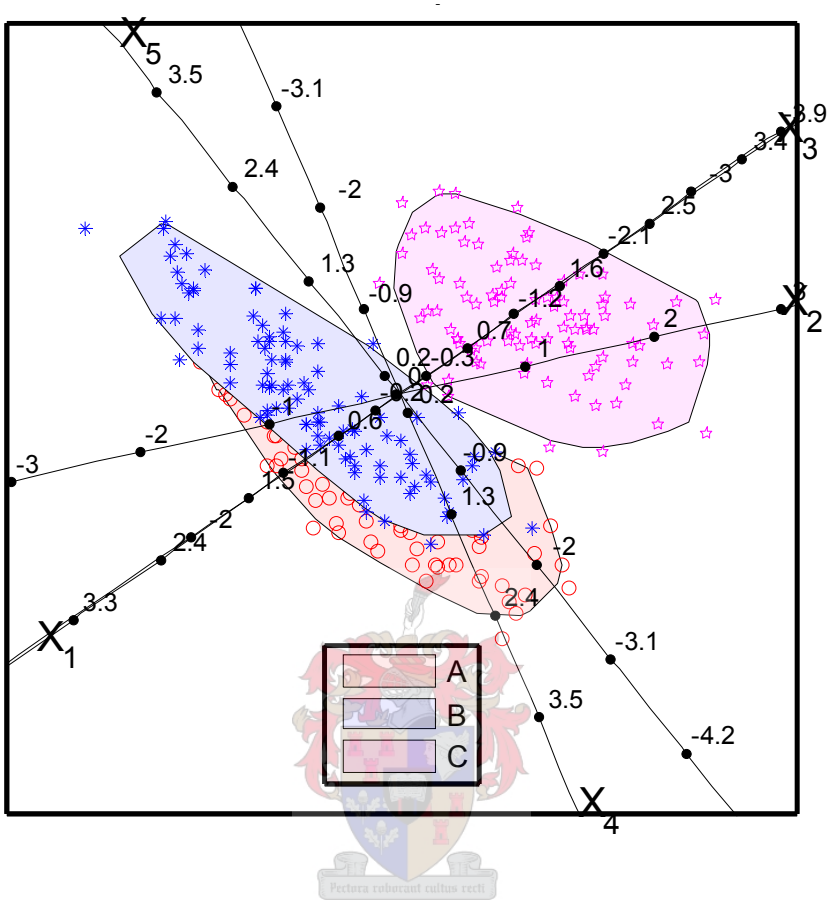


Figure 7.1: PCA biplot of platinum froth flotation features (software to generate figure provided by Aldrich et al., 2004)

7.3 Defining the Benchmark approach

A benchmark classification approach was needed to gauge the efficiency and accuracy of the classification rules developed by GP. The non-linear regression model of classification trees was used as the benchmark. In MATLAB this non-linear regression approach is easily executed with the 'treefit' – function. A training set consisting of 198 data points were constructed. (This will be known as data set A.) Figure 7.2 illustrates the classification tree found:

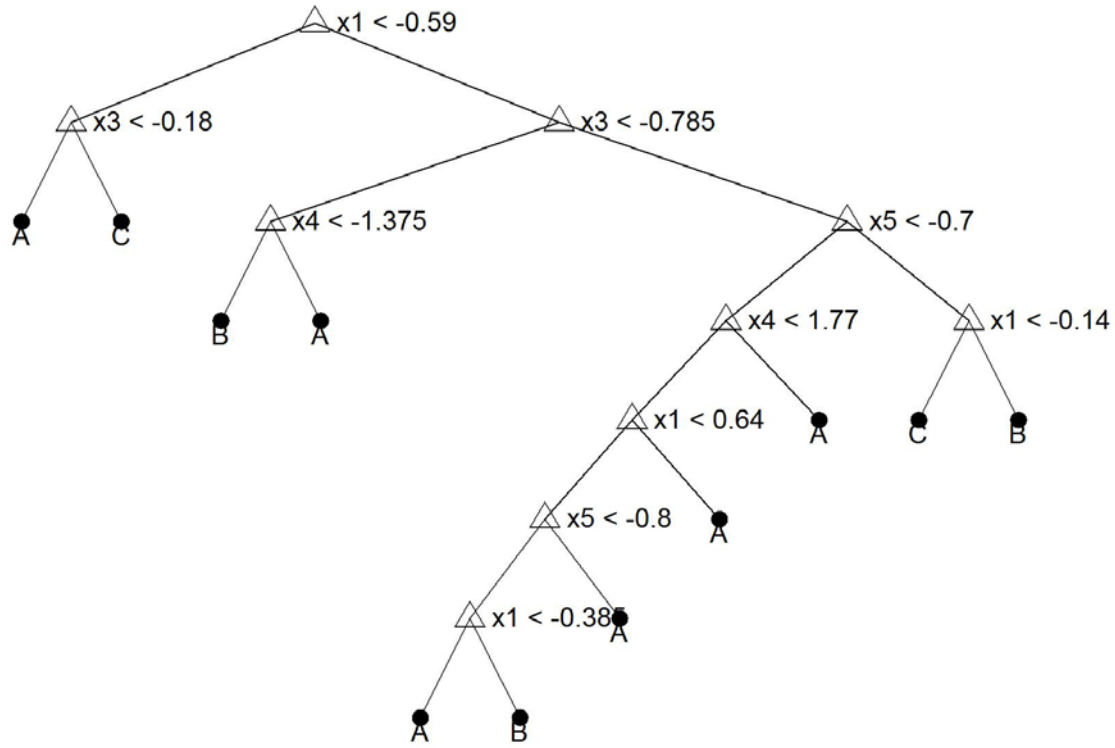


Figure 7.2: Classification tree produced by 'treefit' – function using data set A

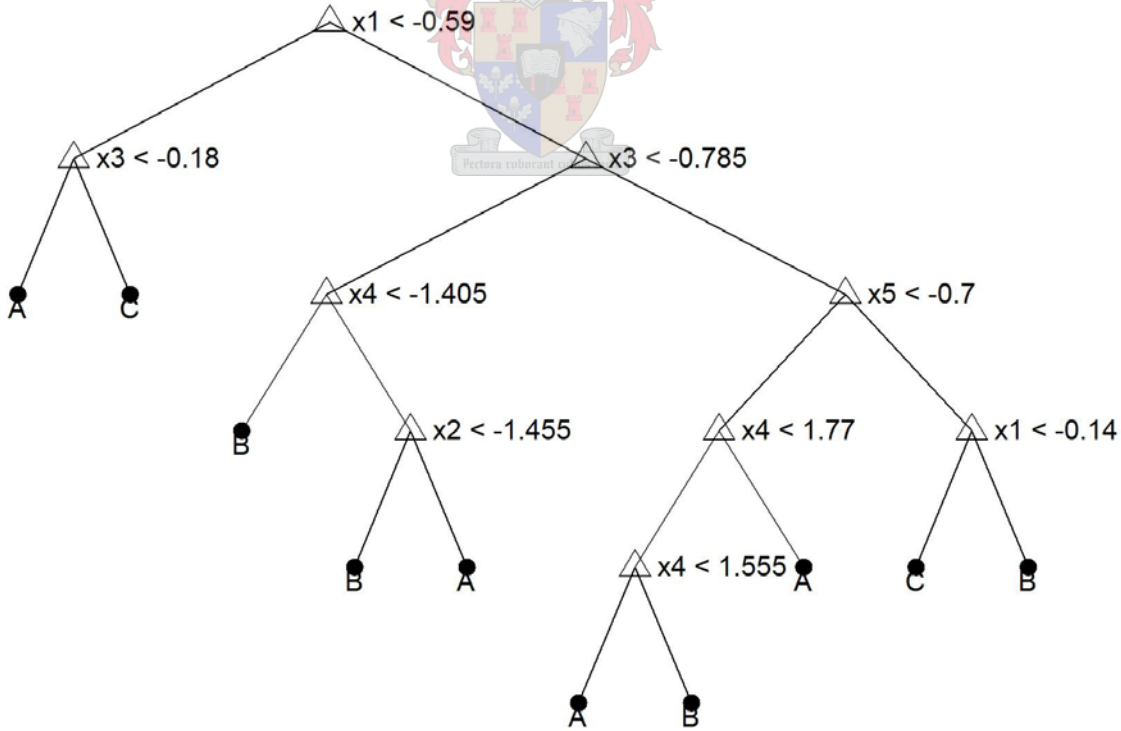


Figure 7.3: Classification tree produced by 'treefit' – function using data set B

Each terminal node represents one classification rule. Figure 7.2 consists thus of 11 separate rules since the tree has 11 terminal nodes. The unused 99 data points formed the test data set A on which this tree was evaluated. It was found that the correct prediction was produced 87 out of the 99 instances. The 12 errors imply an error of about 12%. A second training set consisting of 198 data points were constructed. (This will be known as data set B.) Figure 7.3 illustrates the classification tree found.

Again each terminal node represents one classification rule. Figure 7.3 consists thus of 10 separate rules since the tree has 10 terminal nodes. The unused 99 data points formed the test data set B on which this tree was evaluated. This time it was found that the correct prediction was produced 88 out of the 99 instances. The 11 errors imply an error of about 11%. It is interesting to note that in this case one less rule produced even less errors than the previous classification tree.

7.4 Generating classification rules with GP using GPLAB

Silva's GPLAB – Toolbox, for MATLAB, was used to produce the tree structured programs. Each individual program consisted of inputs, functions and terminals. Table 7.1 defines the set of components:

Table 7.1: Components used by GP to produce classification rules.

Input Type:	**Input:**	**Comment:**
Extracted Variables	'X1', 'X2', 'X3', 'X4', 'X5'	

Function Type:	**Function:**	**Comment:**
Arithmetic Operators:	'+', '-', '×'	Protected '÷' gave Strongly Typed GP errors.
Boolean Operators:	'and', 'or', 'not'	'or'-operator allow for disjunctive rules.
Relational Operators	'< or equal' ,' >'	'equal' was included in '<' Operator

Terminal Type:	**Terminal:**	**Comment:**
Classification	'A', 'B', 'C'	
Extracted Variables	'X1', 'X2', 'X3', 'X4', 'X5'	

Data set A was used for the GP approach for which numerous runs were undertaken varying in number of generations and individuals assigned to each generation. The maximum permitted level of complexity of the runs was also

varied. One of the best trees found utilized 500 individuals for 30 generations. Figure 7.3 illustrates this specific tree:

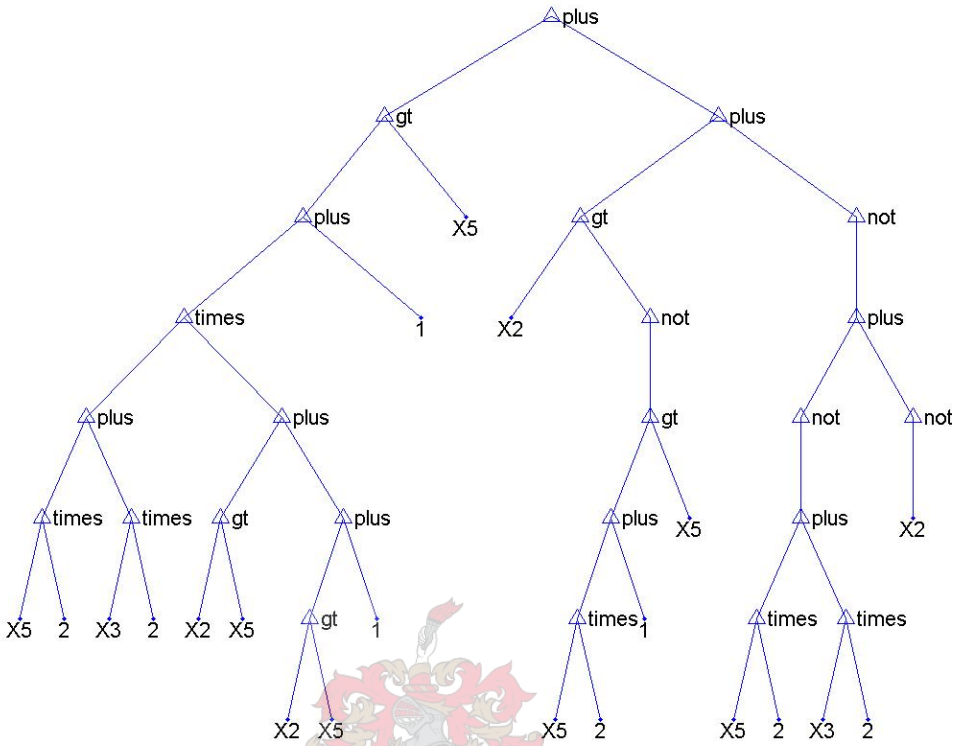


Figure 7.4: Best individual tree produced in a specific run.

The tree in Figure 7.3 was tested on the same data test set as that used in data set A. It was found that this tree misclassified 9 individuals. (90 of the 99 test were correctly classified implying an error of about 9%) This tree performed better than the two classification trees set as benchmarks in the previous section.

7.5 Interpreting GP produced classification rules

Interpreting GP produced classification trees is not as simple as interpreting classification trees produced by the 'treefit' – function. The terminal nodes in the GP tree of Figure 7.3 does not indicate the number of discreet rules. The Boolean operators also lead to a more complex interpretation than what was the case with the 'treefit' – function.

The tree in Figure 7.4 can be divided into 3 sub trees. The 3 sub trees will help in interpreting the complete tree. The 3 sub trees are shown in Figure 7.5:

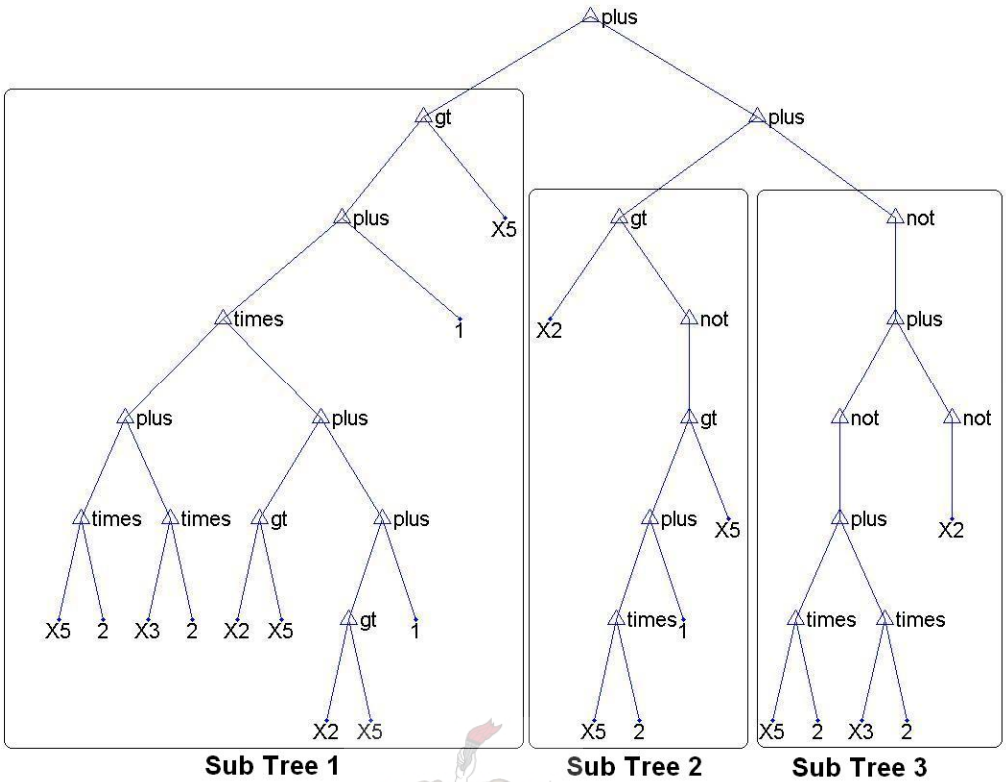


Figure 7.5: Best individual tree split up in to 3 sub trees.

It should be noted that the three classes are now denoted as '1', '2' and '3.' (Previously in the benchmarking cases the classes were 'A', 'B' and 'C.' Figure 7.5 can be simplified as follows:

Sub Tree 1 + Sub Tree 2 + Sub Tree 3

Each sub tree was examined individually. No sub tree had a trivial result implying that the tree as developed was a near optimum tree. Only the 'not'-operator could be simplified as not to be included in the final rule set. The simplification of the sub trees follows:

7.4.1 Sub Tree 1

Scenario 1:

IF X2 > X5 AND 2X5 + 6X3 +1 > X5
THEN 1
IF X5 > X2 AND 2X5 + 2X3 + 1 > X5
THEN 1

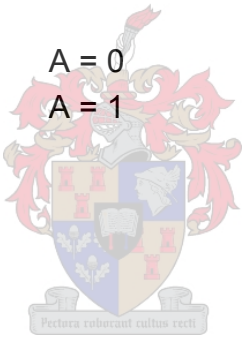
Scenario 2:

IF X2 > X5 AND 2X5 + 6X3 +1 $\leq$ X5
THEN 0
IF X5 > X2 AND 2X5 + 2X3 + 1 $\leq$ X5
THEN 0

7.4.2 Sub Tree 2

Scenario 1:

IF 2X5 +1 > X5 THEN A = 0
IF 2X5 +1 $\leq$ X5 THEN A = 1
IF X2 > A
THEN 1



Scenario 2:

IF 2X5 +1 > X5 THEN A = 0
IF 2X5 +1 $\leq$ X5 THEN A = 1
IF X2 $\leq$ A
THEN 0

7.4.3 Sub Tree 3

Scenario 1:

IF 2X5 + 2X3 $\neq$ 0 AND X2 $\neq$ 0
THEN 1

Scenario 2:

IF (2X5 + 2X3 = 0 OR 2X5 + 2X3 $\neq$ 0) AND X2 = 0
THEN 0

Clearly all 3 sub trees can take values of either 1 or 0. The number of possibilities is thus 2 to the power 3. This comes to a total of 8. In essence 8 different rules could be formulated. Keeping the scenarios in mind and applying Figure 7.4.2 produces the following where the digit after the '=' – sign indicates the class assigned to the individual:

- 1 + 1+ 1 = 3
- 0 + 1 + 1 = 2
- 1 + 1 + 0 = 2
- 1 + 0 + 1 = 2
- 0 + 0 + 1 = 1
- 0 + 1 + 0 = 1
- 1 + 0 + 0 = 1
- 0 + 0 + 0 = 0

The above mentioned expresses the 8 classification rules produced. There was no class '0' and all the data points associated with this outcome were assumed misclassified. Interesting enough only a single data point had the assigned '0' – class. The relationship between the genetic algorithm and genetic programming should be apparent to the reader from the analysis done above.

The 8 possible rules will follow:

Rule 1:

Sub Tree 1, Scenario 1:

IF X2 > X5 AND 2X5 + 6X3 +1 > X5
THEN 1

IF X5 > X2 AND 2X5 + 2X3 + 1 > X5
THEN 1

Sub Tree 2, Scenario 1:

IF 2X5 +1 > X5 THEN A = 0
IF 2X5 +1 $\leq$ X5 THEN A = 1
IF X2 > A
THEN 1

Sub Tree 3, Scenario 1:

IF 2X5 + 2X3 ≠ 0 AND X2 ≠ 0
THEN 1

Sub Tree 1 + Sub Tree 2 + Sub Tree 3

1 + 1 + 1 = 3

Class 3 / C

Rule 2:

Sub Tree 1, Scenario 2:

IF X2 > X5 AND 2X5 + 6X3 +1 ≤ X5
THEN 0

IF X5 > X2 AND 2X5 + 2X3 + 1 ≤ X5
THEN 0

Sub Tree 2, Scenario 1:

IF 2X5 +1 > X5 THEN A = 0
IF 2X5 +1 ≤ X5 THEN A = 1
IF X2 > A
THEN 1

Sub Tree 3, Scenario 1:

IF 2X5 + 2X3 ≠ 0 AND X2 ≠ 0

THEN 1

Sub Tree 1 + Sub Tree 2 + Sub Tree 3

0 + 1 +1 = 2

Class 2 / B

Rule 3:

Sub Tree 1, Scenario 1:

IF X2 > X5 AND 2X5 + 6X3 +1 > X5
THEN 1
IF X5 > X2 AND 2X5 + 2X3 + 1 > X5
THEN 1

Sub Tree 2, Scenario 1:

IF 2X5 +1 > X5 THEN A = 0
IF 2X5 +1 ≤ X5 THEN A = 1
IF X2 > A
THEN 1

Sub Tree 3, Scenario 2:

IF X2 = 0
THEN 0

Sub Tree 1 + Sub Tree 2 + Sub Tree 3

1 + 1 + 0 = 2

Class 2 / B

Rule 4:

Sub Tree 1, Scenario 1:

IF X2 > X5 AND 2X5 + 6X3 +1 > X5
THEN 1

IF X5 > X2 AND 2X5 + 2X3 + 1 > X5

THEN 1

Sub Tree 2, Scenario 2:

IF 2X5 +1 > X5 THEN A = 0
IF 2X5 +1 ≤ X5 THEN A = 1
IF X2 ≤ A
THEN 0

Sub Tree 3, Scenario 1:

IF 2X5 + 2X3 ≠ 0 AND X2 ≠ 0

THEN 1

Sub Tree 1 + Sub Tree 2 + Sub Tree 3

1 + 0 + 1 = 2

Class 2 / B

Rule 5:

Sub Tree 1, Scenario 2:

IF X2 > X5 AND 2X5 + 6X3 +1 ≤ X5
THEN 0

IF X5 > X2 AND 2X5 + 2X3 + 1 ≤ X5
THEN 0

Sub Tree 2, Scenario 2:

IF 2X5 +1 > X5 THEN A = 0
IF 2X5 +1 ≤ X5 THEN A = 1
IF X2 ≤ A
THEN 0

Sub Tree 3, Scenario 1:

IF 2X5 + 2X3 ≠ 0 AND X2 ≠ 0
THEN 1

Sub Tree 1 + Sub Tree 2 + Sub Tree 3

0 + 0 + 1 = 1

Class 1 / A

Rule 6:

Sub Tree 1, Scenario 2:

IF X2 > X5 AND 2X5 + 6X3 +1 ≤ X5
THEN 0

IF X5 > X2 AND 2X5 + 2X3 + 1 ≤ X5

THEN 0

Sub Tree 2, Scenario 1:

IF 2X5 +1 > X5 THEN A = 0
IF 2X5 +1 ≤ X5 THEN A = 1
IF X2 > A

THEN 1

Sub Tree 3, Scenario 2:

IF X2 = 0
THEN 0

Sub Tree 1 + Sub Tree 2 + Sub Tree 3

0 + 1 + 0 = 1

Class 1 / A

Rule 7:

Sub Tree 1, Scenario 1:

IF X2 > X5 AND 2X5 + 6X3 +1 > X5
THEN 1

IF X5 > X2 AND 2X5 + 2X3 + 1 > X5
THEN 1

Sub Tree 2, Scenario 2:

IF 2X5 +1 > X5 THEN A = 0
IF 2X5 +1 ≤ X5 THEN A = 1
IF X2 ≤ A
THEN 0

Sub Tree 3, Scenario 2:

IF X2 = 0
THEN 0

Sub Tree 1 + Sub Tree 2 + Sub Tree 3

1 + 0 + 0 = 1

Class 1 / A

Rule 8:

Sub Tree 1, Scenario 2:

IF X2 > X5 AND 2X5 + 6X3 +1 ≤ X5
THEN 0

IF X5 > X2 AND 2X5 + 2X3 + 1 ≤ X5
THEN 0

Sub Tree 2, Scenario 2:

IF 2X5 +1 > X5 THEN A = 0
IF 2X5 +1 ≤ X5 THEN A = 1
IF X2 ≤ A
THEN 0

Sub Tree 3, Scenario 2:

IF X2 = 0
THEN 0

Sub Tree 1 + Sub Tree 2 + Sub Tree 3

0 + 0 + 0 = 0

Class 0 (Undefined Class)

7.6 Test set results

Table 7.2 shows the results of the test set associated with data set A. The test set was classified with the benchmark classification tree method and with the investigated GP algorithm. The coloured data points indicate all the misclassified data:

Table 7.2: Data test set A evaluated.

Class Given:	Prediction with Classification Tree:	Prediction with GP:	Class Given:	Prediction with Classification Tree:	Prediction with GP:	Class Given:	Prediction with Classification Tree:	Prediction with GP:
1	1	1	2	2	2	3	3	3
1	1	1	2	2	2	3	3	3
1	2	1	2	2	2	3	3	3
1	1	1	2	2	2	3	3	3
1	1	1	2	2	2	3	3	3
1	1	1	2	2	2	3	3	3
1	1	1	2	2	1	3	3	3
1	1	1	2	2	2	3	3	3
1	2	1	2	1	1	3	3	3
1	1	1	2	2	2	3	3	3
1	1	1	2	2	2	3	3	3
1	1	1	2	1	1	3	3	3
1	2	1	2	2	2	3	3	3
1	1	1	2	2	2	3	3	3
1	2	1	2	2	1	3	3	3
1	1	1	2	2	2	3	3	3
1	2	0	2	2	2	3	3	3
1	1	1	2	2	2	3	3	3
1	1	1	2	2	2	3	3	3
1	1	1	2	2	2	3	3	3
1	1	1	2	2	2	3	3	3
1	1	2	2	2	2	3	3	3
1	1	1	2	1	1	3	3	1
1	1	1	2	2	2	3	3	3
1	2	1	2	2	2	3	3	3
1	1	1	2	1	1	3	3	3
1	1	1	2	2	2	3	3	3
1	1	1	2	2	2	3	3	3
1	1	1	2	2	2	3	3	3
1	1	1	2	1	2	3	3	3
1	1	1	2	2	2	3	3	3
1	2	1	2	2	2	3	3	3
1	1	1	2	2	2		12 errors	9 errors

7.7 Generating classification rules with GP using GP250.m

For the third investigation GP250.m was used to generate a model to predict the classification of the flotation froth data set. As stated previously GP250.m also makes use of genetic programming to generate predictive models. Again numerous runs were made varying in number of generations and individuals assigned to each generation. Typically the generations varied from 20 – 1000 and the individuals from 20 – 5000.

One of the best models produced had a best correlation value of 0.871. It was decided to investigate this specific run further since it consisted of only 13 mathematical operators and inputs. The best run achieved had a best correlation, as defined in GP250.m, of 0.880. The best run however consisted of approximately 40 mathematical operators and inputs. It was decided that the approximate 1% better fit on the training data did not warrant an increase of three fold with regards to the number of mathematical operators and inputs

making up the model. The model was reconstructed manually and was found to be:

$$Class = \sin(\sin(x_5) - \cos(x_3)) + \sin(x_3) - x_3 + x_5 \tag{7.1}$$

The model was evaluated on the same test data set as was the case in the previous two investigations. GP250.m does not automatically determine the threshold values necessary for classification. The threshold values were approximated as follows:

Table 7.3: Approximation of threshold values:

Range	Integer	Class
< 1.51	1	A
1.51 - 2.50	2	B
> 2.50	3	C

Unfortunately this model produced by GP250.m did not classify the 99 test cases well at all. It was only accurate for 33 of the 99 test cases implying 33% accuracy. The model misclassified the entire test data set associated with class 3 or C. Table 7.4 represents the result obtained from classification of the GP250.m model and compares it with the previous strategies:

GP250.m was the only classification strategy that failed to achieve 80% accuracy on the test data set. As stated earlier, GP250.m only used mathematical operators to produce its classification model. GPLAB used mathematical operators, Boolean operators and Boolean functions to achieve its classification model.

Table 7.4: Comparison of prediction accuracy of all 3 classification models on flotation froth test data set: (Coloured entries indicate incorrect classification.)

Class Given:	Prediction Classification Tree:	Prediction with GP:	Prediction with GP250.m:	Class Given:	Prediction Classification Tree:	Prediction with GP:	Prediction with GP250.m:	Class Given:	Prediction Classification Tree:	Prediction with GP:	Prediction with GP250.m:
1	1	1	1	2	2	2	1	3	3	3	1
1	1	1	1	2	2	2	3	3	3	3	1
1	2	1	1	2	2	2	2	3	3	3	1
1	1	1	3	2	2	2	3	3	3	3	1
1	1	1	2	2	2	2	2	3	3	3	1
1	1	1	3	2	2	2	1	3	3	3	1
1	1	1	1	2	2	1	1	3	3	3	1
1	1	1	1	2	2	2	1	3	3	3	1
1	2	1	1	2	1	1	1	3	3	3	1
1	1	1	1	2	2	2	3	3	3	3	1
1	1	1	2	2	2	2	1	3	3	3	1
1	1	1	1	2	1	1	2	3	3	3	1
1	2	1	1	2	2	2	1	3	3	3	1
1	1	1	1	2	2	2	2	3	3	3	1
1	2	1	1	2	2	1	2	3	3	3	1
1	1	1	1	2	2	2	1	3	3	3	1
1	2	0	1	2	2	2	1	3	3	3	1
1	1	1	3	2	2	2	3	3	3	3	1
1	1	1	3	2	2	2	2	3	3	3	1
1	1	1	1	2	2	2	1	3	3	3	1
1	1	1	1	2	2	2	3	3	3	3	1
1	1	2	1	2	2	2	1	3	3	3	1
1	1	1	2	2	1	1	1	3	3	1	1
1	1	1	1	2	2	2	1	3	3	3	1
1	2	1	1	2	2	2	1	3	3	3	1
1	1	1	1	2	1	1	3	3	3	3	1
1	1	1	1	2	2	2	2	3	3	3	1
1	1	1	1	2	2	2	1	3	3	3	1
1	1	1	3	2	2	2	1	3	3	3	1
1	1	1	3	2	1	2	1	3	3	3	1
1	1	1	1	2	2	2	1	3	3	3	1
1	2	1	1	2	2	2	2	3	3	3	1
1	1	1	1	2	2	2	2		12 errors	9 errors	65 errors

7.8 Generating a classification model using Discipulus

Discipulus is only capable of binary classification. As stated previously Discipulus produces a classification model by rounding its regression model to the nearest integer assigned to a classification class. This case study necessitated classification of three classes. This was accomplished using Discipulus by handling the training data set as a regression problem. Discipulus fitted a function and the rounding was done manually in exactly the same way binary classification would have been performed.

The same training data set and testing data sets were used as was used previously. Numerous runs were undertaken varying in number of generations and individuals assigned to each generation. The best model was evolved using the default Discipulus parameters of 500 individuals and 50 000 generations. The produced C/C++ code does not lend itself to the same graphical representation as was the code with the produced model from GPLAB. In evaluating the model on unseen data it was found that this model misclassified 11 individuals. (88 of the 99 test were correctly classified implying an error of about 11%) This result was comparable to the result produced by the benchmark method.

7.9 Test set results

Tables 7.5 shows the results of the test set associated with data set A. The test set was classified with the benchmark classification tree method, GPLAB and Discipulus. The previously shown result for GP250 was omitted because of the poor result.

Table 7.5: Data test set A evaluated.

Class Given:	Prediction with Classification Tree:	Prediction with GP:	Prediction with Discipulus:	Class Given:	Prediction with Classification Tree:	Prediction with GP:	Prediction with Discipulus:
1	1	1	1	2	2	2	2
1	1	1	2	2	2	2	2
1	2	1	1	2	2	2	3
1	1	1	1	2	2	2	2
1	1	1	1	2	2	2	2
1	1	1	1	2	2	2	2
1	1	1	1	2	2	1	2
1	1	1	1	2	2	2	2
1	2	1	2	2	1	1	2
1	1	1	1	2	2	2	2
1	1	1	1	2	2	2	2
1	1	1	1	2	1	1	2
1	2	1	2	2	2	2	2
1	1	1	1	2	2	2	2
1	2	1	2	2	2	1	2
1	1	1	1	2	2	2	2
1	2	0	2	2	2	2	3
1	1	1	1	2	2	2	2
1	1	1	1	2	2	2	2
1	1	1	1	2	2	2	2
1	1	1	1	2	2	2	2
1	1	2	2	2	2	2	2
1	1	1	1	2	1	1	2
1	1	1	1	2	2	2	2
1	2	1	2	2	2	2	2
1	1	1	1	2	1	1	2
1	1	1	1	2	2	2	3
1	1	1	1	2	2	2	2
1	1	1	1	2	2	2	2
1	1	1	1	2	2	2	2
1	1	1	1	2	1	2	2
1	1	1	1	2	2	2	2
1	2	1	1	2	2	2	2
1	1	1	1	2	2	2	2

Class Given:	Prediction with Classification Tree:	Prediction with GP:	Prediction with Discipulus:
3	3	3	3
3	3	3	3
3	3	3	3
3	3	3	3
3	3	3	3
3	3	3	3
3	3	3	3
3	3	3	3
3	3	3	3
3	3	3	3
3	3	3	3
3	3	3	3
3	3	3	3
3	3	3	3
3	3	3	3
3	3	3	3
3	3	3	3
3	3	3	3
3	3	3	3
3	3	3	3
3	3	3	3
3	3	3	3
3	3	1	2
3	3	3	3
3	3	3	3
3	3	3	3
3	3	3	3
3	3	3	3
3	3	3	3
3	3	3	3
3	3	3	3
3	3	3	3
	12 errors	9 errors	11 errors

7.10 Summary of investigation of Flotation Froth classification example

The training set consisted of 198 data points and the test set consisted of the remaining 99 data entries. The training data set and test set was the same for all the investigations. Each data point consisted of five extracted features denoted as X1 to X5. A total of three classification classes existed. It was known that that some errors occurred in the classification of the data set. The position and number of the errors were not known.

The benchmark statistical tree regression (MATLAB's function treefit) achieved an accuracy of about 88%. The best tree model consisted of 10 terminal nodes indicating that 10 rules were derived for classification. It was now possible to evaluate GP evolved models since the derived 10 individual

rules was the best possible solution obtainable by statistical regression. All 5 inputs were required to derive the model.

During none of the GP evolved strategies was a disjunctive rule set found to be the best classification model. The GPLAB model had an accuracy of 91% and used only 8 rules to classify the data set. The GPLAB model slightly outperforms the benchmark model regarding accuracy, but the decrease in the number of rules required by two is a significant improvement.

As in previous case studies GP250 was used to investigate whether GP could derive a regression model that when rounded to the nearest class associated integer would produce a classification model. Only mathematical operators and functions were used in evolving the model. Classification threshold values were approximated as showed earlier. Unfortunately the best model produced by GP250.m had a classification accuracy of only 33%.

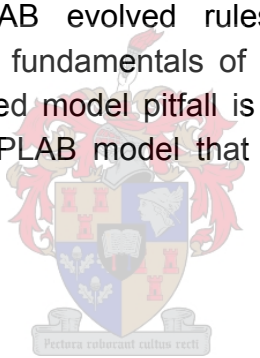
Discipulus is only capable of binary classification. As stated previously Discipulus produces a classification model by rounding its regression model to the nearest integer assigned to a classification class. This case study necessitated classification of three classes. This was accomplished using Discipulus by handling the training data set as a regression problem. Discipulus fitted a function and the rounding was done manually in exactly the same way binary classification would have been performed. The produced C/C++ code does not lend itself to the same graphical representation as was the code with the produced model from GPLAB. The Discipulus model achieved a classification accuracy of 88%. This result was comparable to the result produced by the benchmark method.

Table 7.6: Summary of all methods employed in Flotation Froth classification case study (Case Study 4)

Method:	Input Variables used in model:	Accuracy: (% on total data set)	Model description:	Training set size:
Statistical tree regression	x_1, ,x_2, x_3, x_4, x_5	88	10 simple individual rules	197
GPLAB	x_2, x_3, x_5	91	8 complex classification rules	197
GP250	x_3, x_5	33	1 complex empirical model	197
Discipulus	x_1, x_2, x_3, x_4, x_5	88	1 complex empirical model	197

The first goal was achieved since GP produced comparable classification compared to standard approaches. (GP, using GPLAB, had a 9% error compared to the 11% - 12% of the benchmark strategy.) The second goal was also achieved since GP could generate fewer classification rules and achieve comparable classification results compared to standard approaches. (These fewer rules were more complex than those produced with a classification tree.) This was also achieved with GPLAB. A further important feature of the GPLAB evolved classification rules is that it used only 3 of the 5 inputs provided. The disjunctive 'or' – operator was included in the GPLAB function set, but the best GP rules produced did not include it.

The other two GP applications each had its own pitfalls. GP250 failed completely in deriving a classification model with a comparable classification accuracy compared to the other methods. Discipulus again showed that rounding a regression model to the nearest class associated integer could produce a viable classification model. The first pitfall of the Discipulus model is that little process specific knowledge can be extracted from such a complex empirical model. The GPLAB evolved rules lends it much more to understanding the underlying fundamentals of the process being modelled. The second Discipulus evolved model pitfall is that it used all of the inputs provided compared to the GPLAB model that used only 3 of the 5 inputs provided.



8. Case study 5: Viscosity index on an industrial liquid-liquid extraction plant

Improved operation of liquid-liquid extraction columns can only be achieved when the hydrodynamic and mass transfer regimes induced by the design of the equipment are accounted for. Since the influence of column geometry and rheological characteristics of multiphase extraction systems is not well understood at present, liquid-liquid extraction systems may be difficult to model from first principles. Therefore, as in all the previous case studies, the GP and CART models were developed from process data.

A total of 1345 daily values of stream flow rates (x_1, x_2 ... x_4), degree of impurities (x_5) and temperatures (x_6, x_7 ... x_9) were collected over a period of five years together with the viscosity index of a petroleum product (VI) on an industrial liquid-liquid extraction plant. The temperature gradients and flow rates in the column were controlled in order to maintain a product of constant quality and composition, despite process disturbances associated with changes in the compositions of the feed streams.

The viscosity index is an important process quality indicator that relates the effect of temperature variations and other changes in process conditions on the viscosity of the oil, as indicated in Figure 8.1. Normal operating conditions (NOC) can be defined as those conditions where this quality indicator (VI) remains within certain upper and lower limits, indicated by the horizontal broken lines in Figure 8.1. Plant operation is controlled by ensuring that the VI is maintained within the normal process limits. This is accomplished by monitoring and manipulating the process variables and disturbances related to the quality variable. When the quality or key performance variable (VI) is affected by more than a few variables, as is often the case, manual control becomes difficult without the aid of a process model. The objective of the model was to predict the viscosity index using GP. The data were randomized and split in to a training set of 1076 records and a test set of 269 records.

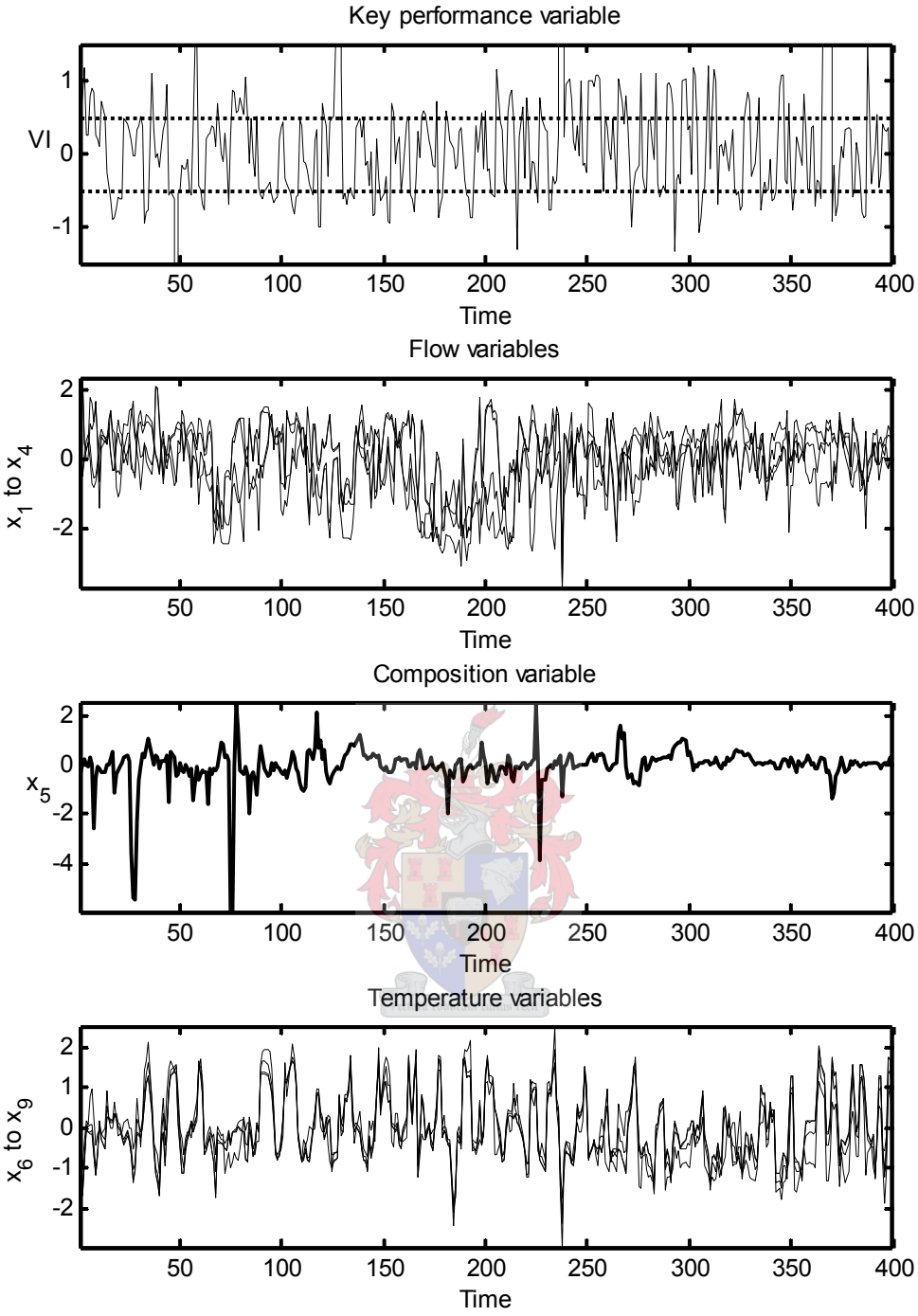


Figure 8.1: Samples of measurements of the variables of the liquid-liquid extraction system. From top to bottom: The controlled variable, the manipulated flow variables, the manipulated composition variable and the manipulated column temperature variables.

8.1 Regression by means of statistical tree regression

The benchmark strategy to compare the success of the GP regression model was statistical regression tree. A tree structure was fitted to the data using

MATLAB's treefit function. The training and test data was used as discussed above. The fitted tree was very complex. It consisted of a total of 481 nodes and 177 terminal nodes. This implies that the regression tree classified the data set using 177 rules. The tree was too complicated to represent graphically in text.

The accuracy was determined on the unseen test data by calculating the simple average of the absolute value of the error. This average error value was determined as 15.96%. Tree pruning was investigated using cross validation. This did not produce any useful results since the best tree size estimated was a single node i.e. a trivial tree.

8.2 Regression by means of Silva's Genetic Programming Toolbox

Exactly the same data sets were evaluated again this time genetic programming was used to solve this regression problem. The toolbox used was that of Sara Silva entitled GPLAB. Various runs were executed. The generations were varied from 40 – 200 and the population size from 300 – 500 individuals. Too many generations lead to excessively complex models with little improvement over more simple models. Increasing the number of individuals per generation increased the computational time dramatically. Some runs took more than 32 hours and produced poor models.

The absolute best model found had an error of 14.91% on the unseen test data. The accuracy was again determined on the unseen test data by calculating the simple average of the absolute value of the error. This model had a tree depth of 17 levels and consisted of 155 nodes. These nodes consisted of mathematical operators, as stated in the previous case studies, and X1 to X9. This model performed 1.05% better than the benchmark strategy.

The model that was investigated further was not the absolute best model found, but a model that had comparable accuracy and less complexity. This model had an error of 15.69%, consisted of 97 nodes distributed on 15 levels. This implies that a model consisting of only 63% of the number of nodes of the absolute best model produced had a penalty on accuracy of only 0.78%. This model was achieved by 40 generations and 500 individuals per generation. The achieved accuracy was just slightly better than the benchmark. (It was 0.27% better than benchmark.) The difference in complexity between this model and the benchmark is enormous. The benchmark, as stated, had 481 nodes and this model only 97. That is a reduction in the number of nodes of almost 80%.

Figure 8.2 below represents the tree structure produced by GPLAB of the run further investigated:

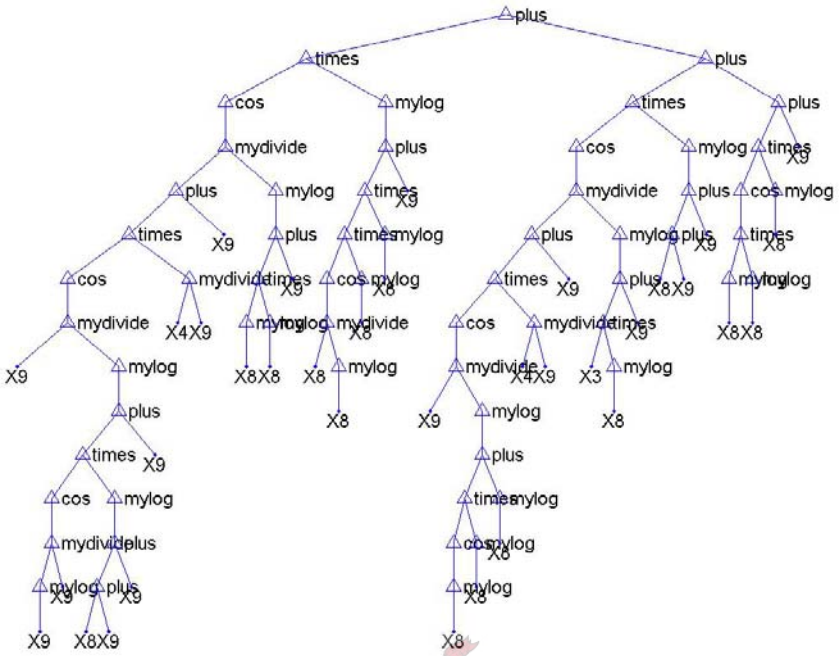


Figure 8.2: GPLAB model of relative high accuracy and low complexity.

The model was broken up into the below indicated fragments and simplified:

The indicated fragments were then computed as follows:

$$A = \cos(\frac{\ln(x_9)}{x_9}) \times \ln(x_8 + 2 \times x_9) + x_9 \quad (8.1)$$

$$B = \cos(\frac{x_9}{\ln(A)}) \times \frac{x_4}{x_9} + x_9 \quad (8.2)$$

$$C = \ln((\ln(x_8))^2 + x_9) \quad (8.3)$$

$$D = \cos(\frac{B}{C}) \quad (8.4)$$

$$E = \ln(\cos(\frac{x_8}{\ln(x_8)}) \times (\ln(x_8))^2 + x_9) \quad (8.5)$$

$$F = \cos(\ln(x_8)) \times \ln(x_8) + \ln(x_8) \quad (8.6)$$

$$G = \cos(\frac{x_9}{\ln(F)}) \times \frac{x_4}{x_9} + x_9 \tag{8.7}$$

$$H = \cos(\frac{G}{\ln(x_3 \times \ln(x_8)}) + x_9) \times \ln(x_8 + 2 \times x_9) \tag{8.8}$$

$$I = \cos((\ln(x_8)^2) \times \ln(x_8) + x_9 \tag{8.9}$$

$$\Pr ediction = H + I + D \times E \tag{8.10}$$

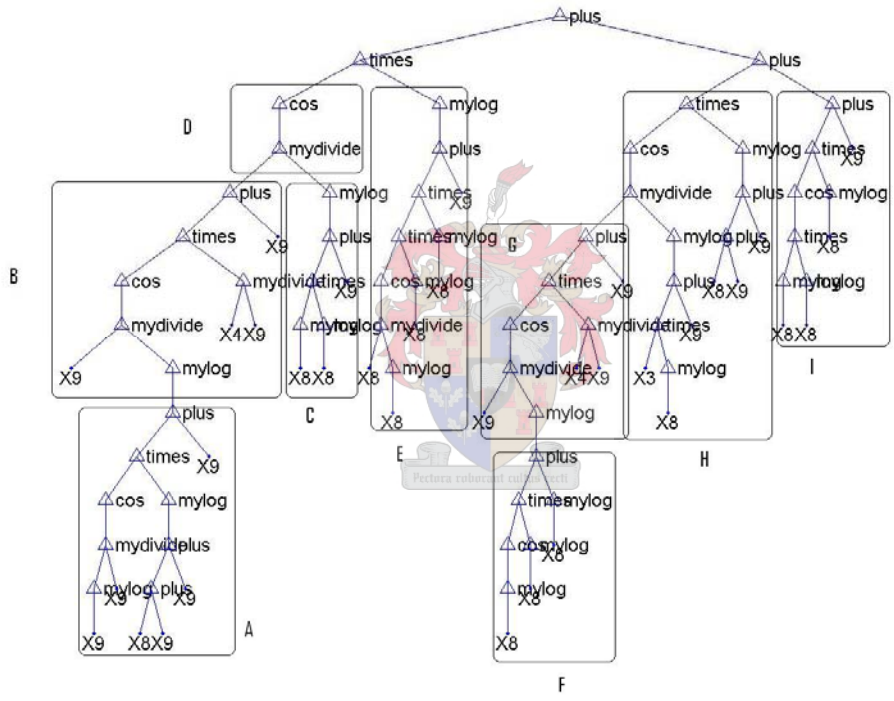


Figure 8.3: Arbitrary model fragmentation for ease of simplification.

The predicted viscosity index is a temperature value in degrees Celsius. Combining the individual fragments of this model into a single empirical model was deemed unnecessary since it would not add any further insight into the model.

8.2 Regression by means of GP250

The case study was attempted a third time by means of the GP code GP250.m. Exactly the same data sets were evaluated. Numerous runs were executed. The generations were varied from 20 – 400 and the population size

from 30 – 300 individuals. This code was much faster in producing models than what was experienced with GPLAB. More complex runs still took 8 – 12 hours.

Unfortunately this code never produced a model with an error of less than 62% on the seen training data. It was deemed unnecessary to test any of the models on the unseen test data because of the poor results. This best model, with a model error of 62%, consisted of 450 nodes and was generated using 100 individuals for 100 generations. The nodes were the same mathematical operators and parameters as used in GPLAB. The typical accuracies obtained by models produced by GP250 had an error of 85% - 90% on the seen training data set. (Lower error percentages indicate better models.) These models ranged widely in the number of nodes used. The typical number of nodes was 100 – 800.

8.3 Regression by means of Discipulus

As final investigation exactly the same data sets were evaluated again. This time Discipulus used genetic programming to solve this regression problem. Various runs were executed.

A run consisted of approximately 50 000 generations with a population size from 500 individuals. A typical run took 3 – 4 hours. The best model found had an error of 15.12% on the unseen test data. The accuracy was again determined on the unseen test data by calculating the simple average of the absolute value of the error. The reported model was in C/C++ code form. As in previous case studies this does not lend itself to tree interpretation. This model performed 0.84% better than the benchmark strategy. The difference was deemed to be of little significance.

The evolved C++ code can not be interpreted as a single equation by Discipulus. Although this is a supported feature the equation produced was too complex to show as a single closed expression. The original model as C++ code was about 110 lines. This implies that transforming the C++ code to a mathematical equation does not enhance model interpretability.

8.4 Summary of investigation of Viscosity Index regression example

The data set consisted of 9 inputs represented by X1 to X9. The output or target value was a viscosity index. The aim was to predict the single target output using GP. The data set consisted of 1345 data points. The data was randomized and split in to a training set of 1076 entries and a test set of 269 entries

The benchmark strategy, statistical regression tree, produced a tree structure consisting of 177 terminal nodes. This implies that the regression tree classified the data set using 177 rules. The accuracy was determined on the unseen test data by calculating the simple average of the absolute value of the error. This average error value was determined as 15.96%.

GPLAB was used to evolve a regression model. The model that was investigated further was not the absolute best model found, but a model that had comparable accuracy and less complexity. This model had an error of 15.69%, consisted of 97 nodes distributed on 15 levels. The achieved accuracy was just slightly better than the benchmark. (It was 0.27% better than benchmark.) The difference in complexity between this model and the benchmark is enormous. The benchmark, as stated, had 481 nodes and this model only 97. That is a reduction in the number of nodes of almost 80%.

As in previous case studies GP250 failed to produce a model with comparable accuracies with regards to the benchmark method or other GP applications. This code never produced a model with an error of less than 62% on the seen training data. It was deemed unnecessary to test any of the models on the unseen test data because of the poor results.

As final investigation Discipulus was used to produce a regression model using GP. The best model found had an error of 15.12% on the unseen test data. The accuracy was again determined on the unseen test data by calculating the simple average of the absolute value of the error. The reported model was in C/C++ code form. As in previous case studies this does not lend itself to tree interpretation. As a single equation the evolved C++ code would comprise in the order of 8 - 10 pages. The original model as C++ code was about 110 lines. This implies that transforming the C++ code to a mathematical equation does not enhance model interpretability.

Only GP250 failed to produce a regression model of decent prediction capabilities. The benchmark method, GPLAB and Discipulus all produced models with prediction accuracies just above 84%. It should be stated that a suspicious data entry formed part of the test data set evaluated by all the models. This suspicious test entry had a viscosity index of 3 °C whereas most entries were in an 80 °C – 120 °C range. If this entry was to be omitted then all three successful methods produced an accuracy of approximately 97%.

It is interesting to note that GPLAB used only four of the nine inputs to derive a GP model with approximately the same accuracy as the other models. Some of the other methods also did not require all the input variables to derive a successful regression model. This is depicted in table 8.1:

Table 8.1: Variables used and accuracies achieved by various methods

Method:	Benchmark tree regression	GPLAB	GP250	Discipulus
% Accuracy:	84.0	84.3	38.0	84.9
Variables used:	All (X1 - X9)	X3, X4, X8, X9	All (X1 - X9)	X1, X3, X4, X5,
				X6, X7, X8, X9

The statistical regression tree needed all nine inputs to achieve comparable model accuracy. Discipulus used all input variables except X2. Since all successful model accuracies are approximately the same the simplest model should be deemed the best model. Since GPLAB used only four of the nine input variables it is the simplest model. The four inputs used and the frequency of use in the GPLAB model are represented in the table below:

Table 8.2: Variables used and their frequency of use.

Input:	Measurement:	Frequency of occurrence:
X3	m^3/h	1
X4	m^3/h	2
X8	°C	15
X9	°C	16

The frequency of occurrence is also of particular interest indicating that the bulk of the model comprised of the temperature measurements X8 and X9. The two volumetric flow rates X1 and X2, X5 measuring percentage impurities and temperature measurements X6 and X7 were never needed in the derived model.

In conclusion using the GPLAB produced model has the following advantages over the other methods:

- Statistical tree regression used 481 nodes compared to the 97 of GPLAB.
- Statistical tree regression produced 177 simple individual regression rules compared to the 1 complex mathematical expression of GPLAB.
- Statistical tree regression used al nine measured inputs to derive its model while GPLAB only used four of the inputs.
- GP250.m failed to produce effective regression models.
- Discipulus could derive a successful model, but it was much more complex than any of the other successful methods like GPLAB.

9. Case Study 6: Classification of surface defects in steel plates

9.1 Discussion of data and benchmark method

As in the previous case study this data set originated from an industrial process. This will then be the second case study using actual process data. In this process steel was hot-rolled into steel plates. This data set differs from the previous data set in that it consists of even more data. It consisted of 20 inputs represented by X1 to X20. The output or target value was a binary variable. A value of 1 indicated an error occurred during the production while a value of 0 indicates successful production. With regards to the output predicted this case study was less complex than the viscosity index prediction case study. The aim was to predict the single target output using GP. GP would produce a classification model. The data set consisted of 3015 data points. The data was randomized and split in to a training set of 2615 entries and a test set of 400 entries, across which examples of both classes were spread equally. The classes were not of the same size, however, since only 27.4% of the examples represented defective steel plates. The table below summarize the measured variables:

Table 9.1: Input parameter representation, measurement and units.

Input:	Measurement:	Units:
X1	Phosphorus content	%
X2	Grinding loss	%
X3	Reheating retention time	h
X4	Width of steel plate	mm
X5	Speed at which plate moves	m/min
X6	Mould level difference	m
X7	Stopper movement	m
X8	Tundish steel mass	ton
X9	Superheated steel temperature	°C
X10	Silicon content	%
X11	Rinse end temperature	°C
X12	Contact time	h
X13	Add to gas end	h
X14	Rinse station stir parameter	Nm^3 / min / °C
X15	Ti3O5 content	kg
X16	TiN content	kg
X17	Product thickness	mm
X18	Mould temperature measurement at specific point	°C
X19	Mould temperature measurement at specific point	°C
X20	Mould temperature measurement at specific point	°C

As previously the benchmark strategy to compare the success of the GP models was statistical regression tree. A tree structure was fitted to the data using MATLAB's treefit function. The training and test data was used as discussed above. The fitted tree consisted of a total of 525 nodes and 36 terminal nodes. This implies that the classification tree classified the data set using 36 rules. The tree was too complicated to represent graphically in text. Tree pruning was attempted with success. The tree was pruned to 17 nodes and 7 terminal nodes. Again the 7 terminal nodes denoted 7 classification rules. The error was measured as the number of misclassified entries. (Lower values indicate less error.) The original tree misclassified 118 of the 400 unseen test cases. (29.5% error) The pruned tree performed slightly better by misclassifying 108 entries. (27% error) This indicates that pruning the tree lowered the possible over fitting of data. The pruned tree is illustrated below:

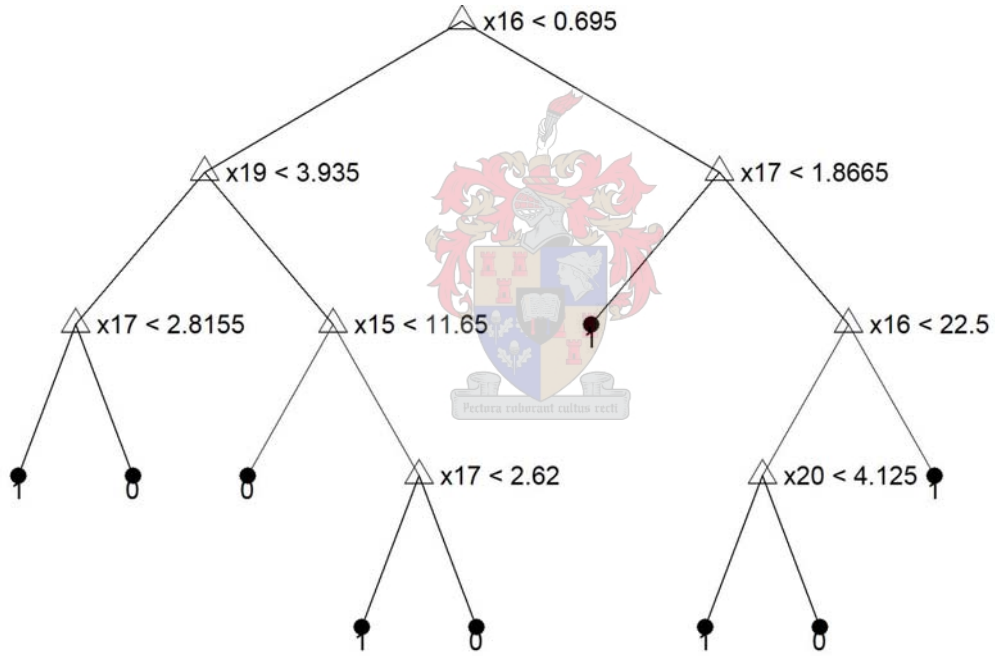


Figure 9.1: Pruned classification tree with 27% error.

It is of interest to note that only X15, X16, X17, X19 and X20 was used to derive the model.

9.2 Regression by means of Silva's Genetic Programming Toolbox

Exactly the same data sets were evaluated again. This time genetic programming was used to solve this classification problem. The GPLAB toolbox was used as in previous case studies. Various runs were executed. The generations were varied from 30 – 300 and the population size from 100 – 500 individuals. As in the previous case study too many generations lead to excessively complex models with little improvement over more simple models. Increasing the number of individuals per generation again increased the computational time dramatically. The duration of the runs was limited to about 18 hours.

The absolute best model found had an error of 26.25% by misclassifying 105 of the unseen test entries. The accuracy was determined on the unseen test data by calculating the number of misclassified entries. This model had a tree depth of 14 levels and consisted of 40 nodes. These nodes consisted of mathematical operators, as stated in the previous case studies, and some of the variables X1 to X20. This model performed 0.75% better than the benchmark strategy. The difference was not considered significant. The figure below represents the tree structure produced by GPLAB of the best model produced. The nodes were arbitrarily labelled from A to T: (Only node Q gave a trivial '0' result with regards to the 400 test data entries. The result of the tree was a binary output.)

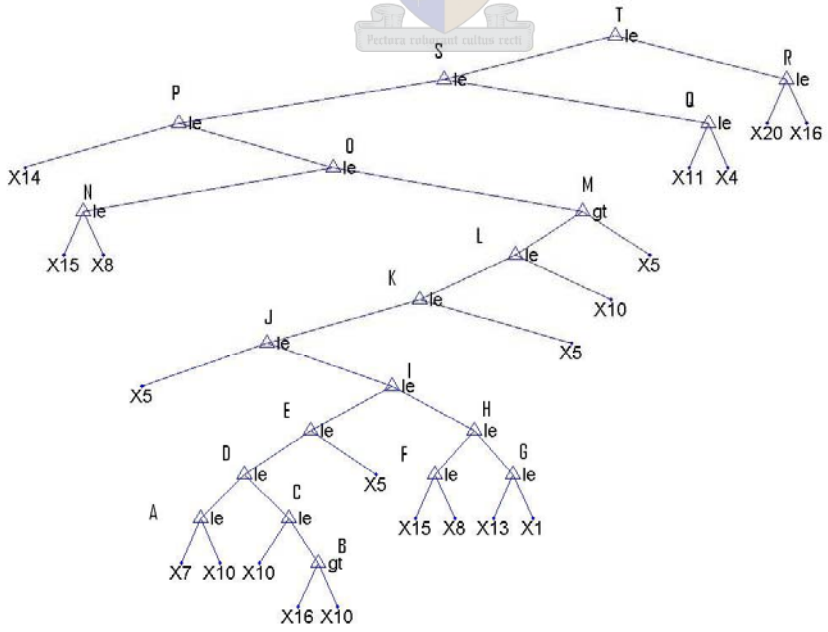


Figure 9.2: Arbitrarily model fragmentation for ease of simplification.

Another model was investigated further that was not the absolute best model found, but a model that had comparable accuracy and less complexity. This model had an error of 27.25%, consisted of 25 nodes distributed on 5 levels. This model was achieved by 200 generations and 300 individuals per generation. The achieved accuracy was just slightly worse than the benchmark. (It was 0.25% worse than benchmark.) The figure below represents the tree structure produced by GPLAB of this specific model. The nodes were again arbitrary labelled:

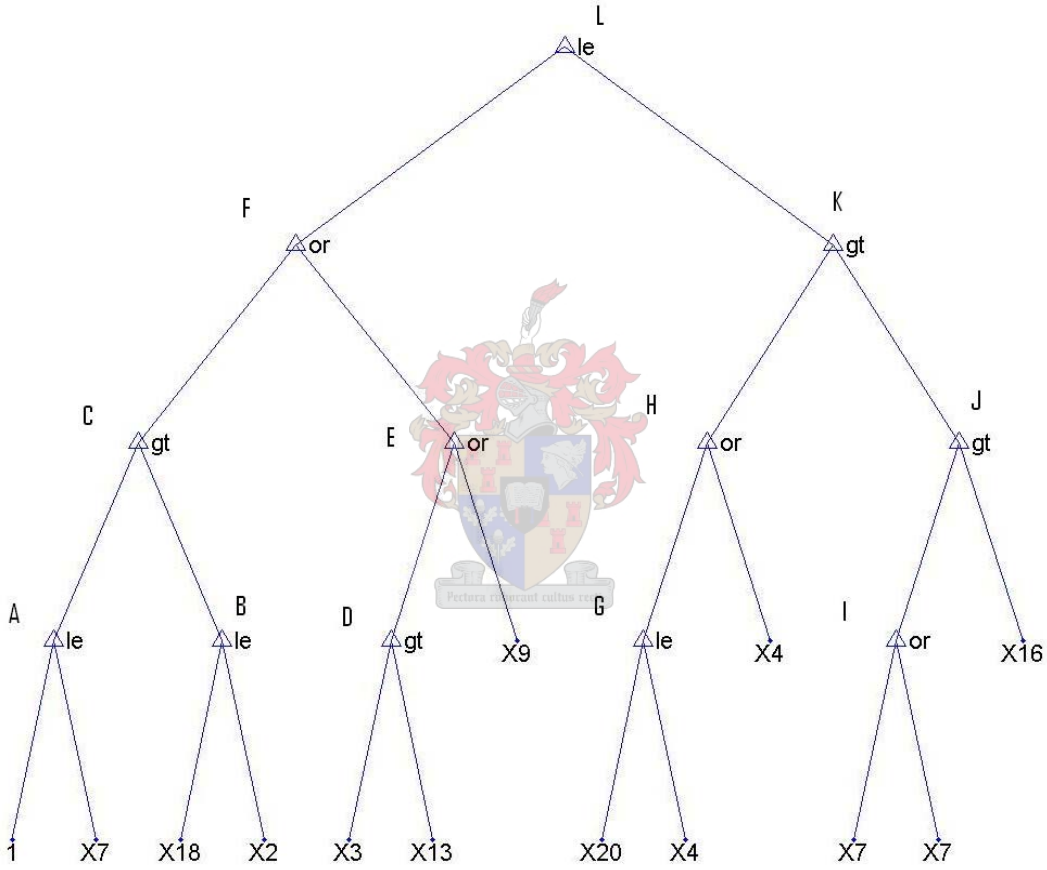


Figure 9.3: GPLAB of relative high accuracy and low complexity.

Evaluating the model on the training data indicated that many of the nodes were trivial nodes. Simplifying the trivial nodes produced:

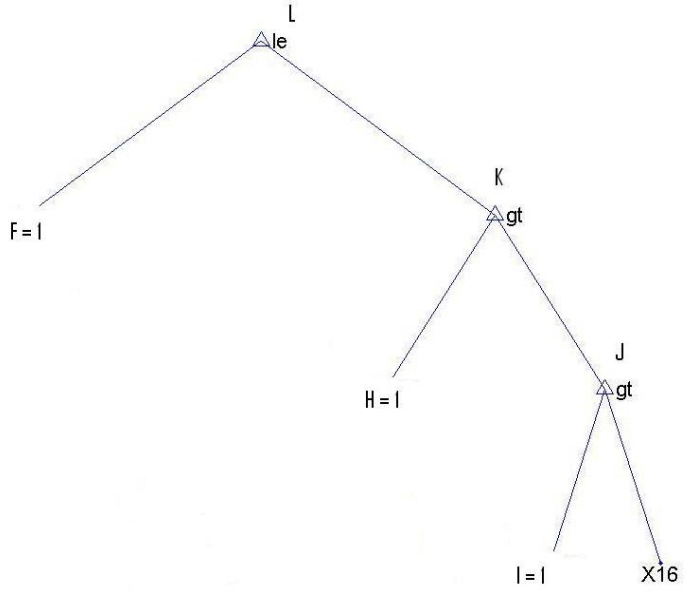


Figure 9.4: Simplified GPLAB model of relative high accuracy and low complexity.

This simplified model consists of 7 nodes. The prediction model can be expressed as:

Predict error if: 1 ≤ (1 if 1 > (1 if 1 > X16))

This can be simplified to the following simple rule:

IF $x_{16} > 1$, THEN 1

ELSE 0

It is clear that the prediction is only dependent on the TiN content represented as X16. This implies that 72.75% of the errors can be predicted by knowing only the TiN content. Also note that this model is far less complex that the 9 rules generated by the pruned classification tree (Fig. 9.1) with the same accuracy.

9.2 Regression by means of GP250

Numerous runs were executed. The generations were varied from 20 – 300 and the population size from 20 - 300 individuals. This code was much faster in producing models than what was experienced with GPLAB. More complex runs still took 6 - 10 hours.

Unfortunately this code produced only trivial models. In these models all test data points evaluated to '0' implying that no error was made during the production process. Although the number of correctly manufactured steel plates are much larger than the steel plates in which surface defects occurred, such a model is worthless. A typical model looked as follows:

Prediction = X16 / (X15 - $X19^2$) - X8

If this model was to be evaluated it would produce negative numeric values on all the test data. The values were closer to '0' than to '1' and thus interpreted as '0.' Clearly therefore, GP250 failed to produce a successful classification model. It could not evolve a model that could distinguish between many entries indicating correct production and much less entries that indicated an error in production. In essence GP250 experienced the same problem that was encountered using GPLAB and GP250 in the chess board classification problem done earlier.

9.3 Regression by means of Discipulus

Various runs consisting of approximately 500 individuals for 50 000 generations were executed. The runs took 3 – 4 hours to complete.

As stated in the previous case studies, Discipulus produces a binary classification model by rounding the produced regression model to the nearest integer assigned to a class. The absolute best model found had an error of 34% by misclassifying 136 of the unseen test entries. The accuracy was determined on the unseen test data by calculating the number of misclassified entries. The C/C++ code model does not lend itself to tree representation as does the GPLAB produced model. The model as a mathematical expression is also too long to be included in text.

9.4 Summary of investigation of Steel Plate surface defect classification

This data set consisted of 20 inputs represented by X1 to X20. The output or target value was a binary variable. A value of 1 indicated an error occurred during the production of the steel plates, while a value of 0 indicates successful production. The data set consisted of 3015 data points. The data were randomized and split in to a training set of 2615 entries and a test set of 400 entries.

As before, the benchmark strategy to compare the success of the GP models was the performance of a statistical regression tree. A tree structure was fitted to the data using MATLAB's treefit function. After tree pruning, the best model

consisted of 17 nodes and 7 terminal nodes. Again the 7 terminal nodes denoted 7 classification rules. The error was measured as the number of misclassified entries. (Lower values indicate less error.) The pruned tree misclassified 108 entries, which amounted to an error of 27%.

The GPLAB toolbox was used to investigate the evolution a possible classification model using GP. Taking model complexity and predicative accuracy into account the best GPLAB model had a comparable accuracy with an error of 27.3%. This model consisted of 25 nodes distributed on 5 levels. After simplification this model could be reduced to a single rule indicating that the prediction is only dependent on the TiN content represented by variable X16. This implies that 72.75% of the errors can be predicted by knowing only the TiN content. This constitutes a highly significant improvement in the interpretability of the data over any of the other models, including the classification tree.

The case study was attempted a third time by means of the GP code GP250.m. Unfortunately this code produced only trivial models. In these models all test data points evaluated to '0' implying that no error was made during the production process. Although the amount of correctly manufactured steel plates are much more than the steel plates in which error occurred such a model is worthless.

As final investigation the data was investigated using Discipulus. As stated in the previous case studies Discipulus produces a binary classification model by rounding the produced regression model to the nearest integer assigned to a class. The absolute best model found had an error of 34% by misclassifying 136 of the unseen test entries.

The table below acts as summary of the evolved models of the various strategies and the associated success of each model:

Table 9.2: Summary of model results.

Method:	Parameter used:	Accuracy:	Number of nodes:	Number of individual rules:
Statistical tree regression (pruned)	X15, X16, X17, X19, X20	73%	17	7
GPLAB model 2	X16	73%	7	1 classification rule
GP250	X8, X15, X16, X19	Non-convergence	9	1 empirical model
Discipulus	X1, X4, X5, X10, X15, X16	66%	n.a.	1 empirical model
	X17, X20			

It is interesting to note that GP can produce comparable accuracy to the benchmark method by using 50% to 150% of the nodes used by the pruned statistical regression tree. The only parameter used by all models was that of the TiN content represented as X16. Clearly the importance of this variable was identified by all the models.

Both Discipulus and GP250 produced classification models by rounding regression models to the nearest integer associated class. This strategy failed completely in GP250, but Discipulus achieved a reasonable model accuracy of 66%. The Discipulus model is very difficult to interpret, however. The lack of insight gained by the user using this model is a great drawback to using Discipulus. The GPLAB model 2 had produced a classification rule set consisting of a single rule. This rule was also human interpretable adding insight into the system being modelled. The GPLAB model also had a better accuracy than the GP250 and Discipulus models. The benchmark method had about the same model predictive accuracy as the GPLAB model, but the benchmark method needed more of the inputs to derive its classification model. Considering the understanding of the system added by the GPLAB model, the simplicity of the rule set and the number of inputs used the GPLAB outperformed the other models in this case study.

10. Discussion of Results and Conclusions

The most exciting phrase to hear in science, the one that heralds the most discoveries, is not "Eureka!", but "That's funny..."

- Isaac Asimov

10.1 Robustness of GP models

The stochastic nature of GP makes it difficult to achieve the same results twice. Once a model with a sufficiently good training data fit is derived it can be simplified and tested on unseen data. The evaluation of any model on unseen test data is the true test of the worth of the model. Deriving GP models took much more time than deriving a statistical regression tree. Typically a GP model took between 1 - 8 hours to evolve depending on the GP application used and size of the training set. Deriving a statistical regression tree took less than a minute.

It should be concluded that GP is not a regression method that produces the best empirical model possible for all possible applications. This was seen from case study 1 where GP failed to improve on the statistical tree regression model for this simplistic problem. The reason that case study 1 was deemed simplistic is that only two inputs were provided to determine the output. Furthermore both inputs were necessary to predict the output. This meant that the regression method used did not have to distinguish between relevant and unnecessary input data.

It would seem that the advantages of using GP are more evident in cases of higher complexity like case study 5. Case study 5 is deemed more complex than case study 1, since case study 5 had 9 inputs and case study 1 only had 2. During case study 5 GP made an intelligent choice of relevant inputs provided. Both successful GP strategies needed fewer inputs than the 9 provided to evolve a model. GPLAB only needed 4 inputs and the Discipulus evolved model needed 8 inputs. Producing a successful regression model that needs less input information than other models is a great advantage if comparable model accuracies are achieved. The successful GP applications achieved 84% accuracy as was the case using the statistical tree regression benchmark method. Using less input information implies that fewer sensors have to be placed in the process on the plant. This will decrease the

necessary capital used, decrease maintenance cost, decrease the amount of possible sensor failure, require less database capabilities and simplify diagnostic processes.

A complex empirical model may be sufficiently accurate to model a specific process, but such complex mathematical equations add little understanding to the underlying fundamentals of the process. This is true irrespective of the method used to derive the regression model. As a worst case producing a successful regression model by means of GP is at least another possible method in the arsenal of the process engineer.

The previous paragraph assumes that the complexity of a model is a trivial characteristic of the model. This was not found to be the case. In case study 5 it was decided that best GP model evolved was the model that required only 4 inputs. The tree of the GP model had a total number of 97 nodes compared to the 481 nodes of the statistical regression tree. The massive difference in the required number of nodes makes the GP model even more attractive. The GP evolved model was a single complex empirical mathematical equation and the regression tree model consisted of 177 simple rules. In section 1.1 it was stated that Androulakis (2004) argued that if two decision trees (classification or regression) offer the same accuracy the tree with the least amount of discreet rules is preferred by the machine learning community. In case study 5 the GP model was not a decision tree, but it consisted of a single rule compared to the 177 regression rules. From the viewpoint of the machine learning community the GP model should also be preferred above the tree regression model. Androulakis (2004) argues that the machine learning community's view that the idea that the accuracy of a model is associated with its complexity dates back at least as far as William of Ockam's razor: "one should not increase, beyond what is necessary, the number of entities required to explain anything." From the number of nodes required to form a model and from the number of discreet rules necessary to form a model the GP model outperforms statistical tree regression. If William of Ockam's razor dealt with the complexity of "the number of entities required" an argument could be made that the 177 simple classification rules produces a better model than a very complex single empirical model.

10.2 Classification case studies

The four classification case studies were: Case study 2: Chessboard Classification, Case study 3: Hydrology Classification example, Case study 4: Flotation Froth example and Case study 6: Steel Plate Error classification.

The evolutionary nature of GP makes it difficult to achieve the same results twice as was the case with the regression case studies. Once a model with a sufficiently good training data fit is derived it can be simplified and tested on unseen data. The true worth of a classification model can also only be evaluated by testing the model on unseen data. Deriving GP models took much more time than deriving a statistical regression tree. Typically a GP model took between 1 – 12 hours to evolve depending on the GP application used and size of the training set. Deriving a statistical regression tree took less than a minute.

The two methods that GP used to evolve its models during the classification case studies were discussed previously. This will be repeated here because of the importance of understanding both methods and the difference in the methods. The first method used to evolve a classification model was to round a regression model to the nearest class associated integer. The regression model used for rounding consisted purely of mathematical operators and functions. Both the GP250 and Discipulus applications used this method. The second way in which GP evolved classification models produced pure classification models. These models were considered pure since the prediction of the model was a class associated integer. No rounding took place during the evaluation of the model. These models consisted purely of Boolean operators and functions. GPLAB evolved classification models using this second method. These classification models resembled the benchmark statistical tree regression model in structure. A fundamental difference between the GPLAB classification model and the benchmark statistical regression model was that the benchmark method always compared a parameter to a numeric value whereas GPLAB compared a parameter to another parameter, function of parameter or a numeric value.

Producing a classification model by rounding the result of a regression model to the nearest class associated integer does not make intuitive sense. This method failed completely in the GP250 application. Still this method succeeded in producing classification models as applied by Discipulus. By employing this method Discipulus was the only GP strategy to produce a classification model in case study 1 better than a trivial result. In case studies 3 and 4 comparable accuracies were achieved with GPLAB and Discipulus. These two case studies were far more complex than case study 1. In case study 6 the GPLAB model outperformed Discipulus with regards to the achieved model accuracy. It would seem that the more complex case studies are more suited to the GPLAB classification model approach whereas the Discipulus method of classification model generation are more suited to simpler case studies. (Simpler case studies indicate less input variables and less available data.)

The simplest of all classification case studies investigated, case study 2, showed that GP has limited application especially in simpler case studies. It should be concluded that GP is not a classification method that produces the best model possible for all possible applications. In case study 2 GP failed to improve on the statistical tree regression model for this simplistic problem. The reason that case study 2 was deemed simplistic is that only two inputs were provided to determine the output. Furthermore both inputs were necessary to predict the output. This meant that the classification method used did not have to distinguish between relevant and unnecessary input data.

As with the regression case studies it would seem that the advantages of using GP are more evident in cases of higher complexity like case studies. The primary judge of a case study's complexity will be the amount of inputs provided to derive an appropriate output. During case studies 3, 4 and 6 GP made an intelligent choice of relevant inputs provided. All the pure classification models used the same amount or fewer inputs than the benchmark tree regression method. The pure classification models were derived by GPLAB. Producing a classification model by rounding the output of a regression model used the same amount of inputs or more as the benchmark statistical tree regression method. Classification models achieved by rounding regression models were evolved by Discipulus. As with the regression case studies using less input information implies that fewer sensors have to be placed in the process on the plant. This will decrease the necessary capital used, decrease maintenance cost, decrease the amount of possible sensor failure, require less database capabilities and simplify diagnostic processes. In this regard GPLAB produced pure classification

models outperformed all other strategies and modelling methods with regards to case studies 4 and 6.

Classification by means of a complex empirical model may be sufficiently accurate to model a specific process, but such complex mathematical equations add little understanding to the underlying fundamentals of the process. As a worst case producing a successful classification model by means of a regression model is at least another possible method in the arsenal of the process engineer. The same argument was stated during the pure regression case studies. Much more process specific insight is gained from a pure classification model. The rule set of these models could be evaluated to understand the underlying fundamentals of a process.

Models with accuracies differing about 5% are deemed to be of comparable accuracy. Various measures exist to compare the complexities of the various models. These measures of complexity can be the amount of total nodes required to produce the model, the number of inputs required by the model, the number of discreet rules forming the classification rule set and the complexity of each of these rules. Model accuracy eliminates the GP250 implementation as a successful classification model producing strategy. The implementation of rounding a GP produced regression model to produce a classification model failed in all the GP250 implementations, but succeeded in the following Discipulus implementations.

Successful pure classification models using Boolean operators and functions were developed for case studies 3, 4 and 6. GPLAB was used for this implementation. If it is again assumed that the model requiring the least amount of inputs is favoured, as argued in the regression case studies, then both the statistical tree regression model and the GPLAB model outperformed the Discipulus model in case study 3. In case studies 4 and 6 the GPLAB model required fewer inputs than the statistical tree regression model or the Discipulus model.

Again according to Androulakis (2004), if two decision trees (classification or regression) offer the same accuracy, the tree with the fewer discreet rules is preferred by the machine learning community. If this is the measure used to access complexity then the GPLAB model should be favoured above the statistical tree regression model in case studies 3, 4 and 6. On the other hand the Discipulus models for the mentioned three case studies comprised of a single empirical mathematical model. If Androulakis's argument can be extended so that a single predictive model is favoured above a discreet rule set then the Discipulus models would be preferred.

It should be remembered that the GPLAB rule set consists of rules comparing inputs with one another whereas the statistical regression rule set only compared a single input to a floating point number per rule. This makes the GPLAB rule set much more complex than the statistical regression tree rule set. The penalty of increased rule set complexity of GPLAB is a small price to pay if considered that an understanding of the interactions of the relevant inputs is evolved. The empirical Discipulus model fails to add intuitive insight into the process.

William of Ockam (Occam's razor) stated: "one should not increase, beyond what is necessary, the number of entities required to explain anything." The difficulty arises from the possible interpretations of the concept of "entities" in the razor. These "entities" could refer to the number of inputs required by a model. If this is true then the model requiring the least amount of inputs will be favoured. If "entities" denotes the number of rules per rule set then the rule set comprising of the least amount of rules will be favoured. Single empirical models would outperform rule sets, but these models add no insight to the underlying fundamentals of the system. GP evolved rule sets proofed to provide insight, but the rules were more complex than statistical regression tree models. Clearly interpreting William of Ockam's razor simply as the model with the least amount of individual rules is the best model is not a sufficient interpretation of the razor.

10.3 Conclusions and general remarks

In case study 1 it was found that Discipulus could not handle the almost 80 000 (281 × 281) entries in the training set. A smaller set was used for Discipulus to train on consisting of 141 × 141 (+- 20 000) training data entries. Nowhere in the Discipulus manual was any information found on the maximum possible size of training data. All other applications were MATLAB based and could handle huge data sets as matrices.

GP250 did not achieve good results at all. It should be remembered that due to the crude output of GP250 the complexity in number of nodes and depth of tree model was limited. This could have been the reason why GP250 failed to produce comparable results. The limitations were still not that strenuous if considered that the restrictions set on GP250 was sufficient for GPLAB to produce its models.

Discipulus was much faster in generating regression models than GPLAB. A typical Discipulus run took 1 – 4 hours. A further advantage of Discipulus is the robustness of the application. Nearly all runs produced accuracies of +- 10% difference. Discipulus was only designed for binary classification and

regression problems. The default threshold value for binary classification was simply set as 0.5. This value could be changed, but the manual strongly advised against it. Higher level classification was forced by handling the classification problem as a regression problem. The 0.5 threshold value was then manually forced.

All Discipulus runs essentially produced black box models. The reason for this was that typical Discipulus models consisted of 100 – 150 C/C++ lines of code. In most cases Discipulus failed to convert this code to a closed expression. In introductory examples Discipulus could convert C/C++ code to a closed expression of 5 – 10 pages. Since this could not be done in these case studies it is assumed that the evolved expression was at least longer than 5 – 10 pages. The human interpretability and explanatory nature argued as an advantage in GP evolved models were lost completely in the Discipulus applications.

GPLAB runs typically took between 4 - 32 hours. GPLAB was also not robust at all producing numerous useless or poor results before good runs were produced. As stated in table 3.1 GPLAB could use the 'or'-operator in producing models. The 'or'-operator produced disjunctive rules if used. Only GPLAB could produce disjunctive rules. In the end the disjunctive rules proofed to have little or no advantage over conjunctive rules. The reason for this was the way in which the 'or'-operator was employed. Table 10.1 illustrates the possible results achieved by the 'or'-operator:

Table 10.1: Possible results of 'or'-operator:

Left Branch	Right Branch	Result of 'or' statement
0	0	0
1	0	1
0	1	1
1	1	1

The disjunctive rule always evaluated to a binary variable of '1' if any of the associated sub branches had a value of '1.'

GPLAB on the other hand evolved discrete human interpretable classification rules. The evolved tree structure could be evaluated and simplified. It was the only GP strategy capable of this. Process specific insight could be deduced from these sets of rules.

10.4 Objectives revisited

The original hypothesis of this thesis is stated here again for convenience:

> *Genetic programming offers a competitive approach towards the automatic generation of process models from data.*

In this context, *competitive* was evaluated both in terms of predictive power, as well as the *interpretability* of the models.

- In the two simulated case studies (one regression and one classification problem), the classification/regression tree models outperformed the GP models both in terms of predictive power and interpretability. To be fair, the simulated classification study (Chess board) favoured the classification tree in that the variables were perfectly amenable to orthogonal binary splits (a chess board rotated by 45° would have posed a significantly more challenging problem to the classification tree model).

- In the four real word case studies, the GP models (not including GP250) and the classification/regression tree performed more or less equally well as far as predictive power was concerned.

- CART always produced a set of easily interpretable IF-THEN rules. In all the classification problems, the sets of rules produced by CART were small and thus easy to understand (9 rules for simulated Chess Board problem, 5 rules for the classification of zinc precipitates, 11 rules to classify the froth images, and 9 rules to classify the hot-rolled steel plates. In the two case studies involving regression models, large uninterpretable trees were generated, as could be expected.

- In contrast, the GP models generate more diverse tree-like structures that were more difficult to interpret. These structures did not represent sets of IF-THEN rules, but single model structures (equations) instead.

- Therefore, although interpretability is a subjective criterion, CART produced more interpretable structures in almost all the case studies. The exception was the case study related to the classification of steel plates (having surface defects or not). In this case, the Silva's GP model produced a singularly simple model, with the same predictive power as that of the classification tree.

Although GP models and their construction are generally more complex than CART models and do not appear to afford any particular advantages in

predictive power over CART, they can provide more concise, interpretable models than CART, albeit with a considerable amount of effort. On this basis, the hypothesis of the thesis could arguably be accepted.

11. Future Work

Many of life's failures are people who
did not realize how close they were to
success when they gave up.

- Thomas A. Edison

Many of the recommendations for future work have to do with the optimization of the coding of GP strategies. This was not considered to be of fundamental importance during this study. Future work could include:

- Determining the robustness of the GPLAB Toolbox. Most GPLAB runs produced poor models. Quantifying the robustness of GPLAB could provide a method to estimate how many runs should be executed to produce a model representative of the true capability of the GP algorithm. GP will only gain widespread use if sufficiently accurate models can be produced on a regular basis. Discipulus was much more robust than GPLAB and most runs produced accurate models.

- Some of the robustness shortcomings of GPLAB can surely be attributed to the fact that runs are far more limited in number of generations and individuals per run than compared to the stand alone Discipulus application. Too many individuals and generations lead to unacceptable computing times of more than 36 hours. After corresponding with Silva the speed of measuring the fitness value was greatly increased. This meant that runs could be executed in about a quarter of the time previously needed. Further speed improvements can be attempted by compiling the MATLAB files in a language like C/C++ or Java.

- Focussing on the capabilities of GP, not only in finding the shape of a model, but also in determining constant variables. No problems were experienced in generating model variables in this research although literature indicated that GP was known for poor performance in this regard (Greeff and Aldrich, 1997).

- Creating hierarchical rule based models. In such a study the primary model will fit the data as good as possible and a secondary model will aim to fit the residuals.

- A second strategy will be to investigate the produced accuracy of a team of GP produced models. The more advanced versions of the Discipulus software package have this as an inbuilt function.

- Evaluating the relative importance of each GP produced rule. In GPLAB's classification models it would be interesting to allocate a certain weight or importance to individual rules. In this way the importance of the input variables associated with the each rule can be studied.

- More advanced versions of Discipulus have a built in function to determine the importance of each input parameter. This will compliment the proposed investigation to be performed on GPLAB as mentioned in the previous point.

You can never ever leave without leaving a piece of you.

- Billy Corgan

(Artist: Smashing Pumpkins, Song: tonight, tonight, Album: Mellon Collie and the Infinite Sadness)

Addendums

Addendum A: MATLAB conversion of C/C++ code of case study 1

```
/*********************************************************
 *
 * Step 1: Import both mex.h and math.h as below.
 *
 *********************************************************/
#include "math.h"
#include "mex.h"


#define LOG2(x) ((float) (log(x)/log(2)))
#define LOG10(x) ((float) log10(x))
#define LOG_E(x) ((float) log(x))
#define PI 3.14159265359
#define E  2.718281828459

/*********************************************************
 *
 * Step 2: Search for and replace 'float' with 'double'.
 *
 *********************************************************/

double DiscipulusCFunction(double v[])
{
  double f[8];
  double tmp = 0;

  f[1]=f[2]=f[3]=f[4]=f[5]=f[6]=f[7]=0;
  f[0]=v[0];

  l0: f[0]-=v[1];
```

```
l1: f[0]*=f[0];
l2: f[0]-=1;
l3: f[0]+=f[0];
l4: f[0]+=f[0];
tmp=f[1]; f[1]=f[0]; f[0]=tmp;
l5: l6: f[0]+=f[1];
l7: f[0]-=0.5;
l8: f[0]+=v[1];
l9: f[0]=fabs(f[0]);
l10: f[1]-=f[0];
l11: f[0]-=f[0];
l12: f[0]-=0.5;
l13: f[0]*=v[1];
l14: f[1]-=f[0];
l15: f[0]=-f[0];
l16: f[0]+=v[0];
l17: f[0]*=f[0];
l18: f[0]-=v[1];
l19: f[0]*=v[1];
l20: l21: f[0]=fabs(f[0]);
l22: f[0]+=v[0];
l23: f[0]+=v[1];
l24: f[0]+=v[1];
l25: f[0]+=v[1];
l26: f[0]*=0.5;
l27: f[0]-=1;
l28: f[0]*=f[0];
f[0]=sqrt(f[0]);
l29: f[0]=sqrt(f[0]);
f[1]/=f[0];
l30: f[0]-=v[1];
l31: l32: f[0]*=0.5;
l33: f[0]+=v[1];
l34: f[0]-=1;
l35: f[0]*=v[0];
```

```
  l36: f[0]-=v[1];
  l37: l38: f[0]-=f[1];
  f[0]=fabs(f[0]);
  l39: f[0]*=0.5;
  l40: f[0]=sqrt(f[0]);
  l41: f[0]+=f[0];
  f[0]=sqrt(f[0]);
  l42: f[0]-=1;
  l43: f[0]-=0.5;
  l44: f[1]+=f[0];
  f[0]+=f[0];
  l45: tmp=f[1]; f[1]=f[0]; f[0]=tmp;
  l46: f[0]+=v[0];
  l47: f[0]=fabs(f[0]);
  f[0]=sqrt(f[0]);
  l48: f[0]-=f[1];
  l49: l50: l51: f[0]+=v[0];
  l52: f[0]*=f[0];
  f[0]*=f[0];
  l53: f[0]-=f[1];
  f[1]-=f[0];
  l54: f[0]=fabs(f[0]);
  f[1]+=f[0];
  l55: tmp=f[1]; f[1]=f[0]; f[0]=tmp;
  l56:
  l57:

  return f[0];
}


/*************************************************************
 *
 * Step 3: Add the function below to the file.
 *
```

```
 *********************************************************/
void mexFunction( int nlhs, mxArray *plhs[],
                  int nrhs, const mxArray*prhs[] )
{
  double *v, *result;
  int i;

  /* Check for proper number of arguments. */

  if(nrhs!=1) {

    mexErrMsgTxt("One input array required.");

  } else if(nlhs>1) {

    mexErrMsgTxt("Too many output arguments");
  }


   /* Create matrix for the return argument. */

  plhs[0] = mxCreateDoubleMatrix(1,1, mxREAL);


  /* Assign pointers to each input and output. */

  v = mxGetPr(prhs[0]);

  result = mxGetPr(plhs[0]);

  /* Call the DiscipulusCFunction subroutine. */

  *result = DiscipulusCFunction(v);
}
```

Addendum B: C/C++ code of case study 2

```
#define LOG2(x) ((float) (log(x)/log(2)))
#define LOG10(x) ((float) log10(x))
#define LOG_E(x) ((float) log(x))
#define PI 3.14159265359
#define E  2.718281828459

float DiscipulusCFunction(float v[])
{
  double f[8];
  double tmp = 0;

  f[1]=f[2]=f[3]=f[4]=f[5]=f[6]=f[7]=0;
  f[0]=v[0];

  l0: f[0]=sqrt(f[0]);
  f[0]=sqrt(f[0]);
  l1: f[1]-=f[0];
  f[0]/=f[0];
  l2: f[0]+=0.5;
  l3: f[0]+=f[0];
  f[0]/=f[1];
  l4: f[0]+=v[1];
  l5: f[0]/=v[0];
  l6: f[0]-=0.5;
  l7: tmp=f[1]; f[1]=f[0]; f[0]=tmp;
  l8: f[0]-=1;
  l9: f[0]*=0.5;
  l10: f[0]/=v[0];
  l11: f[0]*=f[1];
  tmp=f[1]; f[1]=f[0]; f[0]=tmp;
  l12: f[0]/=v[1];
  l13: f[0]=fabs(f[0]);
  l14: f[0]-=0.5;
  l15: f[0]+=f[0];
```

```
f[0]*=f[0];
l16: tmp=f[1]; f[1]=f[0]; f[0]=tmp;
l17: f[0]+=v[1];
l18: f[0]/=v[0];
l19: f[0]-=0.5;
l20: f[0]*=f[1];
f[1]-=f[0];
l21: f[0]*=0.5;
l22: f[0]=fabs(f[0]);
tmp=f[1]; f[1]=f[0]; f[0]=tmp;
l23: f[0]+=0.5;
l24: f[0]*=f[1];
l25: f[0]*=f[0];
f[0]*=f[0];
l26: f[0]/=v[1];
l27: f[0]/=f[1];
f[1]*=f[0];
l28: f[0]*=0.5;
l29: f[0]=sqrt(f[0]);
l30: f[0]-=f[1];
f[0]-=f[1];
l31: f[1]-=f[0];
tmp=f[1]; f[1]=f[0]; f[0]=tmp;
l32: f[0]-=1;
l33: f[0]*=f[0];
f[0]*=f[0];
l34: f[0]-=0.5;
l35: f[0]-=f[1];
f[0]+=f[0];
l36: f[0]-=0.5;
l37: f[0]-=f[1];
f[1]-=f[0];
l38: f[0]-=0.5;
l39: f[0]/=f[1];
f[0]+=f[0];
```

```
  l40: f[0]-=0.5;
  l41: f[0]+=f[0];
  f[0]=-f[0];
  l42: f[0]*=0.5;
  l43: f[0]/=0.5;
  l44: f[0]-=0.5;
  l45: f[0]+=f[0];
  f[0]=sqrt(f[0]);
  l46: f[0]*=0.5;
  l47: f[0]+=f[0];
  f[0]=sqrt(f[0]);
  l48: f[0]+=f[1];
  l49: f[0]+=f[0];
  f[0]=sqrt(f[0]);
  l50: f[0]*=0.5;
  l51: f[0]-=f[1];
  f[1]*=f[0];
  l52: f[0]+=f[1];
  l53: f[0]+=f[0];
  f[0]*=f[0];
  l54: f[0]*=f[0];
  f[0]*=f[0];
  l55: f[0]-=0.5;
  l56:
  l57:

  return f[0];
}
```

Addendum C: Typical mathematical expression of case study 2

```
#define LOG2(x) ((float) (log(x)/log(2)))
#define LOG10(x) ((float) log10(x))
#define LOG_E(x) ((float) log(x))
#define PI 3.14159265359
#define E  2.718281828459

float DiscipulusClosedExpression(float v[])
{
        return (((((((fabs((((fabs((((fabs((((((-(sqrt(v[0]) + 0.5)) + v[1]) / v[0]) -
0.5) / v[1])) - 0.5) + (fabs((((((-(sqrt(v[0]) + 0.5)) + v[1]) / v[0]) - 0.5) / v[1])) -
0.5)) * ((fabs((((((-(sqrt(v[0]) + 0.5)) + v[1]) / v[0]) - 0.5) / v[1])) - 0.5) +
(fabs((((((-(sqrt(v[0]) + 0.5)) + v[1]) / v[0]) - 0.5) / v[1])) - 0.5)))) - (((((((((0 -
sqrt(v[0])) + (0 - sqrt(v[0]))) + 1) / v[0]) * ((((-(sqrt(v[0]) + 0.5)) + v[1]) / v[0]) -
0.5)) + v[1]) / v[0]) - 0.5) * fabs((((fabs((((((-(sqrt(v[0]) + 0.5)) + v[1]) / v[0]) -
0.5) / v[1])) - 0.5) + (fabs((((((-(sqrt(v[0]) + 0.5)) + v[1]) / v[0]) - 0.5) / v[1])) -
0.5)) * ((fabs((((((-(sqrt(v[0]) + 0.5)) + v[1]) / v[0]) - 0.5) / v[1])) - 0.5) +
(fabs((((((-(sqrt(v[0]) + 0.5)) + v[1]) / v[0]) - 0.5) / v[1])) - 0.5)))))) + 0.5) *
((((((((((0 - sqrt(v[0])) + (0 - sqrt(v[0]))) + 1) / v[0]) * ((((-(sqrt(v[0]) + 0.5)) + v[1])
/ v[0]) - 0.5)) + v[1]) / v[0]) - 0.5) * fabs((((fabs((((((-(sqrt(v[0]) + 0.5)) + v[1]) /
v[0]) - 0.5) / v[1])) - 0.5) + (fabs((((((-(sqrt(v[0]) + 0.5)) + v[1]) / v[0]) - 0.5) /
v[1])) - 0.5)) * ((fabs((((((-(sqrt(v[0]) + 0.5)) + v[1]) / v[0]) - 0.5) / v[1])) - 0.5) +
(fabs((((((-(sqrt(v[0]) + 0.5)) + v[1]) / v[0]) - 0.5) / v[1])) - 0.5)))))) * 0.5))) / v[1]) *
(fabs((((fabs((((fabs((((((-(sqrt(v[0]) + 0.5)) + v[1]) / v[0]) - 0.5) / v[1])) - 0.5) +
(fabs((((((-(sqrt(v[0]) + 0.5)) + v[1]) / v[0]) - 0.5) / v[1])) - 0.5)) * ((fabs((((((-
(sqrt(v[0]) + 0.5)) + v[1]) / v[0]) - 0.5) / v[1])) - 0.5) + (fabs((((((-(sqrt(v[0]) +
0.5)) + v[1]) / v[0]) - 0.5) / v[1])) - 0.5)))) - ((((((((((0 - sqrt(v[0])) + (0 - sqrt(v[0])))
+ 1) / v[0]) * ((((-(sqrt(v[0]) + 0.5)) + v[1]) / v[0]) - 0.5)) + v[1]) / v[0]) - 0.5) *
fabs((((fabs((((((-(sqrt(v[0]) + 0.5)) + v[1]) / v[0]) - 0.5) / v[1])) - 0.5) +
(fabs((((((-(sqrt(v[0]) + 0.5)) + v[1]) / v[0]) - 0.5) / v[1])) - 0.5)) * ((fabs((((((-
(sqrt(v[0]) + 0.5)) + v[1]) / v[0]) - 0.5) / v[1])) - 0.5) + (fabs((((((-(sqrt(v[0]) +
0.5)) + v[1]) / v[0]) - 0.5) / v[1])) - 0.5)))))) + 0.5) * ((((((((((0 - sqrt(v[0])) + (0 -
sqrt(v[0]))) + 1) / v[0]) * ((((-(sqrt(v[0]) + 0.5)) + v[1]) / v[0]) - 0.5)) + v[1]) / v[0])
- 0.5) * fabs((((fabs((((((-(sqrt(v[0]) + 0.5)) + v[1]) / v[0]) - 0.5) / v[1])) - 0.5) +
(fabs((((((-(sqrt(v[0]) + 0.5)) + v[1]) / v[0]) - 0.5) / v[1])) - 0.5)) * ((fabs((((((-
(sqrt(v[0]) + 0.5)) + v[1]) / v[0]) - 0.5) / v[1])) - 0.5) + (fabs((((((-(sqrt(v[0]) +
0.5)) + v[1]) / v[0]) - 0.5) / v[1])) - 0.5)))))) * 0.5))) / v[1])) *
((fabs((((fabs((((fabs((((((-(sqrt(v[0]) + 0.5)) + v[1]) / v[0]) - 0.5) / v[1])) - 0.5) +
(fabs((((((-(sqrt(v[0]) + 0.5)) + v[1]) / v[0]) - 0.5) / v[1])) - 0.5)) * ((fabs((((((-
(sqrt(v[0]) + 0.5)) + v[1]) / v[0]) - 0.5) / v[1])) - 0.5) + (fabs((((((-(sqrt(v[0]) +
0.5)) + v[1]) / v[0]) - 0.5) / v[1])) - 0.5)))) - ((((((((((0 - sqrt(v[0])) + (0 - sqrt(v[0])))
+ 1) / v[0]) * ((((-(sqrt(v[0]) + 0.5)) + v[1]) / v[0]) - 0.5)) + v[1]) / v[0]) - 0.5) *
fabs((((fabs((((((-(sqrt(v[0]) + 0.5)) + v[1]) / v[0]) - 0.5) / v[1])) - 0.5) +
(fabs((((((-(sqrt(v[0]) + 0.5)) + v[1]) / v[0]) - 0.5) / v[1])) - 0.5)) * ((fabs((((((-
```

```
(sqrt(v[0]) + 0.5)) + v[1]) / v[0]) - 0.5) / v[1])) - 0.5) + (fabs((((((-(sqrt(v[0]) +
0.5)) + v[1]) / v[0]) - 0.5) / v[1])) - 0.5)))))) + 0.5) * ((((((((((0 - sqrt(v[0])) + (0 -
sqrt(v[0]))) + 1) / v[0]) * ((((-(sqrt(v[0]) + 0.5)) + v[1]) / v[0]) - 0.5)) + v[1]) / v[0])
- 0.5) * fabs((((fabs((((((-(sqrt(v[0]) + 0.5)) + v[1]) / v[0]) - 0.5) / v[1])) - 0.5) +
(fabs((((((-(sqrt(v[0]) + 0.5)) + v[1]) / v[0]) - 0.5) / v[1])) - 0.5)) * ((fabs((((((-
(sqrt(v[0]) + 0.5)) + v[1]) / v[0]) - 0.5) / v[1])) - 0.5) + (fabs((((((-(sqrt(v[0]) +
0.5)) + v[1]) / v[0]) - 0.5) / v[1])) - 0.5))))) * 0.5))) / v[1]) *
(fabs((((fabs((((fabs((((((-(sqrt(v[0]) + 0.5)) + v[1]) / v[0]) - 0.5) / v[1])) - 0.5) +
(fabs((((((-(sqrt(v[0]) + 0.5)) + v[1]) / v[0]) - 0.5) / v[1])) - 0.5)) * ((fabs((((((-
(sqrt(v[0]) + 0.5)) + v[1]) / v[0]) - 0.5) / v[1])) - 0.5) + (fabs((((((-(sqrt(v[0]) +
0.5)) + v[1]) / v[0]) - 0.5) / v[1])) - 0.5)))) - (((((((((0 - sqrt(v[0])) + (0 - sqrt(v[0])))
+ 1) / v[0]) * ((((-(sqrt(v[0]) + 0.5)) + v[1]) / v[0]) - 0.5)) + v[1]) / v[0]) - 0.5) *
fabs((((fabs((((((-(sqrt(v[0]) + 0.5)) + v[1]) / v[0]) - 0.5) / v[1])) - 0.5) +
(fabs((((((-(sqrt(v[0]) + 0.5)) + v[1]) / v[0]) - 0.5) / v[1])) - 0.5)) * ((fabs((((((-
(sqrt(v[0]) + 0.5)) + v[1]) / v[0]) - 0.5) / v[1])) - 0.5) + (fabs((((((-(sqrt(v[0]) +
0.5)) + v[1]) / v[0]) - 0.5) / v[1])) - 0.5)))))) + 0.5) * ((((((((((0 - sqrt(v[0])) + (0 -
sqrt(v[0]))) + 1) / v[0]) * ((((-(sqrt(v[0]) + 0.5)) + v[1]) / v[0]) - 0.5)) + v[1]) / v[0])
- 0.5) * fabs((((fabs((((((-(sqrt(v[0]) + 0.5)) + v[1]) / v[0]) - 0.5) / v[1])) - 0.5) +
(fabs((((((-(sqrt(v[0]) + 0.5)) + v[1]) / v[0]) - 0.5) / v[1])) - 0.5)) * ((fabs((((((-
(sqrt(v[0]) + 0.5)) + v[1]) / v[0]) - 0.5) / v[1])) - 0.5) + (fabs((((((-(sqrt(v[0]) +
0.5)) + v[1]) / v[0]) - 0.5) / v[1])) - 0.5))))) * 0.5))) / v[1]))) - 0.5) +
((((fabs((((fabs((((fabs((((((-(sqrt(v[0]) + 0.5)) + v[1]) / v[0]) - 0.5) / v[1])) - 0.5) +
(fabs((((((-(sqrt(v[0]) + 0.5)) + v[1]) / v[0]) - 0.5) / v[1])) - 0.5)) * ((fabs((((((-
(sqrt(v[0]) + 0.5)) + v[1]) / v[0]) - 0.5) / v[1])) - 0.5) + (fabs((((((-(sqrt(v[0]) +
0.5)) + v[1]) / v[0]) - 0.5) / v[1])) - 0.5)))) - (((((((((0 - sqrt(v[0])) + (0 - sqrt(v[0])))
+ 1) / v[0]) * ((((-(sqrt(v[0]) + 0.5)) + v[1]) / v[0]) - 0.5)) + v[1]) / v[0]) - 0.5) *
fabs((((fabs((((((-(sqrt(v[0]) + 0.5)) + v[1]) / v[0]) - 0.5) / v[1])) - 0.5) +
(fabs((((((-(sqrt(v[0]) + 0.5)) + v[1]) / v[0]) - 0.5) / v[1])) - 0.5)) * ((fabs((((((-
(sqrt(v[0]) + 0.5)) + v[1]) / v[0]) - 0.5) / v[1])) - 0.5) + (fabs((((((-(sqrt(v[0]) +
0.5)) + v[1]) / v[0]) - 0.5) / v[1])) - 0.5)))))) + 0.5) * ((((((((((0 - sqrt(v[0])) + (0 -
sqrt(v[0]))) + 1) / v[0]) * ((((-(sqrt(v[0]) + 0.5)) + v[1]) / v[0]) - 0.5)) + v[1]) / v[0])
- 0.5) * fabs((((fabs((((((-(sqrt(v[0]) + 0.5)) + v[1]) / v[0]) - 0.5) / v[1])) - 0.5) +
(fabs((((((-(sqrt(v[0]) + 0.5)) + v[1]) / v[0]) - 0.5) / v[1])) - 0.5)) * ((fabs((((((-
(sqrt(v[0]) + 0.5)) + v[1]) / v[0]) - 0.5) / v[1])) - 0.5) + (fabs((((((-(sqrt(v[0]) +
0.5)) + v[1]) / v[0]) - 0.5) / v[1])) - 0.5))))) * 0.5))) / v[1]) *
(fabs((((fabs((((fabs((((((-(sqrt(v[0]) + 0.5)) + v[1]) / v[0]) - 0.5) / v[1])) - 0.5) +
(fabs((((((-(sqrt(v[0]) + 0.5)) + v[1]) / v[0]) - 0.5) / v[1])) - 0.5)) * ((fabs((((((-
(sqrt(v[0]) + 0.5)) + v[1]) / v[0]) - 0.5) / v[1])) - 0.5) + (fabs((((((-(sqrt(v[0]) +
0.5)) + v[1]) / v[0]) - 0.5) / v[1])) - 0.5)))) - (((((((((0 - sqrt(v[0])) + (0 - sqrt(v[0])))
+ 1) / v[0]) * ((((-(sqrt(v[0]) + 0.5)) + v[1]) / v[0]) - 0.5)) + v[1]) / v[0]) - 0.5) *
fabs((((fabs((((((-(sqrt(v[0]) + 0.5)) + v[1]) / v[0]) - 0.5) / v[1])) - 0.5) +
(fabs((((((-(sqrt(v[0]) + 0.5)) + v[1]) / v[0]) - 0.5) / v[1])) - 0.5)) * ((fabs((((((-
(sqrt(v[0]) + 0.5)) + v[1]) / v[0]) - 0.5) / v[1])) - 0.5) + (fabs((((((-(sqrt(v[0]) +
0.5)) + v[1]) / v[0]) - 0.5) / v[1])) - 0.5)))))) + 0.5) * ((((((((((0 - sqrt(v[0])) + (0 -
sqrt(v[0]))) + 1) / v[0]) * ((((-(sqrt(v[0]) + 0.5)) + v[1]) / v[0]) - 0.5)) + v[1]) / v[0])
- 0.5) * fabs((((fabs((((((-(sqrt(v[0]) + 0.5)) + v[1]) / v[0]) - 0.5) / v[1])) - 0.5) +
(fabs((((((-(sqrt(v[0]) + 0.5)) + v[1]) / v[0]) - 0.5) / v[1])) - 0.5)) * ((fabs((((((-
(sqrt(v[0]) + 0.5)) + v[1]) / v[0]) - 0.5) / v[1])) - 0.5) + (fabs((((((-(sqrt(v[0]) +
0.5)) + v[1]) / v[0]) - 0.5) / v[1])) - 0.5))))) * 0.5))) / v[1])) *
((fabs((((fabs((((fabs((((((-(sqrt(v[0]) + 0.5)) + v[1]) / v[0]) - 0.5) / v[1])) - 0.5) +
```

(fabs((((((-(sqrt(v[0]) + 0.5)) + v[1]) / v[0]) - 0.5) / v[1])) - 0.5)) * ((fabs((((((-(sqrt(v[0]) + 0.5)) + v[1]) / v[0]) - 0.5) / v[1])) - 0.5) + (fabs((((((-(sqrt(v[0]) + 0.5)) + v[1]) / v[0]) - 0.5) / v[1])) - 0.5)))) - ((((((((((0 - sqrt(v[0])) + (0 - sqrt(v[0]))) + 1) / v[0]) * ((((-(sqrt(v[0]) + 0.5)) + v[1]) / v[0]) - 0.5)) + v[1]) / v[0]) - 0.5) * fabs((((fabs((((((-(sqrt(v[0]) + 0.5)) + v[1]) / v[0]) - 0.5) / v[1])) - 0.5) + (fabs((((((-(sqrt(v[0]) + 0.5)) + v[1]) / v[0]) - 0.5) / v[1])) - 0.5)) * ((fabs((((((-(sqrt(v[0]) + 0.5)) + v[1]) / v[0]) - 0.5) / v[1])) - 0.5) + (fabs((((((-(sqrt(v[0]) + 0.5)) + v[1]) / v[0]) - 0.5) / v[1])) - 0.5)))))) + 0.5) * ((((((((((0 - sqrt(v[0])) + (0 - sqrt(v[0]))) + 1) / v[0]) * ((((-(sqrt(v[0]) + 0.5)) + v[1]) / v[0]) - 0.5)) + v[1]) / v[0]) - 0.5) * fabs((((fabs((((((-(sqrt(v[0]) + 0.5)) + v[1]) / v[0]) - 0.5) / v[1])) - 0.5) + (fabs((((((-(sqrt(v[0]) + 0.5)) + v[1]) / v[0]) - 0.5) / v[1])) - 0.5)) * ((fabs((((((-(sqrt(v[0]) + 0.5)) + v[1]) / v[0]) - 0.5) / v[1])) - 0.5) + (fabs((((((-(sqrt(v[0]) + 0.5)) + v[1]) / v[0]) - 0.5) / v[1])) - 0.5))))) * 0.5))) / v[1]) * (fabs((((fabs((((fabs((((((-(sqrt(v[0]) + 0.5)) + v[1]) / v[0]) - 0.5) / v[1])) - 0.5) + (fabs((((((-(sqrt(v[0]) + 0.5)) + v[1]) / v[0]) - 0.5) / v[1])) - 0.5)) * ((fabs((((((-(sqrt(v[0]) + 0.5)) + v[1]) / v[0]) - 0.5) / v[1])) - 0.5) + (fabs((((((-(sqrt(v[0]) + 0.5)) + v[1]) / v[0]) - 0.5) / v[1])) - 0.5)))) - ((((((((((0 - sqrt(v[0])) + (0 - sqrt(v[0]))) + 1) / v[0]) * ((((-(sqrt(v[0]) + 0.5)) + v[1]) / v[0]) - 0.5)) + v[1]) / v[0]) - 0.5) * fabs((((fabs((((((-(sqrt(v[0]) + 0.5)) + v[1]) / v[0]) - 0.5) / v[1])) - 0.5) + (fabs((((((-(sqrt(v[0]) + 0.5)) + v[1]) / v[0]) - 0.5) / v[1])) - 0.5)) * ((fabs((((((-(sqrt(v[0]) + 0.5)) + v[1]) / v[0]) - 0.5) / v[1])) - 0.5) + (fabs((((((-(sqrt(v[0]) + 0.5)) + v[1]) / v[0]) - 0.5) / v[1])) - 0.5))))) + 0.5) * ((((((((((0 - sqrt(v[0])) + (0 - sqrt(v[0]))) + 1) / v[0]) * ((((-(sqrt(v[0]) + 0.5)) + v[1]) / v[0]) - 0.5)) + v[1]) / v[0]) - 0.5) * fabs((((fabs((((((-(sqrt(v[0]) + 0.5)) + v[1]) / v[0]) - 0.5) / v[1])) - 0.5) + (fabs((((((-(sqrt(v[0]) + 0.5)) + v[1]) / v[0]) - 0.5) / v[1])) - 0.5)) * ((fabs((((((-(sqrt(v[0]) + 0.5)) + v[1]) / v[0]) - 0.5) / v[1])) - 0.5) + (fabs((((((-(sqrt(v[0]) + 0.5)) + v[1]) / v[0]) - 0.5) / v[1])) - 0.5))))) * 0.5))) / v[1])))) - 0.5)) * (((((fabs((((fabs((((fabs((((((-(sqrt(v[0]) + 0.5)) + v[1]) / v[0]) - 0.5) / v[1])) - 0.5) + (fabs((((((-(sqrt(v[0]) + 0.5)) + v[1]) / v[0]) - 0.5) / v[1])) - 0.5)) * ((fabs((((((-(sqrt(v[0]) + 0.5)) + v[1]) / v[0]) - 0.5) / v[1])) - 0.5) + (fabs((((((-(sqrt(v[0]) + 0.5)) + v[1]) / v[0]) - 0.5) / v[1])) - 0.5)))) - ((((((((((0 - sqrt(v[0])) + (0 - sqrt(v[0]))) + 1) / v[0]) * ((((-(sqrt(v[0]) + 0.5)) + v[1]) / v[0]) - 0.5)) + v[1]) / v[0]) - 0.5) * fabs((((fabs((((((-(sqrt(v[0]) + 0.5)) + v[1]) / v[0]) - 0.5) / v[1])) - 0.5) + (fabs((((((-(sqrt(v[0]) + 0.5)) + v[1]) / v[0]) - 0.5) / v[1])) - 0.5)) * ((fabs((((((-(sqrt(v[0]) + 0.5)) + v[1]) / v[0]) - 0.5) / v[1])) - 0.5) + (fabs((((((-(sqrt(v[0]) + 0.5)) + v[1]) / v[0]) - 0.5) / v[1])) - 0.5))))) + 0.5) * ((((((((((0 - sqrt(v[0])) + (0 - sqrt(v[0]))) + 1) / v[0]) * ((((-(sqrt(v[0]) + 0.5)) + v[1]) / v[0]) - 0.5)) + v[1]) / v[0]) - 0.5) * fabs((((fabs((((((-(sqrt(v[0]) + 0.5)) + v[1]) / v[0]) - 0.5) / v[1])) - 0.5) + (fabs((((((-(sqrt(v[0]) + 0.5)) + v[1]) / v[0]) - 0.5) / v[1])) - 0.5)) * ((fabs((((((-(sqrt(v[0]) + 0.5)) + v[1]) / v[0]) - 0.5) / v[1])) - 0.5) + (fabs((((((-(sqrt(v[0]) + 0.5)) + v[1]) / v[0]) - 0.5) / v[1])) - 0.5))))) * 0.5))) / v[1]) * (fabs((((fabs((((fabs((((((-(sqrt(v[0]) + 0.5)) + v[1]) / v[0]) - 0.5) / v[1])) - 0.5) + (fabs((((((-(sqrt(v[0]) + 0.5)) + v[1]) / v[0]) - 0.5) / v[1])) - 0.5)) * ((fabs((((((-(sqrt(v[0]) + 0.5)) + v[1]) / v[0]) - 0.5) / v[1])) - 0.5) + (fabs((((((-(sqrt(v[0]) + 0.5)) + v[1]) / v[0]) - 0.5) / v[1])) - 0.5)))) - ((((((((((0 - sqrt(v[0])) + (0 - sqrt(v[0]))) + 1) / v[0]) * ((((-(sqrt(v[0]) + 0.5)) + v[1]) / v[0]) - 0.5)) + v[1]) / v[0]) - 0.5) * fabs((((fabs((((((-(sqrt(v[0]) + 0.5)) + v[1]) / v[0]) - 0.5) / v[1])) - 0.5) + (fabs((((((-(sqrt(v[0]) + 0.5)) + v[1]) / v[0]) - 0.5) / v[1])) - 0.5)) * ((fabs((((((-(sqrt(v[0]) + 0.5)) + v[1]) / v[0]) - 0.5) / v[1])) - 0.5) + (fabs((((((-(sqrt(v[0]) + 0.5)) + v[1]) / v[0]) - 0.5) / v[1])) - 0.5))))) + 0.5) * ((((((((((0 - sqrt(v[0])) + (0 -

sqrt(v[0]))) + 1) / v[0]) * ((((-(sqrt(v[0]) + 0.5)) + v[1]) / v[0]) - 0.5)) + v[1]) / v[0])
- 0.5) * fabs((((fabs((((((-(sqrt(v[0]) + 0.5)) + v[1]) / v[0]) - 0.5) / v[1])) - 0.5) +
(fabs((((((-(sqrt(v[0]) + 0.5)) + v[1]) / v[0]) - 0.5) / v[1])) - 0.5)) * ((fabs((((((-
(sqrt(v[0]) + 0.5)) + v[1]) / v[0]) - 0.5) / v[1])) - 0.5) + (fabs((((((-(sqrt(v[0]) +
0.5)) + v[1]) / v[0]) - 0.5) / v[1])) - 0.5))))) * 0.5))) / v[1])) *
((fabs((((fabs((((fabs((((((-(sqrt(v[0]) + 0.5)) + v[1]) / v[0]) - 0.5) / v[1])) - 0.5) +
(fabs((((((-(sqrt(v[0]) + 0.5)) + v[1]) / v[0]) - 0.5) / v[1])) - 0.5)) * ((fabs((((((-
(sqrt(v[0]) + 0.5)) + v[1]) / v[0]) - 0.5) / v[1])) - 0.5) + (fabs((((((-(sqrt(v[0]) +
0.5)) + v[1]) / v[0]) - 0.5) / v[1])) - 0.5)))) - ((((((((0 - sqrt(v[0])) + (0 - sqrt(v[0])))
+ 1) / v[0]) * ((((-(sqrt(v[0]) + 0.5)) + v[1]) / v[0]) - 0.5)) + v[1]) / v[0]) - 0.5) *
fabs((((fabs((((((-(sqrt(v[0]) + 0.5)) + v[1]) / v[0]) - 0.5) / v[1])) - 0.5) +
(fabs((((((-(sqrt(v[0]) + 0.5)) + v[1]) / v[0]) - 0.5) / v[1])) - 0.5)) * ((fabs((((((-
(sqrt(v[0]) + 0.5)) + v[1]) / v[0]) - 0.5) / v[1])) - 0.5) + (fabs((((((-(sqrt(v[0]) +
0.5)) + v[1]) / v[0]) - 0.5) / v[1])) - 0.5)))))) + 0.5) * (((((((((0 - sqrt(v[0])) + (0 -
sqrt(v[0]))) + 1) / v[0]) * ((((-(sqrt(v[0]) + 0.5)) + v[1]) / v[0]) - 0.5)) + v[1]) / v[0])
- 0.5) * fabs((((fabs((((((-(sqrt(v[0]) + 0.5)) + v[1]) / v[0]) - 0.5) / v[1])) - 0.5) +
(fabs((((((-(sqrt(v[0]) + 0.5)) + v[1]) / v[0]) - 0.5) / v[1])) - 0.5)) * ((fabs((((((-
(sqrt(v[0]) + 0.5)) + v[1]) / v[0]) - 0.5) / v[1])) - 0.5) + (fabs((((((-(sqrt(v[0]) +
0.5)) + v[1]) / v[0]) - 0.5) / v[1])) - 0.5))))) * 0.5))) / v[1]) *
(fabs((((fabs((((fabs((((((-(sqrt(v[0]) + 0.5)) + v[1]) / v[0]) - 0.5) / v[1])) - 0.5) +
(fabs((((((-(sqrt(v[0]) + 0.5)) + v[1]) / v[0]) - 0.5) / v[1])) - 0.5)) * ((fabs((((((-
(sqrt(v[0]) + 0.5)) + v[1]) / v[0]) - 0.5) / v[1])) - 0.5) + (fabs((((((-(sqrt(v[0]) +
0.5)) + v[1]) / v[0]) - 0.5) / v[1])) - 0.5)))) - ((((((((0 - sqrt(v[0])) + (0 - sqrt(v[0])))
+ 1) / v[0]) * ((((-(sqrt(v[0]) + 0.5)) + v[1]) / v[0]) - 0.5)) + v[1]) / v[0]) - 0.5) *
fabs((((fabs((((((-(sqrt(v[0]) + 0.5)) + v[1]) / v[0]) - 0.5) / v[1])) - 0.5) +
(fabs((((((-(sqrt(v[0]) + 0.5)) + v[1]) / v[0]) - 0.5) / v[1])) - 0.5)) * ((fabs((((((-
(sqrt(v[0]) + 0.5)) + v[1]) / v[0]) - 0.5) / v[1])) - 0.5) + (fabs((((((-(sqrt(v[0]) +
0.5)) + v[1]) / v[0]) - 0.5) / v[1])) - 0.5)))))) + 0.5) * (((((((((0 - sqrt(v[0])) + (0 -
sqrt(v[0]))) + 1) / v[0]) * ((((-(sqrt(v[0]) + 0.5)) + v[1]) / v[0]) - 0.5)) + v[1]) / v[0])
- 0.5) * fabs((((fabs((((((-(sqrt(v[0]) + 0.5)) + v[1]) / v[0]) - 0.5) / v[1])) - 0.5) +
(fabs((((((-(sqrt(v[0]) + 0.5)) + v[1]) / v[0]) - 0.5) / v[1])) - 0.5)) * ((fabs((((((-
(sqrt(v[0]) + 0.5)) + v[1]) / v[0]) - 0.5) / v[1])) - 0.5) + (fabs((((((-(sqrt(v[0]) +
0.5)) + v[1]) / v[0]) - 0.5) / v[1])) - 0.5))))) * 0.5))) / v[1]))) - 0.5) +
((((fabs((((fabs((((fabs((((((-(sqrt(v[0]) + 0.5)) + v[1]) / v[0]) - 0.5) / v[1])) - 0.5) +
(fabs((((((-(sqrt(v[0]) + 0.5)) + v[1]) / v[0]) - 0.5) / v[1])) - 0.5)) * ((fabs((((((-
(sqrt(v[0]) + 0.5)) + v[1]) / v[0]) - 0.5) / v[1])) - 0.5) + (fabs((((((-(sqrt(v[0]) +
0.5)) + v[1]) / v[0]) - 0.5) / v[1])) - 0.5)))) - ((((((((0 - sqrt(v[0])) + (0 - sqrt(v[0])))
+ 1) / v[0]) * ((((-(sqrt(v[0]) + 0.5)) + v[1]) / v[0]) - 0.5)) + v[1]) / v[0]) - 0.5) *
fabs((((fabs((((((-(sqrt(v[0]) + 0.5)) + v[1]) / v[0]) - 0.5) / v[1])) - 0.5) +
(fabs((((((-(sqrt(v[0]) + 0.5)) + v[1]) / v[0]) - 0.5) / v[1])) - 0.5)) * ((fabs((((((-
(sqrt(v[0]) + 0.5)) + v[1]) / v[0]) - 0.5) / v[1])) - 0.5) + (fabs((((((-(sqrt(v[0]) +
0.5)) + v[1]) / v[0]) - 0.5) / v[1])) - 0.5)))))) + 0.5) * (((((((((0 - sqrt(v[0])) + (0 -
sqrt(v[0]))) + 1) / v[0]) * ((((-(sqrt(v[0]) + 0.5)) + v[1]) / v[0]) - 0.5)) + v[1]) / v[0])
- 0.5) * fabs((((fabs((((((-(sqrt(v[0]) + 0.5)) + v[1]) / v[0]) - 0.5) / v[1])) - 0.5) +
(fabs((((((-(sqrt(v[0]) + 0.5)) + v[1]) / v[0]) - 0.5) / v[1])) - 0.5)) * ((fabs((((((-
(sqrt(v[0]) + 0.5)) + v[1]) / v[0]) - 0.5) / v[1])) - 0.5) + (fabs((((((-(sqrt(v[0]) +
0.5)) + v[1]) / v[0]) - 0.5) / v[1])) - 0.5))))) * 0.5))) / v[1]) *
(fabs((((fabs((((fabs((((((-(sqrt(v[0]) + 0.5)) + v[1]) / v[0]) - 0.5) / v[1])) - 0.5) +
(fabs((((((-(sqrt(v[0]) + 0.5)) + v[1]) / v[0]) - 0.5) / v[1])) - 0.5)) * ((fabs((((((-
(sqrt(v[0]) + 0.5)) + v[1]) / v[0]) - 0.5) / v[1])) - 0.5) + (fabs((((((-(sqrt(v[0]) +

```
0.5)) + v[1]) / v[0]) - 0.5) / v[1])) - 0.5)))) - ((((((((((0 - sqrt(v[0])) + (0 - sqrt(v[0]))) + 1) / v[0]) * ((((-(sqrt(v[0]) + 0.5)) + v[1]) / v[0]) - 0.5)) + v[1]) / v[0]) - 0.5) * fabs((((fabs((((((-(sqrt(v[0]) + 0.5)) + v[1]) / v[0]) - 0.5) / v[1])) - 0.5) + (fabs((((((-(sqrt(v[0]) + 0.5)) + v[1]) / v[0]) - 0.5) / v[1])) - 0.5)) * ((fabs((((((-(sqrt(v[0]) + 0.5)) + v[1]) / v[0]) - 0.5) / v[1])) - 0.5) + (fabs((((((-(sqrt(v[0]) + 0.5)) + v[1]) / v[0]) - 0.5) / v[1])) - 0.5)))))) + 0.5) * ((((((((((0 - sqrt(v[0])) + (0 - sqrt(v[0]))) + 1) / v[0]) * ((((-(sqrt(v[0]) + 0.5)) + v[1]) / v[0]) - 0.5)) + v[1]) / v[0]) - 0.5) * fabs((((fabs((((((-(sqrt(v[0]) + 0.5)) + v[1]) / v[0]) - 0.5) / v[1])) - 0.5) + (fabs((((((-(sqrt(v[0]) + 0.5)) + v[1]) / v[0]) - 0.5) / v[1])) - 0.5)) * ((fabs((((((-(sqrt(v[0]) + 0.5)) + v[1]) / v[0]) - 0.5) / v[1])) - 0.5) + (fabs((((((-(sqrt(v[0]) + 0.5)) + v[1]) / v[0]) - 0.5) / v[1])) - 0.5)))))) * 0.5))) / v[1])) * ((fabs((((fabs((((fabs((((((-(sqrt(v[0]) + 0.5)) + v[1]) / v[0]) - 0.5) / v[1])) - 0.5) + (fabs((((((-(sqrt(v[0]) + 0.5)) + v[1]) / v[0]) - 0.5) / v[1])) - 0.5)) * ((fabs((((((-(sqrt(v[0]) + 0.5)) + v[1]) / v[0]) - 0.5) / v[1])) - 0.5) + (fabs((((((-(sqrt(v[0]) + 0.5)) + v[1]) / v[0]) - 0.5) / v[1])) - 0.5)))) - ((((((((((0 - sqrt(v[0])) + (0 - sqrt(v[0]))) + 1) / v[0]) * ((((-(sqrt(v[0]) + 0.5)) + v[1]) / v[0]) - 0.5)) + v[1]) / v[0]) - 0.5) * fabs((((fabs((((((-(sqrt(v[0]) + 0.5)) + v[1]) / v[0]) - 0.5) / v[1])) - 0.5) + (fabs((((((-(sqrt(v[0]) + 0.5)) + v[1]) / v[0]) - 0.5) / v[1])) - 0.5)) * ((fabs((((((-(sqrt(v[0]) + 0.5)) + v[1]) / v[0]) - 0.5) / v[1])) - 0.5) + (fabs((((((-(sqrt(v[0]) + 0.5)) + v[1]) / v[0]) - 0.5) / v[1])) - 0.5)))))) + 0.5) * ((((((((((0 - sqrt(v[0])) + (0 - sqrt(v[0]))) + 1) / v[0]) * ((((-(sqrt(v[0]) + 0.5)) + v[1]) / v[0]) - 0.5)) + v[1]) / v[0]) - 0.5) * fabs((((fabs((((((-(sqrt(v[0]) + 0.5)) + v[1]) / v[0]) - 0.5) / v[1])) - 0.5) + (fabs((((((-(sqrt(v[0]) + 0.5)) + v[1]) / v[0]) - 0.5) / v[1])) - 0.5)) * ((fabs((((((-(sqrt(v[0]) + 0.5)) + v[1]) / v[0]) - 0.5) / v[1])) - 0.5) + (fabs((((((-(sqrt(v[0]) + 0.5)) + v[1]) / v[0]) - 0.5) / v[1])) - 0.5)))) * 0.5))) / v[1]) * (fabs((((fabs((((fabs((((((-(sqrt(v[0]) + 0.5)) + v[1]) / v[0]) - 0.5) / v[1])) - 0.5) + (fabs((((((-(sqrt(v[0]) + 0.5)) + v[1]) / v[0]) - 0.5) / v[1])) - 0.5)) * ((fabs((((((-(sqrt(v[0]) + 0.5)) + v[1]) / v[0]) - 0.5) / v[1])) - 0.5) + (fabs((((((-(sqrt(v[0]) + 0.5)) + v[1]) / v[0]) - 0.5) / v[1])) - 0.5)))) - ((((((((((0 - sqrt(v[0])) + (0 - sqrt(v[0]))) + 1) / v[0]) * ((((-(sqrt(v[0]) + 0.5)) + v[1]) / v[0]) - 0.5)) + v[1]) / v[0]) - 0.5) * fabs((((fabs((((((-(sqrt(v[0]) + 0.5)) + v[1]) / v[0]) - 0.5) / v[1])) - 0.5) + (fabs((((((-(sqrt(v[0]) + 0.5)) + v[1]) / v[0]) - 0.5) / v[1])) - 0.5)) * ((fabs((((((-(sqrt(v[0]) + 0.5)) + v[1]) / v[0]) - 0.5) / v[1])) - 0.5) + (fabs((((((-(sqrt(v[0]) + 0.5)) + v[1]) / v[0]) - 0.5) / v[1])) - 0.5)))))) + 0.5) * ((((((((((0 - sqrt(v[0])) + (0 - sqrt(v[0]))) + 1) / v[0]) * ((((-(sqrt(v[0]) + 0.5)) + v[1]) / v[0]) - 0.5)) + v[1]) / v[0]) - 0.5) * fabs((((fabs((((((-(sqrt(v[0]) + 0.5)) + v[1]) / v[0]) - 0.5) / v[1])) - 0.5) + (fabs((((((-(sqrt(v[0]) + 0.5)) + v[1]) / v[0]) - 0.5) / v[1])) - 0.5)) * ((fabs((((((-(sqrt(v[0]) + 0.5)) + v[1]) / v[0]) - 0.5) / v[1])) - 0.5) + (fabs((((((-(sqrt(v[0]) + 0.5)) + v[1]) / v[0]) - 0.5) / v[1])) - 0.5)))) * 0.5))) / v[1]))) - 0.5))) - 0.5);

}
```

Addendum D: C/C++ code of case study 3

```
#define LOG2(x) ((float) (log(x)/log(2)))
#define LOG10(x) ((float) log10(x))
#define LOG_E(x) ((float) log(x))
#define PI 3.14159265359
#define E  2.718281828459

float DiscipulusCFunction(float v[])
{
  double f[8];
  double tmp = 0;

  f[1]=f[2]=f[3]=f[4]=f[5]=f[6]=f[7]=0;
  f[0]=v[0];

  l0: f[0]+=v[3];
  l1: f[0]/=v[0];
  l2: f[1]+=f[0];
  tmp=f[1]; f[1]=f[0]; f[0]=tmp;
  l3: f[0]*=v[1];
  l4: f[0]*=f[0];
  f[0]+=f[1];
  l5: f[0]/=f[1];
  l6: f[0]-=v[1];
  l7: f[0]+=0;
  l8: f[0]*=v[3];
  l9: f[1]+=f[0];
  f[1]/=f[0];
  l10: f[0]/=v[0];
  l11: f[0]*=f[0];
  f[0]/=f[1];
  l12: tmp=f[1]; f[1]=f[0]; f[0]=tmp;
  f[0]+=f[1];
  l13: f[0]/=v[1];
  l14: f[0]*=f[0];
```

```
f[1]*=f[0];
l15: f[0]*=f[0];
f[0]=-f[0];
l16: f[0]+=v[2];
l17: f[0]+=1;
l18: f[1]-=f[0];
tmp=f[1]; f[1]=f[0]; f[0]=tmp;
l19: l20: f[0]*=v[1];
l21: tmp=f[1]; f[1]=f[0]; f[0]=tmp;
f[0]*=f[0];
l22: f[1]+=f[0];
l23: f[0]+=f[1];
l24: f[1]*=f[0];
l25: f[0]+=1;
l26: f[0]/=v[3];
l27: f[1]-=f[0];
f[1]/=f[0];
l28: f[1]/=f[0];
f[0]/=f[0];
l29: tmp=f[1]; f[1]=f[0]; f[0]=tmp;
f[0]=fabs(f[0]);
l30: f[0]+=f[1];
f[0]*=f[0];
l31: f[0]/=v[0];
l32: f[0]*=v[1];
l33: f[0]-=f[1];
f[0]*=f[0];
l34: f[1]+=f[0];
f[1]/=f[0];
l35: f[1]/=f[0];
f[0]/=f[0];
l36: f[0]+=v[3];
l37: f[0]*=f[0];
l38: f[1]+=f[0];
f[1]/=f[0];
```

```
  l39: f[0]=-f[0];
  l40: f[0]+=0.5;
  l41: f[0]+=0.5;
  l42: f[0]=fabs(f[0]);
  l43: f[0]-=v[2];
  l44: f[0]*=f[1];
  f[0]=fabs(f[0]);
  l45: f[0]/=v[2];
  l46: f[0]*=f[1];
  f[0]-=f[1];
  l47: f[0]+=v[1];
  l48: f[0]+=v[1];
  l49: f[0]/=v[0];
  l50: f[0]-=v[2];
  l51: f[0]/=v[2];
  l52: f[0]/=v[0];
  l53: f[0]*=v[1];
  l54: f[1]-=f[0];
  tmp=f[1]; f[1]=f[0]; f[0]=tmp;
  l55:
  l56:

  return f[0];
}
```

Addendum E: C/C++ code of case study 4

```
#define LOG2(x) ((float) (log(x)/log(2)))
#define LOG10(x) ((float) log10(x))
#define LOG_E(x) ((float) log(x))
#define PI 3.14159265359
#define E  2.718281828459

float DiscipulusCFunction(float v[])
{
  double f[8];
  double tmp = 0;

  f[1]=f[2]=f[3]=f[4]=f[5]=f[6]=f[7]=0;
  f[0]=v[0];

  l0: f[0]=fabs(f[0]);
  f[0]+=f[0];
  l1: f[0]=sqrt(f[0]);
  l2: f[0]=sqrt(f[0]);
  f[1]+=f[0];
  l3: f[0]+=f[0];
  l4: f[0]+=v[2];
  l5: f[0]+=v[4];
  l6: f[0]+=1;
  l7: f[0]+=v[2];
  l8: f[0]*=f[0];
  f[0]=sqrt(f[0]);
  l9: f[0]+=f[1];
  l10: f[0]-=v[0];
  l11: f[0]=sqrt(f[0]);
  l12: tmp=f[1]; f[1]=f[0]; f[0]=tmp;
  f[0]=fabs(f[0]);
  l13: f[0]=sqrt(f[0]);
  l14: f[0]-=1;
```

```
l15: f[0]*=v[2];
l16: f[0]*=f[0];
l17: f[0]+=f[0];
f[0]=sqrt(f[0]);
l18: f[0]+=v[3];
l19: tmp=f[1]; f[1]=f[0]; f[0]=tmp;
tmp=f[1]; f[1]=f[0]; f[0]=tmp;
l20: f[0]+=v[4];
l21: f[0]*=f[0];
f[1]+=f[0];
l22: l23: f[0]=sqrt(f[0]);
l24: f[0]/=f[1];
l25: f[0]/=f[1];
l26: f[0]*=f[1];
l27: f[0]*=0.5;
l28: l29: tmp=f[1]; f[1]=f[0]; f[0]=tmp;
l30: f[0]+=0.5;
l31: f[1]+=f[0];
f[0]=sqrt(f[0]);
l32: f[0]+=v[0];
l33: f[0]-=v[2];
l34: f[0]*=v[0];
l35: f[0]*=0.5;
l36: f[0]*=0.5;
l37: f[0]*=f[1];
l38: f[0]-=v[3];
l39: f[0]*=v[2];
l40: f[0]=fabs(f[0]);
l41: f[0]=sqrt(f[0]);
l42: f[0]/=f[1];
l43: f[0]=sqrt(f[0]);
l44: f[0]-=f[1];
l45: l46: l47: f[1]-=f[0];
l48: l49: f[1]*=f[0];
l50: f[0]=fabs(f[0]);
```

```
  l51: f[1]*=f[0];
  tmp=f[0]; f[0]=f[0]; f[0]=tmp;
  l52: f[0]=fabs(f[0]);
  l53: f[0]=fabs(f[0]);
  l54:
  l55:

  return f[0];
}
```

Addendum F: C/C++ code of case study 5

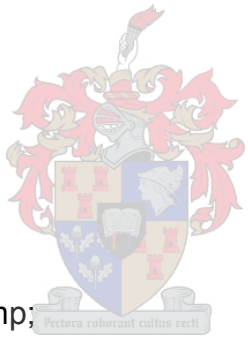
```
#define LOG2(x) ((float) (log(x)/log(2)))
#define LOG10(x) ((float) log10(x))
#define LOG_E(x) ((float) log(x))
#define PI 3.14159265359
#define E  2.718281828459

float DiscipulusCFunction(float v[])
{
  double f[8];
  double tmp = 0;

  f[1]=f[2]=f[3]=f[4]=f[5]=f[6]=f[7]=0;
  f[0]=v[0];

  l0: f[1]+=f[0];
  l1: f[0]+=f[0];
  l2: f[0]*=f[0];
  l3: tmp=f[0]; f[0]=f[0]; f[0]=tmp;
  f[0]=sqrt(f[0]);
  l4: f[0]*=f[1];
  l5: f[0]+=f[1];
  f[0]=sqrt(f[0]);
  l6: tmp=f[1]; f[1]=f[0]; f[0]=tmp;
  f[0]-=f[1];
  l7: f[0]=fabs(f[0]);
  l8: f[0]*=f[1];
  f[0]/=f[0];
  l9: f[0]-=f[0];
  f[0]+=f[1];
  l10: f[0]+=f[0];
  f[1]*=f[0];
  l11: f[0]/=f[0];
  f[0]*=f[0];
```

```
l12: f[0]*=f[0];
f[0]=-f[0];
l13: f[0]*=v[13];
l14: f[0]*=0;
l15: tmp=f[0]; f[0]=f[0]; f[0]=tmp;
f[0]=sqrt(f[0]);
l16: f[0]+=v[16];
l17: f[0]-=f[1];
l18: f[0]-=0.5;
l19: f[1]-=f[0];
f[0]+=f[0];
l20: f[0]+=f[0];
l21: tmp=f[1]; f[1]=f[0]; f[0]=tmp;
l22: f[0]-=f[1];
l23: f[0]/=v[3];
l24: f[1]-=f[0];
l25: f[0]*=v[5];
l26: f[0]/=f[0];
l27: f[1]-=f[0];
l28: f[0]+=v[14];
l29: tmp=f[0]; f[0]=f[0]; f[0]=tmp;
l30: f[0]*=f[0];
l31: f[0]-=v[0];
l32: f[0]-=v[11];
l33: f[0]/=v[11];
l34: f[0]/=v[0];
l35: f[0]-=v[8];
l36: f[0]-=0.5;
l37: f[0]-=v[16];
l38: f[0]*=f[0];
l39: tmp=f[1]; f[1]=f[0]; f[0]=tmp;
f[0]*=f[0];
l40: f[0]-=v[15];
l41: tmp=f[1]; f[1]=f[0]; f[0]=tmp;
l42: f[1]/=f[0];
```



```
  l43: f[0]-=v[18];
  l44: f[0]*=v[9];
  l45: f[0]*=f[1];
  f[0]-=f[1];
  l46: tmp=f[0]; f[0]=f[0]; f[0]=tmp;
  f[0]*=f[0];
  l47: l48: f[0]/=f[0];
  l49: l50: f[0]*=f[0];
  l51: f[0]-=f[1];
  l52: f[0]=fabs(f[0]);
  l53: f[0]-=0.5;
  l54:
  l55:

  return f[0];
}
```

Addendum G: C/C++ code of case study 6

```
#define LOG2(x) ((float) (log(x)/log(2)))
#define LOG10(x) ((float) log10(x))
#define LOG_E(x) ((float) log(x))
#define PI 3.14159265359
#define E  2.718281828459

float DiscipulusCFunction(float v[])
{
  double f[8];
  double tmp = 0;

  f[1]=f[2]=f[3]=f[4]=f[5]=f[6]=f[7]=0;
  f[0]=v[0];

  l0: f[0]/=f[0];
  l1: tmp=f[1]; f[1]=f[0]; f[0]=tmp;
  f[0]-=f[1];
  l2: f[0]*=f[0];
  l3: f[0]+=f[1];
  l4: f[0]=sqrt(f[0]);
  l5: f[0]-=v[16];
  l6: f[0]/=f[1];
  f[0]+=f[0];
  l7: tmp=f[1]; f[1]=f[0]; f[0]=tmp;
  f[0]*=f[0];
  l8: f[0]=-f[0];
  l9: f[0]-=f[0];
  f[0]+=f[1];
  l10: tmp=f[1]; f[1]=f[0]; f[0]=tmp;
  l11: f[0]+=f[1];
  tmp=f[0]; f[0]=f[0]; f[0]=tmp;
  l12: f[0]-=f[1];
  f[0]+=f[0];
```

```
l13: f[1]*=f[0];
f[1]*=f[0];
l14: f[0]=fabs(f[0]);
l15: f[0]+=v[15];
l16: f[0]=-f[0];
tmp=f[1]; f[1]=f[0]; f[0]=tmp;
l17: f[0]=fabs(f[0]);
f[0]=fabs(f[0]);
l18: f[0]/=0.5;
l19: f[0]+=v[19];
l20: f[0]+=v[19];
l21: tmp=f[0]; f[0]=f[0]; f[0]=tmp;
l22: f[1]+=f[0];
l23: f[1]+=f[0];
l24: f[0]/=f[0];
tmp=f[1]; f[1]=f[0]; f[0]=tmp;
l25: f[0]-=v[14];
l26: f[0]*=v[5];
l27: f[0]*=v[13];
l28: f[0]=-f[0];
tmp=f[1]; f[1]=f[0]; f[0]=tmp;
l29: tmp=f[0]; f[0]=f[0]; f[0]=tmp;
l30: f[0]=fabs(f[0]);
l31: tmp=f[0]; f[0]=f[0]; f[0]=tmp;
f[0]/=f[0];
l32: f[0]/=f[1];
l33: f[0]+=f[0];
f[0]*=f[0];
l34: f[0]=sqrt(f[0]);
f[0]=fabs(f[0]);
l35: f[0]+=0;
l36: f[0]+=f[1];
f[0]=fabs(f[0]);
l37: f[0]-=f[0];
l38: f[0]=fabs(f[0]);
```

```
  f[0]=-f[0];
  l39: f[0]=fabs(f[0]);
  f[0]*=f[1];
  l40: f[0]*=f[0];
  f[0]-=f[0];
  l41: f[1]+=f[0];
  f[0]-=f[0];
  l42: tmp=f[1]; f[1]=f[0]; f[0]=tmp;
  l43: f[0]+=0.5;
  l44:
  l45:

  return f[0];
}
```

References

- Act No. 36 of 1998: National Water Act. (All amendments implicitly assumed.)
- Act No. 39 of 2004: National Environment Management: Air Quality Act. (All amendments implicitly assumed.)
- Act No. 73 of 1989: Environment Conservation Act. (All amendments implicitly assumed.)
- Act No. 85 of 1993: Occupational Health and Safety Act. (All amendments implicitly assumed.)
- Aldrich, C., Gardner, S. and Le Roux, N.J. (2004) Monitoring of metallurgical process plants by use of biplots. *AIChE Journal*, 50(9), 2167-2186.
- Androulakis, I.P. (2004) Selecting maximally informative genes. *Computers and Chemical Engineering*, 29 (2005), 535–546.
- Cai, W., Pacheco-Vega, A., Sen, M. and Yang, K.T. (2006) Heat transfer correlations by symbolic regression. *International Journal of Heat and Mass Transfer*, 49, 4352-4359.
- Hongqing Cao, H., Jingxian Yu, J., Lishan Kang, L., Yuping Chen, Y. and Yongyan Chen, Y. (1999) The kinetic evolutionary modelling of complex systems of chemical reactions. *Computers and Chemistry*, 23, 143-151.
- Xiaofang ChenChen, X., Weihua Gui, W., Yalin Wang, Y and Lihui Cen, L. (2004) Multi-step optimal control of complex process: a genetic programming strategy and its application. *Engineering Applications of Artificial Intelligence*, 17 (2004) 491–500.
- Corma, A., Serra, J.M., Serna, P., Moliner, M. (2005) Integrating high-throughput characterization into combinatorial heterogeneous catalysis: unsupervised construction of quantitative (structure/property relation-ship models) *Journal of Catalysis* 232, 335–341.
- Csukás, B., Balogh, S. (1998) Combining genetic programming with generic simulation models in evolutionary synthesis. *Computers in Industry* , 36 1998, 181–197.

- Darwin, C. (1859) *The Origin of Species.* [Electronically] Available online at: http://en.wikipedia.org/wiki/The_Origin_of_Species

- GP250.m Code (2001) Swart, H. and Aldrich, C., Datacube Technologies CC

- Greeff, D.J., Aldrich, C. (1997) Empirical modelling of chemical process systems with evolutionary programming. *Computers and Chemical Engineering*, 22(7-8), 995-1005.

- Grieu, S., et al. (2005) KSOM and MLP neural networks for on-line estimating the efficiency of an activated sludge process. *Chemical Engineering Journal* , 116, 1–11.

- Grosman, B and Lewin, D.R. (2002) Automated nonlinear model predictive control using genetic programming. *Computers and Chemical Engineering*, 26(4-5), 631-640.

- Grosman, B and Lewin, D.R. (2004) Adaptive genetic programming for steady-state process modelling. *Computers and Chemical Engineering*, 28(12), 2779-2790.

- Tadashi Hattori, T. and Shigeharu Kito, S. (2005) Analysis of factors controlling catalytic activity by neural network. *Catalysis Today* 111, 328–332.

- Herrmann, J., et al. The role of explanations in an intelligent assistant system Universitäiit Dortmund, Informatik I, 44221 Dortmund, Germany [Electronically] Available online at www.sciencedirect.com

- Hinchliffe, M,. Willis, M. (2003) Dynamic systems modelling using genetic programming. *Computers and Chemical Engineering* 27, 1841-1854.

- Holland, J.H., (1962) *Outline of a logical theory of adaptive systems* University of Michigan, Michigan, Publisher ACM Press New York, NY, USA .

- Intel, (2006) [Electronically] information available at: http://www.intel.com/technology/silicon/mooreslaw/

- Jordan, M.I., (1999) Neural Networks. *The MIT Encyclopedia of the Cognitive Sciences.* A Bradford Book, ISBN 0-262-73124-X

- Kondos, P.D. and Demopoulos, G.P. 1993. Statistical modeling of O_2-$CaCl_2$-HCl leaching of a complex U/Ra/N/As ore. Hydrometallurgy, 32, 287-315.

- Koza, J. R. (1992) *Genetic programming.* Cambridge: MIT Press.

- Koza, J.R. (2004) Introduction to Genetic Programming Tutorial, GECCO 2004 [Electronically] http://www.genetic-programming.org

- Lew, T.L., Spencer, A.B., Scarpa, F., Worden, K., (20052006) Identification of response surface models using genetic programming. *Mechanical Systems and Signal Processing*., 20(8), 1819-1831.

- Madár, J., Abonyi, J. and Szeifert, F. (2005) Genetio programming for the identification of nonlinear input-output models. *Industrial Engineering and Chemistry Research*, 44, 3178-3186.

- Marlin, T.E. (2000) *Process Control: Designing processes and control systems for dynamic performance 2nd Edition*, McGraw – Hill Higher Education

- MATLAB by MathWorks, Inc, R13 and better. Documentation provided with software.

- McKay, B., Willis, M., Barton, G. (1997) Steady-state modelling of chemical process systems using genetic programming. *Computers and Chemical Engineering.* 21(9), 981-996.

- Moolman, D.W., Aldrich, C., Van Deventer, J.S.J. and Bradshaw, D.J. (1995) The interpretation of flotation froth surfaces by using digital image analysis and neural networks. *Chemical Engineering Science*, 50(22), 3501-3513.

- (B) Oduguwa, V., Roy, R. (2005b) A review of rolling system design optimisation. *International Journal of Machine Tools &and Manufacture* xx, 1–17.

- (A) Oduguwa, V., Tiwari, A., Roy, R. (2005a) Evolutionary computing in manufacturing industry: an An overview of recent applications. (2005a) *Applied Soft Computing* 5, 281–299.

- Pasadakis, N., Sourligas, S., et al. (2005) Prediction of the distillation profile and cold properties of diesel fuels using mid-IR spectroscopy and neural networks. *Fuel,* 85 (2006), 1131–1137.

- Sette, S., Boullart, L. (2001) Genetic programming: principles and applications. *Engineering Applications of Artificial Intelligence* 14, (2001) 727–736.

- Silva, S. (2004) A *Genetic Programming Toolbox for MATLAB* [Electronically] Available online at www.gplab.com

- Utgoff, P. (1999) Decision Trees. *The MIT Encyclopedia of the Cognitive Sciences.* A Bradford Book, ISBN 0-262-73124-X

- Willis, M., Hiden, H., Hinchliffe, M, McKay, B. (1997) Systems Modelling using Genetic Programming. *Computers and Chemical Engineering*, 21, Suppl 1, S1161-S1166.

- Yeun, Y.S., Suh, J.C., Yang, Y.S. (1999) *Function approximations by superimposing genetic programming trees: with applications to engineering problems,* 122(2), 259-280.

- Zquez-Roman, R.V, King, J.M.P., et al. (1996) KBMoSS: A process Engineering Modelling Support System, Department of Chemical Engineering, University of Edinburgh, Scotland (U.K.) [Electronically] Available online at www.sciencedirect.com