EXPLORATORY DATA ANALYSIS AND EMPIRICAL MODELLING OF STATIONARY PROCESSES BY USE OF GENETIC PROGRAMMING

by

Timothy Paul Chemaly

Thesis submitted in partial fulfilment of the requirements for the degree of

Master of Engineering

in the Department of Chemical Engineering
University of Stellenbosch

Supervised by
Prof C. Aldrich

Stellenbosch
December 1999

Stellenbosch University https://scholar.sun.ac.za

DECLARATION

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

Timothy Paul Chemaly
December 1999

Stellenbosch University https://scholar.sun.ac.za

SUMMARY

Enhancing the performance of any process requires a detailed knowledge of the unknown system, with a mathematical model being the most common means of representing this knowledge. The most frequently used statistical techniques, assume that any relationships between input and output variables are linear and that the data itself is normally distributed. However, real world systems can be highly non-linear and linear approaches can therefore fail to predict the behaviour of the system accurately. Explicit specification of optimal structure in large non-linear models is often not practical and as a result, non-parametric methods (kernel regression, artificial neural networks, etc.) are usually employed. Although these models allow accurate representation of complex systems, they can be very difficult to interpret.

This research project explores a novel approach to this problem of mathematical modelling which attempts to evolve optimal parametric models, based on the Darwinian mechanism of evolution. This approach, referred to as genetic programming (GP), facilitates development of explicit or implicit models, or any mix of these two extremes, as dictated by the problem and unlike other methods, it can handle a trade-off between accuracy and interpretability with great ease.

During this research, a commercial application (a-GP) was developed, since very few commercial systems are currently available. Some techniques were developed, which improved the performance of the original algorithm considerably. For instance, memory demands were decreased by a factor of 5 by utilizing a different implementation model. Improved convergence and robustness was obtained by using a correlation-based fitness function in conjunction with a correction filter which reduced the sum of the

Stellenbosch University https://scholar.sun.ac.za

squared errors; at the expense of a more complex model. The evaluation process was expedited by evaluating each tree-like structure as a reverse polish expression; as opposed to a branch-node reduction technique. Additional execution speed was further obtained by implementing the algorithm in C++ (an object oriented compiled language) which is significantly faster than the original LISP (an interpreted language) implementation,.

The newly improved algorithm, a-GP, was applied to four industrial data sets and the results were compared against other methods such as standard genetic programming, multilayer perceptron neural networks and linear regression. It was found that a-GP outperformed standard genetic programming on all four case studies, while improving on neural networks on half of the runs.

The evolved models tended to be complex. This could be attributed to the lack of parameter estimation that the genetic programming algorithm tried to compensate for by evolving complex tree structures; which it used to approximate the parameters.

As a data visualization tool, a-GP was applied to four bench marking data sets used extensively in the literature. The results acquired with a-GP compared favourably with those obtained by other methods with the additional benefit in that a-GP was able to evolve simple mapping functions, which clearly indicated how the variables related to the structure. Additionally, the algorithm was applied in the mapping of two industrial processes. The results showed distinct clustering tendencies within the data, indicating the different operating regimes of the processes under investigation.

Stellenbosch University https://scholar.sun.ac.za

OPSOMMING

Om die vermoë van 'n proses stelsel te verbeter vereis 'n gedetaileerde kennis of model van die onderliggende proses. Die statistiese tegnieke wat meestal gebruik word, neem aan dat enige verwantskap tussen die intree- en die uittree-veranderlikes lineêr is en dat die data self normaal versprei is. Ongelukkig is realistiese probleme dikwels nie-linieêr en linieêre benaderings kan nie die gedrag van sulke stelsels akkuraat karakteriseer nie. Die eksplisiete spesifikasie van die optimale struktuur in groot nie-linieêre modelle is nie altyd prakties moontlik nie, met die gevolg dat nie-parametriese metodes (basisfunksie-regressie, kunsmatige neurale netwerke, ens.) gebruik word. Alhoewel hierdie modelle akkurate voorstellings toelaat van komplekse sisteeme is hulle baie moeilik om te interpreteer.

Hierdie tesis beskryf 'n unieke benadering tot die probleem van wiskundige modellering deur optimale parametriese modelle te evolueer, wat op die basiese beginsels van Darwin se evolusionêre model berus. Hierdie tegniek, wat genetiese programmering (GP) heet, kan eksplisiete, sowel as implisiete modelle ontwikkel of enige kombinasie daarvan, soos gedikteer word deur die probleem. Anders as ander metodes, kan dit ook maklik 'n balans tussen akkuraatheid en interpreteerbaarheid handhaaf.

Gedurende hierdie navorsing is 'n kommersiële sagteware-pakket (a-GP) ontwikkel, aangesien baie min sulke pakkette tans beskikbaar is. Verskeie tegnieke is ontwikkel wat die standaard algoritme aansienlik verbeter het. By voorbeeld, die aanvraag na geheue is verminder met 'n faktor van 5 deur gebruik te maak van 'n alternatiewe implementeringsmodel. Versnelde konvergensie en robuustheid was ook verkry deur gebruik te maak van 'n korrelasie-gebaseerde fiksheidfunksie in samewerking met 'n

Stellenbosch University https://scholar.sun.ac.za

korreksiefilter wat die som van die gekwadreerde foute geminimeer het. Die evaluasieproses was ook versnel deur elke boomstruktuur as 'n omgekeerde Poolse vergelyking op te los, in plaas van 'n tak-node vereenvoudigingstegniek. Die verwerking was verder versnel deur die algoritme te implementeer in C++ ('n objek georiënteerde, gekompileerde taal) wat aansienlik vinniger is as die oorspronklike LISP (geïnterpreteerde taal) implementering.

Die nuwe verbeterde algoritme, a-GP, is toegepas op vier realistiese probleme met die doel om regressiemodelle te genereer en die resultate is vergelyk met dié verkry deur ander tegnieke, soos standaard genetiese programmering, kunsmatige neurale netwerke en linieêre regressie. Daar was gevind dat a-GP op die standaard genetiese programmering verbeter in al vier gevalle, terwyl dit op kunsmatige neurale netwerke verbeter het op een van die toetsstelle. Die modelle het geneig om kompleks te wees, wat interpretasie bemoeilik het. Dit kan toegeskryf word aan die tekortkoming van 'n parameterbenadering, waarvoor die genetiese programmering algoritme probeer kompenseer deur komplekse boomstukture te ontwikkel. Die algoritme gebruik dié strukture om die parameters af te skat.

a-GP is ook gebruik om data te visualiseer. Die resultate op vier datastelle het gewys dat a-GP baie goed vergelyk met ander metodes, terwyl dit die addisionele voordeel gehad het, dat dit eenvoudige projeksie-funksies kon evolueer wat duidelike verwantskappe tussen die veranderlikes en die struktuur uitgewys het. Die algoritme was ook toegepas op die projeksie van twee industrieële stelsels na twee dimensies vir visualisering. Die resultate het duidelike trosvorming in die data uitgewys, wat 'n indikasie was van die verskillende operasionele toestande van die prosesse.

Stellenbosch University https://scholar.sun.ac.za

ACKNOWLEDGEMENTS

The author gratefully acknowledges the contributions of the following:

- Prof C. Aldrich, the supervisor of this thesis, for his guidance and insight during the course of this work.
- De Beers, for their financial assistance.
- And of course, Jeanné.

Stellenbosch University https://scholar.sun.ac.za

TABLE OF CONTENTS

DECLARATION i
SUMMARY ii
OPSOMMING iv
ACKNOWLEDGEMENTS vi
LIST OF FIGURES xiii
LIST OF TABLES xvii

CHAPTER 1 **INTRODUCTION TO EXPLORATORY DATA ANALYSIS AND EMPIRICAL MODELLING** 1

1.1 *Background to exploratory data analysis* 1
1.2 *The nature of empirical modelling of process systems* 2
1.3 *The objectives* 4

CHAPTER 2 **THE GENETIC PROGRAMMING PARADIGM** 6

2.1 *The emergence of algorithms driven by evolution* 6
2.1.1 A brief history of Darwin's evolutionary world 6
2.1.2 An overview of evolutionary strategies 7
2.1.3 An overview of genetic algorithms 7
2.2 *The emergence of automated programming:*

Stellenbosch University https://scholar.sun.ac.za

Genetic Programming 9
2.2.1 An introduction to genetic programming 9
2.2.2 The characterization of an evolutionary algorithm 11
2.2.2.1 *Initialization* 12
2.2.2.2 *Evaluation* 13
2.2.2.3 *Selection* 14
2.2.2.4 *Reproduction* 15
2.3 *Current limitations in genetic programming* 18
2.3.1 Speed and resources 18
2.3.2 Limitations with genetic programming as a global searching algorithm 20
2.3.3 The disruptive behaviour of the crossover operation . . 21
2.3.4 Exploring large search spaces 23
2.3.5 Restraining premature convergence 24
2.3.6 Discussion of the current remedies 25
2.4 *Applications of genetic programming* 26
2.4.1 Robotic control 26
2.4.2 Image analysis and feature extraction 26
2.4.3 Language learning applications 27
2.4.4 Evolving controllers for systems 27
2.4.5 Process modelling 28
2.5 *Objectives of this study* 29
2.5.1 Motivation for this research 29
2.5.2 Outline of the chapters in this thesis 30

CHAPTER 3 **THE DESIGN METHODOLOGY** 32

3.1 *Taking an object oriented approach towards designing the genetic programming kernel* 32
3.1.1 Designing the abstract base class 33
3.1.2 Designing the abstract evolutionary algorithm class . . 34
3.1.3 Designing the genetic programming class 35

Stellenbosch University https://scholar.sun.ac.za

3.1.4 Designing the feature extraction class 35
3.1.5 Graphical overview of the base class and all its derived descendants 36
3.2 Probing the size of the search space 37
3.3 Augmentations to improve the original genetic programming algorithm 39
3.3.1 Changing the internal representation of an individual in genetic programming 39
3.3.2 A different evaluation scheme 42
3.3.3 Increasing convergence and robustness in regression models using an expanded solution space 44
3.3.3.1 Fitness function 44
3.3.3.2 Correlation 45
3.3.3.3 Confirming the hypotheses 48
3.3.3.4 Discussion of results 51
3.3.3.5 Conclusions 54

CHAPTER 4 **PROCESS MODELLING USING a-GP** 55

4.1 An introduction to process modelling 55
4.2 Case studies 57
4.2.1 Approximation of multivariate functional relationships 57
4.2.2 Obtaining regression models for four real data sets . . 60
4.2.2.1 Modelling of transpiration in pine trees . . 61
4.2.2.2 Modelling of transpiration in poplar trees 61
4.2.2.3 Modelling of the Black Mountain base metal flotation plant 61
4.2.2.4 Modelling of a solution preparation circuit 61
4.3 Run parameter listings 61
4.4 Investigating the effect different crossover and mutation rates has on the overall performance of the algorithm 63

Stellenbosch University https://scholar.sun.ac.za

4.5 Discussion of results . . . 65
4.6 Conclusions . . . 68

CHAPTER 5 **VISUALIZATION OF PROCESS SYSTEMS USING a-GP** 70

5.1 An introduction to dimensionality reduction . . . 70
5.1.1 An overview of data projection . . . 71
5.1.2 Characteristics of data . . . 73
5.2 Extending the genetic programming algorithm to accommodate feature extraction . . . 74
5.3 Case studies . . . 75
5.3.1 Case studies on artificial and bench marking data sets 76
5.3.1.1 Description of each data set . . . 76
5.3.1.2 Results obtained . . . 78
5.3.2 Flotation data from an Australian base metal flotation plant . . . 80
5.3.2.1 A description of each data set . . . 80
5.3.2.2 Results obtained . . . 81
5.3.3 Three-phase oil flow data. . . . 82
5.3.3.1 A description of each data set . . . 82
5.3.3.2 Results obtained . . . 83
5.4 Results and conclusions . . . 85

CHAPTER 6 **RECOMMENDATIONS FOR FUTURE RESEARCH** 87

CHAPTER 7 **CONCLUSIONS** 89

REFERENCES 92

Stellenbosch University https://scholar.sun.ac.za

NOMENCLATURE 103

APPENDIX A EVOLVED MODELS A.1

A.1 The unsimplified regression models of Chapter 4 . . . . A.1
A.1.1 Regression model for data set PINE A.1
A.1.2 Regression model for data set POP A.1
A.1.3 Regression model for data set BMVANO A.1
A.1.4 Regression model for data set SOLPREP A.2

APPENDIX B THE SOURCE CODE B.1

B.1 Abstract base classes B.1
B.1.1 Header file for abstract class GenericIndividual and GenericAlgorithm B.1
B.1.2 Header file for abstract class GenericEvoIndividual and GenericEvolutionaryAlgorithm B.6
B.2 The GP class B.14
B.2.1 Header file for class GPIndividual and CustomGPalgorithm B.14
B.2.2 Header file for class GPSupervised B.23
B.2.3 Header file for class GPUnsupervised B.24
B.2.4 Implementation of each class in the GP kernel B.24
B.2.4.1 Implementation of class CustomGPAlgorithm B.24
B.2.4.2 Implementation of class GPSupervised B.34
B.2.4.3 Implementation of class GPUnsupervised B.39
B.2.5 Header file for class FeatureExtract B.40

Stellenbosch University https://scholar.sun.ac.za

APPENDIX C HELP ON THE a-GP PACKAGE C.1

C.1 Possible analysis that can be conducted using a-GP C.1
C.2 How to select a new algorithm C.1
C.3 How to select a different process C.1
C.4 How to change the properties of an algorithm C.2
C.5 How to import data C.3
C.5.1 Format of data file C.3
C.5.2 Importing the data C.3
C.6 Starting the algorithm C.4
C.6.1 The Start button C.4
C.6.2 The Pause button C.4
C.6.3 The Restart button C.4
C.7 Plotting the results C.5
C.7.1 Available charts C.5
C.7.2 Dragging a vector to a chart C.5
C.7.3 Removing a specific plot from a chart C.6
C.7.4 Changing chart types C.6

Stellenbosch University https://scholar.sun.ac.za

LIST OF FIGURES

Figure 1.1 : *Typical structure of a feedforward neural network.* 3

Figure 2.1 : *According to the evolutionary theory, mankind and other primates, share a common ancestor.* 6

Figure 2.2 : *An example of a chromosome in genetic algorithms. This chromosome is comprised of four genes. Each gene is represented by a different colour.* 8

Figure 2.3 : *A schematic representation of the genetic operators: (a) crossover and (b) mutation.* 9

Figure 2.4 : *A parse tree in GP consisting of two functional nodes and three terminal nodes.* 9

Figure 2.5 : *The basic flowchart characterizing the behaviour of an evolutionary algorithm.* 12

Figure 2.6 : *A graphical depiction of the crossover operation. Two points are randomly selected on the two parents and their respective sub-trees are swapped.* 16

Figure 2.7 : *A random node is selected on the parent and replaced by a randomly generated sub-tree, during mutation.* 17

Figure 2.8 : *An explicit mathematical function evolved by GP written in both Polish and standard form.* 20

Stellenbosch University https://scholar.sun.ac.za

Figure 2.9 : *An explicit computer program evolved by CGP.* 20

Figure 3.1 : *A generic initialization algorithm.* 33

Figure 3.2 : *The generic DoAlgorithm() method. Note the method: GeneticOperations is defined as pure virtual. It's actual behaviour depends on the class in which it is implemented.* 34

Figure 3.3 : *A graphical overview of all the classes and their decendants.* 36

Figure 3.4 : *A recursive procedure that generates a genetic programming tree-like structure. This structure may be used as either an individual or as a randomly generated sub-tree during mutation.* 41

Figure 3.5 : *The pseudocode for the evaluation function.* 44

Figure 3.6 : *An algorithm and the correction filter, G, acting as a hybrid model.* . . 48

Figure 3.7 : *The difference in average convergence, for the three data sets in terms of R^2-values vs the number of generations : (a) PINE, (b) SOLPREP and (c) BMVANO, when an error-based fitness function (broken line) and a correlation-based fitness function (solid line) is used. In all three examples, the correlation-based fitness function yields a much higher convergence.* 53

Figure 4.1 : *A histogram plot of the frequency distribution of the fitness of each individuals in a sampled at a specific generation. Notice that when the algorithm starts (generation 1) most individuals have very low fitness values. With succeeding generations this distribution starts moving towards the region with higher fitness.* 60

Figure 4.2 : *The effect of varying combinations of crossover (Pc) and mutation (Pm) rates, emphasized the fact that a too small search rate does not yield satisfactory results. In (a) and (b) we can see that the algorithm got entrapped in a local optimum, when the crossover/mutation rate was set at (20/1)%. Increasing the search rate to (80/20)% allowed the algorithm to avoid entrapment in the local optimum. In (c) the larger search rate did not make*

Stellenbosch University https://scholar.sun.ac.za

significant difference, while in (d) a steady increase can be observed. . . . 64

Figure 4.2 : A comparison of R^2 obtained from the four data sets. a-GP outperforms GP on all four case studies. . . . 66

Figure 4.3 : X-Y scatter plots of the Observed output vs Predicted output for data sets: (a) PINE [R^2=0.85], (b) POP [R^2=0.67], (c) BMVANO [R^2=0.53] and (d) SOLPREP [R^2=0.48]. . . . 67

Figure 5.1 : The q parse-trees that make up an individual for feature extraction. Each tree represent a mapping function ranging from ρ_1 to ρ_q. . . . 74

Figure 5.2 : During feature extraction, crossover only occurs between parse-trees with similar indices. In this example two individuals, I_i and I_j are randomly selected from the mating pool. A parse-tree, ρ_3, is randomly selected from both trees for crossover. . . . 75

Figure 5.3 : Typical Sammon map of the BITET data set, generated by the Genetic Programming algorithm, S = 0.0472, $F_1 = x_1$ and $F_2 = x_1 + x_2$. The clusters are indicated by different labels, as shown in the legend. . . . 79

Figure 5.4 : The Sammon map of the SPHERESHELL data set,generated by the Genetic Programming algorithm, S = 0.0531, $F_1 = x_2$ and $F_2 = x_1$ 79

Figure 5.5 : Typical Sammon map of the IRIS data set generated by the Genetic Programming algorithm, S = 0.00657, $F_1 = x_3 + x_4/[1+\exp(x_1)]$; $F_2 = x_2$. . . . 79

Figure 5.6 : Typical Sammon map of the SPIRAL data set, generated by the Genetic Programming algorithm, S = 0.00403, $F_1 = x_3$ and $F_2 = x_2$. . . . 79

Figure 5.7 : Principal component map of 13 plant variables on a base metal flotation plant. The first two principal components (PC_1 and PC_2) explained 55.9% and 14.1% of the variation in the data respectively. The discretized values of the concentration of the valuable metal (not part of the mapped data set) is superimposed on the map. . . . 80

Stellenbosch University https://scholar.sun.ac.za

Figure 5.8 : *Sammon map of the base metal flotation data generated by the Genetic Programming algorithm with* $S = 0.00473$, $F_1 = x_6 - x_{12}$ *and* $F_2 = 1 + x_1 + x_4 - x_7 + x_{11} - x_8/x_{11}$. . . . 81

Figure 5.9 : *Sammon map of the base metal flotation data generated by the multilayer perceptron neural network, with a Sammon stress of* $S = 0.02473$. . . . 82

Figure 5.10 : *Three-phase flow with* $S = 0.05270$, $F_1 = v_2 - v_3 + v_{10} + 1/(1+\exp(v_{12})$ *and* $F_2 = v_7/[1+\exp(v_1v_4)] + v_7 - 1/\{1+\exp[1/(1+\exp(v_1v_4))]\} - v_4$. . . . 83

Figure 5.11 : *Three-phase flow with* $S = 0.05293$, $F_1 = v_2 + v_6 + v_{10}$ *and* $F_2 = v_7 + 2v_5$. . . . 84

Figure 5.12 : *Three-phase flow with* $S = 0.04943$, $F_1 = 2\sin(v_4) + v_{10}$ *and* $F_2 = \sin(\sin(v_7)) + v_5 + v_7$. . . . 84

Figure 5.13 : *The results obtained using a multilayer-perceptron neural network. The Sammon stress, S = 0.0324. The stratified flows appear more distinct but also more clustered, from the homogeneous and annular clusters.* . . . 85

Figure C.2 : *The drop down list of the available processes. To activate one of the processes move the mouse cursor to the process and click.* . . C.2

Figure C.3 : *The property box with all the available properties of the active process. The box is divided into two regions: a Name field and a Value field.* . . . C.2

Figure C.4 : *The contents of the data file is displayed in the Data Import Wizard. If the first rows have labels click on "Labels in first row". The data type of each column can be specified by right clicking on the Type row of the corresponding column. Click on ">>" to continue.* . . . C.3

Figure C.5 : *If the contents of all the columns contain valid numeric values, the variables may be send to any of the processes listed in the process list box. Click on the down arrow and select a different process you wish to send the variables to.* . . . C.4

Figure C.6 : *The available plots are displayed on the panel. Clicking on any of the variables will remove it from the current chart.* . . . C.6

Stellenbosch University https://scholar.sun.ac.za

LIST OF TABLES

Table 3.1 : *Illustrating the difference between Standard, Polish and reverse Polish notation.* 43

Table 3.2 : *Parameters used for each data set during regression.* 50

Table 3.3 : *Measured results obtained for each data set after using an error-based fitness function and a correlation-based fitness function.* . . 52

Table 4.1 : *Results obtained for the identification of a multivariate functional relationship. 20 runs were conducted of which 10 used a correlation-based fitness criterion and the remainder, an error-based fitness criterion.* 58

Table 4.2 : *Results obtained for the evaluation of the algorithm with and without a priori knowledge in the function set. 20 runs were conducted of which 10 used a function set that had the a priori information included or F = {+, -, *, /, sin}. In the remaining ten runs, this information was excluded, therefore F = {+, -, *, /}.* 58

Table 4.3 : *Run parameters used for each data set during regression.* 63

Table 4.4 : *Results obtained for each of the four data set after testing. A comparison of R^2 and MSE is made amongst the four different regression techniques. These are a-GP, GP, linear regression and ANN's.* 65

Table 4.5 : *Significance of the difference between the correlation*

Stellenbosch University https://scholar.sun.ac.za

coefficients of the four different regression techniques. Here, the null hypothesis, H_0, is tested to see whether the results obtained with a-GP, on the four data sets, are significantly different than those obtained via GP, linear regression and neural networks. The values inside the table are the test statistic (z) values. The values that are labelled with [(a)] imply that the results obtained via a-GP, are significantly different when compared to the corresponding algorithm in that row. 66

Table 5.1 : *Essential characteristics of the four data sets* 77

Table 5.2 : *Parameters used for each data set during feature extraction.* 77

Table 5.3 : *A comparison of stress values (Sammon stress) obtained from six different projection algorithms for the four data sets.* 78

Stellenbosch University https://scholar.sun.ac.za

CHAPTER 1

INTRODUCTION TO EXPLORATORY DATA ANALYSIS AND EMPIRICAL MODELLING

1.1 Background to exploratory data analysis

The tremendous acceleration in computer technology, which was accompanied by a reduction in hardware size and an increase in computational speed; and the emergence of the internet and especially the World Wide Web (WWW) has led to an increase in data traffic and especially data processing. Chemical and metallurgical process industries have likewise experienced a continued growth in large data systems. This has precipitated intense efforts to develop more efficient methods for the exploration and interpretation of large volumes of data. It is therefore not uncommon for the individual analyst to have to interpret many hundreds or even thousands of variables and hundreds of thousands of observations off-line, while in automated monitoring and control systems, data volumes of an order of magnitude higher may have to be accommodated.

Exploratory data analysis, therefore, aims to find interesting structures in data for visualization purposes. These structures may ultimately lead to an increased understanding of the unknown process and may be used for empirical modelling. As such, data are usually pre-processed via exploratory data analysis before the actual modelling occurs.

Principal component analysis (PCA) is the most widely used tool for exploratory data analysis (Kendall, 1975; Jolliffe, 1986; Piovoso *et al.*, 1992; MacGregor, 1989; Stephanopoulos and Guterman, 1989). However, principal component analysis is a linear technique. This has lead to several attempts to extend the technique to deal with

Stellenbosch University https://scholar.sun.ac.za

non-linearities arising from complex data. In this regard, artificial neural networks have been used extensively (Lampinen and Oja, 1995), (Mao and Jain, 1995), (Pal and Eluri, 1998), (Kraaiveld *et al.*, 1995). Also, major advances have been made with methods such as cluster analysis, which try to group individuals or objects that are more homogeneous than objects that reside in other groups; factor analysis, which reduces the dimensions of a problem, similar to principle component analysis, except that the effect of noise is taken into account and projection pursuit analysis which tries to find directions such that the projection of the data in that direction has an "interesting" distribution.

One of the main problems with the non-linear techniques is their inability to generate simple non-linear functions which can transform the higher dimensional data to a lower dimensional space. The lack of transformation functions can lead to an inability (eg. Sammon mapping) to generalize which results in the retraining of the system should new data arrive. Also the transformations obtained via non-parametric solutions are restricted in the sense that the models are difficult to interpret.

Therefore, the idea is then to construct explicit and simple, non-linear transformation functions, using genetic programming. This will not only allow generalization (within the range of the data used for model development and avert exhaustive retraining) but also facilitate the development of interpretability transformation functions.

1.2 The nature of empirical modelling of process systems

Processing plants require periodical adjustments of their operating conditions to maximize profits or minimize costs (Seborg *et al.*, 1989). For example, instrumentation has to be recalibrated and the plant units need to be adjusted to accommodate variations in ore feed; blending operations in the petrochemical industry may have to be modified in response to changes in crude oil feedstocks, etc. These modifications require some form of representation or modelling of the processes, without which adjustments could result in significant inefficiency in overall operations (Greeff and Aldrich, 1998).

Stellenbosch University https://scholar.sun.ac.za

The advantage of having a process model is that it can be analysed to increase understanding of the underlying physical phenomena inherent to the system. Although possible, the development of a model requires a detailed knowledge of the physics and chemistry of a system. This is not always viable, owing to the complex and non-linear nature of industrial process systems. Also, it may require a considerable amount of time and resources to develop a realistic model. Nonetheless, an accurate process model can improve process operability. Empirical models are often based on regression analysis, aimed at minimizing a least square criterion.

A regression analysis tries to model an input-output description of the system using historic data. The most widely used and well understood regression model is the linear model as depicted in Eq. 1.1.

$$F(\mathbf{x}) = a_0 + a_1x_1 + a_2x_2 + \ldots + a_nx_n = a_0 + \sum_{i=1}^{n} a_ix_i \qquad (1.1)$$

Although simple, linear models try to make a linear approximation of the process while in practice most systems are non-linear. This has lead to the development of non-linear methods using various techniques, some of which generate solutions using a simulated form of evolution, *viz.* evolutionary algorithms (EA), other, such as inductive systems, try to build decision trees that are equivalent to IF-THEN rules, while packages like CART construct regression trees that are similar to decision trees, except that the nodes do not represent classes, but continuous values. Additionally, polynomial regression, breakpoint regression and piece-wise regression are also used. Perhaps the most important and widely used are

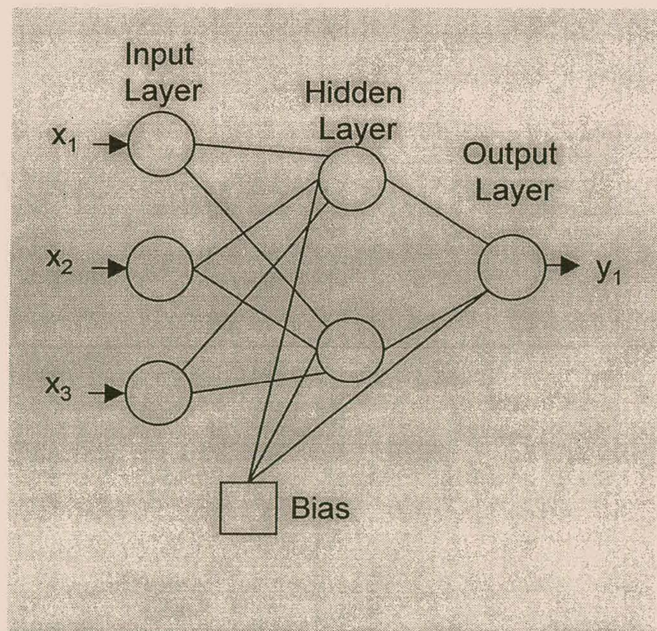


Figure 1.1 : ***Typical structure of a feedforward neural network.***

Stellenbosch University https://scholar.sun.ac.za

algorithms that try to model solutions which mimic the workings of the brain, *viz.* artificial neural networks (ANN). Non-linear methods are typically based on iterative procedures for estimation of the parameters, such as Gauss-Newton, Levenberg-Marquardt or Powell algorithms, which are normally extremely computationally intensive and thus slow; whilst in linear methods the parameters are calculated directly.

Usually no *a priori* information is available regarding the structure of the model. This results in considerable effort to find an adequate model to which parameters may be fitted. For non-parametric techniques, such as artificial neural networks, no explicit structure specification is required[1] but the derived mathematical models are extremely complex and very difficult to analyse. Figure 1.1 illustrates the structure of a typical feedforward neural network.

As explained in more detail in Chapter 2, evolutionary methods (specifically genetic programming) facilitate the automatic construction of explicit models. This can lead to considerable cost savings over manual efforts. In fact it can provide a feasible approach to the development of explicit models, where other methods may not be viable at all.

1.3 The objectives

As mentioned previously, in this thesis it is shown that by making use of genetic programming, interpretable empirical models and transformation functions can be constructed without any need to specify explicit model structures. This technique is also more cost effective in the sense that it does not need any encoding schemes as required by other evolutionary algorithms.

Owing to the novelty of this algorithm, very few commercial tools are available that employ the genetic programming algorithm. Consequently, this necessitated the development of a commercial application which could perform all of these tasks, i.e.

[1]Although artificial neural networks do not need any information regarding the structure of the model, the neural network architecture still need to be specified, i.e. the number of hidden layers and nodes, type of activation function, etc.

Stellenbosch University https://scholar.sun.ac.za

perform exploratory data analysis and generate empirical models, using this algorithm.

- One of the main objectives of this thesis was the design of a commercial package that incorporates the genetic programming algorithm, and the creation of a simple and intuitive graphical user interface (GUI).

Specifically, genetic programming algorithms will therefore be used

- for data visualisation purposes, to find explicit symbolic mapping functions which allows the visualisation of data residing in a high dimensional space (d > 3), to be viewed in a lower (d = 2) dimension.
- to obtain explicit symbolic functions to describe the input-output relationships within processes, especially in the chemical and metallurgical industries.
- for comparison with other methods, such as neural networks and linear regression techniques.

Additionally, some improvements will be made to the original algorithm to

- improve the convergence speed and robustness of regression models.
- reduce memory usage and increase processing speed.

Included in this thesis is the fully functional software package, a-GP, which the reader may install and evaluate on his/her own computer. The reader is advised to consult Appendix C to gain understanding on how to use this software.

Stellenbosch University https://scholar.sun.ac.za

CHAPTER 2

THE GENETIC PROGRAMMING PARADIGM

2.1 The emergence of algorithms driven by evolution

2.1.1 A brief history of Darwin's evolutionary world

In 1859, Charles Darwin published his controversial "The Origin of Species". In this book he claimed that life itself was compelled by evolution and that the main driving force behind evolution was *natural selection*. In short, natural selection implied that the strongest, or fittest, individuals within species would have a better chance of surviving and being selected for mating. They would therefore be more likely to pass there genes on to the next generation, than the weaker members of the species. Subsequently, the offsprings of these fit individuals would therefore possess traits from both parents. Given the environment in which the species live, the offspring would be better adapted to it than the parents. With increasing generations the species would have changed to the point where it has adapted completely to its environment.

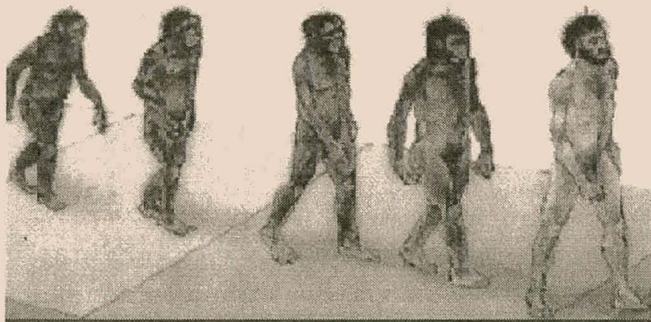


Figure 2.1 : ***According to the evolutionary theory, mankind and other primates, share a common ancestor.***

Although this notion of evolution is not without its discrepancies, some very exciting work has been done since the early 1960's, by applying the concept of evolution and natural selection to optimize real world problems.

Stellenbosch University https://scholar.sun.ac.za

2.1.2 An overview of evolutionary strategies

In 1963, two students at the Technical University of Berlin met and collaborated on experiments which used the wind tunnel of the Institute of Flow Engineering. During the search for the optimal shapes of bodies in a flow, which was then a matter of labourious intuitive experimentation, the idea was conceived of proceeding strategically. However, attempts with coordinate and simple gradient strategies were unsuccessful. One of the students, Ingo Rechenberg, now Professor of Bionics and Evolutionary Engineering at the Technical University of Berlin, hit upon the idea of trying random changes in the parameters defining the shape, following the example of natural mutations. In this way the evolutionary strategy (ES) was born. A third student, Peter Bienert, joined them and started the construction of an automatic experimenter, which would work according to the simple rules of mutation and selection. Evolution strategies were invented to solve technical optimization problems like constructing an optimal flashing nozzle, the design of truss bridges and more recently to the design of partially recurrent neural networks. Until recently the evolutionary strategy was only known to civil engineers, as an alternative to standard solutions. Although genetic algorithms (GA), (Holland, 1992) which were developed in the 1960's, are closely linked to evolutionary strategies, genetic algorithms use crossover as the main searching operator whereas evolutionary strategies use mutation. Crossover is a stochastic process which allows two parents to exchange some of their traits (or genes) during mating and hence produce offspring which resemble both of them. Both genetic algorithms and evolutionary strategies are referred to as evolutionary algorithms (EA). At present evolutionary algorithm is an umbrella term for all population based algorithms that employ the basic principles of evolution, *viz.* natural selection, crossover and/or mutation to evolve new and fitter individuals during successive generations.

2.1.3 An overview of genetic algorithms

The genetic algorithm (GA) developed by Holland (1992) in the early 1960's is a model of machine learning which derives its behaviour from a metaphor of some of the mechanisms of evolution in nature, i.e. the natural selection, mutation and crossover of genetic material. This is done by the creation of a population of individuals

Stellenbosch University https://scholar.sun.ac.za

represented by chromosomes. In essence a set of character strings that are analogous to the base-4 chromosomes that can be seen in our human DNA. The individuals in the population then go through a process of simulated evolution until a good or optimal solution is found.

Genetic algorithms are used in a number of different application areas. These applications are typically multidimensional optimization problems in which the character string of the chromosome can be used to encode the values for the different parameters being optimized.

In practice, one can implement this genetic model of computation by having arrays of bits or characters to represent the chromosomes as shown in Figure 2.2.

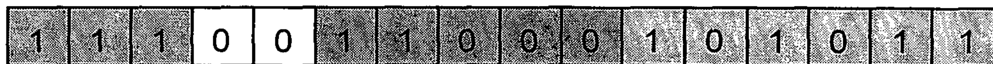
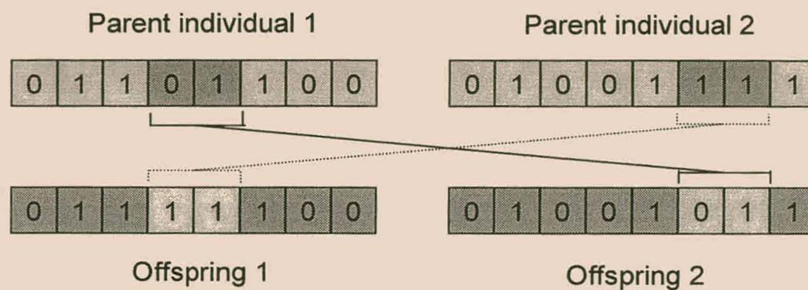


Figure 2.2 : ***An example of a chromosome in genetic algorithms. This chromosome is comprised of four genes. Each gene is represented by a different colour.***

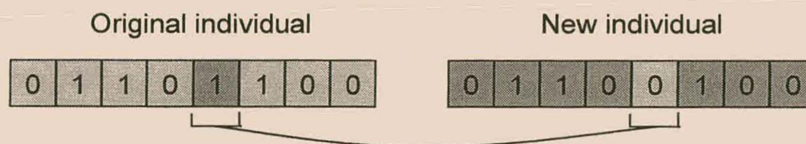
Simple bit manipulation allows the implementation of crossover, mutation and other operations, as indicated in Figure 2.3. The crossover operation, between two individuals, results in genetic material being selected from both parents. This material is then swapped (Figure 2.3.a) and the resulting individuals are the offspring of the two parents. The offspring become members of the next generation. Mutation results in one of the bits being randomly flipped to either 1 or 0, as indicated in Figure 2.3.b. This new individual becomes a member of the next generation.

Although a substantial amount of research has been performed on variable-length strings (Nordin and Banzhaf, 1995) and other structures (Iba and Sato, 1992), the vast majority of work [with genetic algorithms] has focussed on fixed-length character strings. In this regard genetic algorithms differ substantially from genetic programming (GP) (Koza, 1992) that does not have a fixed length representation and does not need any encoding scheme.

Stellenbosch University https://scholar.sun.ac.za



(a) Crossover operation



(b) Mutation operation

Figure 2.3 : ***A schematic representation of the genetic operators: (a) crossover and (b) mutation.***

2.2 *The emergence of automated programming: Genetic Programming*

2.2.1 An introduction to genetic programming

The notion of instructing a computer *what* to do as opposed to *how* to do something has stimulated the human mind since the early stages of the development of the computer. This dream came a step closer to reality when John Koza (Koza, 1992) introduced a form of a self-modifying code generator, which he called Genetic Programming or GP. Using the LISP programming language as an implementation platform, and the ideas of natural selection (i.e. survival of the fittest and genetic manipulation) as the main driving force behind his algorithm, he was able to generate (or evolve) complete

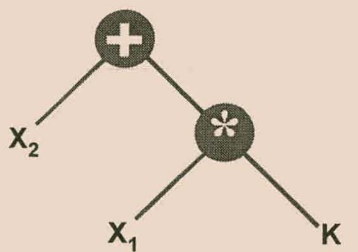


Figure 2.4 : ***A parse tree in GP consisting of two functional nodes and three terminal nodes.***

UNIVERSITEITSBIBLIOTEEK

Stellenbosch University https://scholar.sun.ac.za

and syntactically correct computer programs which could be used in virtually any field or application. Genetic programming did not require any *a priori* knowledge regarding a model structure which is required in virtually all other algorithms such as evolution strategies, genetic algorithms, artificial neural networks (ANN)[1], multivariate linear regression, etc. Since no encoding and no structure was required, the algorithm could be seen as a "black box" approach to solving problems. Essentially genetic programming was a new paradigm in the sense that *any* solution to a problem could be expressed in a parse tree-like structure as shown in Figure 2.4.

Genetic programming is a programming variant of genetic algorithms. Unlike genetic algorithms the objects that constitute the population are *not* fixed-length character strings (see Figure 2.2) that encode possible solutions of the problem at hand, but programs that are the possible solutions to the problem. Genetic programming assumes no *a priori* information regarding inputs, structure or parameters.

In genetic programming every individual is represented as a tree-like structure of variable length. This representation can be seen as a phenotypic depiction of the individual. For example the simple program "$x_2 + x_1 \cdot K$" would be represented as in Figure 2.4.

As one can see, the parse tree consists of nodes and leaves. A node acts as a function or operator and a leave as a terminal. A function can be any known mathematical function or operator, such as "+", "-", "*", "sin", ..., etc. The terminals are usually the input variables of the process under investigation or any other known[2] constants. In Figure 2.4 there are two function nodes {+, *} and three terminal nodes {x_1, x_2, K}. The sets of all possible functions and terminals, which can be used to construct an individual, are termed the *function set*, ***F***, and *terminal set*, ***T***, respectively. Each element in the function and terminal set is referred to as an allele, which is derived from biological terminology.

[1]For an artificial neural network, the network architecture has to be specified.

[2]Actually the term *known* is a little misleading since one generally does not know anything about the process.

Stellenbosch University https://scholar.sun.ac.za

The advantages of using a parse tree is that (1) it can be rewritten in Polish notation, (2) it lends itself to easy manipulation and (3) it is very easy and fast to evaluate when expressed in Polish notation. In Polish notation the tree in Figure 2.4 can be expressed as

$$+x_2 * x_1 K \tag{2.1}$$

Which equals

$$x_2 + x_1 * K \tag{2.2}$$

in standard notation.

2.2.2 The characterization of an evolutionary algorithm

All evolutionary based algorithms' implementations can be characterized by the following sequence of events:

1. First construct an initial random population of N individuals.
2. Evaluate each individual against its objective and assign it a fitness value. Check for the termination criteria using the fitness value.
3. Select individuals for reproduction based on their fitness, i.e. those individuals who exhibit a high degree of fitness have a better chance for reproduction than others with a lower fitness. These individuals are placed in a mating pool. The mating pool is an intermediate container which the selected individuals enter *before* one applies the genetic operators to create the offspring.
4. Apply genetic operations, such as crossover and mutation, on randomly selected individuals in the mating pool.

Stellenbosch University https://scholar.sun.ac.za

5. Repeat step 2.

One iteration of this loop is referred to as a generation. There is no theoretical reason for this as an implementation model. Indeed, this punctuated behaviour is not seen in populations in nature as a whole, but it is a convenient implementation model. The first generation [generation 0] of this process operates on a population of randomly generated individuals. From there on, the genetic operations, in concert with the fitness measure, operate to improve the population. Figure 2.4 presents a flowchart of an evolutionary algorithm.

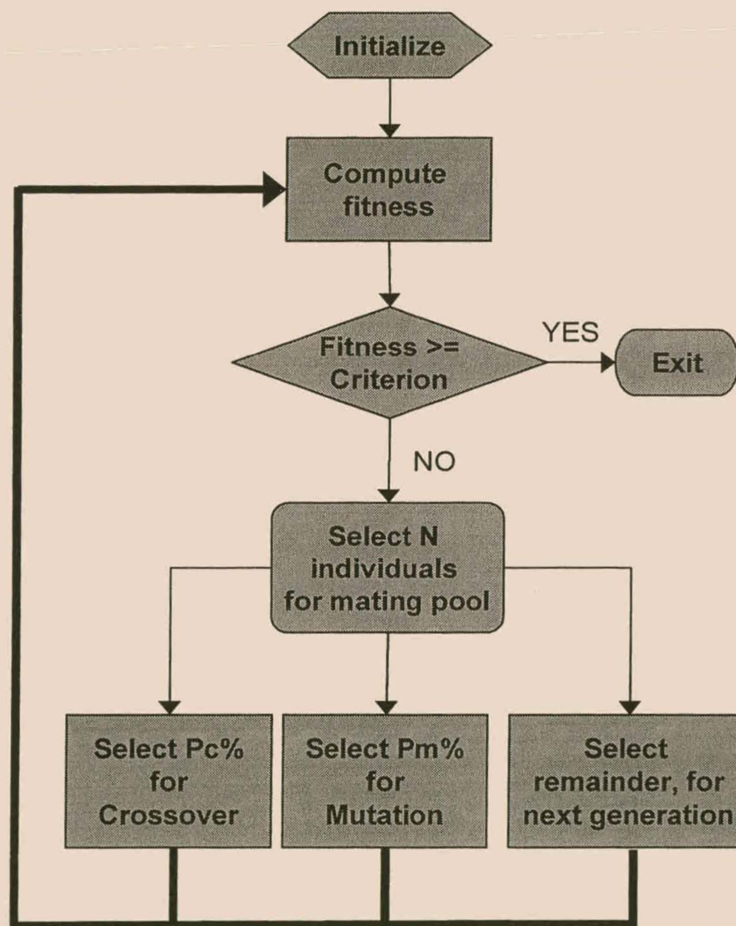


Figure 2.5 : ***The basic flowchart characterizing the behaviour of an evolutionary algorithm.***

2.2.2.1 *Initialization*

When the algorithm is initialized, N individuals are randomly generated. Every individual

Stellenbosch University https://scholar.sun.ac.za

is constructed as a parse tree (see Figure 2.4) from alleles of the function set and terminal set. These individuals can range from less than ten to several thousands. It is extremely important that the initial population be as diverse as possible to reduce premature convergence.

2.2.2.2 *Evaluation*

After this process of initialization, the fitness values, f_i, of all N individuals are computed. The fitness is a measure of the individual's ability to survive in its environment, while the fitness function itself characterizes the behaviour of the population. That is, if the fitness function assigns a high fitness to individuals who can approximate a desired state as accurately as possible, then after numerous generations *all* members of the population will start behaving in such a way that they can approximate the desired state. For regression, the environment will typically be the output data of the process under investigation. If $\mathbf{y} = F(x_1, x_2, \ldots, x_k)$, where $x_1, \ldots, x_k$ represents the input vectors of the process, $\mathbf{y}$ the output and F the functional representation of the process, the values of vector $\mathbf{y}$ will represent the environment. Every individual will represent a possible solution to the process. The difference between the actual output, $\mathbf{y}$, and the predicted, $\hat{\mathbf{y}}$, is here defined as the fitness. This kind of fitness criterion is referred to as an error-based fitness. In regression one can among other use an error-based fitness function or a correlation-based fitness function. In a correlation-based fitness function the correlation between the actual output, $\mathbf{y}$, and the predicted, $\hat{\mathbf{y}}$, is used as a measure of fitness.

The fitness (f_i) of the i'th individual, using an error-based fitness, can be expressed as follows:

$$f_i = \sum_{k=1}^{M} (y_k - \hat{y}_{ik})^2 \qquad (2.3)$$

Where y_k is the k'th value of the process output and $\hat{y}_{i_k}$ is the k'th value of individual

Stellenbosch University https://scholar.sun.ac.za

i. For a correlation-based fitness, (2.3) changes to

$$f_i = \left| \frac{\mathrm{cov}(\mathbf{y}, \hat{\mathbf{y}}_i)}{\sigma_{\mathbf{y}} \cdot \sigma_{\hat{\mathbf{y}}_i}} \right| \tag{2.4}$$

$\mathrm{cov}(\mathbf{y}, \hat{\mathbf{y}}_i)$ represents the covariance between the process output, $\mathbf{y}$, and the *i*'th individual, $\hat{\mathbf{y}}_i$. $\sigma_{\mathbf{y}}$ and $\sigma_{\hat{\mathbf{y}}_i}$ are there respective standard deviations. The absolute value of the correlation is used to bound it between 0 and 1.

In (2.3) the fitness will decrease as the individual becomes fitter in its environment. For convenience, the fitness is expressed as a value between 0 and 1, where 1 represents a 100% accurate description of the process and values close to 0 a very poor description. (2.3) can now be rewritten as:

$$f_i = \frac{1}{1 + \sum_{k=1}^{M} (y_k + \hat{y}_{i_k})^2} \tag{2.5}$$

to invert the relationship between the individual's fitness and the error-based criterion in (2.3). Note that in an error-based fitness function, either the sum of the squared errors (SSE) or the mean of the squared errors (MSE) is used as a means of error measurement.

2.2.2.3 Selection

Selection is the phase driven by natural selection, i.e. survival of the fittest. Those individuals who exhibit the greatest fitness are selected for mating and to contribute some of their traits (sub-trees) which will be passed on to the next generation. During selection a selection scheme is used to select N individuals to enter the mating pool. Three selection schemes are typically employed in GP:

- Fitness proportionate: The fitness of the individual is an indication of its probability

Stellenbosch University https://scholar.sun.ac.za

to be selected for reproduction. The [selection] probability of each individual is defined as $P_i = \frac{f_i}{f_T}$ where f_T is the total fitness of the current generation. Those individuals who have a higher fitness than others will constitute a larger part of the mating pool as opposed to less fit members, therefore the average population fitness increases.

- Tournament selection: Two or more individuals are randomly selected from the current population to compete against one another. The fittest individual is selected to enter the mating pool for reproduction. Normally more than two individuals will be selected to compete against one another. Too few competitors will eventually cause slow convergence while too many will facilitate premature convergence and a rapid decay in diversity. Usually three competitors are sufficient for this selection scheme.
- Rank selection: The M fittest individuals have a probability of, say, 70% for reproduction, while the remaining, N-M, only have a probability of 30%.

Fitness proportionate and Tournament selection are the two selection schemes normally used. There is no evidence as to which is the better of the two. Tournament selection, however is favoured by most researchers, since it appears to be a more natural scheme.

One of the major problems with selection is that, with increasing generations, the diversity within the population decreases. This normally leads to premature convergence.

2.2.2.4 Reproduction

After the selection phase, the selected members (after entering the mating pool) are subjected to genetic operations. A percentage, P_c, (between 50% and 90%) are selected for crossover. To maintain diversity, a small percentage P_m (between 0% and 10%) are selected for mutation. The remaining members are reproduced without change.

Stellenbosch University https://scholar.sun.ac.za

- Crossover: During crossover, two individuals are randomly selected from the mating pool. These individuals are the parents. A random node is chosen from the first parent's tree. This node represents the crossover point of the first parent. The same is done for the second parent. The two nodes and there respective sub-trees are the genetic material that is swapped between the two parents. After swapping their respective genetic material, these new members are termed the progeny or offspring of the two parents. As depicted in Figure 2.6, crossover is representative of the analogous sexual process observed in biological populations, since two individuals are involved. By swapping genetic material in this way, the vicinity of the two parents in the search space can be explored.

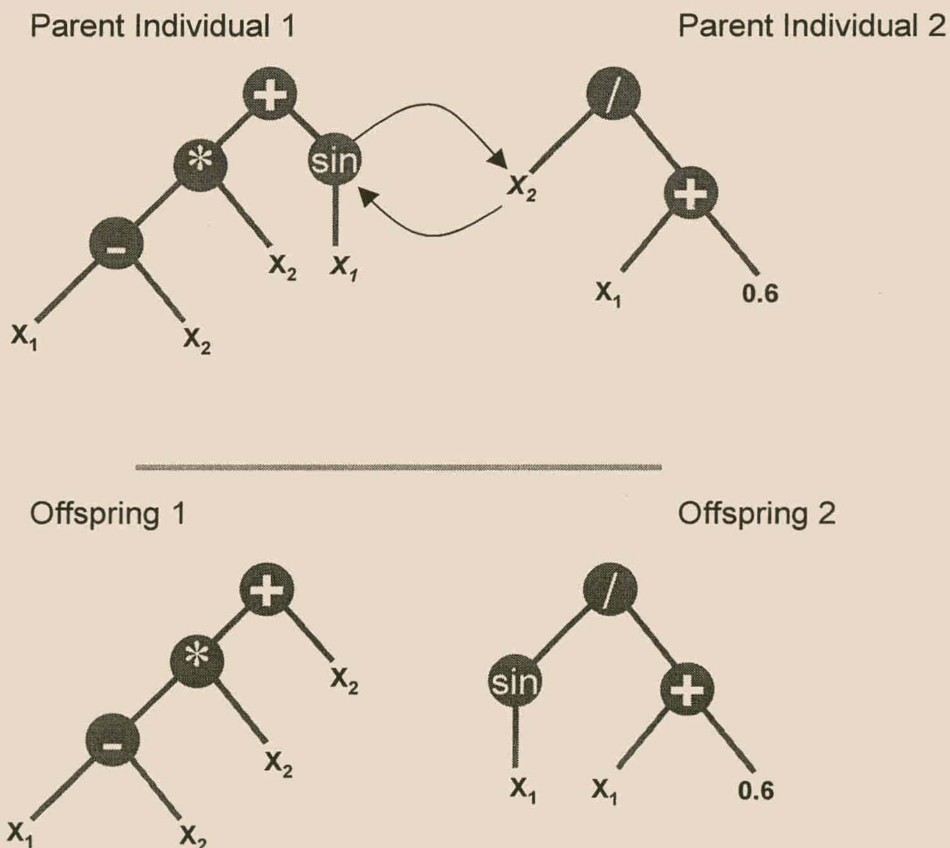


Figure 2.6 : ***A graphical depiction of the crossover operation. Two points are randomly selected on the two parents and their respective sub-trees are swapped.***

- Mutation: Mutation is mainly used to restore some lost diversity in the population

Stellenbosch University https://scholar.sun.ac.za

and acts as a random search mechanism. It proceeds as follows: one individual is selected at random from the mating pool. A node is randomly selected for mutation. Everything from the node downwards is removed and replaced with a randomly generated sub-tree. Mutation is representative of an asexual process associated with biological populations, since only one parent is involved. Figure 2.7 illustrates the mutation operation.

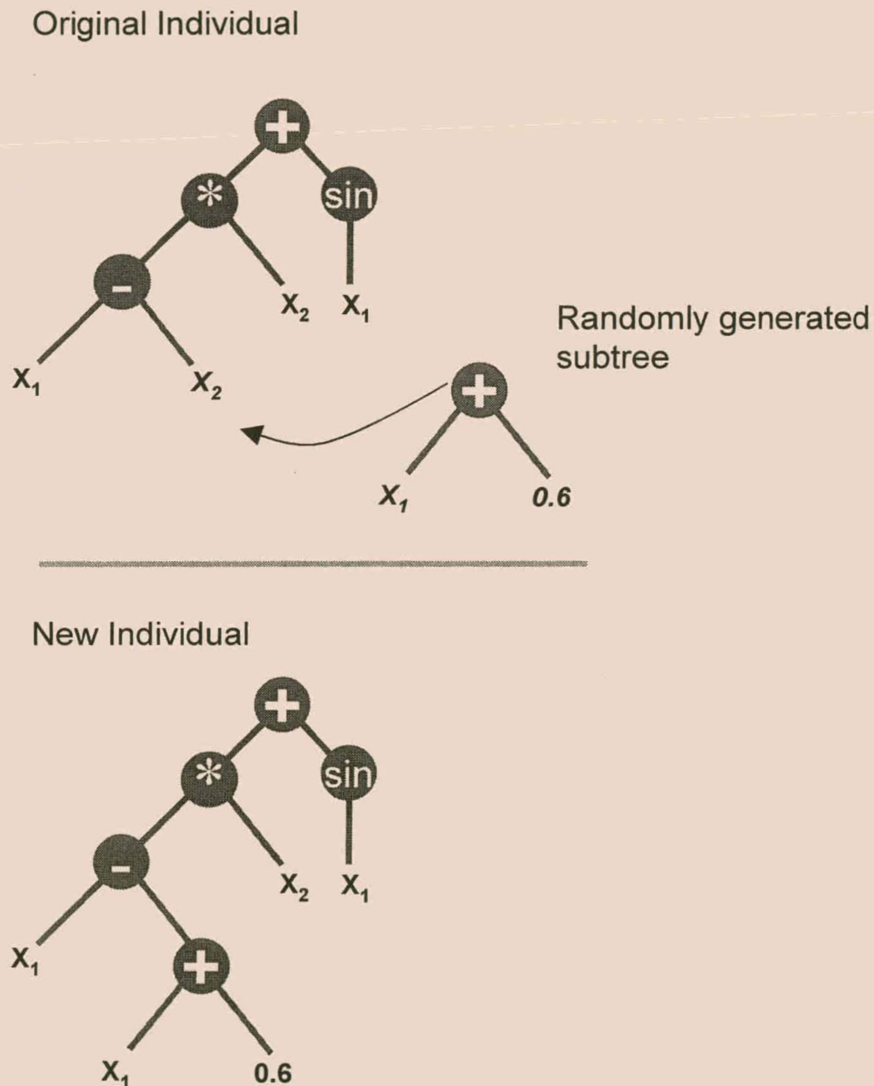


Figure 2.7 : ***A random node is selected on the parent and replaced by a randomly generated sub-tree, during mutation.***

Note, however, that crossover, which is a stochastic searching operator, is the main

Stellenbosch University https://scholar.sun.ac.za

genetic operator. Mutation plays a very small (if any!) role in GP. Both crossover and mutation are search operators, in that they allow the exploration of different parts of the search space. Increasing the search rate will automatically result in a faster search, but at an increased risk of entrapment in a local optimum. Specifying too low a rate will avoid entrapment. Unfortunately one does not have unlimited time and selecting a low search rate will be too time consuming.

2.3 Current limitations in genetic programming

There are several drawbacks in the way genetic programming algorithms are normally implemented. In the following paragraphs each weakness is considered and possible remedies are proposed.

2.3.1 Speed and resources

Since genetic programming is a population-based searching algorithm, it requires an enormous amount of resources, to the detriment of computational speed. Previous research conducted by Greeff and Aldrich (1998) confirmed this when even a PC equipped with 128MB of RAM eventually ran out of memory even though only small computational problems were considered. The way an individual is represented in the system's memory is critical in this regard. The conventional way of representing an individual (Koza, 1992), is to implement every node as a pointer in memory, hence creating a tree-like structure (in memory) or S-expression containing nodes and leaves (see Figure 2.4). Since the original algorithm was implemented in the LISP programming language[3], this implementation made sense because a LISP program is written as an S-expression. Each node is usually comprised of the following information:

- the type of node, functional or terminal (1 byte).
- a pointer (if it is a terminal) to the value of the appropriate vector in the terminal

[3]LISP is an interpreted language like BASIC. These languages are much slower than compiled languages such as C and C++, because each instruction has to be interpreted during runtime to execute the appropriate machine code.

Stellenbosch University https://scholar.sun.ac.za

list (4 bytes).

- a variable (if it is a function) to indicate which function is being called (1 byte).
- pointers to the previous, left and right node in the tree structure (12 bytes).
- other information the programmer may deem pertinent (4 bytes).

Adding all these memory requirements yield at least 22 bytes of memory per node. This technique lends itself to severe restrictions:

- memory is squandered on irrelevant information concerning a node, such as the pointer to the vector in the terminal list if the node is functional or the value of the function if the node is a terminal.
- the left, right and back pointers require additional memory storage.
- should a function node be required that uses more than two arguments, the code will need to be rewritten to accommodate this change. Instead of having only a left and right pointer, the structure will require new pointers which will eventually only confuse the programmer and increase the memory requirements.

Vast amounts of memory are required this way which degrades the performance of the algorithm.

Nordin and Banzhaf (1995) implemented the GP algorithm in pure machine code which they referred to as Compiling Genetic Programming or CGP. That is, every individual was comprised of a linear set of machine code instructions. Since each instruction was exactly 32 bits, this approach was more analogous to a genetic algorithm which consisted of chromosomes of varying lengths and genes which were made up of 32 bits each. The normal searching or genetic operators, i.e. crossover and mutation, could be applied to produce new [and valid!] machine code instructions. The main advantage to this approach was that the individuals did not need to be interpreted by the virtual machine (which one requires for the other techniques) since they are already in machine code. Nordin *et al.* (1995) reported a speed improvement by a factor 1500 - 2000 after comparing their algorithm against the traditional S-expression

Stellenbosch University https://scholar.sun.ac.za

implementation in LISP. Tacket (1994) presented a system written in the C programming language which was about 25 times faster when compared against the LISP implementation. Suffice to say that Nordin's implementation outperforms any of the other at present (which is to be expected since it was implemented in assembler!).

It should be stressed, however, that the main difference between genetic programming and compiling genetic programming is that in the former an explicit mathematical function is obtained, while the latter yields an explicit computer program as shown in Figures 2.8 and 2.9. The drawback in compiling genetic programming is that no explicit mathematical model structure is obtained and therefore the way in which each variable relates to the structure is not obvious.

```
Polish notation:
f(X) = - * sin * + x2 x1 - 5
       x1 cos + x1 x2 x1

Standard form:
f(X) = sin(5x2+5x1-x2x1-x1^2) *
       cos(x1+x2)-x1
```

Figure 2.8 : ***An explicit mathematical function evolved by GP written in both Polish and standard form.***

```
Func1(X)
begin
    x1 = sin(x2);
    x1 = x1+5;
    x2 = 4;
    i = 0;
    while(i < 5)
    begin
       x3 = x3+i*x1;
       Inc(i);
    end
end;
```

Figure 2.9 : ***An explicit computer program evolved by CGP.***

2.3.2 Limitations with genetic programming as a global searching algorithm

It has been maintained by numerous researchers that the main obstacle is not the development of a model structure, but the simultaneous fitting of parameters to this structure. Genetic programming is a global optimizing algorithm, owing to the fact that it evolves its own model structures and searches through the discrete tree-like search space for the optimal structure. It can only optimize these structures by evolving

Stellenbosch University https://scholar.sun.ac.za

complex structures to estimate any parameters.

Koza (1992) introduced his so-called *ephemeral random constants*. These were random numbers that were included in the terminal set and changed every generation. When the mutation operator was applied, it would select a new random number from the terminal set and hopefully improve the newly mutated solution. The problem here was that it was completely random and could not be optimized. To solve this problem various other searching algorithms, such as genetic algorithms (Howard and D'Angelo, 1995) and simulated annealing (Sharman *et al.*, 1995; Gray *et al.*, 1996) were used to accelerate the identification of optimal model parameters.

Searching the discrete tree-like structures and continuous parameters concurrently seemed to be embraced by researchers as a solution to this problem. However, this approach consumed a vast amount of resources, since there were two searching algorithms operating concurrently. Even though Howard and D'Angelo (1995) claimed that using this hybrid genetic algorithm-program (GA-P) actually improves on the original algorithm, careful analysis of their results proves the contrary. Of the fifteen different simulation runs they conducted only seven outperformed genetic programming by a slight margin! The problem is aggravated by the fact that the search space for the parameters can be very large. Consider for example a model with six parameters, each considered at 10 levels (as a gross simplification). This means 10^6 possible parameter combinations. By comparison, the number of model structures to be searched could be significantly smaller

2.3.3 The disruptive behavior of the crossover operation

Genetic programming blindly combines sub-trees when applying the crossover operation. This can often lead to a disruption of beneficial sub-functions in the trees. Watson and Parmee (1997[a]) introduced the concept of constrained complexity crossover or CCC to minimize the disruptive behaviour of crossover. Using this technique they assigned a weight factor to each node in the tree-like structure. All terminal nodes were assigned a value of 1.0 while function weights varied from 1.1 to

Stellenbosch University https://scholar.sun.ac.za

1.x, where x is directly related to the complexity of the node. The more complex the function the higher its weight.

To compute the complexity of a specific node (the so-called node complexity or NC) each node complexity is expressed as a function of the node complexity values below it and the weighting of that node. The complexity of the tree therefore decreases with tree depth, while the root node has the highest node complexity. When the crossover operator is applied, it is constrained by only applying it to sub-trees with node complexity values that fall within a similar range. This ensures that the crossover operation is not overly disruptive when it swaps sub-trees between individuals. Watson and Parmee found that by using this technique, smaller population sizes were needed than with the traditional GP implementation, which greatly reduced memory requirements. However, the crossover operator required extra administration to find sub-trees with node complexity values within the required range.

Angeline (1997) introduced two forms of macro mutation, originally conceived by Jones (1995) that were mechanically identical to sub-tree crossover, *viz. strong headless chicken crossover* (SHCC) and *weak headless chicken crossover* (WHCC). When SHCC was applied, two parents were randomly selected from the population. For each parent a random tree was constructed to mate with. Once the random parent was constructed, standard crossover was performed on the given parent and its corresponding randomly generated counterpart. The operation was then repeated for the second parent and its corresponding random counterpart. After this operation both modified *parent* trees were returned as the offspring. The modified random trees were discarded. The redeeming feature of the offspring stemmed from the fact that they contained some randomly generated material.

WHCC proceeds exactly like SHCC, except that it has an even probability of returning either the modified parent or the modified random tree. This operation was considered weak, since for half of the offspring, a small amount of non-random material was returned to the population.

Angeline compared *strong headless chicken crossover* and *weak headless chicken*

Stellenbosch University https://scholar.sun.ac.za

crossover against standard crossover on three data sets, as described in Koza (1992). The first data set consisted of an intertwined spiral and the aim was to separate the two spirals. The second data set consisted of data collected from the average number of sunspots observed for each month since the year 1700. Here the aim was to predict when the next sunspot would occur. The third data set was the Boolean 6-bit even parity problem. The objective of the problem was to induce a function that returns TRUE when an even number of terminals are TRUE and FALSE otherwise. Angeline's results indicated that standard crossover outperformed both SHCC and WHCC by a slight margin on the spiral and the 6-bit even parity problem. However, it did significantly worse on the sunspot modelling problem.

2.3.4 Exploring large search spaces

A key concern in genetic programming is (1) the size of the search space which must be searched and (2) the number of invalid programs (due to type mixing) that are produced during initialization and the application of the genetic operators. Even for small terminal sets, function sets and tree depths, search spaces of the order 10^{30} - 10^{40} is not uncommon (Montana, 1994). One method to reduce the size of the search space is to use strongly typed genetic programming (STGP) (Montana, 1994). Montana maintained a table giving the types of all available terminals and functions. If a function requires its arguments to be of type X, then this implicitly constrained its offspring to produce a value of type X. A second table provides type constraints (or type possibilities) according to the depth in the tree where type matching occurs. This extra information constrains the choice of functions to create nodes in the tree to ensure that the tree can grow to its maximum depth. During the initialization phase of the population (and during crossover and mutation), each parse tree is grown top-down by choosing functions and terminals at random within the constraints of the types in the table. In this way the initial and subsequent populations only consists of parse trees that are type correct. Strongly typed genetic programming utilizes the structuring of the genetic programming S-expressions to reduce the search space. Haynes *et al.* (1995) used this technique to develop a multiple cooperating agent system where numerous agents

Stellenbosch University https://scholar.sun.ac.za

cooperated to hunt and track down prey. Their results indicated that strongly typed genetic programming was able to evolve programs with a higher success rate than genetic programming. Also solutions were generated in a fewer number of generations than those obtained by genetic programming.

2.3.5 Restraining premature convergence

Like all other searching algorithms, genetic programming tends to converge too fast. The reason for this is that with selection the diversity decreases, because the next generation will have duplicates of the best individuals. Selection is a necessity in order to improve the overall fitness of the population. Selecting individuals by only looking at their fitness is a sure way of guaranteeing entrapment in local optima. It has been proposed (De Jong, 1975) that to ensure diversity in a genetic algorithm, the Hamming distance between chromosomes be used. Since a GA consists of a fixed length chromosome consisting of genes, each gene can be visualized as a vector in a *n*-dimensional hyper-space; where *n* equals the number of genes. By selecting individuals with Hamming distances (that fall within a similar range) for mating, one is ensured of diversity. Another possibility is to distribute several populations (the so-called *distributed genetic algorithm*) over several processors and assign each population to a separate processor (Tanese, 1989). However, this approach requires an enormous amount of expensive hardware which is not economically viable.

Although the concept of using a Hamming distance will work [in GA's] the problem arises when one tries to express the distance between two individuals in genetic programming. Because each individual is a tree-like structure it becomes impossible to determine the interspatial distance between them. Also an individual is not unique in the sense that there is more than one way of representing a simplified tree. Therefore even though one may have two trees that *appear* dissimilar, when simplified, they are exactly the same and occupy the same position in the search space. Koza (1992), Langdon (1995) and Winkeler and Manjurath (1997) made use of a parallel genetic program which consists of several populations running in series. These populations or

Stellenbosch University https://scholar.sun.ac.za

demes reduced premature convergence considerably. During crossover, genetic material is only shared amongst members of the same deme. This kept the code from becoming too tightly focussed in a particular area of the search space and subsequently reduced premature convergence.

2.3.6 Discussion of the current remedies

From sections 2.4.1 to 2.4.5 the reader was introduced to several of the limitations facing genetic programming and some of the suggested remedies. Implementing any algorithm, especially a population based algorithm, in an interpreted language such as LISP is *not* advisable. Interpreted languages are simply too slow and cannot handle extensive usage of floating point arithmetic. Although fast, machine language implementations tend to be restricted to specific hardware platforms. Also from a design point of view, machine language programs do not have the scalable properties of an object oriented programming (OOP) language, such as C++, SmallTalk or Object Pascal. Therefore future extensions to the algorithm becomes a formidable task.

Although several suggestions have been proposed to create some form of local optimization in genetic programming, none of these attempts have been wholly successful. Using two population based searching algorithms concurrently does not yield any significant improvement; it only degrades the available resources.

Although several suggestions have been proposed to minimize the disruptive behaviour of the crossover operation, none of these solutions have been successful. Enforcing strict type checking via strongly typed genetic programming, forces the algorithm to evolve syntactical and type correct programs. This in effect shrinks the search space which results in higher convergence.

By using sub-populations and allowing only interaction amongst members of the same sub-population, one is ensured of a means to reduce premature convergence. Unfortunately, genetic programming does not allow simple (Hamming) distance

Stellenbosch University https://scholar.sun.ac.za

calculation between individuals, therefore one does not have the advantages of a GA to allow individuals who are in close proximity, to mate.

2.4 Applications of genetic programming

2.4.1 Robotic control

Koza (1992) further proposed usage of automatically defined functions (ADF). These are subtrees of a genetic program which are randomly selected and incorporated into the function set for reuse. These functions form a library of *possible* useful utilities which then may be selected during the mutation operation. Hondo *et al.* (1997) used this technique to generate programs for robotic control. Andre (1995) used automatically defined functions to create an intelligent agent which could collect gold placed on random locations on an $n \times n$ grid of squares. Langdon (1995) used automatically defined functions to create simple abstract data structures, namely a circular data queue and an integer stack via genetic programming. Each data structure was implemented by five independent, cooperating procedures. Each procedure was represented as an independent tree within the same individual. Thus each individual was comprised of five parse trees or S-expressions. Langdon showed that the abstract data structures could be successfully evolved. When the data primitives, such as the appropriate increment and decrement operations were omitted, automatically defined functions could successfully evolve these routines, although it took much more effort than when the primitives were included.

2.4.2 Image analysis and feature extraction

Recently work has been conducted on employing feature extraction and image analysis via genetic programming. One of the main obstacles in image analysis is the size of the data sets (up to 1024 x 1024) and the fact that genetic programming is a population based searching algorithm. As such, this implies vast amounts of processing

Stellenbosch University https://scholar.sun.ac.za

power and memory demands, which tend to make population based algorithms unfavourable for these tasks. One way of overcoming this obstacle is to process only a small portion of the image (say 32 x 32). Daida *et al.* (1996) used genetic programming to extract pressure-ridge and rubble features from multiyear-ice signatures. There results showed that the algorithm performed well with a low-resolution European remote sensing satellite (ERS) synthetic aperture radar (SAR) data products. Teller and Velosa (1995) utilized genetic programming for image recognition in which they classified various human faces. They demonstrated that genetic programming can generate programs which can correctly recognize different faces. Genetic programming has also been used for object detection (Winkeler and Manjunath, 1997) by first performing an experiment which extracted statistical features from images to ascertain whether the image was a face and then processing gray-scale images to locate faces. Although the training was expensive, the first experiment (classifying images by extracting features) did well at locating a specific scale of face (i.e. faces of more or less the same size), while the second experiment (classifying faces from gray-scale images) could locate faces at all scales based solely on intensity, but exhibited an undesirable number of misclassifications.

2.4.3 Language learning applications

In the field of grammatically-based learning systems, Whigham (1995) used genetic programming to map each sentence to a fitness value. Each individual tree structure was associated with these sentences to define the structure of the schemata. The simple genetic operators, crossover and mutation, were then applied to evolve new and grammatically correct sentences.

2.4.4 Evolving controllers for systems

Several researches have used genetic programming to evolve controllers for systems. Gritz and Hahn (1997) used genetic programming to evolve controllers for 3-D

Stellenbosch University https://scholar.sun.ac.za

character animation. Although the initial training was extremely time consuming, the resulting motion was fluid, physically and biologically believable; and often appeared to be very organic. Dracopoulos (1997) applied genetic programming to a highly nonlinear control problem, the attitude control problem for satellites. The satellite was detumbled and controlled by a control law evolved by genetic programming. Simulations seemed to show that the control law could stabilize the system for different initial conditions. Dracopoulos proved the theoretical stability of the control algorithm found by genetic programming by utilizing the classical theory of Lyapunov functions.

2.4.5 Process modelling

During the past few years some work had been conducted in the modelling of industrial processes via evolutionary computations (Watson and Parmee, 1997[a]; Watson and Parmee, 1997[b]; Greeff and Aldrich, 1998; Mackay *et al.*, 1997; Kulkarni *et al.* 1999). Genetic programming, however, is capable of finding solutions to relatively small problems only, or alternatively, it has to be compromised to allow it to deal with large problems. Mackay *et al.* (1997) used genetic programming to develop (1) a model to infer the bottom product composition of a binary vacuum distillation column and (2) a model of a continuous stirred tank reactor system. They used the standard genetic programming algorithm [as proposed by Koza] combined with a Levenberg-Marquardt method of least squares optimisation to optimise the model constants. There results revealed that in each case genetic programming was able to generate an accurate input-output model based solely on observed data. The identified structures, however, did not provide detailed phenomenological information regarding the system being modelled.

Greeff and Aldrich (1998) attempted to model the acid pressure leaching of nickeliferous chromites. This process has previously been investigated by Das *et al.* (1995) for which they derived quadratic regression equations for nickel, cobalt and iron dissolutions. The evolved model for nickel and cobalt had an accuracy similar to the regression models of Das *et al.* (1995). The evolved models were significantly more

Stellenbosch University https://scholar.sun.ac.za

accurate in the case of the leaching of iron.

Hiden *et al.* (1997) extended the genetic programming algorithm to model dynamic systems. By using the input data, **u** and approximating the objective function F(**u**) by

$$F(\mathbf{u}) = \sum_{j=1}^{k} b_j \, g_j(\mathbf{u}) \tag{2.6}$$

a linear combination of functions $g_j(\mathbf{u})$, $j = 1 \ldots k$, such that

$g_j(\mathbf{u})$ was then evolved via genetic programming. The process dynamics was approximated by a simple first order Laplace transform. In order to model non-linear process dynamics, (2.6) was augmented with the Laplace transform to give the following non-linear dynamic model.

$$F'(\mathbf{u}) = \sum_{j=1}^{k} \frac{b_j \, g_j(\mathbf{u})}{\tau_j s + 1} \tag{2.7}$$

where τ_j are the model time constants and '*s*' the Laplace operator. Here the genetic programming algorithm was used to determine the non-linear functions $g_1(\mathbf{u})$, $g_2(\mathbf{u})$, ..., $g_k(\mathbf{u})$ and the values of the time constants τ_1, τ_2,..., τ_k while the coefficients b_1, b_2, ..., b_k were determined using batch least squares. Hiden *et al.* (1997) applied their algorithm to the modelling of a plasticating extruder. Their results indicated that models obtained with genetic programming are as accurate as those using a neural network with the additional benefit of being easy to analyse and interpret.

2.5 *Objectives of this study*

2.5.1 Motivation for this research

Genetic programming is a very new and rapidly expanding field in computational

Stellenbosch University https://scholar.sun.ac.za

intelligence. It's main advantage is its ability to evolve symbolic structures which makes it ideally suited for virtually any application. The very fact that it does not require any encoding schemes or structural information means that it truly takes a "black box" approach towards problem solving. From section 2.5 it is apparent that genetic programming is a very versatile tool. It is this author's opinion that genetic programming will eventually be seen as a viable alternative for artificial neural networks and as such more extensive research needs to be done.

This research is concerned with finding solutions to overcome some of the obstacles in genetic programming. That is

- looking at ways of speeding up the algorithm.
- improving memory management and hence reduce the substantial amount of resources required by the current implementation of the genetic programming algorithm.
- and increasing convergence and robustness for improved performance.

Additionally this technique is applied to mineral processing for process modelling (Chapter 4) and symbolic feature extraction or dimensionality reduction (Chapter 5). Both of these fields have been investigated extensively via other techniques such as neural networks, linear regression, gradient descent, etc. Also, the results obtained are compared with the abovementioned algorithms which are used as benchmarks.

2.5.2 Outline of the chapters in this thesis

Chapter 1 introduces the reader to explorative data analysis and empirical modelling and its importance in today's world. Some of the major obstacles encountered within these disciplines are highlighted. The objectives regarding this research are also presented.

Chapter 2 provides an introduction to the concepts of evolutionary algorithms and looks at the development of genetic programming and the limitations [and current remedies]

Stellenbosch University https://scholar.sun.ac.za

facing this novel algorithm. Additionally the reader is presented with a wide range of applications that use the genetic programming algorithm.

Chapter 3 focusses on new augmentations incorporated in the genetic programming algorithm to improve efficacy in terms, memory demand, execution speed and improved convergence and robustness of regression models. The reader is also introduced to some object oriented programming (OOP) terminology and why it was used to develop the genetic programming kernel.

In *Chapter 4* the augmented algorithm, a-GP, is used to model industrial processes via steady-state modelling and the results are compared against those obtained with standard genetic programming, linear regression and a multilayer perceptron neural network.

In *Chapter 5*, a-GP is used for feature extraction or dimensionality reduction on several bench marking data sets and two industrial processes. The results obtained from the bench marking data sets are compared against those obtained via artificial neural networks and other algorithms.

Chapter 6 presents the reader with ideas for future research while the results and conclusions are discussed in *Chapter 7*.

Stellenbosch University https://scholar.sun.ac.za

CHAPTER 3

THE DESIGN METHODOLOGY

3.1 Taking an object oriented approach towards designing the genetic programming kernel

Object oriented programming (OOP) is a design philosophy in its own right. The main difference between OOP and structured programming is that the former tries to represent any solution as a closed object which encapsulates the states (variables) and methods (functions) through which we alter the behaviour of an object, while the latter takes a top-down approach towards solving problems. Object oriented programming, however, has the ability to inherit the properties of an object and derive a new object which can have new implementations and ultimately change the behaviour of the derived object. This ability to inherit and override previous implementations by using the same interfaces presents tremendous benefits for programmers. For one, it reduces the amount of code writing and debugging considerably. Secondly, a properly designed class (or blueprint of an object) is highly scalable; something which cannot be readily achieved with structured programming. Some of the terminology used here may seem foreign to readers who are not familiar with object oriented programming. Although the author has gone to great lengths to explain the terminology, this thesis is *not* an introduction to object oriented programming. That is beyond the scope of this thesis. The interested reader is referred to Tom Swan's: Using Borland C++ 4.5 and other books on C++ to help him/her come to grips with object oriented programming.

Stellenbosch University https://scholar.sun.ac.za

3.1.1 Designing the abstract base class

In designing the base class we start off by looking for common properties in our algorithm. This process is called abstraction. Any population-based algorithm has :

- A solution. How the solution is implemented varies from algorithm to algorithm. For instance: an artificial neural network's solution consists of neurons, weights and activation functions; a linear regression model has parameters, a_i, while a genetic programming individual is a tree-like structure comprised of nodes and leaves. We therefore opt for an open implementation, which we will specify in another derived class.
- A fitness. The solution's ability to solve the problem at hand is rated by this floating point value.
- A population of solutions. We simply use a vector (of unspecified length) to store each solution.
- A method to initialize the population. All population-based algorithms have the same initialization procedure which is summarized in Figure 3.1. Note that the function itself is very generic in the sense that both the `CreateSolution` and

```
Procedure Initialize( Population, PopulationSize )
Begin
      For i=0 to PopulationSize-1
      Begin
         CreateSolution(Dummy);
         ComputeFitness(Dummy);
         AddToPopulation(Population, Dummy);
      End;
      ComputeTotalFitness(Population);
      CurrentStep = 0;
End.
```

Figure 3.1 : ***A generic initialization algorithm.***

Stellenbosch University https://scholar.sun.ac.za

`ComputeFitness` methods are pure virtual[1] methods. They are implemented in the final derived class for each respective algorithm, and

- A variable `CurrentStep` keeping track of the current iteration number, because population-based algorithms depend on some form of iteration.

The reader is referred to Appendix B.1 for a complete listing of the abstract base class `GenericAlgorithm`.

3.1.2 Designing the abstract evolutionary algorithm class

All evolutionary algorithms undergo natural selection and have the ability to evolve via mutation and/or crossover. To incorporate these methods into the algorithm one first have to derive a new class from the base class in section 3.1.1 and define new methods. Once again some of these methods will be pure virtual because it depends on the particular algorithm[2] how they will be implemented.

Figure 3.2 shows the `DoAlgorithm` function. `Reproduction` is implemented in this

```
Procedure DoAlgorithm()
Begin
      while(NOT(Paused))
      Begin
           Reproduction(PopulationSize, SelectionMethod,
      TournamentMembers);
           GeneticOperations(Pc, Pm, PopulationSize);
           CurrentStep = CurrentStep+1;
      End;
End.
```

Figure 3.2 : ***The generic DoAlgorithm() method. Note the method: GeneticOperations is defined as pure virtual. It's actual behaviour depends on the class in which it is implemented.***

[1]A virtual method is a function which has the same name and takes the same arguments in all classes derived from the class where the method was defined. Their implementations, however, differ. A pure virtual method has no implementation.

[2]Crossover and mutation in genetic programming is implemented different than in genetic algorithms.

Stellenbosch University https://scholar.sun.ac.za

class because it is not related to any specific evolutionary algorithm. All this function does is to fill the mating pool (or gene pool for a genetic algorithm) with the fittest members of the current generation by using a predefined selection scheme (Fitness proportionate, Tournament or Rank selection). `GeneticOperations` is defined as pure virtual. The reader is referred to Appendix B.1 for a complete listing of the abstract class `GenericEvolutionaryAlgorithm`.

3.1.3 Designing the genetic programming class

To start designing the final genetic programming class we first need to derive a class from the abstract class `GenericEvolutionaryAlgorithm`. This class will have an open implementation for both the `CreateSolution` and `ComputeFitness` methods. The reason for this will become apparent shortly. Finally an implementation is provided here for `GeneticOperations`. This new abstract class is called `CustomGPAlgorithm`. From this parent class we derive two important classes. The one is `GPSupervised`, which will be used for supervised training, i.e. for algorithms which have variables which we can denote as outputs. The other is a pseudo-abstract class GPUnsupervised which will be used for unsupervised training, i.e. for algorithms where there are no variables which we can denote as outputs. Both of these classes have their own implementation for the `CreateSolution` and `ComputeFitness` methods. The reader is referred to Appendix B.2 for a complete listing of these classes.

3.1.4 Designing the feature extraction class

This class is derived from the `GPUnsupervised` class, since feature extraction is essentially an unsupervised training problem (see Chapter 5). The `ComputeFitness` method needs to be overridden and given a new implementation. The reader is referred to Appendix B.2 for a complete listing of the class `FeatureExtract`.

Stellenbosch University https://scholar.sun.ac.za

3.1.5 Graphical overview of the base class and all its derived descendants

Figure 3.3 presents a graphical overview of the abstract base class and its derived descendants. Note how the hierarchy splits in two after the `CustomGPAlgorithm` class. `GPSupervised` is used for regression, since it is the final class in this hierarchy for supervised training. `FeatureExtract` is derived from `GPUnsupervised` because it uses a different implementation for evaluating the fitness

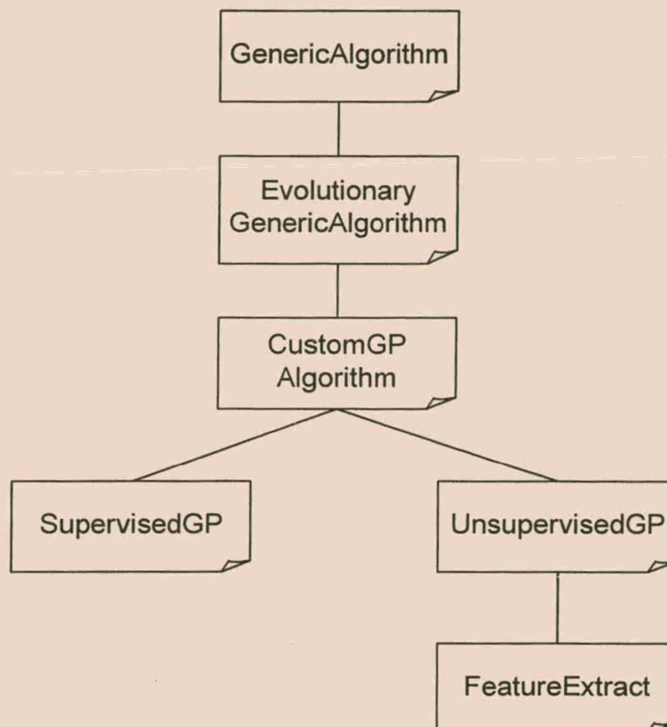


Figure 3.3 : ***A graphical overview of all the classes and their decedents.***

of a solution. One can clearly see that by using an object oriented programming methodology very complex behaviour can be programmed in the minimum time. Also, to maintain an algorithm such as this is much easier than one that is written in a structured programming language.

Stellenbosch University https://scholar.sun.ac.za

3.2 *Probing the size of the search space*

By definition, the search space is the set of all possible individuals that can be constructed of the elements in the function set and terminal set while the solution space (which is a sub-set of the search space) is the set of all good or perfect solutions to the problem at hand (McKay *et al.*, 1997). The search space is constrained by the limit imposed on the maximum number of levels an individual can assume. To determine the effective size of the search space one proceeds as follows:

Let the terminal set be represented by $\boldsymbol{T}$ and the function set by $\boldsymbol{F}$. The terminal set can be expressed as

$$\boldsymbol{T} = \{\, T_i \mid i = 1..k \,\} \tag{3.1}$$

Similarly, for the function set

$$\boldsymbol{F} = \{\, F_j \mid j = 1..m \,\} \tag{3.2}$$

The size (**S**) of the search space, at a level l, is given by $\mathbf{S}(l)$. Clearly for l=1

$$\mathbf{S}(1) = k \tag{3.3}$$

where k equals the number of terminals in the terminal set.

Note

For an individual which consists of one node, the search space is as large as the total number of terminals, k, in the terminal set.

For l = 2, we consider the case where the function set consists of the following functions, $\boldsymbol{F}$ = {+, /, sin} and the terminal set has two terminals, $\boldsymbol{T} = \{t_1, t_2\}$. If the root node consists of the function '+' then the total number of individuals, $\gamma(F_i)$, that can be constructed with two terminals and function F_i, are

Stellenbosch University https://scholar.sun.ac.za

$$\gamma(F_1 = '+') = \sum_{i=0}^{k-1}(k-i) = \frac{k(k+1)}{2} = \frac{2(3)}{2} = 3 \tag{3.4}$$

However, if the root node is "/" then (3.4) becomes

$$\gamma(F_2 = '/') = k^2 = 2^2 = 4 \tag{3.5}$$

and using a root node of "sin"

$$\gamma(F_3 = '\sin') = k = 2 \tag{3.6}$$

The reason why equations 3.4 and 3.5 differ is that

$$t_1 + t_2 = t_2 + t_1 \tag{3.7}$$

but

$$t_1 / t_2 \neq t_2 / t_1 \tag{3.8}$$

Likewise, (3.4) and (3.5) are also applicable to the functions "*" and "-", respectively.

For an arbitrary number of levels (3.4) changes to

$$\gamma(F_1 = '+') = \sum_{i=0}^{\mathbf{S}(l-1)-1}(\mathbf{S}(l-1) - i) = \frac{\mathbf{S}(l-1)(\mathbf{S}(l-1)+1)}{2} \tag{3.9}$$

While (3.5) becomes

$$\gamma(F_2 = '/') = \mathbf{S}(l-1)^2 \tag{3.10}$$

and (3.6)

$$\gamma(F_3 = '\sin') = \mathbf{S}(l-1) \tag{3.11}$$

Where **S**(*l* - 1) refers to the effective size of the search space of the previous level.

Taken together

Stellenbosch University https://scholar.sun.ac.za

$$\gamma(F_j) = \begin{cases} \frac{\mathbf{S}(l-1)(\mathbf{S}(l-1)+1)}{2} & \text{if } F_j \in \{+,*\} \\ \mathbf{S}(l-1)^2 & \text{if } F_j \in \{-,/\} \\ \mathbf{S}(l-1) & \text{if } F_j \in \{\text{all single argument functions}\} \end{cases} \tag{3.12}$$

For $l = 1$

$$\mathbf{S}(l) = \mathbf{S}(1) = k \tag{3.13}$$

and for $l > 1$

$$\mathbf{S}(l) = \sum_{j=1}^{m} \gamma(F_j) + k \tag{3.14}$$

3.3 *Augmentations to improve the original genetic programming algorithm*

3.3.1 Changing the internal representation of an individual in genetic programming

Genetic programming consumes a vast amount of resources. In this investigation a novel approach was pursued to minimize the memory requirements of an individual. A different representation was used to store the individual in memory. The individual was stored as a Polish expression, as in (2.1). This enabled the storage of the whole expression as an array of characters which, in effect, is equivalent to encapsulate a node in one byte or character. This constituted a significant improvement on a previous implementation (Greeff and Aldrich, 1998), that used about 22 bytes of memory per node. This new implementation requires only 1 byte per node resulting in a 95% reduction in memory usage, which eventually allows faster computation time.

To represent terminals and functions via a single byte requires some new approaches in designing the genetic programming kernel. Since a byte can address 256 unique values, i.e. 0 to 255, it has to be divided to represent either a function or a terminal. To

Stellenbosch University https://scholar.sun.ac.za

accomplish this the first half (0 to 127) of the byte, is allocated for terminals, while the second half (128 to 255) is used for functions. That is, each value of the byte in the range 1 to 127 uniquely identifies a terminal[3]. One drawback to this approach is that one is only permitted to have a maximum of 127 terminals! Similarly, the remaining 128 positions, from 128 to 255, uniquely map to 128 possible functions. Each terminal and function is stored in a *terminal list* and a *function list*. The terminal list contains information regarding each terminal. These are

- The terminal ID, which ranges from 1 to 127.
- The pointer to the memory block where the values of the terminal is stored in memory. The terminal has to be a column vector, i.e. an (n x 1) matrix. This pointer is called the terminal pointer.
- An indicator to specify whether the terminal acts as an input or an output of a process[4].

The function list contains

- The function ID, which ranges from 128 to 255.
- The number of arguments the function requires.

Once all the functions and terminals have been selected, the process of constructing an individual commences. To avoid complexity and constrain the size of the search space, a limit has to be set to the number of levels each individual can have. This limit is usually set to thirteen. Using a limit of thirteen allows each individual to have a maximum of 2^{13}-1 or 8191 nodes. Each level adds an exponential increase in the maximum number of nodes and size of the search space. Increasing the search space results in slower convergence. Assuming a function set of $\boldsymbol{F}$ = {+, -, *, sigmoidal} and a terminal set of $\boldsymbol{T} = \{\mathbf{x}_1, \mathbf{x}_2\}$ and using (3.14), the effective size (for a thirteen level parse tree) of the search space is calculated as

[3]The value 0 is not used to represent a terminal. Instead it is used by C++ to signify the end of an array of characters.

[4]This feature is only used for regression and not feature extraction.

Stellenbosch University https://scholar.sun.ac.za

$$S(13) = 9.4 \times 10^{2996} \quad (3.15)$$

Needless to say the search space gets very large and grows exponentially for every function included in the function set!

To understand how an individual is constructed, the following piece (see Figure 3.4) of pseudocode is used to illustrates the process. Two parameters are passed to the

```
Procedure CreateIndividual(Individual, CurrentLevel)
Begin
   TotalArguments=0;
   TypeOfNode=random(1);            {A value of either 1 or 0}
   CurrentLevel = CurrentLevel+1;
   If(CurrentLevel == MaximumLevel) Then
      {Select a random terminal from the terminal list}
      NewNode=SelectedRandom_TerminalID();
   Else
   Begin
   If(TypeOfNode == 1) Then
      {Select a random function from the function list}
      NewNode=SelectedRandom_FunctionID();
      {Get the number of arguments required for this function}
      TotalArguments= GetArgumentCount(NewNode);
   Else
      {Select a random terminal from the terminal list.}
      NewNode=SelectedRandom_TerminalID();
   End;
   Append(Individual, NewNode) {Add the new node to the current
expression}
   For I=0 To TotalArguments-1
   Begin
      {each argument of the function}
      CreateIndividual(Individual, CurrentLevel);
   End;
```

Figure 3.4 : ***A recursive procedure that generates a genetic programming tree-like structure. This structure may be used as either an individual or as a randomly generated sub-tree during mutation.***

Stellenbosch University https://scholar.sun.ac.za

function. The one parameter is a reference to the individual that is to be constructed and the other is a variable that keeps track of the number of levels the individual has at its current insertion point. First the current level is incremented and then compared against the maximum number of levels the individual can assume; to ascertain whether it equals this value. If the result true, a terminal is randomly selected and appended to the individual. At this stage the algorithm exits the function and returns to the caller function (which happens to be itself[5]), if false a random number between 0 and 1 is generated. If the result is 1, the new node will be a random member of the function set, else it is a randomly selected terminal. If a function was selected, the number of arguments the function needs is obtained from the terminal list by using the appropriate function ID. A loop is constructed, which ranges from zero to the number of arguments minus one. Each time the loop is executed the function is called again with the new individual and the current level as arguments. This recursive process continues until each branch in the individual has been terminated by a terminal.

3.3.2 A different evaluation scheme

Evaluating a tree hierarchically from top to bottom is a very slow process. To evaluate a tree, one starts at the root node and traverses along the left most branch of the tree until a terminal is reached. If the corresponding right branch also has a terminal on the same level as the current terminal an operation is executed on the two terminals and the result is placed in a temporary storage facility. This process of node-branch simplification continues until the whole tree has been reduced to a single node which yields the final answer. Needless to say that this is *the* part of the algorithm that consumes most of the resources. Also, extensive use is being made of floating point arithmetic that generally degrades performance even further. The evaluation can be accelerated and simplified if the individual is evaluated as a *reverse* Polish expression. In Polish notation, expressions are characterized by a function followed by its

[5]The creation process uses recursion which means that the algorithm calls itself until some termination criterion is reached. Although this may seem unnecessarily complicated, recursion actually makes the whole creation process very simple!

Stellenbosch University https://scholar.sun.ac.za

arguments. In reverse Polish notation[6] the arguments are followed by the function. In evaluating a reverse Polish expressions as in Table 3.1, we make use of a stack[7]. We

Table 3.1 : ***Illustrating the difference between Standard, Polish and reverse Polish notation.***

Standard notation	Polish notation	Reverse Polish notation
5+6*7	+ 5 * 6 7	7 6 * 5 +

first start by pushing 7 onto the stack. This is followed by pushing 6. When we arrive at the "*" sign we pop two[8] values from the stack, i.e. 6 and 7, apply the corresponding function (we multiply 6 by 7) and push the result onto the stack. Note, there is now only one value on the stack, 42.

Proceeding, the value 5 is pushed onto the stack followed by two pops when the next function, "+", is reached. Adding the two recently popped values, 5 and 42, we obtain 47. This is the final result and is returned by the evaluation function. The following piece of pseudocode illustrates this operation:

[6]The HP Scientific calculators use RPN (reverse Polish notation) to evaluate an expression.

[7]A stack is an array of values. To insert a value in the stack we *push* it. To retrieve the most recent value we pushed on the stack we *pop* it from the stack.

[8]We need to pop two values from the stack because the multiplication function requires two arguments.

Stellenbosch University https://scholar.sun.ac.za


```
Function Evaluate(Individual)
Begin
          K=Total_number_of_nodes_in_the_individual
          For i=K DownTo 0 {Start at the back and move to the front}
          Begin
             {If the current node is a terminal}
             If(Individual[i]<=127)
             Begin
                 TerminalID=Individual[i];
                 {now push the appropriate Terminal pointer on the stack}
                 push(GetTerminalPointer(TerminalID));
             End
          Else
             {the current node is a function}
             Begin
                 FunctionID=Individual[i];
                 {Do the appropriate function specified by FunctionID}
                 {And push the result on the stack}
                 push(ApplyFunction(FunctionID));
             End;
          End;

      {And finally pop the last value off the stack and return it}
      Return pop();
End.
```

Figure 3.5 : ***The pseudocode for the evaluation function.***

3.3.3 Increasing convergence and robustness in regression models using an expanded solution space

3.3.3.1 Fitness function

A solution's fitness is a measure of how accurately it approximates the desired state or optimum solution. An error-based fitness function's measure is usually based on the sum of the squared errors (SSE) or mean of the squared errors (MSE) between the desired state and the solution's approximation to that state. Error-based fitness

Stellenbosch University https://scholar.sun.ac.za

functions such as these are used widely in regression problems (Iba *et al.*, 1995; Koza, 1992; Tang *et al.*, 1996; Watson and Parmee, 1997).

South *et al.* (1995) noted that using a correlation-based fitness function improves convergence speed. A correlation-based fitness describes the fitness as the correlation between the desired state and the solution's approximation. Pearson's correlation coefficient is used as the actual fitness. The correlation ranges from -1 to 1, where -1 implies [perfect] negative correlation, 0 indicates no correlation and 1 perfect correlation. If the absolute value of the correlation is used to bound it between 0 and 1, it can serve as the fitness value.

3.3.3.2 Correlation

Correlation is a measure of the linear association between two random variables *X* and *Y* and is given by the population correlation coefficient, ρ, where

$$\rho_{XY} = \frac{\mathrm{cov}(X,Y)}{\sigma_X \sigma_Y} \tag{3.16}$$

Since μ_X, μ_Y, σ_X and σ_Y are usually unknown, ρ_{XY} can be estimated by the sample correlation coefficient[9] r_{XY}, where

$$r_{XY} = \frac{\sum_{i=1}^{n}(x_i - \mathrm{E}[X])(y - \mathrm{E}[Y])}{\sum_{i=1}^{n}(x_i - \mathrm{E}[X]).\sum_{i=1}^{n}(y_i - \mathrm{E}[Y])} \tag{3.17}$$

Each variable consists of *n* observations. E[*X*] and E[*Y*] denote the estimated or mean of variables *X* and *Y* respectively. The sum of the squared errors or SSE between two random variables *X* and *Y* is

[9] Also referred to as the Pearson correlation coefficient.

Stellenbosch University https://scholar.sun.ac.za

$$SSE_{XY} = \sum_{i=1}^{n} (x_i - y_i)^2 \qquad (3.18)$$

Consider a variable *Y*, as a linear function of *X*, such that

$$Y = aX + b \qquad (3.19)$$

and *a*, *b* ∈ ℜ. If *a* = 1 then *b* is simply a bias added to *X* which transforms it by a constant value of *b*. If *b* = 0 then *a* is a scale which expands or shrinks *X*. Substituting (3.18) in (3.16) and (3.17) yields

$$r_{XY} = 1 \qquad (3.20)$$

and

$$SSE_{XY} = \sum_{i=1}^{n} (x_i - (ax_i + b))^2 \qquad (3.21)$$

One can clearly see that if a correlation-based fitness function is used, a misleadingly high fitness of 1 (regardless of *a* or *b*) is obtained. If the value for an error-based fitness function is defined as $f = 1/(1+SSE)$[#] then the fitness using (3.20) can range from either -∞ to ∞, depending on the actual values of *a* and *b*.

Consequently, a correlation-based fitness as in (3.16), can result in solutions having large SSEs, while not being able to approximate the desired state, which in this case is *X*, even though the fitness is 1.

The idea now is to *remove* the scale and bias (*a* and *b*) introduced in (3.18), from *Y*. To accomplish this we must first standardize the variable *Y*, i.e. subtract the mean and divide by the standard deviation and *then* scale it to have the same standard deviation and the same mean as the desired state, *X*. The following correction filter does just that

[#]To bound it between 0 and 1

Stellenbosch University https://scholar.sun.ac.za

$$G(X_d, X_a) = \text{sign}(r_{X_d X_a}) \cdot \left(\frac{\sigma_{X_d}}{\sigma_{X_a}} (X_a - E[X_a]) + E[X_d] \right) \tag{3.22}$$

Here X_d represents the desired state, X_a the approximation, σ_{Xd} and σ_{Xa} the respective standard deviations of X_d and X_a and $E[X_d]$ and $E[X_a]$ the respective means of X_d and X_a. Since the correlation between X_d and X_a can be less than 0 we need to invert the sign of X_a to compensate for negative correlation. This is done via sign(r_{XY}) which is either 1 or -1. Now substituting Y for X_a and X for X_d yields

$$G(X,Y) = \text{sign}(r_{XY}) \left(\frac{\sigma_X}{\sigma_Y} (Y - E[Y]) + E[X] \right) \tag{3.23}$$

It can be shown that

$$\sigma_Y = a\sigma_X \tag{3.24}$$

and

$$G(X,Y) = X \tag{3.25}$$

Therefore (3.22) simplifies to

$$Y - E[Y] = a(X - E[X]) \tag{3.26}$$

That is, after passing through $G(X,Y)$, Y can approximate the desired state, X, exactly if it is a linear function of X. Therefore the SSE between X and Y remains 0, regardless of a or b. This implies that solutions which would have been previously discarded by an error-based fitness function will now be accepted as valid solutions by a correlation-based fitness function *after* it has passed through the correction filter, $G(X_d,X_a)$, which removes any bias, scale or inversion; i.e. an expansion of the solution space. This process can be visualized in Figure 3.6.

Stellenbosch University https://scholar.sun.ac.za

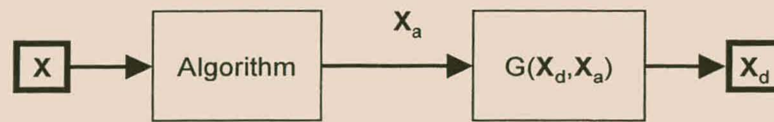


Figure 3.6 : ***An algorithm and the correction filter, G, acting as a hybrid model.***

Since the algorithm is connected to another stage, $G(X_d, X_a)$, this process is effectively a hybrid model. Also the algorithm can be *any* algorithm and is not limited to genetic programming.

Note

All the parameters σ_{Xd}, σ_{Xa}, $E[X_d]$, $E[X_a]$ and $sign(r_{XY})$ are computed during training and are left unchanged during testing or validation!

Note, that the values σ_{Xd}, σ_{Xa}, $E[X_d]$, $E[X_a]$ and $sign(r_{XY})$ are computed during training and are left unchanged during testing or validation.

From these results we can deduce three hypotheses:

1. Since more potentially good solutions are retained in each generation, there is an increase in convergence.
2. The more stringent the fitness criterion becomes, the more specialized the solutions will be as they try to abide by the criterion. A correlation-based fitness is not as stringent a criterion as an error-based fitness, therefore the final solutions have better generalizing abilities.
3. They are thus simpler (Occam's razor!)[2].

3.3.3.3 Confirming the hypotheses

[2]Occam's razor states that, all things being equal, the simplest solution is always the best.

Stellenbosch University https://scholar.sun.ac.za

To confirm the hypotheses, several runs were conducted on each of the three data sets described below and a statistical analysis was performed on the measured results. The results that were measured were

- Convergence speed or fitness over generations
- Standard deviations in the differences of the R^2 obtained during training and testing
- The length (in terms of the number of nodes in the parse tree) of the best individual after each run.

Ten runs were conducted on each data set by first using a correlation-based fitness function and then repeating it using an error-based fitness function.

The following data sets were used:

- Data set, PINE, consists of 6612 exemplars measured from the *Pinus patula* of the form [*AGE*, *TREE*, *DATE*, *TIME*, *TEMP*, *RH*, *VPD*, *PAR*, *WSPEED*, *LEAFMASS*, *HEIGHT*, *DBH*, *XPP*, *WOODDENS*, *SAPFLO*], where *AGE* denotes the age of the tree in years, *TREE* denoted the specific individual from which samples were taken, *DATE* the date of the observations (yymmdd), *TIME* the time of day during which measurements were taken, *TEMP* the temperature (°C), *RH* the relative humidity, *VPD* the vapour pressure deficit (kPa), *PAR* the photo synthetically active radiation (μmol/m^2/s), *WSPEED* the wind speed (m/s), *LEAFMASS* the estimated leaf mass of the tree (kg), *HEIGHT* the height of the tree (m), *DBH* the diameter of the tree at breast height (m), *XPP* the xylem pressure potential (kPa), *WOODDENS* the density of the tree (kg/m^3), as well as *SAPFLO*, the rate at which water was transported through the tree by means of transpiration (l/tree/h). The objective is to predict the hourly sapflow rates per tree.
- Data set, BMVANO, is comprised of 1234 observations measured from the Black Mountain base metal flotation plant. It consists of eight variables. *AvrGrayCuSc* is the average grey scale value of the digitized froth image, indicative of the average loading of solids on the bubble, *AvrRedCuSc* the average level of red

Stellenbosch University https://scholar.sun.ac.za

colour in the froth appearance, *AvrGreenCuSc* the average level of green colour in the froth appearance, *AvrBlueCuSc* the average level of blue colour in the froth appearance, *SNECuSc* is a statistical parameter indicative of the number of small bubbles in an image, *SMCuSc* is an indicator of the image darkness, *MobilitCuSc* the positional change of froth elements in consecutive images, *FlowCuSc* and *CuSc%Pb* the percentage lead in the final concentrate. Here *CuSc%Pb* is used as the output variable, i.e. the %Pb in the concentrate.

- Data set, SOLPREP, consists of set of plant data of a solution preparation circuit that were collected on a daily basis. There are eight variables x_1, x_2, ..., x_8 that describe the behaviour of the circuit.

The following parameters listed in Table 3.2, were used for each run. Two populations

Table 3.2 : ***Parameters used for each data set during regression.***

Data set	PINE	BMVANO	SOLPREP
Number of populations(demes)	2	2	2
Population size	50	50	50
Maximum tree levels	13	13	13
Terminal set (T)	AGE, TREE, DATE, TEMP, RH, VPD, PAR, WSPEED, LEAFMASS, HEIGHT, DBH, XPP, WOODDENS	AvrGrayCuSc, AvrRedCuSc, AvrGreenCuSc, AvrBlueCuSc, SNECuSc, SMCuSc, MobilitCuSc	x_1, x_2, x_3, x_4, x_5, x_6, x_7
Function set (F)	+, -, *, /, ln, exp	+, -, *, /, ln, exp	+, -, *, /, ln, exp
Crossover rate (P_c)	60%	60%	60%
Mutation rate (P_m)	4%	4%	4%
Tournament size	3	3	3
Terminate each run after	200 generations	200 generations	200 generations
Elitism	Yes	Yes	Yes
Training set size	1000	792	235
Test set size	1000	340	100

Stellenbosch University https://scholar.sun.ac.za

(demes) were used, each containing 50 individuals. Each run's function set included the basic arithmetic functions ***F*** = {+, -, *, /} and the natural logarithm, *ln*, as well as the exponential function, *exp*. The terminal set for data set PINE included all fourteen variables except *TIME*. For data set SOLPREP seven variables were included in the terminal set, while *Mn063am* was used as the target variable, while data set BMVANO had eight variables in the terminal set and *CuSc%Pb* as target.

Crossover and mutation rates were set at 60% and 4% respectively, while tournament selection was used with a tournament size of 3. An elitist strategy was followed in that the best individual after each generation was passed on, unchanged, to the next generation. Each run was terminated after 200 generations. 2000 observations were randomly selected from the original 6612 exemplars of the PINE data set. Of these, 1000 observations were used as training data and the remaining 1000 as testing data. 864 randomly selected observations were used as training data for the BMVANO data set and the remaining 370 as testing data. 235 exemplars were used as training data for the SOLPREP data set and the remaining 100 as testing data.

3.3.3.4 Discussion of results

The results obtained from the 20 runs[3] for each data set are listed in Figures 3.7 (a) - (c) and summarized in Table 3.2.

To measure the robustness of the final model after 200 generations, the difference in the R^2 between the training data and the testing data was obtained for each run. The standard deviation is computed from these differences and is denoted by $\sigma_{T\text{-}T}$ (see Table 3.2). From the results we can deduce that a correlation-based fitness initially starts at a much higher average fitness (for all three case studies) as opposed to an error-based fitness. The final average fitness is also significantly higher for all three data sets. Because a correlation-based fitness is a less stringent fitness criterion than an error-based fitness, specialization is reduced, as one can see from the $\sigma_{T\text{-}T}$ for data set PINE ranging from 0.024 for an error-based fitness, down to 0.011 for a correlation-based fitness.

---

[3] 10 runs for an error-based fitness and 10 for a correlation-based fitness.

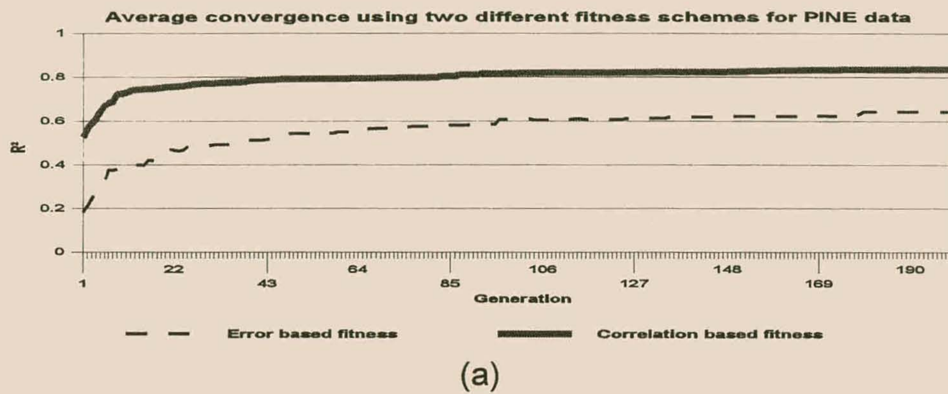
Stellenbosch University https://scholar.sun.ac.za

Table 3.3 : ***Measured results obtained for each data set after using an error-based fitness function and a correlation-based fitness function.***

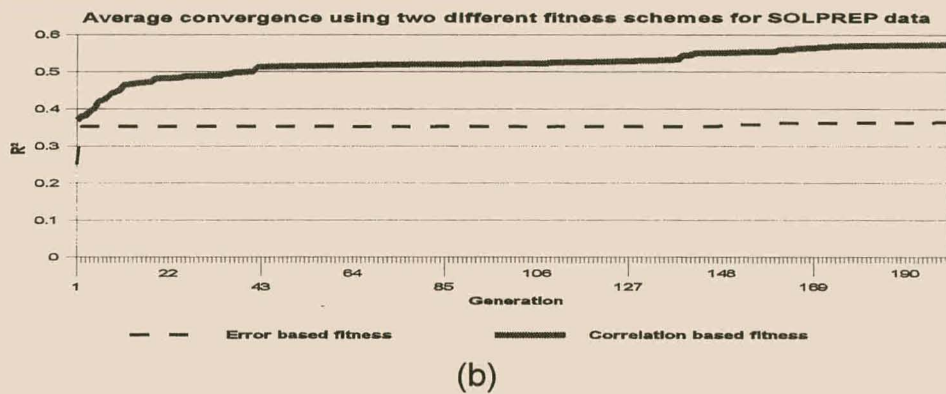
	PINE		*BMVANO*		*SOLPREP*	
Measurements	Error based fitness	Correlation based fitness	Error based fitness	Correlation based fitness	Error based fitness	Correlatio n based fitness
σ_{T-T}	**0.024**	**0.011**	**0.022**	**0.017**	**0.016**	**0.022**
Average initial fitness (R^2) over 10 runs	**0.381**	**0.525**	**0.209**	**0.481**	**0.284**	**0.37**
Average final fitness (R^2) over 10 runs	**0.643**	**0.836**	**0.464**	**0.531**	**0.318**	**0.573**
Average number of nodes of the best of model over 10 runs	**54.8**	**73.5**	**79.1**	**83.6**	**33.2**	**67.2**

However, the specialization increased for data set SOLPREP, from 0.016 to 0.022. This was owing to the fact that the error-based fitness runs got continuously entrapped in a local optimum. This kept the convergence line horizontal (see Figure 3.7.(b)) for most part of the simulation. In contrast, the models obtained via a correlation-based fitness were significantly more complex than those obtained using an error-based fitness, as shown by the average number of nodes for all three case studies.

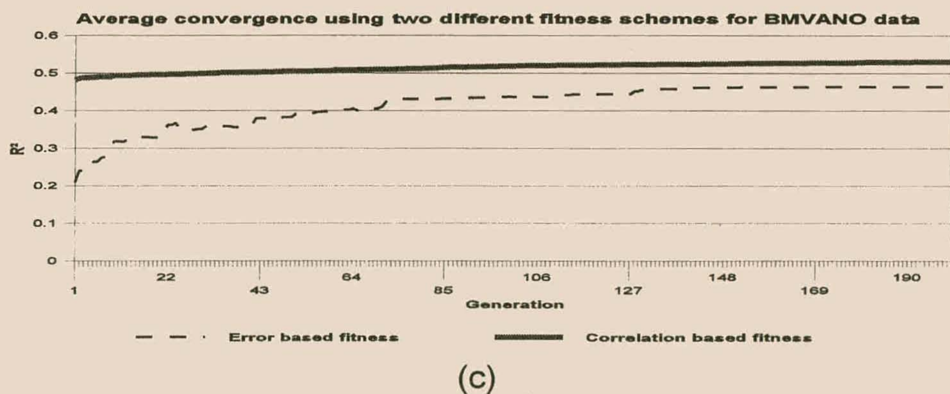
Stellenbosch University https://scholar.sun.ac.za



(a)



(b)



(c)

Figure 3.7 : ***The difference in average convergence, for the three data sets in terms of R^2-values vs the number of generations : (a) PINE, (b) SOLPREP and (c) BMVANO, when an error-based fitness function (broken line) and a correlation-based fitness function (solid line) is used. In all three examples, the correlation-based fitness function yields a much higher convergence.***

Stellenbosch University https://scholar.sun.ac.za

3.3.3.5 Conclusions

Form the results we can deduce that a correlation-based fitness allows much faster convergence than an error-based fitness (using the same run parameters). This can be seen as an expansion of the solution space, since solutions which would have been discarded previously by an error-based fitness function are now deemed valid. Second, the final results are more robust and can generalize better, owing to the smaller standard deviations in the fitness, obtained from the training sets and the test sets. Third, the average fitness of the models are higher, since an error-based fitness is a more critical way (as opposed to a correlation-based fitness) of looking at a solution. It only makes sense that the solution itself will start to specialize *and* take longer to converge. Finally, contrary to prior believe, the solutions (model structures) are on average, *more* complex[4] than those obtained using an error-based fitness. This phenomenon may be specific to genetic programming, because it evolves structures.

A correlation based fitness forces any algorithm to act as a hybrid model, because it has to be connected to a correction filter, $G(X_d,X_a)$ to reduce the SSE. In theory one would be able to achieve even higher convergence by expressing the fitness as a n-th order polynomial association between the actual state and the desired state and then using the correction-filter to remove the higher order parameters.

[4]In terms of the number of nodes. This number was obtained from models which were not simplified.

Stellenbosch University https://scholar.sun.ac.za

CHAPTER 4

PROCESS MODELLING USING a-GP

4.1 An introduction to process modelling

Process modelling strives to find functional representations between inputs and outputs of unknown processes. That is, given a set on n inputs and m outputs, the idea is to construct some kind of mathematical function to relate the inputs and the outputs and thus to identify the underlying trend in the data and predict the outputs as accurately as possible.

Consider the simplest case of a linear model of the form

$$\mathbf{y} = \mathbf{xb} + \mathbf{e} \tag{4.1}$$

where $\mathbf{y}$ is an m x 1 response, $\mathbf{x}$ is an m x n matrix of data, with rank($\mathbf{x}$) = n, $\mathbf{b}$ is an n x 1 vector of parameters and $\mathbf{e}$ is an m x 1 random vector with independent, identically and normally distributed elements, i.e. $e_i \sim m(0, \sigma^2)$ for i = 1, 2, ... m.

A linear relationship between a continuous variable (assumed to have normal distribution) and a single explanatory variable, is modelled by

$$\mathrm{E}(y_i) = b_0 + bx_i, \text{ for } i = 1,2,\ldots n \tag{4.2}$$

This is equivalent to the model E($\mathbf{y}$) = $\mathbf{xb}$, with

Stellenbosch University https://scholar.sun.ac.za

$$\mathbf{y} = \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_n \end{bmatrix}, \mathbf{x} = \begin{bmatrix} 1 & x_1 \\ 1 & x_2 \\ \vdots & \vdots \\ 1 & x_n \end{bmatrix} \text{ and } \mathbf{b} = \begin{bmatrix} b_0 \\ b_1 \end{bmatrix} \tag{4.3}$$

The simplest models are based on the premise that any relationship between the input and output variables are linear and that the data themselves are normally distributed (McKay *et al.*, 1997). However, real world systems are highly non-linear and these linear approximations fail to discover the functional relationships in the data. Subsequently systems are often modelled using non-parametric techniques (such as neural networks (Del Giudice and Amabile, 1997), regression trees (Breiman *et al.*, 1984), kernel regression (Herrmann, 1994) and fuzzy regression (Shakouri *et al.*, 1997). It has been proven (Hornik *et al.*, 1989) that neural networks, with one hidden layer of sigmoidal units, are capable of approximating any continuous function. However, the main drawback with neural networks and other non-parametric techniques is that the mathematical models are extremely complex and very difficult to analyse. Also, no insight is gained as to how the inputs relate to the structure.

Genetic programming, on the other hand, can easily handle a trade-off between interpretability and accuracy. In effect, genetic programming applies symbolic regression to discover the underlying trend in the data, which allows it to operate as a non-parametric algorithm, whilst having the interpretabilities of a linear approximation. The final solution can be represented as a parametric model. Also the way in which the input variables relate to the structure becomes immediately apparent from the symbolically evolved functions. There is thus, no need for any *a priori* knowledge regarding the inputs (or structure) of the process.

Stellenbosch University https://scholar.sun.ac.za

4.2 *Case studies*

4.2.1 Approximation of multivariate functional relationships

To evaluate the performance of the genetic programming algorithm, a multivariate functional relationship of two independent variables, as represented by (4.4) was considered.

$$y = x_1^2 + 3x_1 + 2x_1x_2 + 4x_2, \text{ with } x_1 \in [-1, 1] \text{ and } x_2 \in [-1, 1] \quad (4.4)$$

200 exemplars were uniformly sampled over the appropriate ranges by means of (4.4). Twenty runs were conducted. In the first ten runs, an error-based fitness criterion was used while the last ten employed a correlation-based fitness criterion. For each run, two demes were used, each consisting of 50 individuals. The terminal and function sets consisted of the following, $\boldsymbol{F} = \{+, -, *, /\}$ and $\boldsymbol{T} = \{x_1, x_2\}$. Crossover and mutation rates were set at 60% and 4% respectively, while tournament selection was used with a tournament size of 3. An elitist strategy was followed in that the best individual after each generation was passed on, unchanged, to successive generations. Each run was terminated when the fitness of the best individual after each generation was equal to 1.

In each of the runs the desired relationship was attained, regardless of the fitness criterion. The equations in (4.5) are representative of a typical result obtained during

$$\begin{aligned} y &= ++x_2 + -x_2 ++*/x_1x_2 *x_2x_1x_1 + *x_1x_2x_2 - \\ &\quad ++x_1x_1 + *x_2x_1x_2x_2 + x_2x_2 \\ &= ++x_2 + -x_2 - x_2(x_1^2 + x_1 + x_1x_2 + x_2)(2x_1 + x_1x_2)(2x_2) \\ &= ++x_2(x_2 + x_1^2 + x_1 + x_1x_2 + 2x_1 + x_1x_2)(2x_2) \\ &= x_1^2 + 3x_1 + 2x_1x_2 + 4x_2 \end{aligned} \quad (4.5)$$

one of the runs.

As shown in Table 4.1, the algorithm (using both fitness criteria) was able to correctly identify the functional relationship from the given data; which resulted in a fitness of 1.

Stellenbosch University https://scholar.sun.ac.za

Using a correlation-based fitness criterion, the average number of generations (averaged over 10 generations) needed for convergence, was 79.7. In comparison, an error-based fitness criterion required 139.2 generations. A 43% improvement in performance.

To investigate the effect "*a priori* knowledge" has on the algorithm the following function

Table 4.1 : ***Results obtained for the identification of the multivariate functional relationship in eq. 4.4. 20 runs were conducted of which 10 used a correlation-based fitness criterion and the remainder, an error-based fitness criterion.***

Results	Correlation-based fitness	Error-based fitness
Best fitness value	1.0	1.0
Average number of generations needed for convergence	79.7	139.2

of two independent variables was considered

$$y = \sin(x_1 x_2) + \frac{x_1 x_2}{5}, \text{ with } x_1 \in [-4\pi,\ 4\pi] \text{ and } x_2 \in [-4\pi,\ 4\pi] \tag{4.6}$$

500 exemplars were uniformly sampled over the appropriate ranges by means of (4.6). Once again, 20 runs were conducted. The same parameter criteria were used as

Table 4.2 : ***Results obtained for the evaluation of the algorithm with and without a priori knowledge in the function set. 20 runs were conducted of which 10 used a function set that had the a priori information included or F = {+, -, *, /, sin}. In the remaining ten runs, this information was excluded, therefore F = {+, -, *, /}.***

Results	*A priori* knowledge included *F* = {+, -, *, /, sin}	*A priori* knowledge excluded *F* = {+, -, *, /}
Total runs	10	10
*Number of successful runs**	8	0
Success rate	80%	0%
Best fitness	1	0.925
Worst fitness	0.93	0.917

* i.e. finding the exact relationship

Stellenbosch University https://scholar.sun.ac.za

described in the previous example. A correlation-based fitness function was employed. For the first ten runs, the function set consisted of ***F*** = {+, -, *, /, sin} while in the remaining ten runs the "sin" function was excluded from the function set to evaluate the performance of the algorithm without this "*a priori* knowledge". The terminal set, consisted of ***T*** = $\{x_1, x_2\}$. Each run was terminated after either the desired functional relationship was discovered or after the 200th generation, whichever came first. The results obtained are presented in Table 4.2.

A run was considered successful if the algorithm was able to correctly identify the exact functional relationship from the given data. From Table 4.2 one finds that when the "sin" function is included in the function set, the algorithm identifies the correct function 80% of the time. When the "sin" function is excluded from the function set, the algorithm is unable to find the correct function. However, it consistently produces reliable approximations in the sense that the worst fitness and the best fitness over ten runs are not significantly different. The equation in (4.7) is representative of a typical result

$$y = x_1 x_2 + \frac{x_1}{-3x_1^4 + 2x_1 x_2 + x_2^2 - 2\frac{x_1^7}{x_2} - 2x_1^3 x_2 + 2\frac{x_1^4}{x_2}} \tag{4.7}$$

obtained during runs, using the functional set ***F*** = {+, -, *, /}.

Figure 4.1 presents the fitness distribution within a population sampled at specific generations. During the first few generations most individuals have very low fitness values. With increasing generations there occurs a shift in the distribution towards the region with a higher fitness (i.e. towards a fitness of 1). One should also bear in mind that natural selection decreases the diversity and therefore the final generations are primarily composed of copies of the best of individual.

The lack of "*a priori* knowledge" results in a complex parametric approximation of the desired functional relationship.

Stellenbosch University https://scholar.sun.ac.za

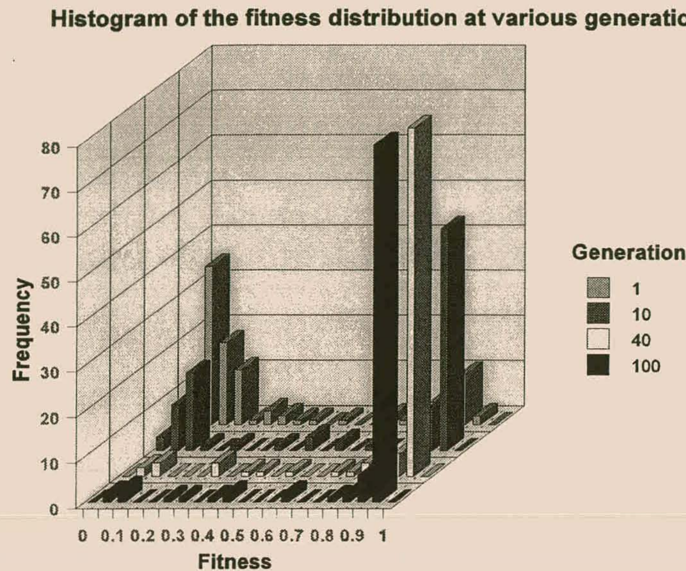


Figure 4.1 : ***A histogram plot of the frequency distribution of the fitness of each individuals in a sampled at a specific generation. Notice that when the algorithm starts (generation 1) most individuals have very low fitness values. With succeeding generations this distribution starts moving towards the region with higher fitness.***

4.2.2 Obtaining regression models for four real data sets

A regression analysis was conducted on four data sets as described in the following case studies. The augmented genetic programming algorithm (a-GP) was compared against standard genetic programming, linear regression and a multilayer perceptron (MLP) neural network in all four case studies. For standard genetic programming an error-based fitness function was used while a-GP employed a correlation based fitness function in conjunction with a correction filter[1], $G(X_d, X_a)$, to correct the SSE. A multilayer perceptron neural network architecture was used consisting of one hidden layer which was comprised of sixteen hidden nodes. Each node contained a sigmoidal activation function. Training for the neural network was completed after 10000 epochs. These four techniques were compared against one another by using their MSE and R^2, averaged over three runs.

[1]See Chapter 3.4.3.

Stellenbosch University https://scholar.sun.ac.za

4.2.2.1 *Modelling of transpiration in pine trees*

Refer to Chapter 3.4.3.3 for a description of the PINE data set.

4.2.2.2 *Modelling of transpiration in poplar trees*

This data set, referred to as POP, consists of 1130 exemplars measured from the *Populus deltoides* of the same form [*AGE*, *TREE*, *DATE*, *SEASON_NO*, *HOUR*, *VPD*, *PAR*, *LEAVE_AREA*, *ETLA*], where *AGE* denotes the age of the tree in years, *TREE* denoted the specific individual from which samples were taken, *DATE* the date of the observations (yymmdd), *SEASON_NO* a dummy variable for the seasons (Autumn=1, Spring=2 and Summer=3), *HOUR* the hour of day during which measurements were taken, *VPD* the vapour pressure deficit (kPa), *PAR* the photo-synthetically active radiation (μmol/m^2/s), *LEAVE_AREA* the leave area (m^2) and *ETLA*, the rate at which water was transported through the tree by means of transpiration (l/m^2/h). Here the objective is to predict the hourly sapflow rates per square metre.

4.2.2.3 *Modelling of the Black Mountain base metal flotation plant*

Refer to Chapter 3.4.3.3 for a description of the BMVANO data set.

4.2.2.4 *Modelling of a solution preparation circuit*

Refer to Chapter 3.4.3.3 for a description of the SOLPREP data set.

4.3 Run parameter listings

The following parameters listed in Table 4.3, were used for each run. Two populations (demes) were used, each containing 50 individuals. Each run's function set

Stellenbosch University https://scholar.sun.ac.za

included the basic arithmetic functions ***F*** = {+, -, *, /} and the natural logarithm, *ln*, as well as the exponential function, *exp*. The terminal set for data set PINE included thirteen variables with *TIME* and *SAPFLO* excluded. *SAPFLO* was used as the target variable. In data set POP, eight variables were included in the terminal set except *ETLA* which was used as the target variable. For data set SOLPREP, seven variables were included in the terminal set while *Mn063am* was used as the target variable, while data set BMVANO had seven variables in the terminal set and *FlowCuSc* as target.

Crossover and mutation rates were set at 60% and 4% respectively, while tournament selection was used with a tournament size of 3. An elitist strategy was followed in that the best individual after each generation was passed on, unchanged, to successive generations. Each run was terminated after 300 generations. 2000 observations were randomly selected from the original 6612 exemplars of the PINE data set. 1000 observations were used as training data and the remaining 1000 as testing data, while 791 observations were randomly selected from the POP data set as training data and the remaining 339 were used as test data. 864 randomly selected observations were used as training data for the BMVANO data set and the remaining 370 as testing data. 235 exemplars were used as training data for the SOLPREP data set and the remaining 100 as testing data.

A correlation-based fitness function was used for a-GP. For standard genetic programming, the fitness function was changed to an error-based fitness function. The remaining parameters were left unchanged. Table 4.3 presents a run parameter description.

Stellenbosch University https://scholar.sun.ac.za

Table 4.3 : *Run parameters used for each data set during regression.*

Parameters	PINE	POP	BMVANO	SOLPREP
Number of populations (demes)	2	2	2	2
Population size	50	50	50	50
Maximum tree levels	13	13	13	13
Terminal set (T)	AGE, TREE, DATE, TEMP, RH, VPD, PAR, WSPEED, LEAFMASS, HEIGHT, DBH, XPP, WOODDENS	AGE, TREE, DATE, SEASON_NO, HOUR, VPD, PAR, LEAVE_AREA	AvrGrayCuSc, AvrRedCuSc, AvrGreenCuSc, AvrBlueCuSc, SNECuSc, SMCuSc, MobilitCuSc	$[H_2SO_4]$_before, $[H_2SO_4]$_after, H_2SO_4_addition, NH_4OH_addition, LeachFlow, $[Mn^{2+}]$_solidT093, Mn093am
Function set (F)	+, -, *, /, ln, exp	+, -, *, /, ln, exp	+, -, *, /, ln, exp	+, -, *, /, ln, exp
Crossover rate (P_c)	60%	60%	60%	60%
Mutation rate (P_m)	4%	4%	4%	4%
Tournament size	3	3	3	3
Terminate each run after	300 generations	300 generations	300 generations	300 generations
Elitism	Yes	Yes	Yes	Yes
Training set size	1000	791	792	235

4.4 Investigating the effect different crossover and mutation rates has on the overall performance of the algorithm

To understand how the crossover rate (Pc) and mutation rate (Pm) affects the performance of the genetic programming algorithm, two different runs were performed on each of the data sets described above using different crossover/mutation rate combinations. For the first run, all the parameters in Table 4.3 were left unchanged.

Stellenbosch University https://scholar.sun.ac.za

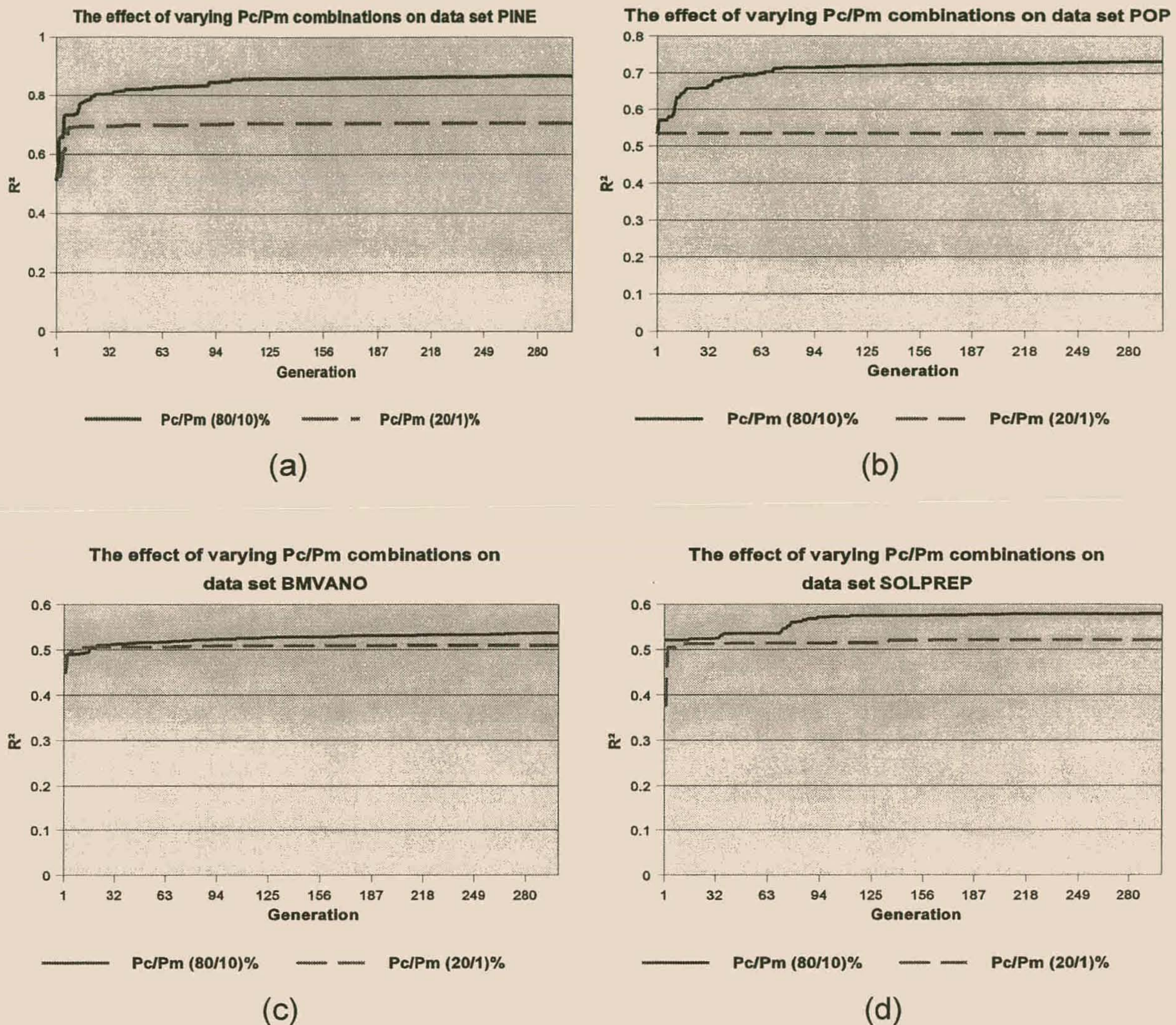


Figure 4.2 : *The effect of varying combinations of crossover (Pc) and mutation (Pm) rates, emphasized the fact that a too small search rate does not yield satisfactory results. In (a) and (b) we can see that the algorithm got entrapped in a local optimum, when the crossover/mutation rate was set at (20/1)%. Increasing the search rate to (80/20)% allowed the algorithm to avoid entrapment in the local optimum. In (c) the larger search rate did not make significant difference, while in (d) a steady increase can be observed.*

Crossover and mutation rates were set at 20% and 1% respectively. For the second run, all the parameters in Table 4.3 were also left unchanged. Crossover and mutation rates, however, were set at 80% and 10% respectively. The results are depicted in Figure 4.2.

Stellenbosch University https://scholar.sun.ac.za

4.5 Discussion of results

Judging from Figure 4.3 and Table 4.3 one can clearly differentiate amongst the performance of the four different algorithms, i.e. a-GP, genetic programming (GP), linear regression and the multilayer perceptron neural network. Clearly, a-GP outperforms standard genetic programming as proposed by Koza (1992) on all four case studies. Of particular interest is the fact that a-GP improves on the multilayer perceptron neural networks on one of the four case studies. For data sets POP, BMVANO and SOLPREP, the neural network outperformed a-GP by a small margin. Interestingly, linear regression outperformed both a-GP and the neural network, when

Table 4.4 : ***Results obtained for each of the four data set after testing. A comparison of R^2 and MSE is made amongst the four different regression techniques. These are a-GP, GP, linear regression and ANN's.***

	R^2				MSE			
Data set	a-GP	GP	Linear regression	ANN	a-GP	GP	Linear regression	ANN
PINE	0.85	0.73	0.68	0.77	0.83	1.36	1.58	1.1
POP	0.67	0.58	0.53	0.69	0.02	0.02	0.035	0.001
BMVANO	0.53	0.49	0.48	0.58	219	210	214	176
SOLPREP	0.48	0.32	0.50	0.495	0.37	0.74	0.31	0.31

applied to data set SOLPREP, by a slight margin. A two-tailed test of significance revealed that the results obtained [for data set SOLPREP] using linear regression were not significant at the 0.05 level when compared to any of the other techniques. This could imply that the data is linear. Unfortunately the evolved models were too complex to simplify and are included in Appendix A.

Stellenbosch University https://scholar.sun.ac.za

Table 4.5 presents the significance of the difference between the correlation coefficients of the four regression techniques. The null hypothesis, H_0, was tested to see whether the results obtained via a-GP was significantly different at the 0.05 level. A two-tailed test of the normal distribution was used. H_0 would only be rejected if the

Table 4.5 : ***Significance of the difference between the correlation coefficients of the four different regression techniques. Here, the null hypothesis, H_0, is tested to see whether the results obtained with a-GP, on the four data sets, are significantly different than those obtained via GP, linear regression and neural networks. The values inside the table are the test statistic (z) values. The values that are labelled with [(a)] imply that the results obtained via a-GP, are significantly different when compared to the corresponding algorithm in that row.***

	PINE	POP	BMVANO	SOLPREP
Genetic Programming	7.36[(a)]	1.98[(a)]	0.74	1.478
Neural Network	5.29[(a)]	-0.49	-0.978	-0.146
Linear Regression	9.61[(a)]	2.95[(a)]	0.92	-0.195

[(a)]Significantly different from a-GP at the 0.05 level.

test statistic, z > 1.96 or z < -1.96. From the test statistics in Table 4.5 we can conclude that the results obtained for data set PINE were significantly different when compared against the other techniques. Like wise, for data set POP, the difference was significant,

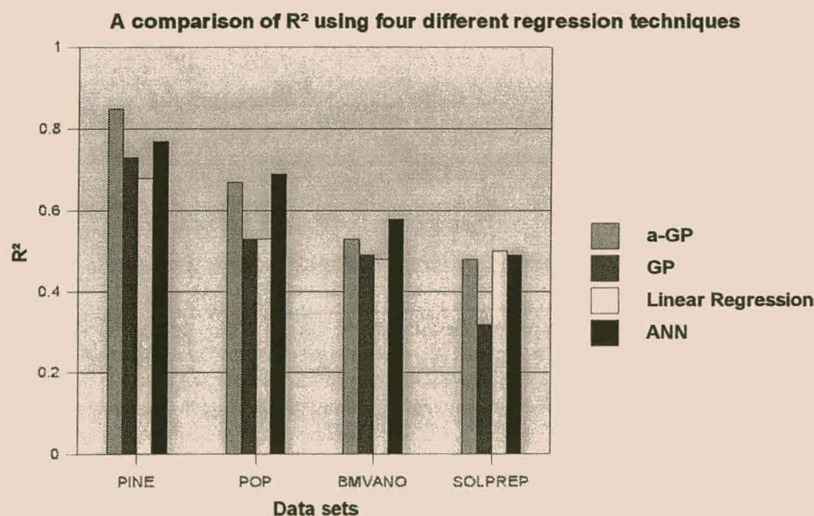


Figure 4.3 : ***A comparison of R^2 obtained from the four data sets. a-GP outperforms GP on all four case studies.***

Stellenbosch University https://scholar.sun.ac.za

except when compared to the neural network.

Figure 4.2 illustrates the effect different crossover/mutation rate combinations has on the overall performance of the a-GP algorithm. Clearly, when the crossover/mutation rate is too low (20/1) the algorithm gets entrapped in a local optimum (see Figures 4.2.(a) and 4.2.(b)). Increasing the search rate to (80/20) alleviates this problem. The effect of varying rates was least significant in Figure 4.2.(c), while in Figure 4.2.(d) a steady increase in convergence is noticed.

Figures 4.4 (a)-(d) are scatter plots of the observed output vs the predicted output of

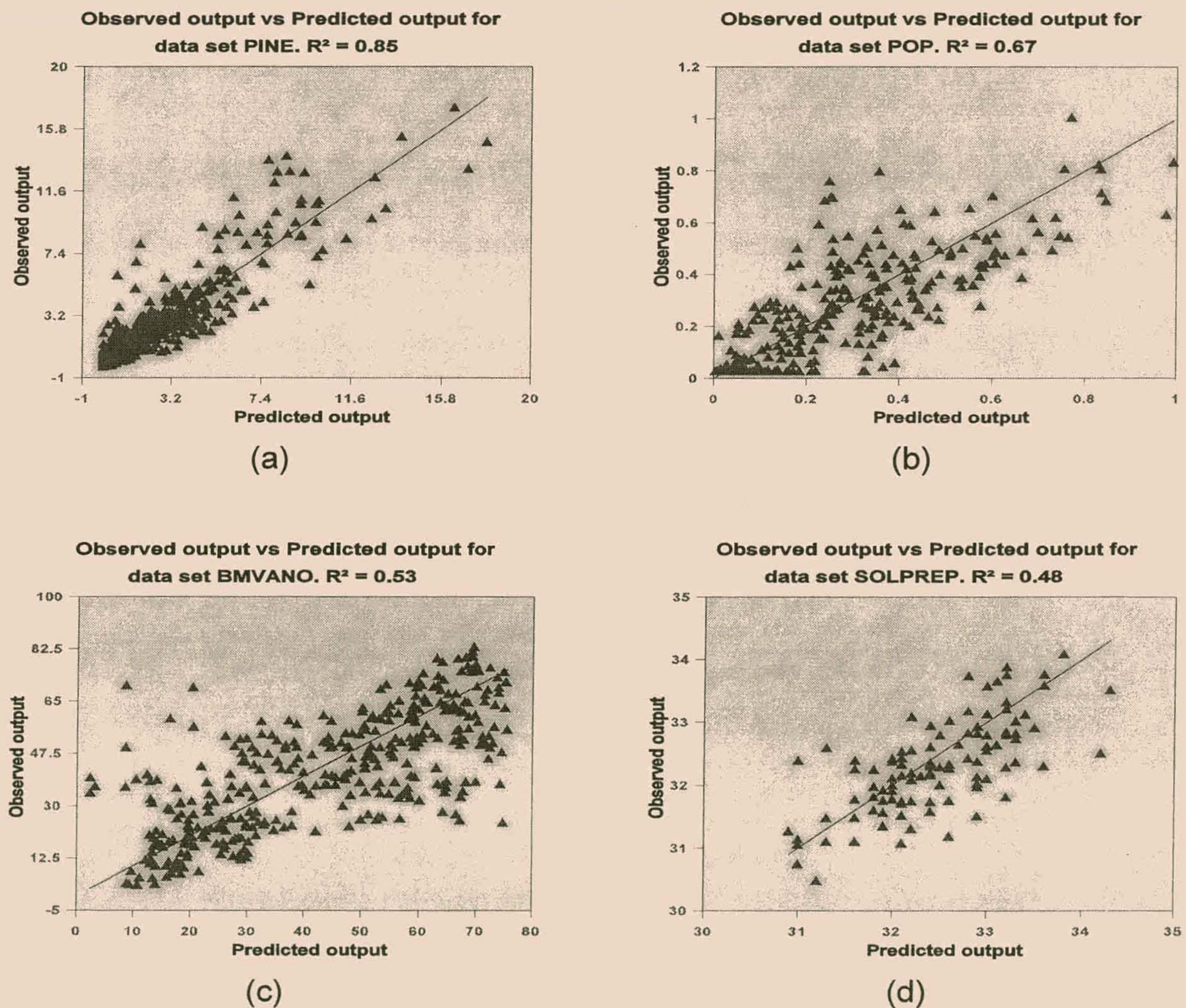


Figure 4.4 : ***X-Y scatter plots of the Observed output vs Predicted output for data sets: (a) PINE [R^2=0.85], (b) POP [R^2=0.67], (c) BMVANO [R^2=0.53] and (d) SOLPREP [R^2=0.48].***

Stellenbosch University https://scholar.sun.ac.za

the best individual obtained for each data set. There is less scatter in the model obtained for data set SOLPREP as opposed to data set BMVANO, even though the evolved model for BMVANO explains 5% more variation.

4.6 Conclusions

The results clearly indicate how a vital tool a-GP can be for steady-state process modelling and that it can be seen as a viable alternative for artificial neural networks.

When applied to a the identification of a multivariate functional relationship of two independent variables, the algorithm was able to obtain the correct function from the given data. Using a correlation-based fitness criterion, the correct solution was found in 43% less generations than when an error-based fitness criterion was employed. Using a correlation-based fitness function allows much faster convergence than using an error-based fitness function.

The effect of "*a priori* knowledge" was investigated in that a multivariate functional relationship (that incorporated a "sin" function) of two independent variables was generated. In this instance, the "sin" function represented the "*a priori* knowledge". When this "a priori knowledge" was included in the function set, the algorithm was able to correctly identify the function from the given data. However, when the "sin" function was excluded from the function set, the algorithm was unable to correctly identify the function. The approximations, however, were extremely consistent in the sense that the best of fitness and the worst of fitness over ten runs were very similar. Incorporating "*a priori* knowledge" in the function- and/or terminal set does benefit the genetic programming algorithm. Excluding this information from the algorithm results in complex parameterized approximations of the desired functional relationship.

Although a-GP failed against the multilayer perceptron neural network on three of the four case studies, the results obtained [using the neural network] were not significantly different at the 0.05 level. Allowing possible longer evolution time or by using a larger population could also improve results. Given the complexity of the evolved models, one

Stellenbosch University https://scholar.sun.ac.za

can conclude that the underlying relationships within the data are extremely complex and that very little "*a priori* knowledge" was available, which resulted in highly parametric models. Owing to the fact that the genetic programming algorithm lacks parameter estimation, it evolves complex tree structures; which it uses to approximate the parameters. This demonstrates he need for a local optimization procedure to generate and optimize parameters in the genetic programming algorithm. The result obtained via a-GP were significantly different at the 0.05 level, on half of the data sets, when compared to standard genetic programming.

By varying the crossover/mutation rate combinations the results (as expected) clearly indicate that high search rates are favoured. A too small crossover/mutation rate does not allow sufficient exploration of the search space in the allotted time (300 generations). Also the low mutation rate (1%) was not sufficient to avoid entrapment in local optima. This would seem to imply that the search rates used in this thesis, *viz.* 60% for crossover and 4% for mutation, are sufficient.

Stellenbosch University https://scholar.sun.ac.za

CHAPTER 5

VISUALIZATION OF PROCESS SYSTEMS USING a-GP.

5.1 An introduction to dimensionality reduction

The continued growth in large data systems in the chemical and metallurgical process industries has precipitated intense efforts to develop more efficient methods for the exploration and interpretation of large volumes of data. It is not uncommon for the individual analyst to have to interpret many hundreds or even thousands of variables and hundreds of thousands of observations off-line, while in automated monitoring and control systems, data volumes of an order of magnitude higher may have to be accommodated. The extraction of features and the reduction of dimensionality are two vitally important ways of dealing with these problems. Feature extraction and dimensionality reduction provides an antidote to the "curse of dimensionality" and can improve the generalizability of process models and classifiers, allow us to visualize high dimensional data to better understand the underlying structure, explore the intrinsic dimensionality and analyse the clustering tendency of multivariate data (Mao and Jain, 1995).

Dimensionality reduction can generally be achieved in two ways, viz. by selecting a small but important subset of variables prior to analysis, or by extracting a lower-dimensional set of features that preserve the essential characteristics of the original data (Pal and Eluri, 1998).

A large number of approaches for the dimensionality reduction of data (i.e. feature extraction and multivariate data projection) has been reported in the literature dealing

Stellenbosch University https://scholar.sun.ac.za

with pattern recognition (Sammon, 1969; Biswas *et al.*, 1981; Mao and Jain, 1995; Kraaijveld *et al.*, 1995). The differences in these approaches are based on the characteristics of the mapping function $\wp$ (linear or non-linear), the way $\wp$ is learned (supervised or unsupervised), the nature of the optimization criterion, etc. (Mao and Jain, 1995). Although non-linear techniques are more suitable for complex (non-linear) process systems, these mapping functions (such as represented by artificial neural networks) tend to be non-parametric, among other, and may also be difficult to optimize in the presence of a large number of local minima in the error surface of the optimization criterion associated with the mapping.

During this research a novel strategy, based on the use of genetic programming (GP) to visualize and explore industrial mineral process data, is proposed. This approach has the advantage that an explicit non-linear mapping function, $\wp$, is generated which gives an indication of the structure of the data as well as the way the original variables are related to this structure, as will be shown by way of a few case studies.

5.1.1 An overview of data projection

During feature extraction and data projection, data residing in a higher dimensional space, $\Re^p$, is mapped to a lower dimensional space, $\Re^q$ (where $q < p$), while the essential characteristics of the original data are preserved (Pal and Eluri, 1998). Usually q (for exploratory data analysis purposes) is set to either 2 or 3 in order to visualize the mapped data. In order to map the data some criterion *C*, is optimized. However, unlike regression where the mapping function is estimated from input-output pairs (known outputs), in feature extraction or data projection the outputs are often not available.

The Sammon measure (Sammon, 1969) is the most widely used criterion which tries to preserve all the inter-pattern distances between the data in $\Re^p$ and the mapped data in $\Re^q$. Euclidian distance is used in this projection. Sammon's method is an intuitively simple, but powerful way of preserving the structure of the data, and can be summarized as follows:

Stellenbosch University https://scholar.sun.ac.za

Define the similarities in the input space $\mathfrak{I} \in \Re^p$ as $F(i,j)$ and in the output $\mathfrak{C} \in \Re^q$ as $G(u,v)$, where $q \leq p$ and i and j are points in the input space, $\mathfrak{I}$, while u and v are points in the output space, $\mathfrak{C}$.Assuming that there are n points or patterns to be mapped, and that $\wp$ is a one-to-one mapping of points from the input space to the output space, yielding n points or patterns in the output space, so that the following objective function can be defined:

$$S = \left[\sum_{i=1}^{n-1} \sum_{j=i+1}^{n} F(i,j) \right]^{-1} \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} [F(i,j) - G(\wp(i), \wp(j))]^2 / F(i,j) \qquad (5.1)$$

S is also referred to as the Sammon stress. This objective function assumes that both F and G are Euclidean distances in the two spaces, with a unity distance between neighbouring points in each space. Alternatively (5.1) can be rewritten as

$$S = \left[\sum_{i=1}^{n-1} \sum_{j=i+1}^{n} d_{ij}^* \right]^{-1} \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} (d_{ij}^* - d_{ij})^2 / d_{ij}^* \qquad (5.2)$$

i.e. d_{ij}^* is the [Euclidian] distance between points i and j in the input space, $\mathfrak{I}$, and d_{ij} is the [Euclidian] distance between the corresponding projected points u and v in the output space, $\mathfrak{C}$.

Sammon used the method of steepest descent for the approximate minimization of S, that is, if $\mathbf{y}_i(t)$ is the estimate of $\mathbf{y}_i$ at the t'th iteration, then $\mathbf{y}_i(t+1)$ is given by

$$y_{ij}(t+1) = y_{ij}(t) - \alpha[A/B] \qquad (5.3)$$

where

$$A = \frac{\partial S(t)}{\partial y_{ij}(t)} \text{ and } B = \frac{\partial^2 S(t)}{\partial y_{ij}(t)^2} \qquad (5.4)$$

with α the step size for the gradient search, i.e. a nonnegative scalar constant with a recommended value of between 0.3 and 0.4.

Stellenbosch University https://scholar.sun.ac.za

With the approach originally formulated by Sammon, as well as other optimization strategies, such as proposed by Chen *et al.* (1999), it is *not* possible to obtain an explicit mapping function, $\wp$, relating patterns in the input space, $\Im$, with patterns in the output space, $\mathfrak{C}$. This means that if new points are to be projected, the optimization procedure has to be repeated. This is a major disadvantage, given that the optimization is computationally intensive, as every step within an iteration requires the calculation of $n(n-1)/2$ distances. In addition, the error surface is riddled with local minima, and the algorithm is likely to get stuck in one of these.

Various approaches based on cluster analysis (Chang and Lee, 1973; Schachter, 1978; Pykett, 1980) have been proposed to alleviate the computational burden associated with the optimization, but these were only partially successful. More recently, methods based on the use of neural networks (Mao and Jain, 1995; Pal and Eluri, 1998) to model the mapping function have removed the need for re-optimization prior to the mapping of new data.

5.1.2 Characteristics of data

During feature extraction the data that are used for projection can be described by the following characteristics (Mao and Jain, 1995).

- Data source (source): Specifies whether the data is real or artificially generated.
- Dimensionality of pattern vectors (d): Specifies the number of input vectors.
- Intrinsic dimensionality (d_I): The intrinsic dimensionality of the data is measured by the number of significant eigenvalues (more than 97% of the total variance is retained by the first d_I principal components) of the covariance matrix of the data.
- Number of classes/clusters (c): Indicates how many known classes or clusters there are in the data.
- Number of patterns (n): Specifies the dimensions of the input vectors.

Stellenbosch University https://scholar.sun.ac.za

- Linear separability (λ_s): This is defined as the largest eigenvalue of the covariance matrix. λ_s is restricted to the range [0.0, 1.0]. As λ_s increases from 0.0 to 1.0 the data set becomes more and more linearly separable.
- Sparseness: This is measured by the ratio of the dimensionality to the number of patterns (d/n) in the data set; the larger this ratio, the sparser the data.

5.2 *Extending the genetic programming algorithm to accommodate feature extraction*

During this research, genetic programming was used to construct the mapping functions. Since the mapped data resides in a q-dimensional space, q mapping functions are needed. An individual in the genetic programming algorithm is extended to have q parse trees representing q mapping functions, $\wp_1..\wp_q$, as shown in Figure 5.1.

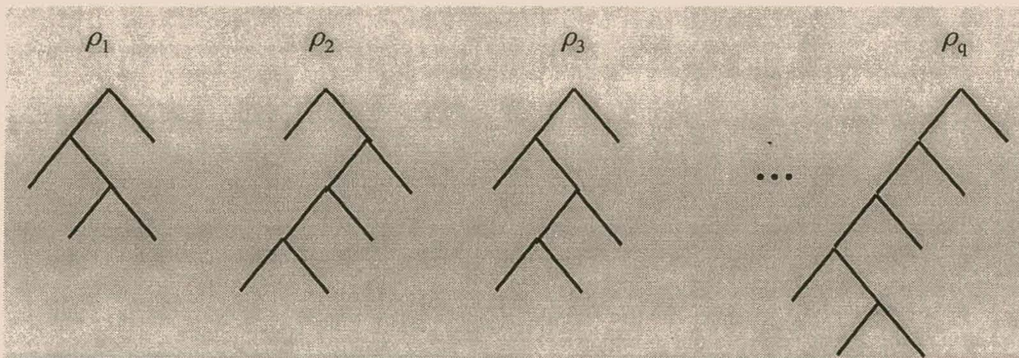


Figure 5.1 : ***The q parse-trees that make up an individual for feature extraction. Each tree represent a mapping function ranging from ρ_1 to ρ_q.***

Crossover is limited to a single tree at a time in the expectation that this will reduce the extent to which it disrupts "building blocks" of useful code. Therefore, one mapping function (with the same index), $\wp_k$, is randomly selected from two [randomly] selected individuals I_i and I_j. The actual crossover only occurs between the two parse trees $I_i{:}\wp_k$ and $I_j{:}\wp_k$, as depicted in Figure 5.2.

Since the [Sammon] stress has to be minimized and the fitness, *f*, is always expressed as a value between 0 and 1, the fitness can be expressed as

Stellenbosch University https://scholar.sun.ac.za

$$f = \frac{1}{1+S} \tag{5.5}$$

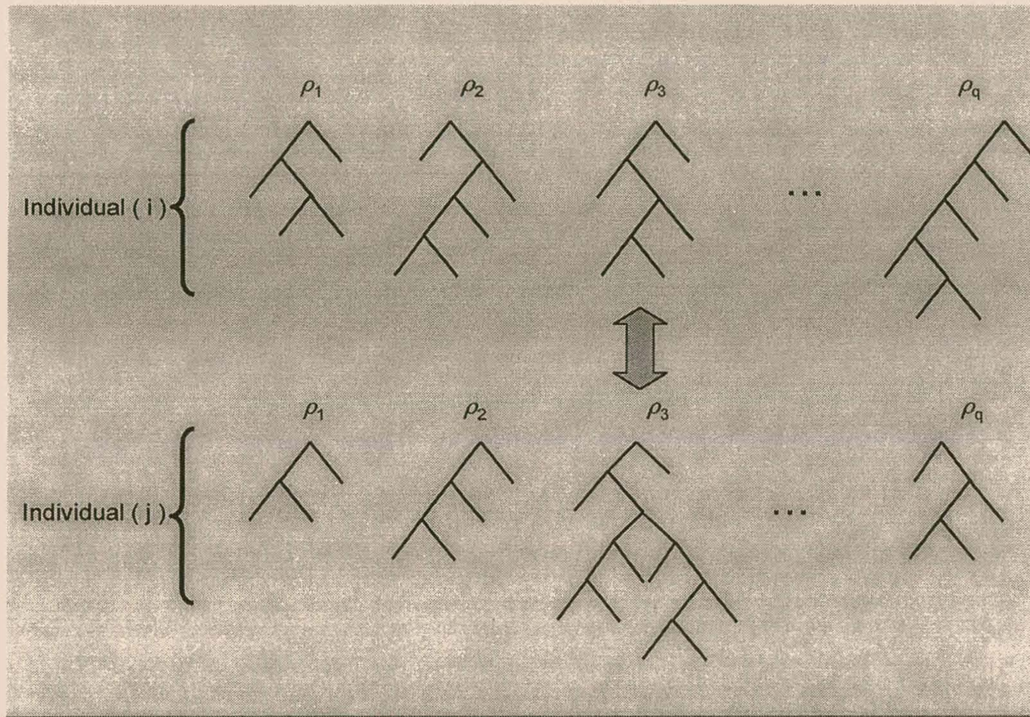


Figure 5.2 : ***During feature extraction, crossover only occurs between parse-trees with similar indices. In this example two individuals, I_i and I_j are randomly selected from the mating pool. A parse-tree, ρ_3, is randomly selected from both trees for crossover.***

5.3 Case studies

A data dimensionality reduction analysis was conducted on five data sets to increase understanding into the underlying relationship amongst the data. During all of these cases the reduced space dimensions, q, was set to two (for visualization purposes).

In order to illustrate the characteristics of the evolutionary computation algorithm used for the projection of multivariate data, the following simple data sets were considered.

Stellenbosch University https://scholar.sun.ac.za

5.3.1 Case studies on artificial and bench marking data sets

5.3.1.1 Description of each data set

Four data sets were investigated in this analysis. These data sets are widely used as bench marking sets in the literature and are described below:

- Data set BITET consisted of an asymmetrically arranged set of four 3-dimensional clusters (A, B, C and D) is considered. The clusters are arranged along the vertices of two tetrahedra joined at their bases and with apices pointing in opposite directions. Clusters A, B and C are roughly spherical and arranged along the vertices of the common basis of this bi-tetrahedron. Cluster D is an elongated ellipsoidal cluster that joins the fourth and fifth vertices (apices) of the bi-tetrahedron.

- The SPIRAL data set has been investigated previously by (Mao and Jain, 1995) and (Pal and Eluri, 1998). It is an artificially generated data set describing two spirals (500 data points each) in 3D-space, so that $x_1 = ½\cos\theta + \varepsilon$, $x_2 = ½\sin\theta + \varepsilon$ and $x_3 = \sin 2\theta + \cos 2\theta + \varepsilon$, with $\theta \in \{-\pi/2, \pi/2\}$, and ε a randomly generated noise factor, $\varepsilon \in \{0, 0.25\}$.

- The third data set, SPHERESHELL (Pal and Eluri, 1998) was likewise comprised of three coordinates, and described a hemisphere with radius 0.6 (500 data points) enclosed in a shell (also of 500 data points) with an inner radius of 2 and and outer radius of 2.013. The hemisphere, as well as the shell each contained 500 randomly generated data points. This data set was also artificially generated.

- Although the IRIS data set is neither artificial, nor related to chemical process systems, it has been investigated extensively elsewhere, and serves as a useful benchmark for data mapping algorithms. It consists of 150 data points describing three species of *Iris* (*setosa*, *virginica* and *versicolor*) in terms of sepal length (x_1) and width (x_2), and petal length (x_3) and width (x_4).

The essential characteristics of each data set are summarized by Table 5.1. The terminal set of the genetic programming algorithm, for each run, contained all the

Stellenbosch University https://scholar.sun.ac.za

variables of the corresponding data set, $\boldsymbol{T} = \{x_1, x_2, \ldots, x_n\}$, while the function set contained the four basic arithmetic operators and the sigmoidal function, σ, that is $\boldsymbol{F} = \{+, -, *, /, \sigma\}$.

Table 5.1 : *Essential characteristics of the four data sets*

Data set	n	c	d_I	Λ_s	Sparseness (d/n)	d	Source
Bitet	150	4	3	0.35	0.02	3	Artificial
Spiral	1000	2	2	0.7	0.003	3	Artificial
Sphereshell	1000	2	3	0.38	0.003	3	Artificial
Iris	150	3	2	0.97	0.027	4	Real

The size of the population for each data set was 100, and the reproduction, crossover and mutation probabilities were 36%, 60% and 4% respectively. An elitist strategy was followed, in that the best individual was automatically retained in successive generations. The trees were constrained to a maximum depth of 7, which is not particularly restrictive, given the simplicity of the data sets. Individuals were selected by means of a 3-way tournament method, while the fitness of each individual was defined as the inverse of the Sammon stress (Eq. 5.5). Three runs were conducted for each data set and the average stress was recorded. Table 5.2 outlines the parameters used for each run.

Table 5.2 : *Parameters used for each data set during feature extraction.*

Data set	Bitet	Spiral	Sphereshell	Iris
Population size	100	100	100	100
Tree levels	7	7	7	7
Terminal set (*T*)	{A, B, C, D}	$\{x_1, x_2, x_3\}$	$\{x_1, x_2, x_3\}$	$\{x_1, x_2, x_3, x_4\}$
Function set (*F*)	+, -, *, σ	+, -, *, σ	+, -, *, σ	+, -, *, σ
Target dimension (q)	2	2	2	2
Crossover rate (P_c)	60%	60%	60%	60%
Mutation rate (P_m)	4%	4%	4%	4%

Stellenbosch University https://scholar.sun.ac.za

5.3.1.2 *Results obtained*

Since the algorithm had to extract more than one feature, the trees in the population had composite structures, as shown in Figure 5.1. This meant that for each feature (in this case two), the exchange of genetic material was confined to trees with similar indices, representing a specific feature (Figure 5.2). The co-ordinates of the individual points in each data set were presented to the genetic programming algorithm, which projected the data to a two-dimensional feature spaces with coordinates y_1 and y_2. The results visualized in Figures 5.3-5.6 and are summarized in Table 5.3, were they are also compared with those obtained by other researchers making use of other methods

Table 5.3 : ***A comparison of stress values (Sammon stress) obtained from six different projection algorithms for the four data sets.***

Data set	GP [a]	SAM	SNN1	SNN2	SNN3	SNN4
Bitet [b]	**0.0513**	-	**0.0597**	-	-	-
Spiral [c]	**0.004**	**0.029**	**0.02428**	**0.34153**	**0.34435**	**0.34579**
Sphereshell [d]	**0.0536**	**0.00089**	**0.05036**	**0.09**	**0.03**	**0.05**
Iris [e]	**0.009**	**0.01**	**0.005**	**0.0125**	**0.0617**	**0.0424**

[a] Average stress for three runs for each of the data sets (b) - (e) below
[b] $(\rho_1, \rho_2) = (x_1, x_1+x_2)$; (x_2, x_3-x_1) and (x_3-x_2, x_1).
[c] $(\rho_1, \rho_2) = (x_2, x_3)$; (x_3, x_2) and $(x_2^2/x_1^2, x_3)$.
[d] $(\rho_1, \rho_2) = (x_1, x_2)$; (x_2, x_1) and (x_1, x_2).
[e] $(\rho_1, \rho_2) = (x_1-x_2, x_3^2/x_4+x_2x_3)$; (x_2, x_1+x_4) and $(-x_1-x_4, x_2)$.

to map the data. Specifically, SAM refers to the original algorithm proposed by Sammon (1969), based on the use of Eq. 5.2. SNN1 refers to a multilayer perceptron-type neural network described by Tattersall and Limb (1994). SNN2 refers to the same type of network as SNN1, except that a PCA network (Rubner and Schulten, 1990), (Rubner and Tavin, 1989) was first used to project the data, and the weights from this network were consequently used to initialize the Sammon neural network (Mao and Jain, 1995; Pal and Eluri, 1998). SNN3 refers to alternative strategies proposed by Pal and Eluri (1998), making use of statistical sampling and subsets to reduce the $n(n-1)/2$ number of calculations involved in the computation of the quality of the maps (Sammon stress). With SNN4 (Pal and Eluri, 1998) the idea is the same, except that a Kohonen map is

Stellenbosch University https://scholar.sun.ac.za

used to extract a small, but adequate representation of the data set prior to generating a Sammon map with a multilayer perceptron.

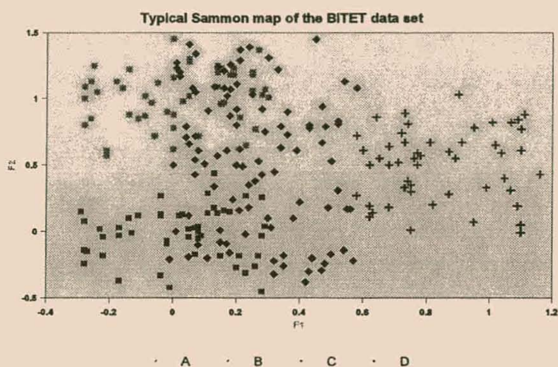


Figure 5.3 : *Typical Sammon map of the BITET data set, generated by the Genetic Programming algorithm, S = 0.0472, $F_1 = x_1$ and $F_2 = x_1 + x_2$. The clusters are indicated by different labels, as shown in the legend*

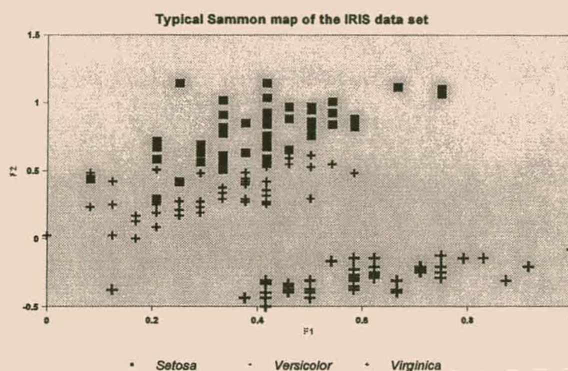


Figure 5.5 : *Typical Sammon map of the IRIS data set generated by the Genetic Programming algorithm, S = 0.00657, $F_1 = x_3 + x_4/[1+\exp(x_1)]$; $F_2 = x_2$.*

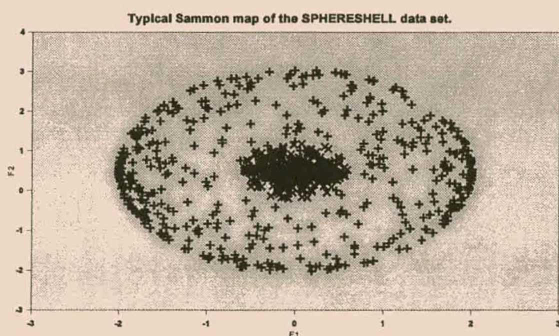


Figure 5.4 : *The Sammon map of the SPHERESHELL data set,generated by the Genetic Programming algorithm, S = 0.0531, $F_1 = x_2$ and $F_2 = x_1$.*

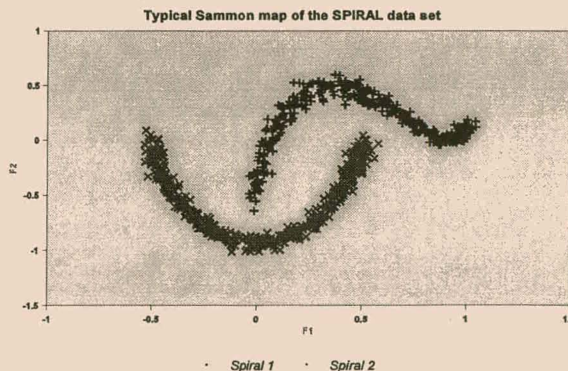


Figure 5.6 : *Typical Sammon map of the SPIRAL data set, generated by the Genetic Programming algorithm, S = 0.00403, $F_1 = x_3$ and $F_2 = x_2$.*

Stellenbosch University https://scholar.sun.ac.za

5.3.2 Flotation data from an Australian base metal flotation plant

5.3.2.1 A description of each data set

The following data set was collected from a base metal flotation plant. It consisted of approximately 1500 observations, 13 variables that described the ore and reagent feed rates to the plant, as well as other operating conditions. The variables were denoted as $\boldsymbol{x}_1, \boldsymbol{x}_2, \ldots, \boldsymbol{x}_{13}$.

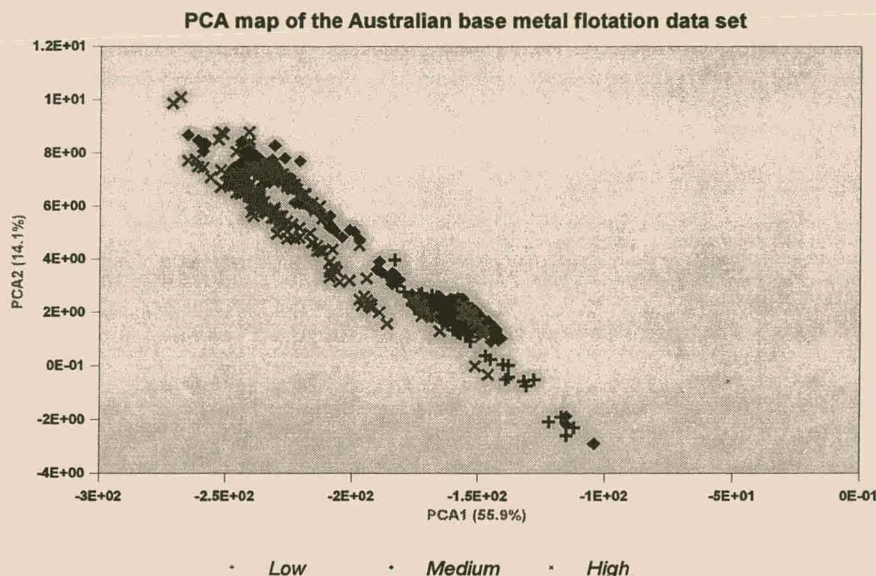


Figure 5.7 : ***Principal component map of 13 plant variables on a base metal flotation plant. The first two principal components (PC_1 and PC_2) explained 55.9% and 14.1% of the variation in the data respectively. The discretized values of the concentration of the valuable metal (not part of the mapped data set) is superimposed on the map.***

As before, the terminal set of the genetic programming algorithm contained the variables, $\boldsymbol{T} = \{x_1, x_2, \ldots, x_{13}\}$, while the function set contained the four basic arithmetic operators, that is $\boldsymbol{F} = \{+, -, *, /\}$. The same default values, i.e. a population size of 100,

Stellenbosch University https://scholar.sun.ac.za

a maximum tree depth of 7 and reproduction, crossover and mutation probabilities of respectively 36%, 60% and 4% were used to map the data.

5.3.2.2 *Results obtained*

The plant data exhibited a clustered structure, owing to the way in which the plant was operated. This is shown in a principal component map of the data in Figure 5.7. Here the concentration of one of the valuable metals have been superimposed on the data, in a discretized form as "high", "medium" and "low".

By mapping these thirteen features, three large clusters can be discerned, that is indicative of the different operating regimes on the plant, as shown in Figure 5.8.

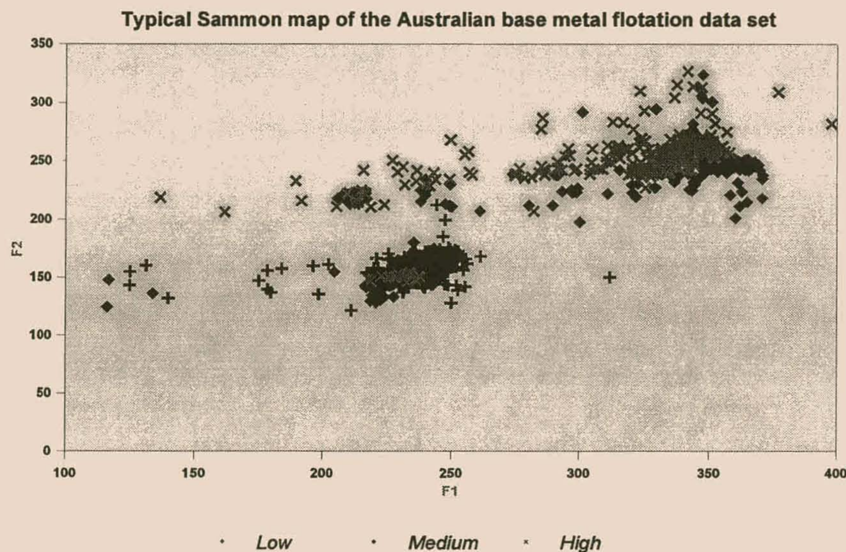


Figure 5.8 : ***Sammon map of the base metal flotation data generated by the Genetic Programming algorithm with S = 0.00473, $F_1 = x_6 - x_{12}$ and $F_2 = 1 + x_1 + x_4 - x_7 + x_{11} - x_8/x_{11}$.***

Although the first principle component only explains 55.9% of the variation, some degree of separation is possible using principle component analysis. Genetic programming, on the other hand, allows greater separation (see Figures 5.8). Unfortunately, it is not known how much variation is explained via the genetic programming mapping because the two algorithms use different mapping objectives.

Stellenbosch University https://scholar.sun.ac.za

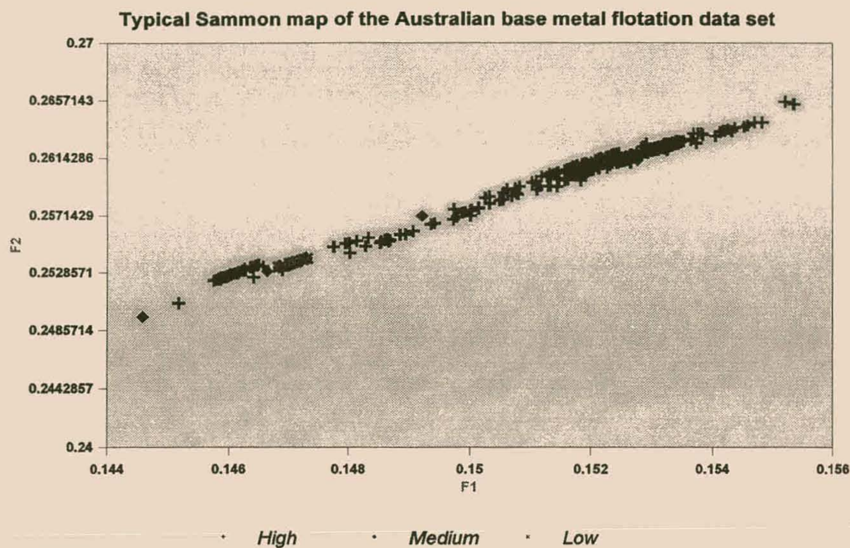


Figure 5.9 : ***Sammon map of the base metal flotation data generated by the multilayer perceptron neural network, with a Sammon stress of S = 0.02473.***

A multilayer perceptron neural network as proposed by Tattersall and Limb (1994) was also used for comparative purposes. After using various parameter combinations (i.e. changing the learning rate, number of hidden nodes, number of epochs, etc.), the mappings obtained appeared consistent over the range of runs, as shown by way of Figure 5.9. Two clusters, representing the "high" and "low" concentration can be discerned, although they are not well separated. The "medium" concentration appears also more clustered around the "high" concentrate. A Sammon stress of 0.02473 was attained, compared to a value of 0.00473 that was obtained via genetic programming. Very little separation was obtained through the neural network.

5.3.3 Three-phase oil flow data.

5.3.3.1 A description of each data set

The flow of oil and water emulsions in pipes can be classified as homogeneous, annular and stratified. 1000 measurements were made on twelve variables $\boldsymbol{v}_1, \boldsymbol{v}_2, \ldots, \boldsymbol{v}_{12}$. These data were mapped to two dimensions (F_1 and F_2) using genetic programming with $\boldsymbol{T} = \{\boldsymbol{v}_1, \boldsymbol{v}_2, \ldots, \boldsymbol{v}_{12}\}$ and $\boldsymbol{F} = \{+, -, *, /, \sin, \cos, \tan, \exp, \log, \sigma\}$. For comparative purposes,

Stellenbosch University https://scholar.sun.ac.za

the runs were repeated using a multilayer perceptron-type neural network as described by Tattersall and Limb (1994).

5.3.3.2 Results obtained

The results from different runs, using genetic programming, can be seen in Figures 5.10, 5.11 and 5.12. Figure 5.13 presents the results obtained using the technique proposed by Tattersall and Limb (1994). Although all the first three maps shown in these figures have more or less the same Sammon stress value, the appearances of the projections are different. From Figure 5.10 the stratified flow is manifested in four relatively small elongated clusters, surrounding two larger clusters representing the annular and the homogeneous flows.

These two clusters appear to be rather spherical and not very distinct. In Figure 5.11 the clusters representing the annular and homogeneous flows are more distinct, while the clusters representing the stratified flows appear to be less elongated. Figure 5.11

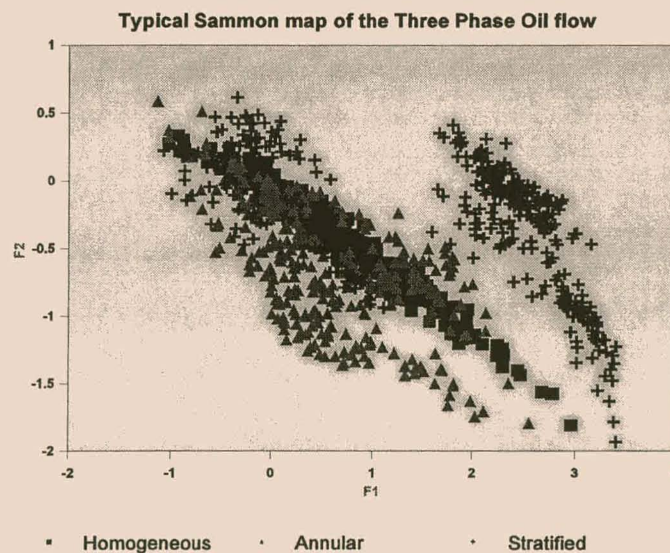


Figure 5.10 : *Three-phase flow with* $S = 0.05270, F_1 = v_2 - v_3 + v_{10} + 1/(1+\exp(v_{12})$ *and* $F_2 = v_7/[1+\exp(v_1 v_4)] + v_7 - 1/\{1+\exp[1/(1+\exp(v_1 v_4))]\} - v_4$.

is similar to Figure 5.12 in appearance, despite the simpler model relating the measured variables, v_1, v_2, ..., v_{12}, with the features F_1 and F_2. In Figure 5.13 the appearance of

Stellenbosch University https://scholar.sun.ac.za

the clusters are very similar to those obtained in Figures 5.11 and 5.12 but the Sammon stress is lower (0.0324). The stratified flows, however, are better separated than in the previous two figures but it is also more clustered. The two larger clusters, representing the annular and the homogeneous flows, are still not as distinct.

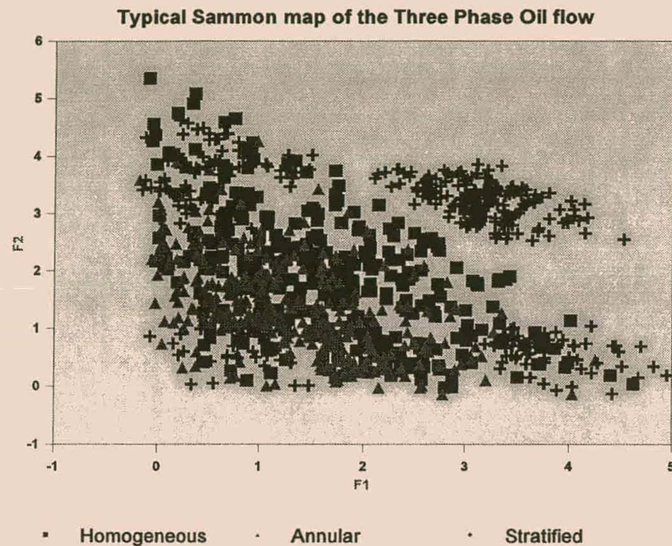


Figure 5.11 : *Three-phase flow with $S = 0.05293, F_1 = v_2 + v_6 + v_{10}$ and $F_2 = v_7 + 2v_5$.*

Since the Sammon stress criterion is not uniquely related to a specific projection, the

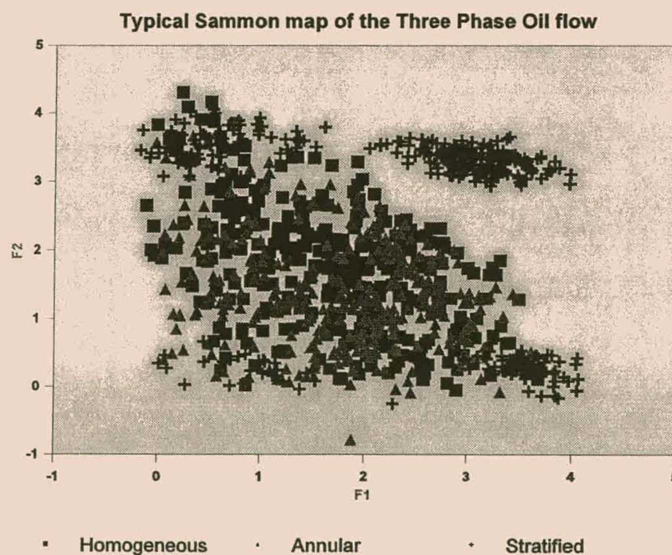


Figure 5.12 : *Three-phase flow with $S = 0.04943, F_1 = 2\sin(v_4) + v_{10}$ and $F_2 = \sin(\sin(v_7)) + v_5 + v_7$.*

Stellenbosch University https://scholar.sun.ac.za

generation of different maps is an advantage that can enhance the interpretation of the structure of process systems. Genetic programming provides a natural way of generating different types of maps, which could not readily be duplicated by use of neural networks, for example.

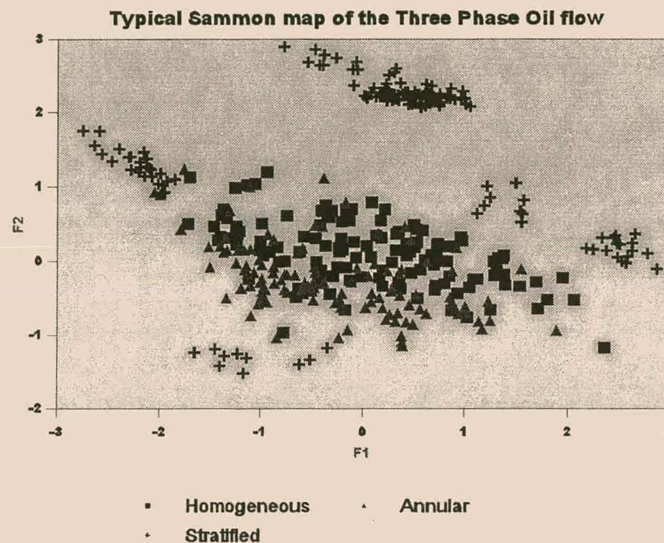


Figure 5.13 : ***The results obtained using a multilayer-perceptron neural network. The Sammon stress, S = 0.0324. The stratified flows appear more distinct but also more clustered, from the homogeneous and annular clusters.***

5.4 Results and conclusions

By comparing the results obtained using genetic programming and those using a multilayer-perceptron neural network, one can see that the results appear very similar. The neural network approach offered better separation of the individual clusters in the three phase oil data set, whereas the neural network approach was not able to separate the Australian base metal flotation data set sufficiently compared to the results obtained via genetic programming. This demonstrates the powerful capabilities of this novel approach toward data visualization. By making use of evolutionary computation to project high-dimensional data sets to lower-dimensional spaces, a population of projections is generated. Evolutionary computation is a natural way of generating multiple projections of a data set, which collectively can be more revealing than single

Stellenbosch University https://scholar.sun.ac.za

projections, such as those generated by neural networks. The quality of the maps was influenced mainly by the composition of the function set. Reliable maps could be generated consistently by inclusion of the basic arithmetic functions {+, -, *, /} only, which also tended to yield relatively simple, linear models for most of the cases considered in this investigation.

Perhaps the most important advantage is that by use of genetic programming, relatively simple and explicit models relating the original variables and the projected variables or features can be formed. This is not the case when neural networks or other methods of optimization is used, and can be potentially useful where these types of maps are used in the monitoring of multivariate processes, since process variables, giving rise to deviations from the norm can be more readily identified. Also, the importance of each variable become immediately apparent from the mapping functions, similar to the loadings of the variables in principal component analysis. Once again this is not so obvious when using neural networks or other techniques.

Stellenbosch University https://scholar.sun.ac.za

CHAPTER 6

RECOMMENDATIONS FOR FUTURE RESEARCH

During this research several limitations were encountered with the genetic programming algorithm and with a-GP, in particular.

- One of these (for both algorithms) is the lack of a local optimization procedure. This drawback resulted in the proliferation of complex tree-like structures in successive generations as reported in Chapter 4. These complex structures were necessary to estimate parameters within the model structures. Although some researches have tried to use other searching algorithms in parallel with the genetic programming algorithm, no significant (if any!) improvements were obtained[1]. A possible solution to this problem of local optimization is to generate and evolve solutions in such a way that, when they are simplified, they can be expressed in the following form

$$I_k = a_1 F_1 + a_2 F_2 + \ldots + a_n F_n; \quad n \in \{1, 2, \ldots, N\} \tag{6.1}$$

  where I_k represents individual k, and F_1 to F_n are sub-trees in the individual with weights, $a_1 \ldots a_n$, attached to them. The weights or constants, $a_1 \ldots a_n$, can then be calculated through a linear batch regression technique. This will yield considerable improvements in terms of speed and memory usage as opposed to a technique that employs two concurrent searching algorithms.

- Time-series prediction can be achieved via a static encoding of the time-series. A future implementation could look at the use of an autoregressive moving average (ARMA) model which can be expressed as

---

[1]See Chapter 2.4.2

Stellenbosch University https://scholar.sun.ac.za

$$X_t = a_0 + \sum_{j=1}^{k} a_j X_{t-j} + \sum_{j=0}^{l} b_j \varepsilon_{t-j} \tag{6.2}$$

to use this model for a-GP, we can replace X_t by I_k, as in (6.1), and rewrite (6.2) as

$$I_{k_t} = a_0 + \sum_{j=1}^{k} a_j I_{k_{t-j}} + \sum_{j=0}^{l} b_j \varepsilon_{t-j} \tag{6.3}$$

The ARMA model is a well studied and widely used (since the late 1920's) implementation for linear time-series modelling.

- A lack of diversity increases the probability of entrapment within a local optimum. This arises out of natural selection, which allows the best individual to dominate large portions of the population over succeeding generations. For now, the only way of ensuring diversity is to use several populations (or demes) in parallel. Mutation too, allows some degree of diversity but selecting a too high mutation rate will result in an inefficient local search. Some technique is required to compute the inter-spatial distance between two individuals within the GP search space and hence ensure that only individuals, which are in close proximity of one another, are allowed to mate. This would be analogous to the technique employed in genetic algorithms which uses the Hamming distance between solutions to enforce local mating.
- The disruptive nature of the crossover operation was not addressed in this thesis. Research will need to be done on ways of minimizing the displacement of individuals in the search-space after applying crossover. This, once again, reinforces the need to find some way of computing the inter-spatial distance between individuals.

Stellenbosch University https://scholar.sun.ac.za

CHAPTER 7

CONCLUSIONS

Several alterations have been proposed in this thesis to improve the original genetic programming algorithm as proposed by Koza (1992).

- Although the original algorithm was implemented in LISP, which is an interpreted language; to increase speed and scalability, it had to be designed and implemented in an object oriented compiled language. C++ was used for this purpose. The implementation of each solution was altered to such an extend that every node in the tree-like structure could be stored in 1 byte of computer memory as opposed to the 22 bytes required by other implementations. This resulted in a significant reduction in resources required by the algorithm. Also the evaluation scheme was changed from node-branch reduction, to a simple stack-based RPN[1] evaluation which is much faster and non-recursive. A significant increase in convergence and robustness in regression models, was also obtained by changing the implementation of the fitness function from an error-based fitness function to a correlation-based fitness function in conjunction with a correction filter. Unfortunately, the unsimplified tree-like structures were more complex when a correlation-based fitness function was used. The correction filter was needed to eliminate any scale or bias in the final models, which affected the SSE but not the R^2.
- Chapter 4 saw the application of the newly improved algorithm, a-GP, in the development of regression models on four case studies. The algorithm was compared to other algorithms such as: standard genetic programming (using an

[1]Reverse Polish Notation

Stellenbosch University https://scholar.sun.ac.za

error-based fitness function), a multilayer perceptron neural network and linear regression. a-GP improved significantly on genetic programming on all four case studies and performed very similar to the neural network. Unfortunately, the evolved models were too complex. This can be attributed to the lack of parameter estimation which the genetic programming algorithm tries to compensate for by evolving complex tree structures; which it uses to approximate the parameters. This demonstrates the need for a local optimization procedure to generate and optimize parameters in the genetic programming algorithm.

- As a data visualization tool, genetic programming compares favourably with other techniques proposed by various researchers in the literature. Four bench marking data sets were used for comparative purposes. The final results compared favourably with the other techniques suggested by various researchers. Additionally the algorithm was applied to flotation data obtained from an Australian base metal flotation plant in which thirteen variables in the plant was transformed to two dimensions. The concentration of one of the valuable metals were superimposed on the data, in a discretized form as "high", "medium" and "low". By mapping these thirteen features, three large clusters were discerned, which was indicative of the different operating regimes on the plant. The results were similar to those derived from the first two principal components of the data implicating that the data was linearly separable. Finally, the flow of oil and water emulsions in pipes, which can be classified as "homogeneous", "annular" and "stratified" was analysed. The original twelve variables were projected to a two-dimensional map. The resulting projections from three different runs were all different in appearance, although the Sammon stress was more or less the same. The first projection showed the stratified flow manifested in four relatively small elongated clusters, which surrounded two larger clusters representing the annular and the homogeneous flows. The two clusters appeared to be rather spherical and not very distinct. In the second projection the clusters representing the annular and homogeneous flows were more distinct, while the clusters representing the stratified flows appeared to be

Stellenbosch University https://scholar.sun.ac.za

less elongated. The final projection was similar in appearance to the latter, albeit with a simpler model. Genetic programming, however had the additional benefit of being able to generate a population of projection maps which, collectively, could be more revealing than single projections, such as those generated by neural networks. Perhaps the most important advantage was that by use of genetic programming relatively simple and explicit models relating the original variables and the projected variables or features could be formed. This is not the case when neural networks or other methods of optimization are used, and could be potentially useful where these types of maps are used in the monitoring of multivariate processes, since process variables, giving rise to deviations from the norm can be more readily identified.

In conclusion, a-GP is an extremely viable tool for both regression modelling and data visualization. It compares favourably with other existing methods. However a-GP (or genetic programming for that matter) *does not* yield simple symbolic models when used in regression modelling. The algorithm lacks a local optimization procedure which severely restricts its usage to evolve simple symbolic functions.

As a data visualization tool, a-GP *does* generate simple symbolic projection functions. These functions are more revealing than the non-parametric models obtained from neural networks. A possible explanation for this discrepancy for not being able to evolve simple functions for both cases can be that: regression requires a mapping, in such a way, that the projected data is an *exact* replica (in the mean squared error sense) of the desired output, whilst data visualization (using a Sammon mapping criterion) requires a mapping, in such a way, that the interspatial distance, between data residing in the input space and that in the projected space, is minimized.

Stellenbosch University https://scholar.sun.ac.za

REFERENCES

(Angeline, 1997)

Angeline, P.J., 1997. Subtree Crossover: Building block engine or Macromutation? *Genetic Programming 1997: Proceedings of the Second Annual Conference,* Morgan Kaufmann Publishers, July 13-16, 9-17.

(Andre, 1995)

Andre, D., 1995. The Evolution of Agents that Build Mental Models and create simple plans using Genetic Programming. *Proceedings of the Sixth International Conference on Genetic Algorithms,* Morgan Kaufmann Publishers, Inc., July 15-19, 248-255.

(Atwar, 1994)

Atmar, W., 1994. Notes on the Simulation of Evolution. *IEEE Transactions on Neural Networks,* **5**(1), January, 130-147.

(Biswas *et al.,* 1981)

Biswas, G., Jain, A.K. and Dubes, R.C., 1981. Evaluation of projection algorithms. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, **PAMI-3**(6), 701-708.

(Breiman *et al.*, 1984)

Breiman, L., Friedman, J.H., Olshen, R.A. and Stone, C.J., 1984. *Classification*

Stellenbosch University https://scholar.sun.ac.za

and Regression Trees. Chapman & Hall, 115 Fifth Avenue, New York, NY 10003.

(Chang and Lee, 1973)

Chang, C.L. and Lee, R.C.T., 1973. A heuristic relaxation method for nonlinear mapping in cluster analysis. *IEEE Transactions on System, Man and Cybernetics*, **SMC-3**, 197-200.

(Chen *et al.*, 1999)

Chen, Z.P., Jiang, J.H., Li, Y. and Yu, R.Q., 1999. Nonlinear mapping using real valued genetic algorithm. *Chemometrics and Intelligent Laboratory Systems*, **45**, 409-418.

(Daida *et al.*, 1996)

Daida, J.M., Bersano-Begey, T.F., Ross, S.J. and Vesecky, J.F., 1996. Evolving feature-extraction algorithms: adapting genetic programming for image analysis in geoscience and remote sensing. *International Geoscience and Remote Sensing Symposium (IGARSS),* Part 3 and 4, **4**, IEEE, Piscataway, NJ, USA,96CB35875. 1520-1522, 2077-2079.

(Das *et al.*, 1995)

Das, G.K., Acharya, S., Anand, S. and Das, R.P., 1995. Acid pressure leaching of nickel-containing chromite overburden in the presence of additives. *Hydrometallurgy*, **39**, 117-128.

(DeJong, 1975)

DeJong, K., 1975. An analysis of the behaviour of a class of genetic adaptive systems. *Dissertation Abstracts International,* **36**(10), 5140B.

(Del Giudice and Amabile, 1997)

Del Giudice, V. and Amabile, R., 1997. The appraisal of the Hedonic prices with

Stellenbosch University https://scholar.sun.ac.za

Neural Network Models. An alternative approach to multiple regression analysis. *5th European Congress on Intelligent Techniques and Soft Computing*, Aachen, Germany, September 8-11, Proceedings, **1**, 448-453.

(Dong and McAvoy, 1996)

Dong, D. and McAvoy, T.J., 1996. Nonlinear principal component analysis - based on principle curves and neural networks. *Computers and Chemical Engineering,* **20**(1), 65-77.

(Dracopoulos, 1997)

Dracopoulos, D.C., 1997. Evolutionary Control of a Satellite. *Genetic Programming 1997: Proceedings of the Second Annual Conference.* Morgan Kaufmann Publishers, July 13-16, 77-81.

(Francone *et al.*, 1996)

Francone, F.D., Nordin, P. and Banzhaf, W., 1996. Benchmarking the generalization capabilities of a Compiled Genetic Programming System using sparse data sets. *Proceedings of the First International Conference on Genetic Programming,* MIT Press, 72-80.

(Gray *et al.*, 1996)

Gray, G.J., Li, Y., Murray-Smith, D.J. and Sharman, K.C., 1996. Structural system identification using genetic programming and a block diagram oriented simulation tool. *Electronics Letters,* **32**(15), July, 1422-1424.

(Greeff and Aldrich, 1998)

Greeff, D.J. and Aldrich, C., 1998. Empirical Modelling of Chemical Process Systems with Evolutionary Programming. *Computers and Chemical Engineering,* **22**(7-8), 995-1005.

(Gritz and Hahn, 1997)

Stellenbosch University https://scholar.sun.ac.za

Gritz, L. and Hahn, J.K., 1997. Genetic Programming Evolution of Controllers for 3-D character animation. Genetic Programming 1997: *Proceedings of the Second Annual Conference*, Morgan Kaufmann Publishers, July 13-16, 139-146.

(Haynes *et al.*, 1995)

Haynes, T., Wainwright, R., Sen, S. and Schoenefeld, D., 1995. Strongly Typed Genetic Programming in Evolving Cooperation Strategies. *Proceedings of the Sixth International Conference on Genetic Algorithms,* Morgan Kaufmann Publishers, Inc., July 15-19, 271-278.

(Herrmann, 1994)

Herrmann, E., 1994. Asymptotic distribution of bandwidth selectors in kernel regression estimation. *Statistical Papers,* **35**, 17-26.

(Hiden *et al.*, 1997)

Hiden, H., Willis, M., McKay, B. and Montague, G., 1997. Non-linear and direction dependant dynamic modelling using Genetic Programming. *Genetic Programming 1997: Proceedings of the Second Annual Conference,* Morgan Kaufmann Publishers, July 13-16, 168-173.

(Holland, 1992)

Holland, J.H., 1992. *Adaptation in Natural and Artificial Systems.* An Arbor, MI: University of Michigan Press.

(Hondo *et al.*, 1997)

Hondo, N., Iba, H. and Kakazu, Y., 1997. Automatic generation of robot behavior using extended genetic programming (acquisition of partial behavior with library). *Nippon Kikai Gakkai Ronbunshu,* C Hen/Transactions of the Japan Society of Mechanical Engineers, Part C, **63**(609), May, 1685-1692.

(Hornik *et al.*, 1989)

Stellenbosch University https://scholar.sun.ac.za

Hornik, K., Stinchcombe, M. and White, H., 1989. Multilayer feedforward neural networks are universal approximators. *Neural Networks,* **2**, 502-516.

(Howard and D'Angelo, 1995)

Howard, L.M. and D'Angelo, D.J., 1995. The GA-P: A Genetic Programming hybrid. *IEEE Expert,* **10**(3), 11-15.

(Iba and Sato, 1992)

Iba, H. and Sato, T., 1992. Meta-level Strategy for genetic algorithms based on Structured Representation. *Proceedings of the 2nd Pacific Rim International Conference on Artificial Intelligence,* **1**, September, 548-554.

(Iba and Sato, 1995[a])

Iba, H. and Sato, T., 1995[a]. Temporal Data Processing using Genetic Programming. *ETL Technical Report-95-1.*

(Iba and Sato, 1995[b])

Iba, H. and Sato, T., 1995[b]. Extending Genetic Programming with Recombinative Guidance. *ETL Technical Report-95-15.*

(Iba *et al.,* 1996)

Iba, H., deGaris, H. and Sato, T., 1996. A Numerical Approach to Genetic Programming for System Identification. *Evolutionary Computation,* 3(4), 417-452.

(Jolliffe, 1986)

Jolliffe, I.T., 1986. *Principal Component Analysis.* Springler-Verlag.

(Jones, 1995)

Jones, T., 1995. Crossover, Macromutation and Population-based search. *Proceedings of the Sixth International Conference on Genetic Algorithms.*

Stellenbosch University https://scholar.sun.ac.za

Morgan Kaufmann Publishers, Inc., July 15-19, 73-80.

(Keane, 1995)

Keane, A.J., 1995. Genetic Algorithm optimization of multi-peak problems: studies in convergence and robustness. *Artificial Intelligence in Engineering,* **9**, 75-83.

(Kendall, 1975)

Kendall, M., 1975. *Multivariate Analysis.* Charles Griffin & Co.

(Koza, 1992)

Koza, J.R., 1992. *Genetic Programming: On the Programming of Computers by means of Natural Selection,* The MIT Press, Cambridge, Massachusetts.

(Kraaiveld *et al.*, 1995)

Kraaiveld, M.A., Mao, J. and Jain, A.K., 1995. A Nonlinear Projection Method based on Kohonen's Topology Preserving Maps. *IEEE Transactions on Neural Networks,* **6**(3), May, 548-559.

(Kulkarni *et al.* 1999)

Kulkarni, B.D., Tambe, S.S., Dahule, R.K. and Yadavalli, V.K., 1999. Consider genetic programming for process identification. *Hydrocarbon Processing*, July, 89-97.

(Langdon, 1995)

Langdon, W.B., 1995. Evolving data structures with Genetic Programming. *Proceedings of the Sixth International Conference on Genetic Algorithms,* Morgan Kaufmann Publishers, Inc., July 15-19, 295-302.

(Lampinen and Oja, 1995)

Stellenbosch University https://scholar.sun.ac.za

Lampinen, J. and Oja, E., 1995. Distortion Tolerant Pattern Recognition Based on Self-Organizing Feature Extraction. *IEEE Transactions on Neural Networks,* **6**(3), May, 539-547.

(MacGregor, 1989)

MacGregor, J., 1989. Multivariate statistical methods for monitoring large data sets from chemical processes. *AICHE Meeting*.

(Mao and Jain, 1995)

Mao, J. and Jain, A.K., 1995. Artificial Neural Networks for feature extraction and Multivariate data projection. *IEEE Transactions on Neural Networks*, **6**(2), March, 296-317.

(McKay *et al.*, 1997)

McKay, B., Willis, M. and Barton, G., 1997. Steady-state Modelling of Chemical Process Systems using Genetic Programming. *Computers and Chemical Engineering,* **21**(9), 981-996.

(Montana, 1994)

Montana, D.J., 1994. Strongly typed genetic programming. *Technical Report 7866, Bolt Beranek and Newman, Inc.*, March 25.

(Nordin and Banzhaf, 1995)

Nordin, P. and Banzhaf, W., 1995. Evolving Turing-Complete Programs for a register machine with self-modifying code. *Proceedings of the Sixth International Conference on Genetic Algorithms.* Morgan Kaufmann Publishers, Inc., July 15-19, 318-325.

(Pal and Eluri, 1998)

Stellenbosch University https://scholar.sun.ac.za

Pal, N.R. and Eluri, V.K., 1998. Two Efficient Connectionist Schemes for Structure Preserving Dimensionality Reduction. *IEEE Transactions on Neural Networks,* **9**(6), November, 1142-1153.

(Piovoso *et al.*, 1992)

Piovoso, M.J., Kosanovich, K.A. and Yuk, J.P., 1992. Process data chemometrica. *IEEE Trans. Instrum. Meas.,* **41**(2), 262-268.

(Pykett, 1978)

Pykett, C.E., 1978. Improving the efficiency of Sammon's nonlinear mapping bu using clustering archetypes. *Electronic Letters*, **14**, 799-800.

(Rubner and Schulten, 1990)

Rubner, J. and Schulten, K., 1990. Dvelopment of feature detectors by self-organizing. *Biological Cybernetics*, **62**, 193-199.

(Rubner and Tavan, 1989)

Rubner, J. and Tavan, P., 1989. A self-organizing network for principal component analysis. *Europhysics Letters*, **10**, 693-698.

(Sammon, 1969)

Sammon, J.W., 1969. A nonlinear mapping for data structure analysis. *IEEE Transactions on Computers*, **C-18**, 401-409.

(Schachter, 1978)

Schachter, B., 1978. A nonlinear mapping algorithm for large databases. *Computational Graphics and Image Processing*, **7**, 271-278.

(Seborg *et al.*, 1989)

Seborg, D.E., Edgar, T.F. and Mellichamp, D.A., 1989. *Process Dynamics and Control.* John Wiley and Sons, Inc., New York.

Stellenbosch University https://scholar.sun.ac.za

(Shakouri *et al.*, 1997)

Shakouri, H.G., Nikravesh, K.Y. and Menhaj, M.B., 1997. The center of the possibility distribution in Fuzzy Regression Analysis with applications to real economic modeling. *5th European Congress on Intelligent Techniques and Soft Computing*, Aachen, Germany, September 8-11, Proceedings, **1**, 335-339.

(Sharman *et al.*, 1995)

Sharman, K.C., Esparcia Alcázar, A.I. and Li, Y., 1995. Evolving signal processing algorithms by genetic programming. *Proceedings of IEE/IEEE Genetic Algorithms in Engineering Systems: Innovations and Applications (GALESIA).*

(Stephanopoulos and Guterman, 1989)

Stephanopoulos, G.N. and Guterman, H., 1989. Pattern recognition in fermentation processes. *ACS Meeting*, Miami Beach.

(Srinivas and Patnaik, 1994[a])

Srinivas, M. and Patnaik, L.M., 1994.Genetic Algorithms : A Survey. *IEEE*, **27**(6), 17-26.

(Srinivas and Patnaik, 1994[b])

Srinivas, M. and Patnaik, L.M., 1994. Adaptive Probabilities of Crossover and Mutation in Genetic Algorithms : A Survey. *IEEE Transactions on Systems, Man and Cybernetics*, **24**(4), April, 656-667.

(South *et al.*, 1995)

South, M.C., McConnel, S., Tham, M.T. and Willis, M.J., 1995. Data analysis via symbolic regression, *Submitted to the Trans. Ichem.*

Stellenbosch University https://scholar.sun.ac.za

(Tacket, 1994)

Tacket, W.A., 1994. Recombination, Selection and the Genetic Construction of Computer Programs. *Dissertation, Faculty of the Graduate School, UCLA, CA.*

(Tamburino and Zmuda, 1995)

Tamburino, L.A. and Zmuda, M.A., 1995. Generating Pattern-Recognition Systems using Evolutionary Learning. *IEEE Expert*, **10**(4), August, 63-68.

(Tanese, 1989)

Tanese, R., 1989. Distributed Genetic Algorithms. *Proceedings of the Third International Conference on Genetic Algorithms.* Morgan Kaufmann Publishers, Inc., 434-439.

(Tattersall and Limb, 1994)

Tattersall, G.D. and Limb, P.R., 1994. Visualization techniques for data mining. *BT Technology Journal*, **12**(4), 23-31.

(Teller and Veloso, 1995)

Teller, A. and Veloso, M., 1995. Algorithm evolution for face recognition: what makes a picture difficult? *Proceedings of the IEEE Conference on Evolutionary Computation*, IEEE, Piscataway, NJ, USA, **2**, 608-613.

(Watson and Parmee, 1997[a])

Watson, A.H. and Parmee, I.C., 1997[a]. Steady state Genetic Programming with Constrained Complexity Crossover using Species sub-populations. *Genetic Programming 1997: Proceedings of the Second Annual Conference,* Morgan Kaufmann Publishers, July 13-16, 329.

Stellenbosch University https://scholar.sun.ac.za

(Watson and Parmee, 1997[b])

Watson, A.H. and Parmee, I.C., 1997[b]. An Improved Genetic Programming strategy for preliminary design model development. *5th European Congress on Intelligent Techniques and Soft Computing* Aachen, Germany, September 8-11, Proceedings. **1**, 682-686.

(Whigham, 1995)

Whigham, P.A., 1995. Schema theorem for context-free grammars. *Proceedings of the IEEE Conference on Evolutionary Computation*, IEEE, Piscataway, NJ, USA, **1**, 178-181.

(Winkeler and Manjunath, 1997)

Winkeler, J.F. and Manjunath, B.S., 1997. Genetic Programming for object detection. *Genetic Programming 1997: Proceedings of the Second Annual Conference,* Morgan Kaufmann Publishers, July 13-16, 330-335.

Stellenbosch University https://scholar.sun.ac.za

NOMENCLATURE

$\hat{\mathbf{y}}$	Predicted output vector
$\gamma(F_i)$	Total number of individuals that can be constructed with function F_i
y_k	k'th output of the observed output vector
$\hat{\mathbf{y}}_i$	Predicted output vector of individual i
$\hat{y}_{i_k}$	k'th output of the predicted individual i
λ_s	Linear separability
ρ	Population correlation
σ_{T-T}	Standard deviation of the difference in the R^2 between the training data and the testing data.
a-GP	Augmented Genetic Programming
ADF	Automatically Defined Functions
ANN	Artificial Neural Network
c	Number of classes/clusters
C	Optimization criterion
CCC	Constraint Complexity Crossover
CGP	Compiled Genetic Programming
d	Dimensionality
d_{ij}	Euclidian distance between projected points u and v in the output space, $\mathfrak{C}$.
d_{ij}^*	Euclidian distance between points i and j in the input space, $\mathfrak{I}$.
d_l	Linear dimensionality

Stellenbosch University https://scholar.sun.ac.za

ε	Randomly generated noise factor
EA	Evolutionary Algorithm
ES	Evolution Strategy
$E[X]$	Expected or mean value of variable X
$\boldsymbol{F}$	Function set
F	Functional representation of a process
f_i	Fitness of individual i
f_T	Total fitness of the population
$G(X_d, X_a)$	Correction filter to remove any bias and scale introduced in X_a.
GA	Genetic Algorithm
GP	Genetic Programming
GUI	Graphical User Interface
I_i	Individual i
MLP	Multilayer Perceptron
MSE	Mean of the squared errors
n	Number of patterns
NC	Node Complexity
P_c	Percentage crossover
PCA	Principal Component Analysis
P_m	Percentage mutation
r_{XY}	Sample correlation coefficient
$R^2(X, Y)$	Amount of variation Y explains in X.
S	Sammon stress
$\mathbf{S}(l)$	Size of the search space at level l.
SSE	Sum of the squared errors
STGP	Strongly Typed Genetic Programming
$\boldsymbol{T}$	Terminal set
X_a	Actual variable
X_d	Desired variable
$\mathbf{y}$	Observed output vector
$\Im$	Input space

Stellenbosch University https://scholar.sun.ac.za

$\mathfrak{C}$	Output space
$\wp$	Mapping function
$\Re^p$	Higher dimensional space
$\Re^q$	Lower dimensional space

Stellenbosch University https://scholar.sun.ac.za

APPENDIX A

EVOLVED MODELS

A.1 The unsimplified regression models of Chapter 4

A.1.1 Regression model for data set PINE

Predicted Model = * TEMP + * - LEAFMASS XPP * * TEMP WOODDEN + * log - PAR - LEAFMASS XPP WSPEED * TEMP WOODDEN - PAR * log - * - * LEAFMASS WOODDEN PAR * TREE TEMP + * - PAR WOODDEN * * log - PAR LEAFMASS WSPEED + * - LEAFMASS TEMP - LEAFMASS WOODDEN * TEMP TREE * TEMP WOODDEN * log + * - * LEAFMASS WOODDEN log TREE * * TEMP log - LEAFMASS AGE + * log WSPEED - LEAFMASS WOODDEN - LEAFMASS WOODDEN WOODDEN * TEMP WOODDEN

A.1.2 Regression model for data set POP

Predicted Model = * / age + + + + log exp vpd log exp - - hour seasno seasno + exp * * / exp tree PAR / * Tree age - hour seasno / / date / date Tree age day age + vpd / - - exp * * / exp hour * Tree age / * Tree seasno - date Tree / hour age + + * + exp tree exp tree age + age + exp tree exp tree log exp tree PAR / date - / PAR / hour + age exp tree + / * + exp tree exp tree age - age Tree log exp tree vpd

A.1.3 Regression model for data set BMVANO

Predicted Model = * AvrGreenCuSc * / AvrRedCuSc + * * * / AvrBlueCuSc + * FlowCuSc SMCuSc / SMCuSc SNECuSc MobilitCuSc * / AvrBlueCuSc + AvrGreenCuSc * log MobilitCuSc - - MobilitCuSc AvrRedCuSc AvrRedCuSc / AvrGreenCuSc + + AvrGreenCuSc

Stellenbosch University https://scholar.sun.ac.za

/ SMCuSc AvrBlueCuSc * log MobilitCuSc - - MobilitCuSc AvrRedCuSc AvrRedCuSc * * * / AvrGreenCuSc + * log MobilitCuSc - AvrBlueCuSc AvrRedCuSc log SMCuSc MobilitCuSc MobilitCuSc * / + AvrRedCuSc AvrGreenCuSc log SMCuSc MobilitCuSc AvrBlueCuSc * / AvrBlueCuSc AvrBlueCuSc * / AvrBlueCuSc + + * log MobilitCuSc AvrRedCuSc log / AvrBlueCuSc AvrBlueCuSc AvrGreenCuSc * / AvrBlueCuSc + AvrGreenCuSc AvrBlueCuSc * * / AvrBlueCuSc AvrRedCuSc * / AvrBlueCuSc + AvrGreenCuSc * log MobilitCuSc - MobilitCuSc AvrBlueCuSc / AvrBlueCuSc + AvrGreenCuSc * log MobilitCuSc - MobilitCuSc AvrBlueCuSc / AvrBlueCuSc + AvrGreenCuSc * log MobilitCuSc - / AvrBlueCuSc AvrBlueCuSc AvrBlueCuSc

A.1.4 Regression model for data set SOLPREP

Predicted Model = + + * / NH4OH_addition [H2SO4]after T093_o/f + + + + + + + Mn093am log [H2SO4]after log [H2SO4]after log - + H2SO4_addition H2SO4_addition [H2SO4]after log - + log - + [H2SO4]before H2SO4_addition [H2SO4]after + * / NH4OH_addition [H2SO4]after [H2SO4]after LeachFlowrate [H2SO4]after log + log + - / T093_o/f H2SO4_addition H2SO4_addition + [H2SO4]after [H2SO4]after log - + exp + H2SO4_addition H2SO4_addition LeachFlowrate log [H2SO4]after log [H2SO4]after log + - - + log + LeachFlowrate log H2SO4_addition - + NH4OH_addition LeachFlowrate [H2SO4]after [H2SO4]before + [H2SO4]before H2SO4_addition + log log + - / NH4OH_addition [H2SO4]after H2SO4_addition + - / NH4OH_addition H2SO4_addition [H2SO4]after LeachFlowrate log - + Mn093am H2SO4_addition [H2SO4]after + - Mn093am H2SO4_addition Mn093am

Stellenbosch University https://scholar.sun.ac.za

APPENDIX B

THE SOURCE CODE

B.1 Abstract base classes

B.1.1 Header file for abstract class GenericIndividual and GenericAlgorithm

```
#ifndef GenericAlgH
#define GenericAlgH
#include <vector>

#include <vcl/syncobjs.hpp>
using namespace std;

template <class T>
inline T sign(T x)
{
    return (x < 0) ? -1 : 1;
};

template <class T>
inline void Swap(T &x, T &y)
{
    T dummy = x;
    x = y;
    y = dummy;
};


template<class Type>
inline void ClearContainer(vector<Type *> &C)
{
    int N = C.size();
    for(register int i =0; i < N; i++)
```

Stellenbosch University https://scholar.sun.ac.za

```
        delete (Type *)C[i];
    C.clear();
};

typedef vector<double> VECTOR_DOUBLE;
//===================================================================
//===================================================================
//===================================================================
/*//////////////////////////////////////////////////////////////////
    -------------------------------------------------------
                GenericIndividual Class
    -------------------------------------------------------
/*//////////////////////////////////////////////////////////////////
class GenericIndividual
{
private:
protected:
public:
    GenericIndividual();
    ~GenericIndividual();
    virtual void Clone(GenericIndividual* &Target);
    double Fitness;
};
/*/
   -------------------------------------------------------
                GenericIndividual Implementation
   -------------------------------------------------------
/*/
GenericIndividual::GenericIndividual()
{
    Fitness=0.0;
};

GenericIndividual::~GenericIndividual() {};

inline void GenericIndividual::Clone(GenericIndividual* &Target)
{
    if(!Target) return;
    Target->Fitness = this->Fitness;
};
//===================================================================
//===================================================================
//===================================================================

/*//////////////////////////////////////////////////////////////////
```

Stellenbosch University https://scholar.sun.ac.za

```
    -------------------------------------------------------
                 GenericAlgorithm Class
    -------------------------------------------------------
/*/////////////////////////////////////////////////////////////////
template <class T>
class GenericAlgorithm
{
private:

protected:
    TCriticalSection* CriticalSection;
    bool Elitism;
    int PopulationSize;
    vector<T *> Population;
    double TotalFitness;
    virtual void ComputeFitness(T * &AnInd) = 0;
    virtual void ClearSystemVariables();
    virtual void UpdateSystemVariables();
    void ComputeTotalFitness();
    virtual T* GenerateAnIndividual() = 0;
    bool IndGreaterThan(GenericIndividual *I1, GenericIndividual* I2)
    {
      return I1->Fitness > I2->Fitness;
    };
public:
    GenericAlgorithm();
    ~GenericAlgorithm();
    void Initialize();

    virtual void DoAlgorithm() = 0;
    virtual double GetFitness(int i);
    void SetElitism(bool E);
    bool GetElitism();
    void SetPopulationSize(int S);
    int GetPopulationSize();
    double GetTotalFitness();
    double GetAvgFitness();
    int CurrentStep;
    bool Initialized;
    vector<double> SVBestOfFitness; //System variable
    vector<double> SVAvgFitness; //System variable
    int GetActualPopSize();
};

/*/
```

Stellenbosch University https://scholar.sun.ac.za

```
   -------------------------------------------------------
                 GenericAlgorithm Implementation
   -------------------------------------------------------
/*/
template<class T>
GenericAlgorithm<T>::GenericAlgorithm()
{
    CriticalSection = new TCriticalSection();
    CurrentStep = 0;
    TotalFitness = 0.0;
    Elitism = false;
    Initialized = false;
    PopulationSize = 50;
};

template<class T>
GenericAlgorithm<T>::~GenericAlgorithm()
{
    CriticalSection->Enter();
    ClearContainer((vector<T *>)Population);
    CriticalSection->Leave();
    ClearSystemVariables();
    delete CriticalSection;
};

template<class T>
inline int GenericAlgorithm<T>::GetActualPopSize()
{
    return Population.size();
};

template<class T>
inline void GenericAlgorithm<T>::ClearSystemVariables()
{
    SVBestOfFitness.clear();
    SVAvgFitness.clear();
};

template<class T>
inline void GenericAlgorithm<T>::UpdateSystemVariables()
{
    SVBestOfFitness.push_back( Population[0]->Fitness );
    SVAvgFitness.push_back( TotalFitness/(double)Population.size() );
};
```

Stellenbosch University https://scholar.sun.ac.za

```
template<class T>
inline void GenericAlgorithm<T>::SetElitism(bool E)
{
    Elitism = E;
};

template<class T>
inline bool GenericAlgorithm<T>::GetElitism()
{
    return Elitism;
};


template<class T>
inline void GenericAlgorithm<T>::SetPopulationSize(int S)
{
    PopulationSize = S;
};

template<class T>
inline int GenericAlgorithm<T>::GetPopulationSize()
{
    return PopulationSize;
};

template<class T>
inline void GenericAlgorithm<T>::Initialize()
{
  T *r=NULL;

  CriticalSection->Enter();
  ClearContainer((vector<T *>)Population);
  CriticalSection->Leave();
  ClearSystemVariables();
  for(register int i=0; i<PopulationSize; i++)
  {
      r=GenerateAnIndividual();
      ComputeFitness(r);
      Population.push_back(r);
      r=NULL;
  }
  ComputeTotalFitness();
  UpdateSystemVariables();
  CurrentStep = 0;
  Initialized=true;
```

Stellenbosch University https://scholar.sun.ac.za

```
  sort(Population.begin(), Population.end(), &IndGreaterThan);
  CurrentStep = 0;
};

template<class T>
inline double GenericAlgorithm<T>::GetFitness(int i)
{
    return ((GenericIndividual *)Population[i])->Fitness;
};

template<class T>
inline double GenericAlgorithm<T>::GetTotalFitness()
{
    return TotalFitness;
};

template<class T>
inline double GenericAlgorithm<T>::GetAvgFitness()
{
    double D = TotalFitness;

    if(Population.size()) D /= (double)Population.size();
    return D;
};

template<class T>
inline void GenericAlgorithm<T>::ComputeTotalFitness()
{
    int N = Population.size();
    TotalFitness = 0.0;

    for(register int i = 0; i < N; i++)
        TotalFitness += ((GenericIndividual *)Population[i])->Fitness;
};

//---------------------------------------------------------------------------
#endif
```

B.1.2 Header file for abstract class GenericEvoIndividual and GenericEvolutionaryAlgorithm

```
#ifndef evoalgorithmH
#define evoalgorithmH
#include <math.h>
#define NDEBUG
```

Stellenbosch University https://scholar.sun.ac.za

```
#include "debugger.h"
#include "vectormath.h"
#include "GenericAlg.h"


enum SetType {stTRAINING, stVALIDATION, stTESTING};
enum TerminalType {tpINPUT, tpOUTPUT, tpSYSTEM};
enum NodeType {ntTERMINAL, ntFUNCTION};
enum FitnessType {ftERROR_BASED, ftCORR_BASED, ft2ndORDER_POLY};
enum SelectionType {stFITNESS_PROPORTIONATE, stTOURNAMENT, stRANK};

class AbstractException {};
class NotInitialized : public AbstractException {};

/*/////////////////////////////////////////////////////////////////
    -------------------------------------------------------
                GenericEvoIndividual Class
    -------------------------------------------------------
/*/////////////////////////////////////////////////////////////////
template<class G>
class GenericEvoIndividual : public GenericIndividual
{
private:
protected:
public:
    GenericEvoIndividual();
    ~GenericEvoIndividual();
    virtual void Clone(GenericEvoIndividual* &Target);
    vector<G*> Genome;
    void SetGenome(G* Src, int Index);
    G* GetGenome(int i);
};
/*/
   -------------------------------------------------------
               GenericEvoIndividual Implementation
   -------------------------------------------------------
/*/
template<class G>
GenericEvoIndividual<G>::GenericEvoIndividual() :
GenericIndividual() {};

template<class G>
GenericEvoIndividual<G>::~GenericEvoIndividual()
{
    int N = Genome.size();
```

Stellenbosch University https://scholar.sun.ac.za

```
    for(register int i =0; i < N; i++)
        delete (G *)Genome[i];
    Genome.clear();
};

template<class G>
inline void GenericEvoIndividual<G>::Clone(GenericEvoIndividual * &Target)
{
    if(!Target) Target = new GenericEvoIndividual();
    GenericIndividual::Clone(Target);
    int N = this->Genome.size();
    Target->Genome.reserve(N);

    G *Dummy;
    for(register int i = 0; i < N; i++)
    {
        Dummy = new G();
        *Dummy = *(this->Genome[i]);
        Target->Genome.push_back(Dummy);
    }
};

template<class G>
inline void GenericEvoIndividual<G>::SetGenome(G* Src, int Index)
{
    Genome[Index] = Src;
};

template<class G>
inline G* GenericEvoIndividual<G>::GetGenome(int i)
{
    return Genome[i];
};


//===================================================================

/*/////////////////////////////////////////////////////////////////
    -------------------------------------------------------
                GenericEvolutionaryAlgorithm Class
    -------------------------------------------------------
/*/////////////////////////////////////////////////////////////////
template<class T, class G>
class GenericEvolutionaryAlgorithm : public GenericAlgorithm<T>
{
```

Stellenbosch University https://scholar.sun.ac.za

```
private:
protected:
    vector<T *> Pool;
    SelectionType SelectionMethod;
    int TournamentMembers, Pc, Pm;
    FitnessType FitnessFunction;
    virtual void ClearSystemVariables();
    virtual void UpdateSystemVariables();
    virtual void Crossover(vector<G *> &Parent1, vector<G *> &Parent2) = 0;
    virtual void Mutate(vector<G *> &Parent) = 0;
    virtual void Crossover(vector<G *> &Parent1, vector<G *> &Parent2,
                           int &Level1, int &Level2) = 0;
    virtual void Mutate(vector<G *> &Parent, int &Level) = 0;
    virtual void RawFitness(T * &AnInd, const VECTOR_DOUBLE *ObservedOutput,
                            const VECTOR_DOUBLE *PredictedOutput);
        void Reproduction(int PopSize, SelectionType SelectionMethod, int
TournamentMembers);
    virtual void GeneticOperations(int Pc, int Pm, int PopSize) = 0;
    virtual VECTOR_DOUBLE* EvaluateGenome(const vector<G *> &Genome) = 0;
public:
    GenericEvolutionaryAlgorithm();
    ~GenericEvolutionaryAlgorithm();
    void DoAlgorithm();
    void SetSelectionMethod(SelectionType SM);
    SelectionType GetSelectionMethod();
    void SetTournamentMembers(int T);
    int GetTournamentMembers();
    void SetPc(int P);
    int GetPc();
    void SetPm(int P);
    int GetPm();
    void SetFitnessFunction(FitnessType F);
    FitnessType GetFitnessFunction();
};
/*/
  -------------------------------------------------------
            GenericEvolutionaryAlgorithm Implementation
  -------------------------------------------------------
/*/
template<class T, class G>
GenericEvolutionaryAlgorithm<T, G>::GenericEvolutionaryAlgorithm() :
GenericAlgorithm<T>()
{
    SelectionMethod = stTOURNAMENT;
    TournamentMembers = 3;
```

Stellenbosch University https://scholar.sun.ac.za

```
    Pc = 60; Pm = 4;
    FitnessFunction = ftCORR_BASED;
};

template<class T, class G>
inline GenericEvolutionaryAlgorithm<T, G>::~GenericEvolutionaryAlgorithm()
{
    CriticalSection->Enter();
    ClearContainer((vector<T *>) Pool);
    CriticalSection->Leave();
    ClearSystemVariables();
};

template<class T, class G>
inline void GenericEvolutionaryAlgorithm<T, G>::ClearSystemVariables()
{
    GenericAlgorithm<T>::ClearSystemVariables();
};

template<class T, class G>
inline void GenericEvolutionaryAlgorithm<T, G>::UpdateSystemVariables()
{
    GenericAlgorithm<T>::UpdateSystemVariables();
};

template<class T, class G>
inline    void    GenericEvolutionaryAlgorithm<T,
G>::SetSelectionMethod(SelectionType SM)
{
    SelectionMethod = SM;
};

template<class T, class G>
inline SelectionType GenericEvolutionaryAlgorithm<T, G>::GetSelectionMethod()
{
    return SelectionMethod;
};

template<class T, class G>
inline void GenericEvolutionaryAlgorithm<T, G>::SetTournamentMembers(int T)
{
    TournamentMembers = T;
};

template<class T, class G>
```

Stellenbosch University https://scholar.sun.ac.za

```
inline int GenericEvolutionaryAlgorithm<T, G>::GetTournamentMembers()
{
    return TournamentMembers;
};

template<class T, class G>
inline void GenericEvolutionaryAlgorithm<T, G>::SetPc(int P)
{
    Pc = P;
};

template<class T, class G>
inline int GenericEvolutionaryAlgorithm<T, G>::GetPc()
{
    return Pc;
};

template<class T, class G>
inline void GenericEvolutionaryAlgorithm<T, G>::SetPm(int P)
{
    Pm = P;
};

template<class T, class G>
inline int GenericEvolutionaryAlgorithm<T, G>::GetPm()
{
    return Pm;
};

template<class T, class G>
inline void GenericEvolutionaryAlgorithm<T, G>::SetFitnessFunction(FitnessType
F)
{
    FitnessFunction = F;
};

template<class T, class G>
inline FitnessType GenericEvolutionaryAlgorithm<T, G>::GetFitnessFunction()
{
    return FitnessFunction;
};

template<class T, class G>
inline void GenericEvolutionaryAlgorithm<T, G>::DoAlgorithm()
{
```

Stellenbosch University https://scholar.sun.ac.za

```
    PRINT("Initializing");
    STARTTIMER();
    if(!Initialized) Initialize();
    ENDTIMER();
    PRINT("Reproduction");
    STARTTIMER();
    Reproduction(PopulationSize, SelectionMethod, TournamentMembers);
    ENDTIMER();
    PRINT("Genetic operations");
    STARTTIMER();
    GeneticOperations(Pc, Pm, PopulationSize);
    ENDTIMER();
    CurrentStep++;
    UpdateSystemVariables();
};

template<class T, class G>
inline void GenericEvolutionaryAlgorithm<T, G>::RawFitness(T * &AnInd, const
VECTOR_DOUBLE *ObservedOutput,
       const VECTOR_DOUBLE *PredictedOutput)
{
     CriticalSection->Enter();
     switch(FitnessFunction)
     {
      case ftERROR_BASED:
          {
                                                        ((GenericIndividual
*)AnInd)->Fitness=1.0/(1.0+sse(PredictedOutput, ObservedOutput));
          }
          break;
      case ftCORR_BASED:
          {
                                    ((GenericIndividual  *)AnInd)->Fitness  =
rsquared(PredictedOutput, ObservedOutput);
          }
          break;
      default : ;
     }
     CriticalSection->Leave();
};

template<class T, class G>
inline void GenericEvolutionaryAlgorithm<T, G>::Reproduction(int PopSize,
       SelectionType SelectionMethod, int TournamentMembers)
{
    int contender;
```

Stellenbosch University https://scholar.sun.ac.za

```
T *competitor, *current_winner, *Dummy=NULL;

switch (SelectionMethod)
{
 case stFITNESS_PROPORTIONATE:
      {
         double pf;
         int ind=0;
                        while(((int)Pool.size() < PopSize) && (ind <
(int)Population.size()))
         {
                                              pf=((GenericIndividual
*)Population[ind])->Fitness/TotalFitness;
             int K=floor(pf*PopSize);
               for(register int i = 0; (i<K) && ((int)Pool.size() <
PopSize); i++)
             {
                Dummy = NULL;
                Population[ind]->Clone(Dummy);
                Pool.push_back(Dummy);
             }
             ind++;
         }
         }
         break;
 case stTOURNAMENT:
      while((int)Pool.size()<PopSize)
      {
          contender = random(Population.size());
          if((Pool.size() == 0) && Elitism) contender = 0;
          current_winner = (T *)Population[contender];
          for(register int i=1; i<TournamentMembers; i++) {
             contender = random(Population.size());
             competitor = (T *)Population[contender];
             if(((GenericIndividual *)current_winner)->Fitness <
                ((GenericIndividual *)competitor)->Fitness)
                current_winner = competitor;
          }
          Dummy = NULL;
          current_winner->Clone(Dummy);
          Pool.push_back(Dummy);
      }
      break;
 default: ;
}
sort(Pool.begin(), Pool.end(), &IndGreaterThan);
```

Stellenbosch University https://scholar.sun.ac.za

```
    CriticalSection->Enter();
    ClearContainer((vector <T *>)Population);
    CriticalSection->Leave();
};

//----------------------------------------------------------------------------
#endif
```

B.2 The GP class

B.2.1 Header file for class GPIndividual and CustomGPalgorithm

```
#ifndef GPalgorithmunitH

#define GPalgorithmunitH
#include "evoalgorithm.h"
#include "vectormath.h"

#define MEMCOPY_double(dest, src, count)\
                       asm\
                       {\
                            push    ecx;\
                            xor     ecx, ecx;\
                            mov     ecx, (count);\
                            push    esi; \
                            push    edi; \
                                        \
                            mov     edi, (dest);\
                            mov     esi, (src);\
                            shl     ecx, 1; \
                            cld;            \
                            REP     MOVSD;  \
                                           \
                            pop     edi;    \
                            pop     esi;    \
                            pop     ecx;    \
                        }
#define MINVALUE 10e-15
#define MAXVALUE 1/MINVALUE
class VectorInfo
{
public:
    VectorInfo() : Dim(0), Ptr(NULL) {};
    ~VectorInfo() {delete[] Ptr;};
    int Dim;
```

Stellenbosch University https://scholar.sun.ac.za

```
    double* Ptr;
};

struct vectorinfo {
    int dim;
    double* ptr;
};

typedef struct VectorInfo VINFO;
typedef vector<VINFO*> StackType;
/*/////////////////////////////////////////////////////////////////
                BaseNode Class
/*/////////////////////////////////////////////////////////////////
class BaseNode
{
protected:
private:
public:
    BaseNode(char *N, NodeType T);
    ~BaseNode();
    virtual void Clone(BaseNode *Target);
    char *Name;
    NodeType NodeKind;
};
//----------------------------------------------------------
//                  Implementation
//----------------------------------------------------------
BaseNode::BaseNode(char *N, NodeType T) :
NodeKind(T)
{
    Name = new char[strlen(N)+1];
    strcpy(Name, N);
    Name[strlen(N)]='\0';
};

BaseNode::~BaseNode()
{
    delete Name;
};

inline void BaseNode::Clone(BaseNode *Target)
{
    if(!Target) return;
    Target->Name = new char[strlen(this->Name)+1];
    strcpy(Target->Name, this->Name);
```

Stellenbosch University https://scholar.sun.ac.za

```
    Target->Name[strlen(this->Name)]='\0';
    Target->NodeKind = this->NodeKind;
};
//=====================================================================

/*//////////////////////////////////////////////////////////////////
                TerminalNode Class
/*//////////////////////////////////////////////////////////////////
class TerminalNode : public BaseNode
{
protected:
private:
public:
    TerminalNode();
    TerminalNode(char *N);
    TerminalNode(char *N, TerminalType T);
    ~TerminalNode();
    void Clone(TerminalNode *Target);
    vector<double> *Values;
    int GetDim();
    TerminalType ActingAs;
};
//-----------------------------------------------------------------
//                  Implementation
//-----------------------------------------------------------------
TerminalNode::TerminalNode():
BaseNode("", ntTERMINAL)
{
    ActingAs = tpINPUT;
};

TerminalNode::TerminalNode(char *N) :
BaseNode(N, ntTERMINAL)
{
    ActingAs = tpINPUT;
};

TerminalNode::TerminalNode(char *N, TerminalType T) :
BaseNode(N, ntTERMINAL)
{
    ActingAs = T;
};

TerminalNode::~TerminalNode()
{
```

Stellenbosch University https://scholar.sun.ac.za

```
//    delete Values;  Values = NULL;
};

inline int TerminalNode::GetDim()
{
    int N = 0;
    if(Values) N = Values->size();

    return N;
};

inline void TerminalNode::Clone(TerminalNode *Target)
{
    if(!Target) Target = new TerminalNode();
    BaseNode::Clone(Target);
    Target->Values = new vector<double>();
    *(Target->Values) = *(this->Values);
    Target->ActingAs = this->ActingAs;
};
//==================================================================

class BaseFunctionNode : public BaseNode
{
public:
    BaseFunctionNode(int A, char *N) : BaseNode(N, ntFUNCTION),
                                       TotalArgs(A) {};
    virtual void ApplyFunction(StackType &S) = 0;
    int TotalArgs;
};

class AddNode : public BaseFunctionNode
{
private:
    double *ptr1, *ptr2;
 //   VECTOR_DOUBLE *x1, *x2;
    VINFO *x1, *x2;
    double k;
    int N;
public:
    AddNode() : BaseFunctionNode(2, "+"),
                ptr1(NULL), ptr2(NULL), x1(NULL), x2(NULL) {};
    ~AddNode() {};
    void ApplyFunction(StackType &S)
    {
        x1=S.back(); S.pop_back(); x2=S.back(); S.pop_back();
```

Stellenbosch University https://scholar.sun.ac.za

```
        ptr1 = x1->Ptr;
        ptr2 = x2->Ptr;
        N = x1->Dim;
        for(register int i = 0; i < N; i++)
        {
            k = (*ptr1) + (*ptr2);
            k = (fabs(k)<MINVALUE) ? (sign(k)*MINVALUE) : k;
            k = (fabs(k)>MAXVALUE) ? (sign(k)*MAXVALUE) : k;
            *ptr1 = k;
            ptr1++; ptr2++;
        }
        ptr1 = ptr2 = NULL;
        delete x2; x2=NULL;
        S.push_back(x1); x1=NULL;
    };
};

class SubtractNode : public BaseFunctionNode
{
private:
    double *ptr1, *ptr2;
    VINFO *x1, *x2;
    double k;
    int N;
public:
    SubtractNode() : BaseFunctionNode(2, "-"),
                     ptr1(NULL), ptr2(NULL), x1(NULL), x2(NULL)
                     {};
    void ApplyFunction(StackType &S)
    {
        x2=S.back(); S.pop_back(); x1=S.back(); S.pop_back();
        ptr1 = x1->Ptr;
        ptr2 = x2->Ptr;
        N = x1->Dim;
        for(register int i = 0; i < N; i++)
        {
            k = (*ptr1) - (*ptr2);
            k = (fabs(k)<MINVALUE) ? (sign(k)*MINVALUE) : k;
            k = (fabs(k)>MAXVALUE) ? (sign(k)*MAXVALUE) : k;
            *ptr1 = k;
            ptr1++; ptr2++;
        }
        ptr1 = ptr2 = NULL;
        delete x2; x2 = NULL;
        S.push_back(x1); x1 = NULL;
```

Stellenbosch University https://scholar.sun.ac.za

```
    };
};

class MultiplyNode : public BaseFunctionNode
{
private:
    double *ptr1, *ptr2;
    double k;
    int N;
    //VECTOR_DOUBLE *x1, *x2;
    VINFO *x1, *x2;
public:
    MultiplyNode() : BaseFunctionNode(2, "*"),
                     ptr1(NULL), ptr2(NULL), x1(NULL), x2(NULL)
                     {};
    void ApplyFunction(StackType &S)
    {
 /*       x2=S.back(); S.pop_back(); x1=S.back(); S.pop_back();
        ptr1 = (double *)x1->begin();
        ptr2 = (double *)x2->begin();
        N = x1->size();
        for(register int i = 0; i < N; i++)
        {
            k = (*ptr1) * (*ptr2);
            k = (fabs(k)>MAXVALUE) ? (sign(k)*MAXVALUE) : k;
            k = (fabs(k)<MINVALUE) ? (sign(k)*MINVALUE) : k;
            *ptr1 = k;
            ptr1++; ptr2++;
        }
        ptr1 = ptr2 = NULL;
        delete x2; x2 = NULL;
        S.push_back(x1); x1 = NULL;

  */
        x2=S.back(); S.pop_back(); x1=S.back(); S.pop_back();
        ptr1 = x1->Ptr;
        ptr2 = x2->Ptr;
        N = x1->Dim;
        for(register int i = 0; i < N; i++)
        {
            k = (*ptr1) * (*ptr2);
            k = (fabs(k)>MAXVALUE) ? (sign(k)*MAXVALUE) : k;
            k = (fabs(k)<MINVALUE) ? (sign(k)*MINVALUE) : k;
            *ptr1 = k;
            ptr1++; ptr2++;
```

Stellenbosch University https://scholar.sun.ac.za

```
        }
        ptr1 = ptr2 = NULL;
        delete x2; x2 = NULL;
        S.push_back(x1); x1 = NULL;
    };
};

class DivideNode : public BaseFunctionNode
{
private:
    double *ptr1, *ptr2;
    double k;
    int N;
    //VECTOR_DOUBLE *x1, *x2;
    VINFO *x1, *x2;

public:
    DivideNode() : BaseFunctionNode(2, "/"),
                   ptr1(NULL), ptr2(NULL), x1(NULL), x2(NULL)
                   {};
    void ApplyFunction(StackType &S)
    {
/*        x1=S.back(); S.pop_back(); x2=S.back(); S.pop_back();
        ptr1 = (double *)x1->begin();
        ptr2 = (double *)x2->begin();
        N = x1->size();
        for(register int i = 0; i < N; i++)
        {
            k = (fabs(*ptr2) < MINVALUE) ? (sign(*ptr2)*MINVALUE) : (*ptr2);
            k = (*ptr1)/k;
            *ptr1 = k;
            ptr1++; ptr2++;
        }
        ptr1 = ptr2 = NULL;
        delete x2; x2 = NULL;
        S.push_back(x1); x1 = NULL;
 */
        x1=S.back(); S.pop_back(); x2=S.back(); S.pop_back();
        ptr1 = x1->Ptr;
        ptr2 = x2->Ptr;
        N = x1->Dim;
        for(register int i = 0; i < N; i++)
        {
            k = (fabs(*ptr2) < MINVALUE) ? (sign(*ptr2)*MINVALUE) : (*ptr2);
            k = (*ptr1)/k;
```

Stellenbosch University https://scholar.sun.ac.za

```
            *ptr1 = k;
            ptr1++; ptr2++;
        }
        ptr1 = ptr2 = NULL;
        delete x2; x2 = NULL;
        S.push_back(x1); x1 = NULL;
    };
};
//===================================================================

/*/////////////////////////////////////////////////////////////////
                GPIndividual Class
/*/////////////////////////////////////////////////////////////////
typedef vector<BaseNode *> VECTOR_BASENODE;
class GPIndividual : public GenericEvoIndividual<VECTOR_BASENODE>
{
private:
protected:
public:
    GPIndividual();
    ~GPIndividual();
    void Clone(GPIndividual* &Target);
    int GetLevel(int i);
    void SetLevel(int i, int Depth);
    vector<int> Levels;
};
//---------------------------------------------------------
//                  Implementation
//---------------------------------------------------------
GPIndividual::GPIndividual() :
GenericEvoIndividual<VECTOR_BASENODE>() {};

GPIndividual::~GPIndividual()
{
    Levels.clear();
};

inline void GPIndividual::Clone(GPIndividual* &Target)
{
    if(!Target) Target = new GPIndividual();
    GenericEvoIndividual<VECTOR_BASENODE>::Clone(Target);
    Target->Levels = this->Levels;
};

inline void GPIndividual::SetLevel(int i, int Depth)
```

Stellenbosch University https://scholar.sun.ac.za

```
{
    Levels[i] = Depth;
};

inline int GPIndividual::GetLevel(int i)
{
    return Levels[i];
};

//==================================================================

/*/////////////////////////////////////////////////////////////////
                GPalgorithm Class
/*/////////////////////////////////////////////////////////////////
class CustomGPAlgorithm : public GenericEvolutionaryAlgorithm<GPIndividual,
BaseNode>
{
private:
    StackType StackVector;
    AddNode *PlusNode;
    SubtractNode *MinusNode;
    DivideNode *DivNode;
    MultiplyNode *MultNode;
protected:
    vector<TerminalNode *> TerminalSet;
    vector<BaseFunctionNode *> FunctionSet;
    int InitTreeLevels, MaxTreeLevels;
    void InsertSubtree(VECTOR_BASENODE &Genome,
         const VECTOR_BASENODE &SubTree, int Pos);
    void RemoveSubtree(VECTOR_BASENODE &Genome,
         const VECTOR_BASENODE &SubTree, int Pos);
    VECTOR_BASENODE* GetSubtree(const VECTOR_BASENODE &Genome,
                     int &subtree_levels, int pos);
    void GetLevel(const VECTOR_BASENODE &Genome, int &Level, int Pos);
    virtual void Crossover(VECTOR_BASENODE &Parent1,
            VECTOR_BASENODE &Parent2) {};
    virtual void Mutate(VECTOR_BASENODE &Parent) {};
    virtual void Crossover(VECTOR_BASENODE &Parent1,
            VECTOR_BASENODE &Parent2, int &Level1, int &Level2);
    virtual void Mutate(VECTOR_BASENODE &Parent, int &Level);
    virtual void EvolveGenome(VECTOR_BASENODE &Genome, int CurrentLevel,
            int MAXLEVELS, int &TotalLevels,
            const vector<TerminalNode *> &TerminalSet,
            const vector<BaseFunctionNode *> &FunctionSet );
    VECTOR_DOUBLE* EvaluateGenome(const VECTOR_BASENODE &Genome);
```

Stellenbosch University https://scholar.sun.ac.za

```
    virtual GPIndividual* GenerateAnIndividual() = 0;
    virtual void ComputeFitness(GPIndividual* &AnInd) = 0;
    virtual void GeneticOperations(int Pc, int Pm, int PopSize);
    virtual void ApplyGeneticScalingFunc(VECTOR_DOUBLE &PredictedOutput, int
Env) = 0;
public:
    CustomGPAlgorithm();
    ~CustomGPAlgorithm();
    void SetMaxTreeLevels(int L);
    virtual void SetTerminalSet(const vector<TerminalNode *> &TrmSet);
    void SetFunctionSet(bool plus, bool minus, bool div, bool mult);
    int GetMaxTreeLevels();
    virtual double GetRecomputedFitness(int Ind) = 0;
    virtual vector<VECTOR_DOUBLE *> *GetPredictedOutput(int ind) = 0;
    virtual char* GetGenomeString(int Ind, int Env);
    VECTOR_BASENODE *GetGenome(int Ind, int Env);
    VECTOR_DOUBLE* GetTerminal(int whichtrm);
};
```

B.2.2 Header file for class GPSupervised

```
class GPSupervised : public CustomGPAlgorithm
{
protected:
    virtual void UpdateSystemVariables();
    virtual void ClearSystemVariables();
    virtual GPIndividual* GenerateAnIndividual();
    virtual void ComputeFitness(GPIndividual* &AnInd);
    virtual void ApplyGeneticScalingFunc(VECTOR_DOUBLE &PredictedOutput, int
Env);
private:
    double SVCurrentRsqr, SVCurrentSSE;

    vector<TerminalNode *> EnvironmentSet;
public:
    GPSupervised();
    ~GPSupervised();
    double GetCurrentRsqr();
    double GetCurrentSSE();
    vector<double> SVRsqr, SVSSE; //System variables
    double GetRecomputedRsqr(int I);
    double GetRecomputedSSE(int I);
    void SetEnvironmentSet(const vector<TerminalNode *> &EnvSet);
    virtual vector<VECTOR_DOUBLE *> *GetPredictedOutput(int ind);
    virtual double GetRecomputedFitness(int Ind);
    VECTOR_DOUBLE* GetEnvironment(int whichenv);
```

Stellenbosch University https://scholar.sun.ac.za

```
    int GetEnvironmentSize();
};
```

B.2.3 Header file for class GPUnsupervised

```
class GPUnsupervised : public CustomGPAlgorithm
{
protected:
    virtual GPIndividual* GenerateAnIndividual();
    virtual void ComputeFitness(GPIndividual* &AnInd) = 0;
    virtual void ApplyGeneticScalingFunc(VECTOR_DOUBLE &PredictedOutput, int
Env)=0;
    int TargetSpaceDim;
private:

public:
    GPUnsupervised();
    ~GPUnsupervised();
    void SetTargetSpaceDim(int T);
    int GetTargetSpaceDim();
    vector<VECTOR_DOUBLE *> *GetPredictedOutput(int ind);
    virtual double GetRecomputedFitness(int Ind)=0;
};
```

B.2.4 Implementation of each class in the GP kernel

B.2.4.1 Implementation of class CustomGPAlgorithm

```
//-----------------------------------------------------------
//                    Implementation
//-----------------------------------------------------------
CustomGPAlgorithm::CustomGPAlgorithm():
GenericEvolutionaryAlgorithm<GPIndividual, BaseNode>()
{
    Elitism = true;
    MaxTreeLevels = 10;
    InitTreeLevels = 5;
    PlusNode = new AddNode();
    MinusNode = new SubtractNode();
    DivNode = new DivideNode();
    MultNode = new MultiplyNode();
};

CustomGPAlgorithm::~CustomGPAlgorithm()
{
    CriticalSection->Enter();
```

Stellenbosch University https://scholar.sun.ac.za

```
    ClearContainer((vector<TerminalNode *>) TerminalSet);
    CriticalSection->Leave();
    delete PlusNode;
    delete MinusNode;
    delete DivNode;
    delete MultNode;
};

GPSupervised::GPSupervised() :
CustomGPAlgorithm()
{};

GPSupervised::~GPSupervised()
{
    CriticalSection->Enter();
    ClearContainer((vector<TerminalNode *>) EnvironmentSet);
    CriticalSection->Leave();
};

GPUnsupervised::GPUnsupervised() :
CustomGPAlgorithm()
{
    TargetSpaceDim = 1;
};

GPUnsupervised::~GPUnsupervised()
{};


inline void CustomGPAlgorithm::GeneticOperations(int Pc, int Pm, int PopSize)
{
    int Parent1, Parent2;
    GPIndividual *P1=NULL, *P2=NULL;
    int Genome1, Genome2;

    TotalFitness=0.0;
    if(Elitism) {
       Pool[0]->Clone(P1);
       TotalFitness += P1->Fitness;
       Population.push_back(P1);
       P1=NULL;
    }
    int K=floor(Pc/200.0*PopSize);
    for(register int i=0; (i<K) && ((int)Population.size()<PopSize); i++)
    {
```

Stellenbosch University https://scholar.sun.ac.za

```
        P1 = P2 = NULL;
        Parent1 = random(Pool.size()); Parent2 = random(Pool.size());
        Pool[Parent1]->Clone(P1);
        Pool[Parent2]->Clone(P2);
        Genome1 = Genome2 = random(P1->Genome.size());
        Crossover(*(P1->GetGenome(Genome1)), *(P2->GetGenome(Genome2)),
                  P1->GetLevel(Genome1), P2->GetLevel(Genome2));
        //now compute the new fitness
        ComputeFitness(P1);
        ComputeFitness(P2);
        TotalFitness += P1->Fitness;
        TotalFitness += P2->Fitness;
        Population.push_back(P1);
        Population.push_back(P2);
    }

    K=floor(Pm/100.0*PopSize);
    for(register int i = 0; (i < K) && ((int)Population.size() < PopSize);
i++)
    {
        P1 = NULL;
        Parent1 = random(Pool.size());
        Pool[Parent1]->Clone(P1);
        Genome1 = random(P1->Genome.size());
        Mutate(*(P1->GetGenome(Genome1)), P1->GetLevel(Genome1));
        ComputeFitness(P1);
        TotalFitness += P1->Fitness;
        Population.push_back(P1);
    }

    for(;(int)Population.size()<PopSize;)
    {
        P1 = NULL;
        Parent1 = random(Pool.size());
        Pool[Parent1]->Clone(P1);
        TotalFitness += P1->Fitness;
        Population.push_back(P1);
    }
    sort(Population.begin(), Population.end(), &IndGreaterThan);
    CriticalSection->Enter();
    ClearContainer((vector<GPIndividual *>) Pool);
    CriticalSection->Leave();
};

inline void CustomGPAlgorithm::Crossover(VECTOR_BASENODE &Parent1,
```

Stellenbosch University https://scholar.sun.ac.za

```
     VECTOR_BASENODE &Parent2, int &Level1, int &Level2)
{
    int cp1, cp2;
   int subtree_levels1=0, subtree_levels2=0, level_at_cp1=0, level_at_cp2=0;
    VECTOR_BASENODE *P1subtree, *P2subtree;


      cp1=random(Parent1.size());    //obtain the crossover points in the 2
strings
    cp2=random(Parent2.size());

    GetLevel(Parent1, level_at_cp1, cp1);
    GetLevel(Parent2, level_at_cp2, cp2);
    P1subtree=GetSubtree(Parent1, subtree_levels1, cp1);
    P2subtree=GetSubtree(Parent2, subtree_levels2, cp2);
   ////////////////////////////////////////////
   //now remove the subtree from expr
   ///////////////////////////////////////////////
    if(subtree_levels2+level_at_cp1-1<=MaxTreeLevels)
    {
       RemoveSubtree(Parent1, *P1subtree, cp1);
       InsertSubtree(Parent1, *P2subtree, cp1);
     //use this line to get the nr of levels for the whole tree
       delete GetSubtree(Parent1, Level1, 0);
    }
    if(subtree_levels1+level_at_cp2-1 <= MaxTreeLevels)
    {
       RemoveSubtree(Parent2, *P2subtree, cp2);
       InsertSubtree(Parent2, *P1subtree, cp2);
       delete GetSubtree(Parent2, Level2, 0);
    }
    delete P1subtree; delete P2subtree;
    P1subtree = P2subtree = NULL;
};

inline void CustomGPAlgorithm::Mutate(VECTOR_BASENODE &Parent, int &Level)
{
    int level=-1, cp;
    int nr_of_levels=0, level_at_cp, subtree_levels;
    VECTOR_BASENODE *Psubtree = NULL;
    VECTOR_BASENODE *sub_tree = new VECTOR_BASENODE();

    cp=random(Parent.size());

    GetLevel(Parent, level_at_cp, cp);
```

Stellenbosch University https://scholar.sun.ac.za

```
    if(level_at_cp < MaxTreeLevels)
    {
       Psubtree=GetSubtree(Parent, subtree_levels, cp);
       RemoveSubtree(Parent, *Psubtree, cp);
       delete Psubtree;
     //create a new random subtree of maxlength abs_max_level-level_at_cp+1
       EvolveGenome(*sub_tree, level, MaxTreeLevels-level_at_cp+1,
                    nr_of_levels, TerminalSet, FunctionSet);
       InsertSubtree(Parent, *sub_tree, cp);
       delete GetSubtree(Parent, Level, 0);
    }
    delete sub_tree;
};

inline void CustomGPAlgorithm::EvolveGenome( VECTOR_BASENODE &Genome, int
CurrentLevel,
     int MAXLEVELS, int &TotalLevels,
     const vector<TerminalNode *> &TerminalSet,
     const vector<BaseFunctionNode *> &FunctionSet )
{
    BaseNode *NodePtr;
    int Args=0, i;
    double Tp, Op;

    CurrentLevel++;
    /////this part describes the propability
    //of a terminal or an operator being chosen////
    Op=-1.0/((double)(MAXLEVELS-1.0))*((double)CurrentLevel)+1.0;
    Tp=1.0-Op;

    ////////////////////////////////////////////////////////////
    if(Op*random(100) >= Tp*random(100)) {
       i = random(FunctionSet.size());
       NodePtr = (BaseFunctionNode *)FunctionSet[i];
       Args = ((BaseFunctionNode *)NodePtr)->TotalArgs;
    }
    else {
       i = random(TerminalSet.size());
       NodePtr = (TerminalNode *)TerminalSet[i];
       if(TotalLevels<CurrentLevel) TotalLevels=CurrentLevel;
    }
    Genome.push_back(NodePtr);
    NodePtr = NULL;
    for(register int k=0; k < Args; k++)
    {
```

Stellenbosch University https://scholar.sun.ac.za

```
        EvolveGenome(Genome, CurrentLevel, MAXLEVELS, TotalLevels,
                     TerminalSet, FunctionSet);
    }
};

inline void CustomGPAlgorithm::InsertSubtree(VECTOR_BASENODE &Genome,
     const VECTOR_BASENODE &SubTree, int Pos)
{
    Genome.reserve(Genome.size()+SubTree.size());
    Genome.insert(Genome.begin()+Pos, SubTree.begin(), SubTree.end());
};

inline void CustomGPAlgorithm::RemoveSubtree(VECTOR_BASENODE &Genome,
     const VECTOR_BASENODE &SubTree, int Pos)
{
    Genome.erase( (Genome.begin()+Pos), (Genome.begin()+Pos+SubTree.size())
);
};

inline VECTOR_BASENODE* CustomGPAlgorithm::GetSubtree(const VECTOR_BASENODE
&Genome, int &subtree_levels, int pos)
{
   int exprpos=-1+pos, s_tree_pos=-1;
   VECTOR_BASENODE* s_tree = new VECTOR_BASENODE(); //newly added

   void newsubtree(BaseNode **Genome, VECTOR_BASENODE &s_tree,
                   int &, int &, int &, int);

   void subtree(const VECTOR_BASENODE &Genome, VECTOR_BASENODE &s_tree,
                   int &, int &, int &, int);
   subtree_levels=0;
   newsubtree((BaseNode **)(Genome.begin()), *s_tree, exprpos, s_tree_pos,
      subtree_levels, -1);
   subtree_levels++;

   return s_tree;
};

inline void newsubtree(BaseNode** Genome, VECTOR_BASENODE &SubTree,
                 int &exprindex, int &s_tree_index, int &subtree_levels, int
level)
{
   int Args=0;
   exprindex++;
   s_tree_index++;
   level++;
```

Stellenbosch University https://scholar.sun.ac.za

```
    SubTree.push_back(Genome[exprindex]);
    if(Genome[exprindex]->NodeKind==ntFUNCTION)
    {
       Args = ((BaseFunctionNode *)Genome[exprindex])->TotalArgs;
    }
    else
    if(subtree_levels<level) subtree_levels=level;

    for (register int i=0; i<Args; i++)
    {
       newsubtree( Genome, SubTree, exprindex, s_tree_index,
                subtree_levels, level);
    }
};

inline void subtree( const VECTOR_BASENODE &Genome, VECTOR_BASENODE &SubTree,
                int &exprindex, int &s_tree_index, int &subtree_levels, int
level)
{
    int Args=0;
    exprindex++;
    s_tree_index++;
    level++;
    SubTree.push_back(Genome[exprindex]);
    if(Genome[exprindex]->NodeKind==ntFUNCTION)
    {
       Args = ((BaseFunctionNode *)Genome[exprindex])->TotalArgs;
    }
    else
    if(subtree_levels<level) subtree_levels=level;

    for (register int i=0; i<Args; i++)
    {
       subtree( Genome, SubTree, exprindex, s_tree_index,
       subtree_levels, level);
    }
};

inline void CustomGPAlgorithm::GetLevel(const VECTOR_BASENODE &Genome, int
&Level, int Pos)
{
  int l=-1;
   void get_the_damn_level(const VECTOR_BASENODE &Genome, int &len, int
CurrentLevel,
                              int Pos, int &Level);
  void newget_the_damn_level(BaseNode **Genome, int &len, int CurrentLevel,
```

Stellenbosch University https://scholar.sun.ac.za

```
                                 int Pos, int &Level);

  Level=0;
//  get_the_damn_level(Genome,l, -1, Pos, Level);
  newget_the_damn_level((BaseNode **)(Genome.begin()),l, -1, Pos, Level);
  Level++;
};

inline void newget_the_damn_level(BaseNode **Genome, int &len, int
CurrentLevel,
                                int Pos, int &Level)
{
   int Args=0;

   CurrentLevel++;
   len++;
   if(len == Pos) Level=CurrentLevel;
   if(len < Pos) {
      if(Genome[len]->NodeKind == ntFUNCTION)
      {
         Args = ((BaseFunctionNode *)Genome[len])->TotalArgs;
      }
   }
   for(register int i = 0; i < Args; i++)
       newget_the_damn_level(Genome, len, CurrentLevel, Pos, Level);
};

inline void get_the_damn_level( const VECTOR_BASENODE &Genome, int &len, int
CurrentLevel,
                                int Pos, int &Level)
{
   int Args=0;

   CurrentLevel++;
   len++;
   if(len == Pos) Level=CurrentLevel;
   if(len < Pos) {
      if(Genome[len]->NodeKind == ntFUNCTION)
      {
         Args = ((BaseFunctionNode *)Genome[len])->TotalArgs;
      }
   }
   for(register int i = 0; i < Args; i++)
       get_the_damn_level(Genome, len, CurrentLevel, Pos, Level);
};
```

Stellenbosch University https://scholar.sun.ac.za

```
inline int CustomGPAlgorithm::GetMaxTreeLevels()
{
    return MaxTreeLevels;
};

inline void CustomGPAlgorithm::SetMaxTreeLevels(int L)
{
    MaxTreeLevels=L;
    if(InitTreeLevels>MaxTreeLevels) InitTreeLevels=MaxTreeLevels;
};

inline void CustomGPAlgorithm::SetFunctionSet(bool plus, bool minus, bool div,
bool mult)
{
    FunctionSet.clear();
    if(plus) FunctionSet.push_back(PlusNode);
    if(minus) FunctionSet.push_back(MinusNode);
    if(div) FunctionSet.push_back(DivNode);
    if(mult) FunctionSet.push_back(MultNode);
};

void CustomGPAlgorithm::SetTerminalSet(const vector<TerminalNode *> &TrmSet)
{
    int N = TerminalSet.size();
    CriticalSection->Enter();
    ClearContainer((vector<TerminalNode *>) TerminalSet);
    CriticalSection->Leave();
    TerminalSet.clear();
    N = TrmSet.size();
    for(register int i = 0; i < N; i++)
    {
        if(TrmSet[i]->ActingAs == tpINPUT)
        {
            TerminalSet.push_back( new TerminalNode(TrmSet[i]->Name,
            TrmSet[i]->ActingAs));
            TerminalSet.back()->Values = TrmSet[i]->Values;
        }
    }
};

inline char* CustomGPAlgorithm::GetGenomeString(int Ind, int Env)
{
    char* t;
    VECTOR_BASENODE* Genome = Population[Ind]->GetGenome(Env);
    int N = Genome->size();
```

Stellenbosch University https://scholar.sun.ac.za

```
    t = new char[10000];
    t[0]='\0';
    char *src;

    for(register int i = 0; i < N; i++)
    {
        src = (*Genome)[i]->Name;
        strcat(t, " ");
        strcat(t, src);
    }
    return t;

};

inline VECTOR_BASENODE *CustomGPAlgorithm::GetGenome(int Ind, int Env)
{
    return Population[Ind]->GetGenome(Env);
};

inline VECTOR_DOUBLE* CustomGPAlgorithm::GetTerminal(int whichtrm)
{
    return TerminalSet[whichtrm]->Values;
};

inline VECTOR_DOUBLE* CustomGPAlgorithm::EvaluateGenome(const VECTOR_BASENODE
&Genome)
{
    CriticalSection->Enter();
    BaseNode **Ptr = (BaseNode **)(Genome.end()-1);
    VINFO *Dummy, *PtrValues;
    double* memblock, *doubleptr;
    StackVector.clear();
    int N = Genome.size();
    int NValues;
    StackVector.reserve(N+1);

    for(register int i=N-1; i>=0; i--, Ptr--)
    {
        if((*Ptr)->NodeKind == ntTERMINAL)
        {
           doubleptr =((TerminalNode *)(*Ptr))->Values->begin();
           NValues = ((TerminalNode *)(*Ptr))->Values->size();
           memblock = new double[NValues];

           for(register int i = 0; i < NValues; i++, doubleptr++)
```

Stellenbosch University https://scholar.sun.ac.za

```
                memblock[i] = *doubleptr;
            Dummy = new VINFO();
            Dummy->Dim = NValues;
            Dummy->Ptr = memblock;
            StackVector.push_back(Dummy);
        }
        else
        {
            ((BaseFunctionNode *)(*Ptr))->ApplyFunction(StackVector);
        }
    }

    Dummy = StackVector.back(); StackVector.pop_back();
    VECTOR_DOUBLE *rtn = new VECTOR_DOUBLE();
    N = Dummy->Dim;
    rtn->reserve(N);
    for(register int i = 0; i < N; i++)
        rtn->push_back(Dummy->Ptr[i]);
    delete Dummy;

    CriticalSection->Leave();
    return rtn;
};
```

B.2.4.2 Implementation of class GPSupervised

```
/**
    Supervised Genetic Programming (GP)
*/
inline void GPSupervised::UpdateSystemVariables()
{
    GenericEvolutionaryAlgorithm<GPIndividual,
BaseNode>::UpdateSystemVariables();
    const vector<VECTOR_DOUBLE *> *Ptr=GetPredictedOutput(0);
    int N = EnvironmentSet.size();
    SVCurrentRsqr = SVCurrentSSE = 0.0;
    for(register int i = 0; i < N; i++)
    {
        SVCurrentRsqr += rsquared((*Ptr)[i], EnvironmentSet[i]->Values);
        SVCurrentSSE += sse((*Ptr)[i], EnvironmentSet[i]->Values);
    }

    if(N)
    {
        SVCurrentRsqr /= (double)N;
        SVCurrentSSE /= (double)N;
    }
```

Stellenbosch University https://scholar.sun.ac.za

```
    SVRsqr.push_back(SVCurrentRsqr);
    SVSSE.push_back(SVCurrentSSE);
    N = Ptr->size();
    for(register int i = 0; i < N; i++)
    {
        delete (*Ptr)[i];
    }
    delete Ptr;
};

inline void GPSupervised::ClearSystemVariables()
{
                  GenericEvolutionaryAlgorithm<GPIndividual,
BaseNode>::ClearSystemVariables();
    SVRsqr.clear();
    SVSSE.clear();
};

inline double GPSupervised::GetCurrentRsqr()
{
    return SVCurrentRsqr;
};

inline double GPSupervised::GetCurrentSSE()
{
    return SVCurrentSSE;
};

inline double GPSupervised::GetRecomputedRsqr(int I)
{
    const vector<VECTOR_DOUBLE *> *Ptr=GetPredictedOutput(I);
    int N = EnvironmentSet.size();
    SVCurrentRsqr = SVCurrentSSE = 0.0;
    CriticalSection->Enter();
    for(register int i = 0; i < N; i++)
    {
       SVCurrentRsqr += rsquared((*Ptr)[i], EnvironmentSet[i]->Values);
    }
    CriticalSection->Leave();
    if(N)
    {
        SVCurrentRsqr /= (double)N;
    }

    N = Ptr->size();
```

Stellenbosch University https://scholar.sun.ac.za

```
    for(register int i = 0; i < N; i++)
    {
        delete (*Ptr)[i];
    }
    delete Ptr;
    return SVCurrentRsqr;
};

inline double GPSupervised::GetRecomputedSSE(int I)
{
    const vector<VECTOR_DOUBLE *> *Ptr=GetPredictedOutput(I);
    int N = EnvironmentSet.size();
    SVCurrentRsqr = SVCurrentSSE = 0.0;
    CriticalSection->Enter();
    for(register int i = 0; i < N; i++)
    {
       SVCurrentSSE += sse((*Ptr)[i], EnvironmentSet[i]->Values);
    }
    CriticalSection->Leave();

    if(N)
    {
        SVCurrentSSE /= (double)N;
    }

    N = Ptr->size();
    for(register int i = 0; i < N; i++)
    {
       delete (*Ptr)[i];
    }
    delete Ptr;

    return SVCurrentSSE;
};

inline int GPSupervised::GetEnvironmentSize() {return EnvironmentSet.size();};
inline void GPSupervised::ComputeFitness(GPIndividual* &AnInd)
{
   VECTOR_DOUBLE *Ptr=NULL;
   int N = EnvironmentSet.size();
   double AvgFitness = 0.0;
   CriticalSection->Enter();
   for(register int i = 0; i < N; i++)
   {
       Ptr=EvaluateGenome(*(AnInd->GetGenome(i)));
```

Stellenbosch University https://scholar.sun.ac.za

```
        RawFitness(AnInd, EnvironmentSet[i]->Values, Ptr);
        AvgFitness += AnInd->Fitness;
        delete Ptr; Ptr=NULL;
    }
    if(N) AnInd->Fitness = AvgFitness/(double)N;
    CriticalSection->Leave();
};

inline GPIndividual* GPSupervised::GenerateAnIndividual()
{
    GPIndividual *Dummy=new GPIndividual();
    VECTOR_BASENODE* Genome = NULL;
    int CurLevel=-1, Level;

    int N = EnvironmentSet.size();
    for(register int i = 0; i < N; i++)
    {
        Level = 0;
        Genome = new VECTOR_BASENODE();
        EvolveGenome(*Genome, CurLevel, InitTreeLevels, Level,
                     TerminalSet, FunctionSet);
        Level++;
        Dummy->Genome.push_back(Genome);
        Dummy->Levels.push_back(Level);
        Genome = NULL;
    }
    return Dummy;
};

inline VECTOR_DOUBLE* GPSupervised::GetEnvironment(int whichenv)
{
    return EnvironmentSet[whichenv]->Values;
};

inline double GPSupervised::GetRecomputedFitness(int Ind)
{
    VECTOR_DOUBLE *Ptr=NULL;
    double v=0.0;
    GPIndividual *P = Population[Ind];
    GPIndividual *Dummy = new GPIndividual();
    int N = EnvironmentSet.size();
    double AvgFitness = 0.0;
    CriticalSection->Enter();
    for(register int i = 0; i < N; i++)
    {
```

Stellenbosch University https://scholar.sun.ac.za

```
        Ptr=EvaluateGenome(*(P->GetGenome(i)));
        RawFitness(Dummy, EnvironmentSet[i]->Values, Ptr);
        AvgFitness += Dummy->Fitness;
        delete Ptr; Ptr=NULL;
    }
    if(N) Dummy->Fitness = AvgFitness/(double)N;
    v = Dummy->Fitness;
    delete Dummy;
    CriticalSection->Leave();
    return v;
};

inline void GPSupervised::SetEnvironmentSet(const vector<TerminalNode *>
&EnvSet)
{
    int N = EnvironmentSet.size();
    CriticalSection->Enter();
    ClearContainer((vector<TerminalNode *>) EnvironmentSet);
    CriticalSection->Leave();
    EnvironmentSet.clear();
    N = EnvSet.size();
    for(register int i = 0; i < N; i++)
    {
        if(EnvSet[i]->ActingAs == tpOUTPUT)
        {
               EnvironmentSet.push_back( new TerminalNode(EnvSet[i]->Name,
EnvSet[i]->ActingAs));
            EnvironmentSet.back()->Values = EnvSet[i]->Values;
        }
    }
};

inline vector<VECTOR_DOUBLE *> *GPSupervised::GetPredictedOutput(int ind)
{
    vector<VECTOR_DOUBLE *> *Ptr = NULL;
    int N = EnvironmentSet.size();
    Ptr = new vector<VECTOR_DOUBLE *>();

    for(register int i = 0; i < N; i++)
    {
        Ptr->push_back(EvaluateGenome(*(Population[ind]->GetGenome(i))));
        if(FitnessFunction == ftCORR_BASED)
           ApplyGeneticScalingFunc(*((*Ptr)[i]), i);
    }
    return Ptr;
```

Stellenbosch University https://scholar.sun.ac.za

```
};

inline    void    GPSupervised::ApplyGeneticScalingFunc(VECTOR_DOUBLE
&PredictedOutput, int Env)
{
    int N=PredictedOutput.size();
    double stdevPredicted, meanPredicted, sf,
           meanObserved=mean( EnvironmentSet[Env]->Values );
    double *ptr;

                double    thesign    =    sign(corr(&PredictedOutput,
EnvironmentSet[Env]->Values));
    stdevPredicted=stdev(&PredictedOutput)*thesign;
    meanPredicted=mean(&PredictedOutput);
    try
    {
        sf=stdev(EnvironmentSet[Env]->Values)/stdevPredicted;
    }
    catch( ... )
    {
        sf=0.0;
    }
    ptr = (double *)PredictedOutput.begin();
    for(register int i = 0; i < N; i++)
    {
       *ptr = sf*( (*ptr)-meanPredicted ) + meanObserved;
       ptr++;
    }
};
```

B.2.4.3 Implementation of class GPUnsupervised

```
/**
    Unsupervised Genetic Programming
*/
inline vector<VECTOR_DOUBLE *> *GPUnsupervised::GetPredictedOutput(int ind)
{
    vector<VECTOR_DOUBLE *> *Ptr = NULL;
    int N = TargetSpaceDim;
    Ptr = new vector<VECTOR_DOUBLE *>();

    for(register int i = 0; i < N; i++)
    {
        Ptr->push_back(EvaluateGenome(*(Population[ind]->GetGenome(i))));
        if(FitnessFunction == ftCORR_BASED)
           ApplyGeneticScalingFunc(*((*Ptr)[i]), i);
    }
```

Stellenbosch University https://scholar.sun.ac.za

```
    return Ptr;
};

//---------------------------------------------------------
GPIndividual* GPUnsupervised::GenerateAnIndividual()
{
    GPIndividual *Dummy=new GPIndividual();
    VECTOR_BASENODE* Genome = NULL;
    int CurLevel=-1, Level;

    int N = TargetSpaceDim;
    for(register int i = 0; i < N; i++)
    {
        Level = 0;
        Genome = new VECTOR_BASENODE();
        EvolveGenome(*Genome, CurLevel, InitTreeLevels, Level,
                     TerminalSet, FunctionSet);
        Level++;
        Dummy->Genome.push_back(Genome);
        Dummy->Levels.push_back(Level);
        Genome = NULL;
    }
    return Dummy;
};

inline void GPUnsupervised::SetTargetSpaceDim(int T) {TargetSpaceDim = T;};
inline int GPUnsupervised::GetTargetSpaceDim() {return TargetSpaceDim;};
//---------------------------------------------------------------------------
#endif
```

B.2.5 Header file for class FeatureExtract

```
#ifndef FeatureXH
#define FeatureXH
#include "GPalgorithmunit.h"
//---------------------------------------------------------------------------
-------
enum StressType {stSammon};
class FeatureExtract : public GPUnsupervised
{
private:
    StressType StressIs;
protected:
    virtual void ComputeFitness(GPIndividual* &AnInd);
    virtual void ApplyGeneticScalingFunc(VECTOR_DOUBLE &PredictedOutput, int
Env){};
```

Stellenbosch University https://scholar.sun.ac.za

```
    double ApplySammon(vector<VECTOR_DOUBLE *> &OriginalSpace,
                       vector<VECTOR_DOUBLE *> &TargetSpace);
public:
    FeatureExtract();
    ~FeatureExtract();
    void SetEnvironmentSet(const vector<TerminalNode *> &E) {};
    double GetStress();
    StressType GetStressType();
    void SetStressType(StressType S);
    virtual double GetRecomputedFitness(int Ind);
    vector<VECTOR_DOUBLE *> GetTargetSpace(int Ind);
};

inline StressType FeatureExtract::GetStressType() {return StressIs;};
inline void FeatureExtract::SetStressType(StressType S) {StressIs = S;};
inline double FeatureExtract::GetStress(){};
inline double FeatureExtract::GetRecomputedFitness(int Ind)
{
    double v;
    GPIndividual *P = Population[Ind];
    GPIndividual *Dummy=NULL;

    P->Clone(Dummy);
    ComputeFitness(Dummy);
    v = Dummy->Fitness;
    delete Dummy;
    return v;
};

inline vector<VECTOR_DOUBLE *> FeatureExtract::GetTargetSpace(int Ind)
{
    GPIndividual* AnInd = Population[Ind];
    vector<VECTOR_DOUBLE *> TargetSpace;

    TargetSpace.reserve(TargetSpaceDim);
    for(register int i = 0; i < TargetSpaceDim; i++)
    {
        TargetSpace.push_back(EvaluateGenome(*(AnInd->GetGenome(i))));
    }
    return TargetSpace;
};

inline void FeatureExtract::ComputeFitness(GPIndividual* &AnInd)
{
    vector<VECTOR_DOUBLE *> TargetSpace;
```

Stellenbosch University https://scholar.sun.ac.za

```
    vector<VECTOR_DOUBLE *> OrigSpace;
    double Stress = 0.0;

    TargetSpace.reserve(TargetSpaceDim);
    for(register int i = 0; i < TargetSpaceDim; i++)
    {
        TargetSpace.push_back(EvaluateGenome(*(AnInd->GetGenome(i))));
        if(FitnessFunction == ftCORR_BASED)
           ApplyGeneticScalingFunc(*(TargetSpace[i]), i);
    }

    int N = TerminalSet.size();
    OrigSpace.reserve(N);
    for(register int i = 0; i < N; i ++)
    {
        OrigSpace.push_back( TerminalSet[i]->Values );
    }

    switch(StressIs)
    {
     case stSammon:
          AnInd->Fitness = 1.0/(1.0+ApplySammon(OrigSpace, TargetSpace));
          break;
     default:;
    }
    ClearContainer((vector<VECTOR_DOUBLE *>) TargetSpace);
    OrigSpace.clear();
};

inline    double    FeatureExtract::ApplySammon(vector<VECTOR_DOUBLE    *>
&OriginalSpace,
       vector<VECTOR_DOUBLE *> &TargetSpace)
{
    int T_Dim = TargetSpace.size();
    int O_Dim = OriginalSpace.size();
    int TotalPoints = TargetSpace[0]->size();
    int origPoints = OriginalSpace[0]->size();
    int Obsrv1, Obsrv2;  //select two random observations
    int Tmax=TotalPoints;
    double TotalDistance = 0.0, Stress = 0.0;
    double *Ptr=NULL;
    double DistOriginalSpace=0.0, DistTargetSpace=0.0, K=0.0;

    for(register int i = 0; i < Tmax; i++)
    {
```

Stellenbosch University https://scholar.sun.ac.za

```
        DistOriginalSpace = 0.0;
        Obsrv1 = random(TotalPoints); Obsrv2 = random(TotalPoints);
        while(Obsrv1 == Obsrv2) Obsrv2 = random(TotalPoints);
        for(register int k = 0; k < O_Dim; k++)
        {
            Ptr = (double *)OriginalSpace[k]->begin();
                                            DistOriginalSpace      +=
(Ptr[Obsrv1]-Ptr[Obsrv2])*(Ptr[Obsrv1]-Ptr[Obsrv2]);
        }
        DistOriginalSpace = sqrt(DistOriginalSpace);

        DistTargetSpace = 0.0;
        for(register int k = 0; k < T_Dim; k++)
        {
            Ptr = (double *)TargetSpace[k]->begin();
                                                  DistTargetSpace     +=
(Ptr[Obsrv1]-Ptr[Obsrv2])*(Ptr[Obsrv1]-Ptr[Obsrv2]);
        }
        DistTargetSpace=sqrt(DistTargetSpace);

        TotalDistance += DistOriginalSpace;
        try
        {
            K = (DistOriginalSpace-DistTargetSpace)*
                (DistOriginalSpace-DistTargetSpace)/DistOriginalSpace;
        }
        catch( ... )
        {
            K = 0.0;
        }
        Stress += K;
    }

    try
    {
        Stress /= TotalDistance;
    }
    catch( ... )
    {
        Stress=0;
    }
    return Stress;
};
#endif
```

Stellenbosch University https://scholar.sun.ac.za

APPENDIX C

HELP ON THE a-GP PACKAGE

C.1 Possible analysis that can be conducted using a-GP

Two modelling techniques are available on this package: Process Modelling and Data Projection.

- For process modelling any multi-input, multi-output system can be analysed. However time-series prediction is not yet available on this package.
- For data visualization purposes a multi input system can be analysed.
- Although there is no limit on the size of a data set, try and keep each data set less than 2000 observations per variable. Because a-GP is a population-based searching algorithm, a vast amount of processing is required which *will* slow down the computer!
- There is no limit on the number of variables.
- Try and keep population sizes less than 1000 for acceptable processing levels.

C.2 How to select a new algorithm

Click on `File|New|Regression` (for multi-input multi-output modelling) or `File|New|Feature extraction` (for data projection). The newly selected process will appear in a drop-down process box.

C.3 How to select a different process

Click on the down arrow of the process list box and select a new process from the list of available processes. See Figure C.1.

Stellenbosch University https://scholar.sun.ac.za

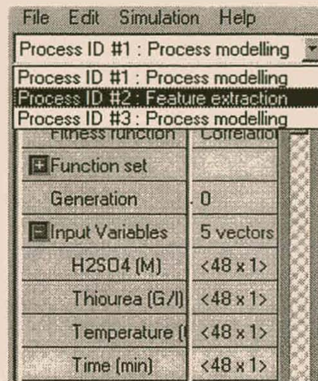


Figure C.2 : *The drop down list of the available processes. To activate one of the processes move the mouse cursor to the process and click.*

C.4 How to change the properties of an algorithm

All processes have properties such as `Population size`, `Elitism`, `Input vectors`, etc. Some of these properties can be directly manipulated. Others cannot. The properties which can be changed will have either a drop-down box or an edit control appear next to them when the user clicks on the value field of the property box. The property box is shown in Figure C.2.

Note that the box is divided into two fields, a Name field and a Value field. The Name

Elitism	true
Function set	
Generation	0
Input Variables	0 vectors
Max levels	10
Pc	60
Pm	4
Population size	50
Selection	Tournament
System Variabl	0 vectors
Fitness	-1
Individual	0
Stress	-2
Target spac	0

Figure C.3 : *The property box with all the available properties of the active process. The box is divided into two regions: a Name field and a Value field.*

fields indicates the name of the property while its associated value lies within the Value field. To change the value click on the Value field.

Stellenbosch University https://scholar.sun.ac.za

C.5 How to import data

C.5.1 Format of data file

All data files have to be in text format. Each column can be separated using a tab, space or comma delimiter. The first row may include labels for each column while all subsequent rows *must have* numeric values.

C.5.2 Importing the data

Select a data file by click on `File|Open` and selecting the text file you wish to analyse. Once the variables have been send to their desired process, they should appear in the `Input Variables` property of the property box. Clicking on the expand button

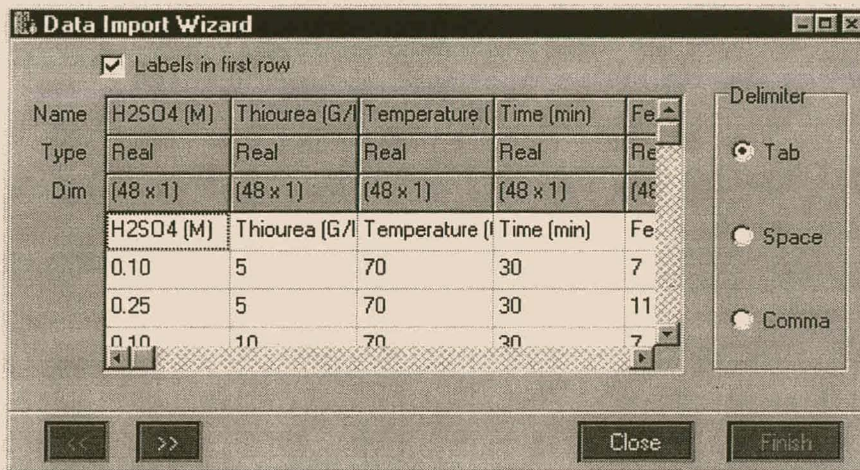


Figure C.4 : *The contents of the data file is displayed in the Data Import Wizard. If the first rows have labels click on "Labels in first row". The data type of each column can be specified by right clicking on the Type row of the corresponding column. Click on ">>" to continue.*

(indicated as a "+") all the variables or vectors will appear below the `Input Variables` property. If the current process is a Regression process, the variables that one wishes to use as output variables may be moved to the `Output Variables` property by clicking down with the mouse button on the name property of the variable and dragging and releasing it on the `Output Variables` property.

Stellenbosch University https://scholar.sun.ac.za

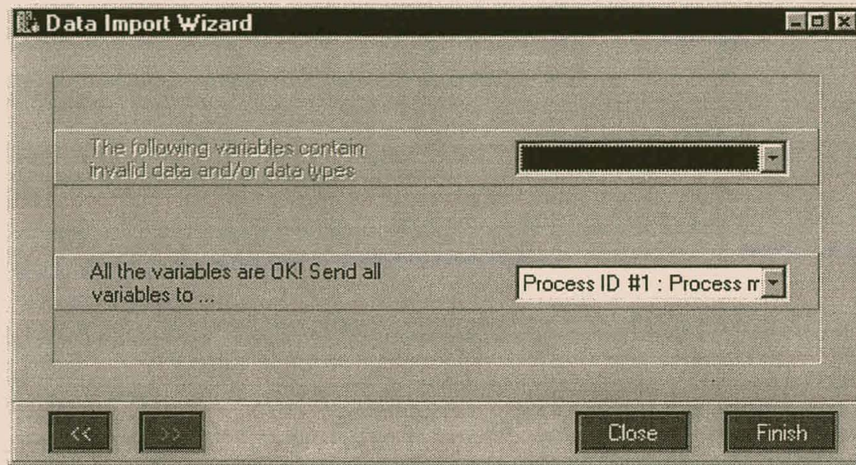


Figure C.5 : *If the contents of all the columns contain valid numeric values, the variables may be send to any of the processes listed in the process list box. Click on the down arrow and select a different process you wish to send the variables to.*

C.6 Starting the algorithm

C.6.1 The Start button

Clicking on `Simulation|Start` will start the simulation of the currently active process. This process' ID appears in the process drop-down box.

C.6.2 The Pause button

The algorithm can be Paused any time during simulation. Go to `Simulation|Pause` to pause it. When the current algorithm is paused a new algorithm can be selected from the available processes in the process box. This algorithm will then become active. Clicking on `Simulation|Start` will allow the newly selected process to continue/commence its simulation.

> **Note**
>
> The properties of a process can only be changed when the active process has been paused.

C.6.3 The Restart button

Stellenbosch University https://scholar.sun.ac.za

At any time during the simulation, one can restart the simulation. This will send an initialize signal to the active process, which will force it to re-initialize and subsequently restart. Click on `Simulation|Restart` to restart the active process.

C.7 Plotting the results

C.7.1 Available charts

Charts are divided into two categories: Active and Inactive charts. An Active chart is one which is continuously updated as the simulation progresses, while an inactive chart only displays the plot of a single instance during simulation. This basically implies that when an Active chart is used the execution speed will be degraded as the chart is updated. Inactive charts, although faster, does not allow continuous monitoring of the simulation.

The following charts are available:

- Active and Inactive line charts.
- Inactive scatter charts: Useful for feature extraction. It requires two vectors. One for the x-axis an another for the y-axis.
- Inactive frequency distribution charts: These charts allows the visualization of the distribution of a vector.
- Active and Inactive step line charts.

C.7.2 Dragging a vector to a chart

To display a specific variable or vector on a chart, click the mouse on the `Input Variables`, `Output Variables` or `System Variables` property. A list of available vectors will appear. A variable or vector can be identified by looking at the Value field of the property box. The size of the variable will be presented in the following format "<row x col>". Hold the mouse button down (in the Name field) on the specific variable you wish to plot, and drag it to the specific chart on which you would like to have it plotted.

Stellenbosch University https://scholar.sun.ac.za

C.7.3 Removing a specific plot from a chart

Right click on the specific chart from which you would like to remove a plot or plots. A list of available plots will appear (see Figure C.5).

Select the plot you want to remove by clicking on Delete and the plot you wish to remove. Alternatively you may decide to remove all the plots. Select Clear all.

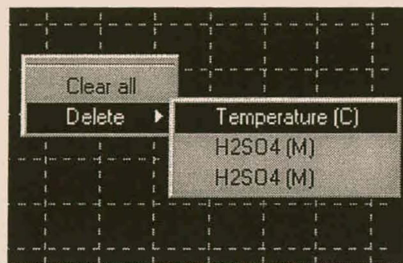


Figure C.6 : *The available plots are displayed on the panel. Clicking on any of the variables will remove it from the current chart.*

C.7.4 Changing chart types

Click on the specific chart that you would like to change. Now click on the `Chart Components` button at the bottom of the screen. A list of available charts types will appear on the left panel. Click on any chart type that you would like the selected chart to change to.

Note

When changing from an active to an inactive chart and *vice versa* the current plots will be lost.