Stony Brook University

OFFICIAL COPY

The official electronic file of this thesis or dissertation is maintained by the University Libraries on behalf of The Graduate School at Stony Brook University.

© All Rights Reserved by Author.

Machine Learning, Evolutionary Algorithms, and the Inference of Mathematical Truths

A Thesis Presented

by

Asher Hensley

to

The Graduate School

in Partial Fulfillment of the Requirements for the Degree of

Master of Science

in

Electrical Engineering

Stony Brook University

December 2013

Stony Brook University

The Graduate School

Asher Hensley

We, the thesis committee for the above candidate for the

Master of Science degree, hereby recommend

acceptance of this thesis.

Alex Doboli, Thesis Advisor
Associate Professor, Department of Electrical and Computer Engineering

John Murray, Second Reader
Associate Professor, Department of Electrical and Computer Engineering

This thesis is accepted by the Graduate School

Charles Taber
Interim Dean of the Graduate School

Abstract of the Thesis

Machine Learning, Evolutionary Algorithms, and the Inference of Mathematical Truths

by

Asher Hensley

Master of Science
in
Electrical Engineering

Stony Brook University
2013

In this thesis we set out to find whether the true data generating formula behind a set of data points can be automatically inferred from the data points alone. We start with the topic of machine learning and quickly realize that black box models can only approximate the real world which creates the motivation to move on to evolutionary algorithms as a vehicle to implement symbolic regression. Through a series of experiments we discover that the mean-squared error cost function is easily fooled by decoy solutions and is unable to make use of all the information presented in the training examples. Based on this result we develop the concept of feature signatures which uniquely define a set of training examples and possess several desirable properties, the most important being invariance to linear transformations. Armed with this concept we conduct several more numerical experiments based on common analytical functions and real world data sets which ultimately lead to the experimental evidence we need to support the thesis.

Contents

List of Figures xi

List of Tables xiii

1 Introduction 1
- 1.1 The World Around Us 2
- 1.2 Big Data 2
- 1.3 Black Box Regression 3
- 1.4 Problem Statement 3
- 1.5 Related Work 4
- 1.6 Thesis 6
- 1.7 Outline 6

2 Machine Learning 9
- 2.1 Introduction 10
- 2.2 Least Squares 11
  - 2.2.1 Historical Perspective 11
  - 2.2.2 Linear Models 11
  - 2.2.3 Assumptions 12
- 2.3 Radial Basis Function Networks 13
  - 2.3.1 Universal Approximation 13
  - 2.3.2 Gaussian Basis Functions 13
  - 2.3.3 K-Means Clustering 14
  - 2.3.4 Supervised vs. Unsupervised Learning 15
- 2.4 Neural Networks 15
  - 2.4.1 Multilayer Perceptrons 15
  - 2.4.2 Regression Networks 16
  - 2.4.3 Gradient Descent Backpropagation 17
  - 2.4.4 Gradient Descent with Momentum 17
  - 2.4.5 The Hessian and Newton's Method 18

2.4.6 The Levenberg-Marquardt Algorithm 19
2.4.7 Remarks 20
2.5 Experiment 2-1: Viscosity of Hydrogen 21
2.6 Summary 23

3 Evolutionary Algorithms 25
3.1 Introduction 26
3.2 Genetic Programming 26
3.2.1 The Population of Solutions 27
3.2.2 Numerical Constants 27
3.2.3 Solution Fitness 27
3.2.4 Shortcomings 28
3.3 Gene Expression Programming 28
3.3.1 Evolutionary Process 29
3.3.2 Solution Structure 30
3.3.3 Gene Expression 31
3.3.4 Replication 32
3.3.5 The Founder Effect 33
3.3.6 Roulette Wheel Selection and Elitism 33
3.3.7 Mutation 34
3.3.8 Recombination 35
3.3.9 Random Numerical Constants 36
3.3.10 Multiple Genes 37
3.3.11 Transpose 38
3.3.12 Inversion 39
3.3.13 Criticism 39
3.4 Experiment 3-1: Viscosity of Hydrogen 40
3.5 Summary 45

4 Convergence Experiments 47
4.1 Introduction 48
4.2 Experiment 4-1: A Simple Equation 51
4.3 Experiment 4-2: A Simple Equation + Noise 53
4.4 Experiment 4-3: Unnecessary Constants 55
4.5 Experiment 4-4: Necessary Constants 57
4.6 Experiment 4-5: Information Removal 59
4.7 Experiment 4-6: Simplified Viscosity of Hydrogen 64
4.8 Experiment 4-7: Imaginary Viscosity of Hydrogen 68
4.9 Summary 71

5 Feature Spaces **73**
5.1 Introduction 74
5.2 Kernel Functions 74
5.2.1 The Kernel Trick 74
5.2.2 Cost Functions Based on the Gram Matrix 75
5.3 Feature Signatures 75
5.3.1 Derivative Approximations 75
5.3.2 Mapping to Higher Dimensions 76
5.3.3 Invariance 77
5.3.4 Post Optimization 78
5.4 Experiment 5-1: Penalization Analysis 79
5.5 Experiment 5-2: Signature Learning 82
5.6 Summary 83

6 The Analytic World vs. the Real World **85**
6.1 Introduction 86
6.2 Analytic Experiments 86
6.2.1 Experiment 6-1: Polynomial 88
6.2.2 Experiment 6-2: Rational 92
6.2.3 Experiment 6-3: Trigonometric 96
6.2.4 Experiment 6-4: Logarithmic 100
6.2.5 Experiment 6-5: Exponential 104
6.3 Real Experiments 108
6.3.1 Experiment 6-6: Vapor Pressure 110
6.3.2 Experiment 6-7: Thermal Conductivity 112
6.3.3 Experiment 6-8: Emission of Electrons 114
6.3.4 Experiment 6-9: Magnetic Flux After Torsion 116
6.3.5 Experiment 6-10: Index of Refraction 118
6.3.6 Experiment 6-11: Resistance vs. Temperature 120
6.4 Summary 122

7 Conclusion **125**

Bibliography **129**

A Chapter 2 Programs **139**
A.1 LS.m 140
A.2 RBF.m 141
A.3 NN.m 143
A.4 Experiment 2-1: Viscosity of Hydrogen 144

B Chapter 3 Programs **147**
B.1 GEP Algorithm 148
B.1.1 Call Structure 148
B.1.2 reg_data.m 149
B.1.3 setup_config.m 150
B.1.4 gepfun.m 157
B.1.5 popgen.m 160
B.1.6 rand_idx.m 161
B.1.7 popexp.m 162
B.1.8 symconv.m 165
B.1.9 popeval.m 166
B.1.10 fitfun.m 168
B.1.11 fit_chk.m 171
B.1.12 nextgen.m 172
B.1.13 rep_fun.m 174
B.1.14 mutate_fun.m 175
B.1.15 recomb1_fun.m 177
B.1.16 recomb2_fun.m 178
B.1.17 recombG_fun.m 179
B.1.18 ist_fun.m 180
B.1.19 rist_fun.m 182
B.1.20 inv_fun.m 184
B.1.21 rnctrnsp_fun.m 186
B.1.22 rncinv_fun.m 188
B.2 Experiment 3-1: Viscosity of Hydrogen 190

C Chapter 4 Programs **191**
C.1 Experiment 4-1: A Simple Equation 192
C.2 Experiment 4-2: A Simple Equation + Noise 193
C.3 Experiment 4-3: Unnecessary Constants 194
C.4 Experiment 4-4: Necessary Constants 195
C.5 Experiment 4-5: Information Removal 196
C.6 Experiment 4-6: Simplified Viscosity of Hydrogen 198
C.7 Experiment 4-7: Imaginary Viscosity of Hydrogen 199

D Chapter 5 Programs **201**
D.1 Experiment 5-1: Sensitivity Analysis 202
D.2 Experiment 5-2: Feature Space Learning 204

E Chapter 6 Programs **205**
E.1 Analytic Experiments 206
E.2 Real Experiments 210
E.3 Real Experiment Data Sets 212

List of Figures

2-1 Experiment 2-1 Results (Least Squares - Left) 23
2-2 Experiment 2-1 Results (Radial Basis Functions - Middle) . . 23
2-3 Experiment 2-1 Results (Neural Networks - Right) 23

3-1 Parse tree representation of $\{+, a, \times, +, b, a, b, a, b, b, a\}$ 32
3-2 Experiment 3-1 Data 40
3-3 Experiment 3-1 Functions Discovered 42
3-4 Experiment 3-1 Learning Curves 42

4-1 Experiment 4-1 Data 51
4-2 Experiment 4-1 Learning Curves 52
4-3 Experiment 4-2 Data 54
4-4 Experiment 4-2 Learning Curves 54
4-5 Experiment 4-3 Learning Curves 55
4-6 Experiment 4-4 Learning Curves 57
4-7 Experiment 4-5 Results from run 1 (left) 61
4-8 Experiment 4-5 Results from run 2 (right) 61
4-9 Experiment 4-5 Results from run 3 (left) 61
4-10 Experiment 4-5 Results from run 4 (right) 61
4-11 Experiment 4-5 Results from run 5 (left) 62
4-12 Experiment 4-5 Results from run 6 (right) 62
4-13 Experiment 4-5 Results from run 7 (left) 62
4-14 Experiment 4-5 Results from runs 8 (right) 62
4-15 Experiment 4-5 Results from runs 9 (left) 63
4-16 Experiment 4-5 Results from runs 10 (right) 63
4-17 Experiment 4-5 Results from run 11 63
4-18 Experiment 4-6 Data 64
4-19 Experiment 4-6 Learning Curves 66
4-20 Experiment 4-7 Results (real part) 69
4-21 Experiment 4-7 Results (imaginary part) 69

4-22 Experiment 4-7 Learning Curves 69

5-1 Experiment 5-1 Fitness Scores 81

6-1 Experiment 6-1 Training Examples 88
6-2 Experiment 6-2 Training Examples 92
6-3 Experiment 6-3 Training Examples 96
6-4 Experiment 6-4 Training Examples 100
6-5 Experiment 6-5 Training Examples 104
6-6 Experiment 6-6 Results 111
6-7 Experiment 6-7 Results 113
6-8 Experiment 6-8 Results 115
6-9 Experiment 6-9 Results 117
6-10 Experiment 6-10 Results 119
6-11 Experiment 6-11 Results 121

List of Tables

2.1 Black Box Regression Method Summary 10

3.1 Experiment 3-1 Algorithm Configuration 41

4.1 Experiment 4-1 Algorithm Configuration 52
4.2 Experiment 4-5 Results 60
4.3 Experiment 4-6 Algorithm Configuration 65

5.1 Experiment 5-1 Algorithm Configuration 79

6.1 Analytic Experiments Configuration 87
6.2 Experiment 6-1 Scores 89
6.3 Experiment 6-2 Scores 93
6.4 Experiment 6-3 Scores 97
6.5 Experiment 6-4 Scores 101
6.6 Experiment 6-5 Scores 105
6.7 Real Experiments Configuration 109
6.8 Experiment 6-6 Data Set (courtesy of the UCI machine learning repository) 110
6.9 Experiment 6-7 Data Set (courtesy of the UCI machine learning repository) 112
6.10 Experiment 6-8 Data Set (courtesy of the UCI machine learning repository) 114
6.11 Experiment 6-9 Data Set (courtesy of the UCI machine learning repository) 116
6.12 Experiment 6-10 Data Set (courtesy of the UCI machine learning repository) 118
6.13 Experiment 6-11 Data Set (courtesy of the UCI machine learning repository) 120

Acknowledgments

In the year and a half it took to write this thesis, I owe an immense amount of gratitude to my wife Jacqueline for putting up with me and my constant preoccupation with this project. There's normal people time and there's Asher time. While working on this project my wife would constantly ask: how much longer do you expect to work tonight? To which I would reply: 5-10 more minutes. Several hours later I would realize Jacqueline was still waiting for me, only to find she had already fallen asleep. For taking this so well, I owe her much.

I also need to thank my advisor Alex for bringing this topic to me and asking some really tough questions. This problem seemed impossible to solve at the beginning, but with his help as time passed the solution slowly presented itself. Additionally I would like thank Rick from work who listened as I ranted and raved on and on about this project when he didn't have to, then offered advice on possible approaches I could take. Finally, I need to thank Rob from work for being understanding with the crazy schedule I had while working on this project.

Chapter 1

Introduction

"...mathematics is the foundation of all exact knowledge of natural phenomena."

David Hilbert, 1900 [1]

1.1 The World Around Us

Finding models that accurately describe the world around us can be a tricky thing. Humans have been trying to do it for centuries to understand phenomena like the motion of heavenly bodies, weather cycles, crop yields, and the spread of disease [2]. Early human reasoning attributed such processes to the gods, which in some cases was thought to be the sun [3]. Additional support for this hypothesis came from the occurrence of rare catastrophic events, or “Black Swans” [4]. However with the invention of mathematics came the ability to describe the world with numbers which was the beginning to revealing truth behind natural processes [2].

Today, we are still describing the world with numbers, but our ability to collect these numbers is far more advanced than our ability to understand these numbers [5]. As of this writing, genome sequencing machines are generating 15 petabytes of compressed genetic data per year [6], the Large Hadron Collider at CERN is generating 40 terabytes of data per second [5], and the next generation radio telescopes are expected to generate 70 petabytes per year [7]. The question is, now that we have all the data we could ever want, how do we understand it?

1.2 Big Data

The first problem of Big Data is how define it. Several authors have tried, and the current definition of the Big Data problem is the entire process recording, cleaning, extracting, processing, visualizing, and interpreting large data sets which cannot be tackled using traditional methods [8, 9, 10, 11]. However, most of these aspects of Big Data are irrelevant for us in this work. Although it’s important to know that Big Data exists and have a general idea of what it is, from here we will oversimplify and ignore the fact that we are drowning data and focus on the problem of knowledge extraction.

One of the current methods is to use machine learning techniques to find patterns in the data and make predictions about future measurements [12]. Although this covers a wide class of problems, we are going to again ignore the majority and only consider the regression problem being: given a set of continuous valued measurements, build a model that will predict future continuous valued measurements. One approach to the regression problem is to solve for the parameters of a black box model using either linear or nonlinear optimization methods. Some examples of “off the shelf” black box models are Neural Networks and Support Vector Machines. The question is, if we can

predict future measurements from an unknown system, does that qualify as knowledge?

1.3 Black Box Regression

The term *regression* is due to Sir Francis Galton who used it to refer to the *regression towards the mean* of a population as more samples are taken [13]. There are two problems with this view of the world, (1) lack of human understanding of the process at hand and (2) the ability of the model to generalize. The first issue is a direct result of the black box model paradigm. The black box model design process is defined as follows, "No physical insight is available or used, but the chosen model structure belongs to families that are known to have good flexibility and have been successful in the past" [14]. This is to say that black box models are an *approximation* of nature.

The second issue is the model's ability to predict new measurements which is where we can run into trouble with statistical methods. There is a whole theory to deal with this problem using cross validation with hold out data points, which is used extensively in the literature (see [15] and [16]). But is this really the path to mathematical truth?

1.4 Problem Statement

In this study we will examine the topic of extracting mathematical truth from data closely. The problem to be solved is: given a set of training examples (x and y), determine the true function $f(x) = y$ explicitly. This is an inverse problem where we want to infer the "white box" model $f(\star)$ which is defined as, "the case when a model is perfectly known; it has been possible to construct it entirely form prior knowledge and physical insight" [14]. It will be up to the computer to find the correct solution using only the knowledge and insight that can be gained from the training examples.

In order to proceed, the reader must be willing to accept the following condition: behind every set of data points, there is a *true* formula relating the independent and dependent variable(s). Although this is not true in general we will be focusing on problems of this type. This problem typically arises when trying to understand data generated by natural time invariant processes. Some of examples of this type of problem are determining the relationship between (1) voltage and current, (2) force and mass, (3) pressure and volume, and (4) temperature and viscosity. This is automated knowledge extraction from

data in its purest form, and it is thought this technique will one day automate science [17].

Finally, before continuing we need introduce the notion of a *decoy*, which is defined as: a model that fits all the known data yet is not the true data generating formula. Decoys arise where there are multiple solutions that explain the training data. If we take a step back and think for a moment, it becomes clear that any black box model is a decoy. These models fit the training examples but do not reveal any truth about the underlying process. The main challenge we will need to overcome in this study is to recognize the true solutions from the decoy solutions.

1.5 Related Work

For us, the path forward for inferring white box modes from data alone will begin with symbolic regression. Symbolic regression is a type of evolutionary algorithm (or genetic programming technique) which was made popular by the work of Koza [18, 19]. Under this approach, the input to the algorithm is a set of training examples which are representative of the system or process that is trying to be learned. The output is a symbolic equation which describes the system or process that generated the training examples. This is the short description which should be sufficient for the time being. Evolutionary algorithms and symbolic regression will be described in detail later in Chapter 3.

However, the inspiration for this study comes from the work of Schmidt [20, 21, 22, 23]. In [21] Schmidt reported that evolutionary algorithms were able to discover known laws of physics from experimental data collected from masses on springs and pendulums. This was followed up by [23] where a nice theory of symbolic regression was given to solve problems of the form $g(\mathbf{x}, y) = 0$. This is the implicit problem where the dependent variable of the given data set is in general unknown, and the algorithm needs to find the function $g(\star)$ that describes the entire data set using unsupervised learning [23]. The contribution of this work was in the use of invariants as a way of ignoring trivial solutions [21]. For example, Schmidt used implicit partial derivatives as a way to measure the ability of candidate solutions to predict the correct relationships between variables [23].

This work is one of the success stories of genetic programming along with several other examples outlined by Koza in [24] which were deemed to be *human competitive*. However, symbolic regression has not been widely accepted by the data science community as an “off the shelf” tool because

the theoretical aspects of evolvable algorithms are not well understood and there are scalability issues [17]. In fact there are many open problems in the area of symbolic regression such as [25]:

- optimal algorithm configuration
- solution generalization
- convergence criteria
- how to handle numerical constants
- solution bloating
- predicting convergence behavior from problem to problem

The relationship between solution bloating and generalization has received significant attention in the literature [26]. The general consensus among researchers is simple solutions tend to offer better generalization properties [27]. In [28] Amil et al used the Vapnik-Chervonekis dimension as a metric for controlling solution bloating combined with complexity penalization. In [29] Vladislavleva et al. used a nonlinearity order based on Chebyshev polynomial approximations as complexity measure for symbolic regression solutions. In [30] Castelli et al. proposed a rotationally invariant Graph Based Complexity (GBC) measure which is shown to be a good predictor of generalization performance. Additionally Keijer and Babovic proposed dimensionally aware genetic programming in [31] which takes into account the data units while scoring candidate solutions.

If we take a step back from the multi objective problem of minimizing solution complexity and maximizing solution generality a pattern emerges. For a population of solutions there exists an optimal solution for a given level of complexity. If the accuracy of every optimal solution is plotted as a function of complexity we will get the curve called the Pareto front. A formal definition of Pareto optimality can be found in [32] where it is pointed out by Zitzler et al. that one of the goals of evolutionary computing to approximate the *Pareto set* without doing a brute force search of the solution space. Smits and Kotanchek introduce a symbolic regression variant in [33] called ParetoGP where the Pareto front is accounted for and exploited during the evolutionary process. Additionally, one observation reported by Schmidt in [21] was the true solution was often found to be at the top of a large cliff on the Pareto front.

Finally Ferreira has answered the solution bloating question with a variant on genetic programming called gene expression programming (GEP) [34, 35]. The GEP algorithm is very elegant in its handing of variable sized solutions while maintaining a constant length representation. Additionally, symbolic regression solutions generated using standard genetic programming must be checked for validity, whereas any expression generated by the GEP algorithm is guaranteed to valid [34]. The GEP algorithm will be the main algorithm for symbolic regression in this study, and a more in depth presentation will be given in Chapter 3.

1.6 Thesis

One approach to finding the true data generating solution behind a set of data points is to construct the Pareto front and look for the largest cliff. However, construction of the Pareto set for a given problem even using evolutionary methods is still very computationally intensive and is very much like a brute force type of search for the true solution. Therefore, one of the goals in this work will be to infer the true solution directly without finding the Pareto set. Instead we aim to understand what types of features in the data allow for the true solution to be consistently found, and then address the question of whether the true solution can be found when these features are absent. Based on the research I have conducted in this area I can confidently state the following thesis:

Truth can be extracted from experimental data if and only if it exists and we can recognize it.

1.7 Outline

The remainder of this work will present experimental evidence to support this thesis as follows:

- **Chapter 2: Machine Learning**, In this chapter we will explore several approaches to regression in the area of machine learning and gain an understanding of their shortcomings which will serve as our motivation to move away from black box models. This will be illustrated through a simple experiment.

- **Chapter 3: Evolutionary Algorithms**, In this chapter we will briefly introduce Genetic Programming and give an depth presentation

of the Gene Expression Programming (GEP) algorithm from the symbolic regression perspective, which will be the main algorithm for this work. Here we will also see the shortcomings of symbolic regression through a simple experiment.

- **Chapter 4: Convergence Experiments**, In this chapter we will explore the convergence behavior of the GEP algorithm and try to understand why the correct solution can be found in some data sats and not others.

- **Chapter 5: Feature Spaces**, In this chapter we will take what was learned from the experiments in Chapter 4 and derive a new cost function based on mapping the training data to higher dimensional space. Through a short series of experiments we will find this new cost function has the ability to lead the GEP algorithm to the true data generating formula behind a set of data points.

- **Chapter 6: The Analytical World vs. The Real World**, In this chapter we will compare the performance of our new cost function to traditional symbolic regression in series of 11 experiments. The first 5 experiments are based on common analytical functions while the last 6 experiments are based on real world data sets from multiple experiments in physics and chemistry (courtesy of the UCI machine learning repository)

- **Chapter 7: Conclusion**, In this chapter we offer conclusions and discuss future work.

Chapter 2

Machine Learning

"We cannot expect to find a good child-machine at the first attempt. One must experiment with teaching one such machine and see how well it learns"

Alan Turing, 1950 [36]

2.1 Introduction

To design predictive regression models, a common place to start is the area of machine learning. Regression is the process of configuring a predetermined mathematical model to fit and predict empirical measurements usually by solving a linear or non-linear optimization problem. Here we will do a brief survey and discuss the following methods: least squares, radial basis functions, and neural networks. A basic summary of each method is given below in Table 2.1. Although these methods are quite useful in many situations, what they all share in common is that the designer must first define the model based on experience and/or intuition. Because regression models are predefined, the result is always a set of numbers corresponding to the unknowns of the model. As we will see via a simple experiment, this approach makes it difficult to gain any real insight on the inner workings of the process at hand. This is by no means an exhaustive presentation of this topic, nor is it meant to be. This chapter is meant to provide the motivation to move away from black box models.

Method	Least Squares	Radial Basis Functions	Neural Networks
Expressive Power	Medium	High	High
Model Complexity	Low	Medium	High
Human Interpretability	High	Medium	Low
Error Space	Convex	Convex	Non-Convex
Optimization	Linear	Linear	Non-Linear
Learning Style	Supervised	Semi-Supervised	Supervised
Learning Process	Direct	Iterative	Iterative
Optimality	Optimal	Sub-Optimal	Optimal

Table 2.1: Black Box Regression Method Summary

2.2 Least Squares

2.2.1 Historical Perspective

When confronted with a linear system of equations, the number of equations should equal the number of unknowns for a unique solution. However, when there are more equations than unknowns we face an interesting problem: how do we solve? The solution to this problem dates back to the turn of the 19th century where the first publication is often credited to Adrien Marie Legendre in his famous Nouvelles methods pour la determination des orbites des cometes [37]. However there is evidence that Carl Friedrich Gauss actually solved this problem first in 1798, which is supported by his famous prediction of where the asteroid Ceres would appear after being lost by the glare of the sun in 1801 [38]. Despite this, the method of least squares would not be where it is today without the contributions of several other key mathematicians such as Laplace, Euler, and Bernoulli [37].

2.2.2 Linear Models

The main contribution by Legendre was to recognize if there cannot be a unique solution to an over determined system of equations, there must be some optimal solution that will minimize the squared error [37]. This is particularly useful for fitting curves to noisy measurements. Typically, we will have a set of data points, say M measurements, which we will use design a model. Each measurement will consist of an input and output pair called a *training* example which we denote using the notation $(\mathbf{x}^{(k)}, y^{(k)})$ for the kth measurement. Note that here the superscript k does *not* mean exponentiation and the lowercase **bold** font means column vector. To refer to the ith element of the kth measurement we will use $x_i^{(k)}$. For the types of data we will consider in this work, the outputs $y^{(k)}$ will be scalar.

Given M measurements, the N dimensional input vector $\mathbf{x}$ is typically related to the output y as using the following model:

$$y^{(k)} = w_0 + \sum_{i=1}^{N} w_i x_i^{(k)} + \epsilon^{(k)} \tag{2.1}$$

where the weight w_0 is a bias term, and $\epsilon^{(k)}$ is the model error for the kth training example. Commonly the input vector $\mathbf{x}$ is modified to be an $N+1$ dimensional vector with $x_0 = 1$, so the above model can be written as an inner product:

$$y^{(k)} = \mathbf{w}^T\mathbf{x}^{(k)} + \epsilon^{(k)} \tag{2.2}$$

There are several ways to solve for the weight vector $\mathbf{w}$, and it can actually be shown that the least squares solution is also the Maximum Likelihood (ML) solution [15] when certain probabilistic assumptions are made regarding the error samples (normal, independent, and identically distributed) [39]. Typically the weights are found by minimizing the squared error cost function:

$$J(\mathbf{w}) = \frac{1}{2}\sum_{k=1}^{M}(y^{(k)} - \mathbf{w}^T\mathbf{x}^{(k)})^2 \tag{2.3}$$

This is done is the usual way by taking the gradient, setting it equal to 0, and solving for $\mathbf{w}$. It can be shown the closed form solution for $\mathbf{w}$ is:

$$\mathbf{w} = (\mathbf{X}^T\mathbf{X})^{-1}\mathbf{X}^T\mathbf{y} \tag{2.4}$$

where $\mathbf{X}$ is referred to as the *design* matrix and is defined by:

$$\mathbf{X} = \begin{bmatrix}\mathbf{x}^{(1)} & \mathbf{x}^{(2)} & \mathbf{x}^{(3)} & \cdots & \mathbf{x}^{(M)}\end{bmatrix}^T \tag{2.5}$$

and $\mathbf{y}$ is a column vector of the desired outputs for the model:

$$\mathbf{y} = \begin{bmatrix}y^{(1)} & y^{(2)} & y^{(3)} & \cdots & y^{(M)}\end{bmatrix}^T \tag{2.6}$$

Matlab is particularly well suited to solve this type of problem because of its elegant handling of matrices. It is unnecessary to implement the explicit solution because Matlab offers the matrix left divide operator. This will yield the solution to an over determined system of equations directly (see Appendix A).

2.2.3 Assumptions

In most cases the assumption that the error residuals are normally distributed and iid[1] is suspect, which makes any results of subsequent statistical analysis (i.e. confidence intervals, prediction variance, and statistical significance of coefficients) somewhat questionable. However, even

[1]iid is an acronym for independent and identically distributed

if this is the case, the resulting solution typically provides useful results [39]. There can also be problems when there are outliers in the data or when the input variables are too correlated (i.e. multicollinearity). Because of this, there are a vast amount of publications on regularization procedures and outlier detection/removal. The interested reader is referred to [40] for a general overview of this area and [41, 42, 43] for special attention to outlier analysis, iteratively reweighted least squares, and ridge regression.

2.3 Radial Basis Function Networks

2.3.1 Universal Approximation

Radial basis function networks began as an approach to exact interpolation due to the work of Powell [44], but were later modified to do regression and classifications tasks by the work of Broomhead and Moody in [45, 46] respectively. It was then shown by Park in [47, 48] that with a minimal number of restrictions these types of networks could do universal approximation. An excellent treatment of this topic can be found in Bishop's book on pattern recognition in [49].

An extension to this idea is captured by a class of techniques called sparse kernel machines, which is also covered by Bishop extensively in [15]. Of particular interest are support vector machines [16], and relevance vector machines [50] which the interested reader is encouraged to pursue. We will revisit the idea of kernels later in chapter 5 when consider new feature spaces.

2.3.2 Gaussian Basis Functions

Most of the available literature on radial basis functions (and neural networks for that matter) typically present complex diagrams to illustrate the input/output relationships of these systems. This type of presentation seems superfluous when the model can readily be written as,

$$y^{(k)} = w_0 + \sum_{i=1}^{N} w_i \phi_j(\mathbf{x}^{(k)}) \tag{2.7}$$

where $\phi_j(\star)$ is the jth radial basis function. This is commonly chosen to be the Gaussian,

$$\phi_j(\mathbf{x}) = exp\left\{-\frac{1}{2}(\mathbf{x}-\boldsymbol{\mu}_j)^T\boldsymbol{\Sigma}_j^{-1}(\mathbf{x}-\boldsymbol{\mu}_j)\right\} \tag{2.8}$$

where $\boldsymbol{\mu}_j$ and $\boldsymbol{\Sigma}_j$ are the jth mean vector and covariance matrix. Note that the least squares method described in the previous section is the special case when $\phi_j(\mathbf{x}) = x_j$. What we are doing here is taking the each input vector $\mathbf{x}$, creating a new input vector,

$$\boldsymbol{\Phi} = \begin{bmatrix}\phi_1(\mathbf{x}) & \phi_2(\mathbf{x}) & \cdots & \phi_N(\mathbf{x})\end{bmatrix}^T \tag{2.9}$$

setting up a new design matrix, and finding the least squares solution for the weight vector $\mathbf{w}$. The question then becomes how to configure each basis function.

2.3.3 K-Means Clustering

An attractive approach is to use a clustering algorithm to partition the training data in to K separate groups and use the sample mean and covariance from each group to initialize the basis functions. This offers a fast and direct two step solution to the model design: (1) initialize basis functions and then (2) solve for the weights.

There has been a lot of research on the clustering problem, which often arises in pattern classification problems. An excellent treatment of this area can be found in [51]. However, for this situation where we merely want to initialize a set a basis functions, a simple clustering algorithm is preferred. One such algorithm is the K-Means algorithm, which was applied in [46] to radial basis function initialization. The batch version of K-Means clustering is described in [52].

It can be shown (see [53]) that the sum of squares error will always decrease [49]. The K-Means clustering algorithm can also be done using online updates using the Robbins-Monroe procedure [49, 54]. Upon convergence, the mean for each basis function is computed using the sample mean,

$$\boldsymbol{\mu}_j = \frac{1}{M_j}\sum_{m\in S_j}\mathbf{x}^{(m)} \tag{2.10}$$

and the covariance for each basis function is computed with the sample covariance,

$$\mathbf{\Sigma}_j = \alpha \mathbf{I} + \frac{1}{M_j - 1} \sum_{m \in S_j} (\mathbf{x}^{(m)} - \boldsymbol{\mu}_j)(\mathbf{x}^{(m)} - \boldsymbol{\mu}_j)^T \tag{2.11}$$

where M_j is the number of elements in the jth cluster S_j and α is a smoothing parameter. In practice, the solution may not have the desired smoothness so the addition of the smoothing parameter α provides a tuning parameter, which can be varied until an acceptable solution has been reached. Alternate matrices other than $\alpha\mathbf{I}$ can also be used if we want to address the spread of each input variable separately, however this is beyond the scope of this work.

To implement a radial basis function network in Matlab, we have used the built in k-means clustering algorithm from the Statistics toolbox, computed the basis function parameters, and set up the design matrix. The weights are then computed using the least square solution (see Appendix A).

2.3.4 Supervised vs. Unsupervised Learning

The use of unsupervised learning is not necessarily optimal, but the use of supervised procedures will force us to solve a nonlinear optimization problem at which point neural networks start to become more attractive [49]. Therefore with this approach we have a tradeoff: we have sacrificed optimality for a faster and more simplistic training process. As with most regression models, the training data used to design the model is extremely important. The use of a clustering algorithm during the training process will cause the basis functions to converge to locations where measurements are more likely.

If the training data is not a good representation of the overall population, there will be problems when out of sample data points are encountered. The effects can be severe with the Gaussian kernel because measurements that have a large distance from the training set will be driven to zero. In addition, understanding the behavior of each dimension of the input variable is important when estimating means and covariances. The problem of model order selection must be addressed in practice however, this is beyond the scope of this work.

2.4 Neural Networks

2.4.1 Multilayer Perceptrons

It has been argued the neural network (i.e. the multilayer perceptron) originated with the work of Alan Turing in 1948 [51]. However the work of

Rosenblatt [55, 56] is commonly referenced as the beginning of neural networks, where the learning problem was addressed with the two layer perceptron [51]. There were several contributions to the learning problem after the work of Rosenblatt [57, 58, 59, 60], however they would receive little attention [51]. It wouldnt be until the work of Rumelhart et al in [61] that the backpropagation learning algorithm would gain support by the machine learning community [51]. Since then, a significant number of papers have been published in this area, arguably more than the topic deserves. As a result of its popularity, the biological plausibility of the backpropagation algorithm has been called into question and criticized by Grossberg in [62] and called highly implausible by Stork in [63] [51].

2.4.2 Regression Networks

Despite the controversy, neural networks have excellent expressive power and tend to perform well on most regression problems. The typical architecture for a three layer, vector input, scalar output regression network is as follows,

$$y^{(k)} = \alpha_0 + \sum_{j=1}^{n_H} \alpha_j f_{net} \left(w_{j,0} + \sum_{i=1}^{M} x_i^{(k)} w_{j,i} \right) \tag{2.12}$$

where $w_{j,i}$ is the weight from the ith input dimension to the jth hidden node, and α_j is the weight from the jth hidden node to the output. The function $f_{net}(\star)$ is commonly referred to as the activation function and is typically the tangent sigmoid,

$$f_{net}(a) = \frac{2}{1 + e^{-2a}} - 1 \tag{2.13}$$

The term n_H corresponds to the number of hidden units in the architecture and is a design parameter of the network. The network design process consists of learning the input to hidden and hidden to output weights using error backpropagation so as to minimize the error between the network output and the desired response. There are many ways to do this, many of which will be discussed in the following sections as we lead up the preferable Levenberg-Marquardt algorithm.

2.4.3 Gradient Descent Backpropagation

The simplest technique is to use gradient descent to iteratively learn the network weights by using the following update rule,

$$\mathbf{w}^{(k+1)} \leftarrow \mathbf{w}^{(k)} - \eta \nabla J(\mathbf{w}^{(k)}) \tag{2.14}$$

where η is the learning rate, $\mathbf{w}^{(k)}$ are the set of weights from the kth iteration, and $J(\star)$ is the cost function commonly taken to be the sum of squares criterion,

$$J(\mathbf{w}) = \frac{1}{2} \sum_{k=1}^{M} \left(y^{(k)} - \alpha_0 + \sum_{j=1}^{n_H} \alpha_j f_{net} \left(w_{j,0} + \sum_{i=1}^{M} x_i^{(k)} w_{j,i} \right) \right)^2 \tag{2.15}$$

The gradient of the criterion function is computed with respect to the network weights for each update,

$$\nabla J(\mathbf{w}) = \left[\frac{\partial}{\partial w_1} J(\mathbf{w}) \quad \frac{\partial}{\partial w_2} J(\mathbf{w}) \quad \cdots \quad \frac{\partial}{\partial w_L} J(\mathbf{w}) \right]^T \tag{2.16}$$

which can be computed directly by making the proper substitutions into the above criterion function [2]. Although this is a very stable algorithm, it can be painfully slow to converge and is easily trapped in local minima because the error space is not convex. Setting the learning rate correctly is somewhat of a balancing act in that small learning rates cause slow convergence, and fast learning rates can cause the search to overshoot the optimal solution.

2.4.4 Gradient Descent with Momentum

One approach to dealing with getting trapped in local minima and slow convergence is to add momentum to the search. This idea is loosely based on principles of physics, and is essentially a smoothing filter that is applied to the weight update rule to help the search escape local minima,

$$\mathbf{w}^{(k+1)} \leftarrow \mathbf{w}^{(k)} - [\mu_m \Delta \mathbf{w}^{(k-1)} + (1 - \mu_m) \eta \nabla J(\mathbf{w}^{(k)})] \tag{2.17}$$

[2] Note here all network weights have been organized into a $1 \times L$ set

Here $\mu_m \in 0, 1$ is the momentum constant (not to be confused with the mean of a Gaussian basis function from the section on Radial Basis Functions), and $\Delta\mathbf{w}^{(k-1)}$ is the weight change that occurred at the $(k-1)^{th}$ iteration. Note, a momentum constant of 0 will return to the standard gradient descent search. Although this approach addresses many of the shortcomings of the simple gradient descent search, it is generally considered ad-hoc and is still typically slower to converge than second order methods based on the Hessian matrix [64].

2.4.5 The Hessian and Newton's Method

The Hessian matrix $\mathbf{H}$ arises when we do a quadratic Taylor approximation of the criterion function around some point in weight space $\mathbf{a}$,

$$J(\mathbf{w}) \approx J(\mathbf{a}) + (\mathbf{w} - \mathbf{a})^T \nabla J(\mathbf{a}) + \frac{1}{2}(\mathbf{w} - \mathbf{a})^T \mathbf{H}(\mathbf{w} - \mathbf{a}) \tag{2.18}$$

where,

$$\mathbf{H}_{i,j} = \left. \frac{\partial}{\partial w_i w_j} J(\mathbf{w}) \right|_{\mathbf{w}=\mathbf{a}} \tag{2.19}$$

One of the classic optimization algorithms based on the Hessian is Newtons method,

$$\mathbf{w}^{(k+1)} \leftarrow \mathbf{w}^{(k)} - \mathbf{H}_k^{-1} \nabla J(\mathbf{w}^{(k)}) \tag{2.20}$$

which can offer very fast convergence properties, however calculating the Hessian matrix (and its inverse) directly can be computationally expensive. In addition Newtons method can become unstable if the quadratic approximation is poor, and can actually lead to a maximum if the Hessian is not positive definite [49]. As an alternative, quasi-Newton algorithms such as the Broyden-Fletcher-Goldfarb-Shanno (BFGS) algorithm, have been proposed which approximate the inverse Hessian recursively on each update offering lower computational cost and improved stability [65, 66].

2.4.6 The Levenberg-Marquardt Algorithm

The Levenberg-Marquardt (LM) algorithm takes another approach and approximates the Hessian using,

$$\mathbf{H} \approx (\mathbf{J}^T\mathbf{J} + \mu\mathbf{I}) \tag{2.21}$$

where $\mathbf{J}$ is the Jacobian matrix, which for a single output network is defined by,

$$\mathbf{J} = \begin{pmatrix} \frac{\partial e_1}{\partial w_1} & \frac{\partial e_1}{\partial w_2} & \cdots & \frac{\partial e_1}{\partial w_L} \\ \frac{\partial e_2}{\partial w_1} & \frac{\partial e_2}{\partial w_2} & \cdots & \frac{\partial e_2}{\partial w_L} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial e_M}{\partial w_1} & \frac{\partial e_M}{\partial w_2} & \cdots & \frac{\partial e_M}{\partial w_L} \end{pmatrix} \tag{2.22}$$

The term μ is a regularization variable (not to be confused with the momentum coefficient), commonly referred to as the combination coefficient. The update rule for the Levenberg-Marquardt algorithm is,

$$\mathbf{w}^{(k+1)} \leftarrow \mathbf{w}^{(k)} - (\mathbf{J}_k^T\mathbf{J}_k + \mu\mathbf{I})^{-1}\mathbf{J}_k^T\mathbf{e}_k \tag{2.23}$$

The beauty of this algorithm rests in its ability to be two algorithms; when μ is large the update rule approximates a gradient descent search with a learning rate of μ^{-1}, and when μ is small the update rule approximates the Gauss-Newton algorithm, which is a version of the Newton algorithm where the Hessian is approximated as $\mathbf{J}^T\mathbf{J}$. On its own, the Gauss-Newton algorithm can be unstable when the quadratic approximation of the criterion function is poor.

The way the LM algorithm addresses this is to decrease μ for every search step the in the correct direction, which causes the search to shift to the Gauss-Newton update only when a minimum is close and the quadratic approximation is good. When there is a step in the wrong direction, the LM algorithm inflates μ to effectively slow down the search. Typically, the change in μ is done with a constant β, where for a correct step $\mu \leftarrow \mu\beta^{-1}$ and for an incorrect step, $\mu \leftarrow \mu\beta$. When an incorrect step is taken, the update is not accepted, μ is inflated, and the iteration is repeated until a correct step is taken, or a maximum number of repetitions is reached.

We have implemented a simple Matlab function to design a neural network regression model using the Levenberg-Marquardt algorithm supplied with the Neural Network Toolbox. Neural networks are sensitive to the range of input values due the nature of the sigmoid activation function, therefore here we

have normalized the input and output values by subtracting out the mean and dividing by the standard deviation. The model output is then scaled appropriately to the correct range (see Appendix A) [3].

2.4.7 Remarks

Of all the regression models presented thus far, neural networks are by far the most complex. As we have seen, the input data is processed in multiple parallel paths, which can be controlled by the number of hidden units. Although the form of the model is clearly defined, it can be difficult to grasp exactly what is going on inside the processing chain, especially with multi-dimensional inputs and outputs. Despite this, neural networks continue to be valuable tool in regression and pattern recognition applications due to their expressive power.

[3] This routine has been written for scalar inputs only and will need to be modified to handle vector inputs.

2.5 Experiment 2-1: Viscosity of Hydrogen

Setup

So far we have presented several different types of regression methods, all which can be extremely useful. However, in this work we seek to discover meaningful relationships between the input and output variables that is more than just a curve fit. To illustrate the shortcomings of this class of methods, we will do a simple experiment using synthetic data based on the Sutherland's gas viscosity equation,

$$\mu = \lambda \frac{T^{3/2}}{T + C} \tag{2.24}$$

where T is the temperature in Kelvins, and λ and C are empirical constants, which vary depending on the gas in question. For hydrogen, $\lambda = 0.636236562 \times 10^{-6}$ and $C = 72$. For this experiment, we will generate 200 random data points using the above relation, add noise, then design least squares (LS), radial basis function (RBF), and neural network (NN) regression models. Subsequently we will examine each model to see if any insight can be gained on the underlying process as if we a measuring a new process with no prior knowledge.

This experiment has been executed by the Matlab program in Appendix A. We have generated 200 random temperatures from 0 to 555 degrees Kelvin and evaluated the viscosity equation with additive Gaussian noise with a zero mean and a 1×10^{-6} standard deviation. Then using our previously designed regression programs we have built LS, RBF, and NN regression models. The LS model has no special parameters, however for the RBF model we have chosen 3 basis functions and set the smoothing parameter to 1e6, and for the NN model we have selected 3 hidden units.

The resulting Least Squares, Radial Basis Function, and Neural Network[4] regression models are shown in the results section on the following page. The models are also plotted in Figures 2-1 through 2-3 over the measured data and truth data.

[4]The neural network equation assumes the input data has been normalized.

Results

Least Squares Model:

$$\mu(T) = 1.16 \times 10^{-6} + 2.35 \times 10^{-8} T \tag{2.25}$$

Radial Basis Function Model:

$$\begin{aligned}\mu(T) = &- 9.3 \times 10^{-4} + 1.1 \times 10^{-2} e^{-\frac{(T-95.5)^2}{2 \times 1.004 \times 10^6}} \\ &- 2.5 \times 10^{-2} e^{-\frac{(T-311.3)^2}{2 \times 1.003 \times 10^6}} + 1.5 \times 10^{-2} e^{-\frac{(T-474.2)^2}{2 \times 1.002 \times 10^6}}\end{aligned} \tag{2.26}$$

Neural Network Model:

$$\begin{aligned}\mu(T) = -0.015 &- 0.47 \left(\frac{2}{1 + e^{-2(0.56-1.6T)}} - 1 \right) \\ &- 0.11 \left(\frac{2}{1 + e^{-2(-1.17-5.8T)}} - 1 \right) \\ &- 0.35 \left(\frac{2}{1 + e^{-2(-3.57-4.6T)}} - 1 \right)\end{aligned} \tag{2.27}$$

Discussion

The obvious result is none of the algorithms found the true solution. Nor should they be because each model architecture was designed before any data was observed. In general the LS model under fit the data and the RBF and NN models over fit the data. It could be argued that with more data, alternate model architectures, and the use of cross validation procedures a better result could be attained. In general this is true, and for this experiment we have not attempted any such methods. Additionally, there are ways of pruning and growing these models [15, 49, 50, 51, 67], however it should be clear that it is impossible to infer the true data generating formula behind the data from this class of techniques alone.

This data set was very challenging in that there are literally an infinite number of solutions that fit the data because there are no real defining features. In running each algorithm we have done nothing more than select the first solution that minimized the error between the model and the data given the current algorithm architecture. In order to infer the true model we need to do

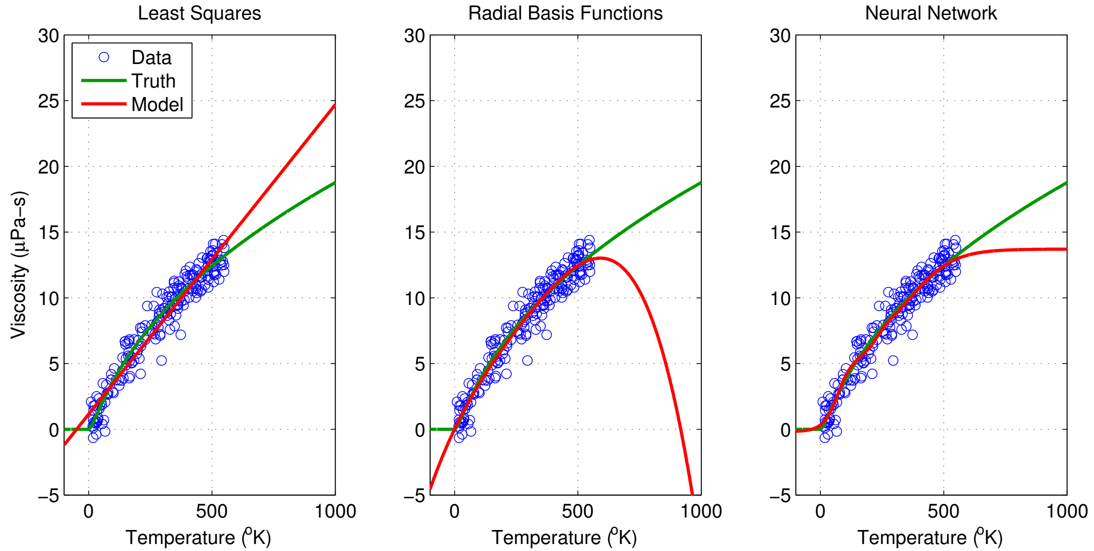


Figure 2-1: Experiment 2-1 Results (Least Squares - Left)

Figure 2-2: Experiment 2-1 Results (Radial Basis Functions - Middle)

Figure 2-3: Experiment 2-1 Results (Neural Networks - Right)

more. The next step will be to allow the model the adapt to the data during the learning process. These black box machine learning methods cannot offer such flexibility because they are mere approximators. This is the motivation for symbolic regression which where we will turn our attention to in the following chapter.

2.6 Summary

In this chapter we have done a brief survey of a few key machine learning regression methods. We have provided the background, the theory, and the Matlab implementation (in Appendix A) for several common techniques and have also done a simple experiment to illustrate their shortcomings. From the results we can conclude this class methods are incapable of finding the true data generating formula behind a given set of data. From here we will turn our attention to evolutionary algorithms and symbolic regression to see if we can obtain better results on the viscosity of hydrogen experiment.

Chapter 3

Evolutionary Algorithms

"...one general law, leading to the advancement of all organic beings, namely, multiply, vary, let the strongest live and the weakest die. "

Charles Darwin, 1859 [68]

3.1 Introduction

Evolutionary algorithms are stochastic search methods used to find solutions to variety of problems in regression, classification, optimization, as well as many other areas. Evolutionary algorithms are based the principles of natural selection where a population of candidate solutions are constantly evolved to try and solve the problem at hand. In this chapter we will focus on generating symbolic equations, often referred to as "symbolic regression". However, the concepts that will be presented are applicable to many other types of problems. We will first set the stage by discussing Genetic Programming (GP), which has its strengths and weaknesses. However this will lead us to the Gene Expression Programming (GEP) algorithm which has several desirable properties that GP algorithms do not have. We will then conclude this chapter by repeating the Hydrogen Viscosity experiment with the GEP algorithm and providing a brief summary.

3.2 Genetic Programming

Genetic Programming has its foundations in Genetic Algorithms (GA), which is where most discussions on this subject begin. This is a little confusing because the GP algorithm is also a genetic algorithm because any algorithm based on a population of candidate solutions evolving and adapting to a problem, commonly referred to as a "fitness landscape", can technically be called a genetic algorithm. However, the class of algorithms covered by GAs are a specific type of genetic algorithm. To clarify this, we will refer to all algorithms based on population evolution as evolutionary algorithms, *not* genetic algorithms. The name genetic algorithms will be reserved for the class of algorithms called GAs, which we will not be explicitly covering. Instead we would like to merely point out that GAs were introduced by Holland in 1975 [51], and the interested reader is encourage to review [69].

Genetic programming can be traced back to the work of Koza [18, 19] in which symbolic regression was only one aspect. Genetic programming in general is based on using code snippets to write computer programs to solve a particular problem. Although there are several publications in this area, an excellent treatment of GP theory can be found in [70]. At first this topic may seem challenging, however the concept is actually quite simple and is described by [51] in literally 2 pages. The remainder of this section briefly describes the GP algorithm and sets the stage for the GEP algorithm.

3.2.1 The Population of Solutions

As is the case with all evolutionary methods we begin with a population. This consists of generating several random guesses as to what the solution to the given problem is. For symbolic regression problems, this consists of generating several parse trees with nodes randomly selected from a function set or terminal set. The function set and terminal sets are defined prior to running the algorithm, and are typically influenced by the type of problem to be solved. For example, common function sets for problems expected to have algebraic solutions are $F = \{+, -, \times, \div\}$. Whereas if the solution is expected to transcendental, a better function set may be $F = \{+, -, \times, \div, \exp(\star), \sin(\star), \log(\star)\}$.

The terminal set is comprised of elements that end a branch on the parse tree. This is usually the independent variables of the data set and/or constants. For example, if we want to discover a function of the form $f(x, y) = z$, then the terminal set would be $T = \{x, y\}$, where the variables x and y represent the "data" we have measured. The variable z is also measured, and the problem is to find the mapping $(x, y) \mapsto z$. Another approach is to assume there is no explicit dependent variable and to try and find the implicit equation $f(x, y, z) = 0$. This has been successfully done by Schmidt in [21, 20, 23] for variety of physics problems based on pendulums and masses on springs.

3.2.2 Numerical Constants

Aside from independent variables, constants can also be included in the terminal set. This can either be known constants such as π, or randomly generated constants. This is one of the tricky areas in genetic programming which Koza has called *"a skeleton in the GP closet"* [25]. For example if we are trying to discover a function with a large constant like 1034.59, this can be very difficult without large constants in the terminal set. One common approach is to use the Ephemeral Random Constants technique, which is slightly out of scope here, but the interested reader is welcome to consult [18, 19]. There are some shortcomings to this approach which have been addressed in [71, 72, 73], however the constants problem is still considered open in the GP community [25, 74].

3.2.3 Solution Fitness

Once the population is initialized using the function and terminal sets, each candidate solution is tested against this fitness criterion. For symbolic

regression problems this is typically the Mean Square Error (MSE) between the output of the candidate solution and the dependent variable. This provides a score for each member of the population which allows the solutions to be ranked. Naturally, some solutions will be terrible, so it common to employ a survival threshold which solutions must meet otherwise they are removed completely from the population. The algorithm then evolves the remaining solutions using the genetic operators: replication, mutation, crossover, and insertion to make the next generation. The next generation is then scored based on the fitness criteria, a new generation is created, and so on until convergence. The computer's representation of each candidate solution can vary depending on the given language, but it has been recognized that the LISP language offers a convenient representation [51].

3.2.4 Shortcomings

Here we have given a basic description of the GP algorithm. In the next section we will present the GEP algorithm in detail and address the specifics of evolution and convergence. There are many parallels between the GP and GEP algorithm, and our preference to the latter is because the GEP algorithm handles many of the defects of the GP algorithm. Mainly, the GEP algorithm always results in legal expressions and the solution size bounded. With the GP algorithm, because we are manipulating the parse tree directly in the evolutionary process we must constantly check for illegal expressions. Similarly, parse trees can grow without bound via the crossover operation which must also guarded against. As a result, implementations of the GP algorithm must be contain a lot of rules to make sure the algorithm behaves, although the concept is actually quite simple. As we will see, the GEP algorithm does not have these problems by design resulting in a far more elegant implementation.

3.3 Gene Expression Programming

Gene Expression Programming (GEP) is an extension to Genetic Algorithms (GA) and Genetic Programming (GP) introduced by Candida Ferreira. Several papers have been published in this area [34, 75, 76, 77, 78, 79, 80] which have ultimately led to the comprehensive book [35]. The major contribution of GEP is that is it uses fixed length linear chromosomes to encode parse trees of varying size using the Karva language. GAs use fixed length linear chromosomes also, however the

resulting parse trees are also a fixed size which seriously impedes their ability to maneuver the search space. As we have seen, GPs vary the parse tree size by manipulating the parse trees directly, however they often result in illegal expressions and still lack the ability the efficiently search the solution space. Additionally, the GP parse trees must be bounded by additional rules otherwise they will grow without bound and will typically generate "bloated" solutions. GEP overcomes these problems by using a fixed length representation which can generate variable length solutions, with the addition of multiple new reproduction operations.

GEP can also be used for many types of problems outside the scope of this study such as decision tree induction, design of neural networks, and combinatorial optimization [35, 81, 82]. Here we will just focus on function discovery via symbolic regression according to our implementation which may slightly differ from Ferreira's implementation. So instead of referring the reader to [35] and only presenting results, the next section is a self contained introduction to the GEP algorithm. The reader is encouraged to consult [35] or [34] if there are any areas that are unclear or if other GEP applications are of interest.

3.3.1 Evolutionary Process

The concept of GEP is similar to GAs and GPs; there is a population of candidate solutions all competing for survival in an "environment". Solutions are randomly generated, tested against a "fitness" criteria, the solutions who don't meet the threshold are discarded, and solutions who do reproduce to create the next generation. The process then repeats with the next generation, and the next until a convergence criteria is met.

The steps for the GEP algorithm are as follows:

1. Initialize Population
2. Score Solutions Using Fitness Criterion
3. Test For Convergence
4. Replication
5. Mutation
6. 1-Pt Recombination
7. 2-Pt Recombination
8. Gene Recombination

9. Insertion Sequence Transpose
10. Root Insertion Sequence Transpose
11. Inversion
12. Go To Step 2

Unfit solutions are pruned out during the replication process and the remaining solutions are replicated according to how well they perform against the fitness criterion. This is the beginning of the next generation of solutions where most times there is a lot of solution redundancy. The entire set of solutions is then processed by each operator sequentially (mutation, recombination, etc.) where solutions are randomly chosen to be changed, meaning one solution can be affected by multiple operators. This process promotes genetic diversity and removes redundancy from the population.

What makes GEP different is the the way the solutions are represented and the reproduction process. Ferreira has introduced several new operations that help promote solution diversity which are straight forward to implement in the GEP framework. To try and and apply these operators in a GP algorithm would be prohibitively complex. The simplest way to understand how GEP works is by example. In this section we will present the algorithm by first discussing the Karva language representation and then moving onto the GEP operations for single and multi gene systems. For this study, we have implemented the entire algorithm in Matlab all of which has been included in Appendix B.

3.3.2 Solution Structure

As with all evolutionary algorithms, GEP begins with a randomly initialized population. Each candidate solution is composed of chromosome which can have one ore more genes. Each gene is made up of a head and a tail. The head size is a design parameter of the model and the tail size is determined by the maximum arity (i.e number of arguments) of the building blocks in the function library. For now, let us consider a single gene with an arbitrary head size of h_s. Given the head size and maximum arity, the tail size t_s can be computed using the following relation:

$$t_s = h_s(m_a - 1) + 1 \tag{3.1}$$

where m_a is the maximum arity of the function library. For example, if the head size is 5 and the maximum arity is 2, then the tail size would be 6. The

chromosome map for this single gene example with head elements H and tail elements T would then be written as

$$(H, H, H, H, H, T, T, T, T, T, T) \tag{3.2}$$

At this point we need to define 2 sets: the function set and the terminal set. A typical function set are the basic arithmetic operators, $\{+, -, \times, \div\}$ which is usually a good place to start. The terminal set is the set of independent variables we want to relate the the dependent variable(s). For now let us use the terminal set $\{a, b\}$ as an example. Upon initialization, the H elements of the chromosome are randomly set to any element of the function set or terminal set with equal probability. In our example so far, this would mean for each H position is Equation 3.2,

$$p(H = +) = p(H = -) = \cdots = p(H = a) = p(H = b) = \frac{1}{6} \tag{3.3}$$

However, the T elements can only be drawn from the terminal set, so

$$p(T = a) = p(T = b) = \frac{1}{2} \tag{3.4}$$

By doing so, every chromosome will always result in a legal expression [34].

3.3.3 Gene Expression

As an example, assume the following was generated:

$$(H, H, H, H, H, T, T, T, T, T, T) = (+, a, \times, +, b, a, b, a, b, b, a) \tag{3.5}$$

The right hand side of the above equation is the Karva representation of the parse tree shown in Figure 3-1. Karva notation is simply a list describing a parse tree from top to bottom and left to right. Here the first item on the list is + which has 2 arguments. The first argument is the next item on the Karva list a and the second argument is the following item $\times$. Because a is a terminal it has no arguments, however $\times$ has 2 arguments which are filled using the next 2 items on the Karva list, and so on. This process repeats until there are no more empty arguments. The parse tree can then be used to determine the symbolic equation, which in this case is

$$(+, a, \times, +, b, a, b, a, b, b, a) \mapsto a + b(a + b) \tag{3.6}$$

Depending on the arguments on the Karva list, not every element will necessarily be expressed in the parse tree. In this example, the last 4 elements were not expressed because there were not enough empty arguments. However this can change when we start introducing the evolution operators. As we will see, minor changes due to mutation and recombination can act as switches causing previously unexpressed parts of a gene to become active. This is the reason for Equation 3.1; in the event that every H element is from the function set, there will be enough terminals to satisfy every argument.

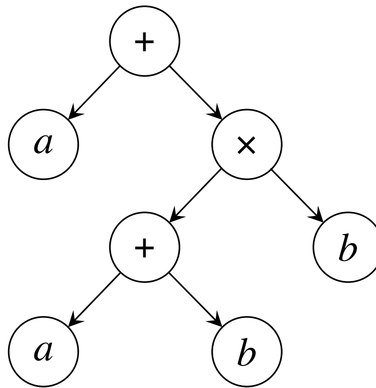


Figure 3-1: Parse tree representation of $\{+, a, \times, +, b, a, b, a, b, b, a\}$

3.3.4 Replication

After the initial population has been generated, and at the beginning of each new generation, the replication process is done. This consists of computing the "fitness" of every candidate solution in the current population and generating a new population using roulette wheel selection with elitism. The new generation is then passed on to the other operators such as mutation and recombination.

The baseline fitness function for this study was chosen to be the Mean Square Error cost function:

$$\Phi_{MS}(f) = K\left(1 + \frac{1}{N}\sum_{k=1}^{N}\left(y^{(k)} - f\left(\mathbf{x}^{(k)}\right)\right)^2\right)^{-1} \tag{3.7}$$

where f is the candidate solution under test, and K is the maximum fitness value.

3.3.5 The Founder Effect

Before discussing roulette wheel selection with elitism, we should mention our implementation of the GEP algorithm includes a founder threshold. The founder threshold only applies to the initial population, and is simply the minimum number of candidate solutions that meet the survival threshold (or minimum fitness score). At initialization, the algorithm has no knowledge of the environment and typically gets low fitness scores. This stage is critical in determining the trajectory of the algorithm, and the founder threshold helps guarantee that good candidates will be available in the population, thus allowing the algorithm to begin with good momentum. In [35] Ferreira uses $m = 1$ and argues that "this does not hinder the evolutionary process", however we have left this as a design parameter which can be varied. If the initial population does not meet the founder threshold, new populations are generated until the founder threshold is met.

3.3.6 Roulette Wheel Selection and Elitism

Once the founder threshold has been satisfied and the algorithm is running, after every generation we create the next generation using roulette wheel selection with elitism. Elitism means an identical copy of the solution with the highest fitness score is always included in the next generation. This guarantees the algorithm will never diverge. in the event the evolutionary operators fail to produce an improved solution, the algorithm will maintain the same position in the search space.

The number of individuals in each generation is constant and is a design parameter. If this is set to M, then after elitism $M-1$ copies of the candidates in the previous generation are made through the replication process. This is done randomly using roulette wheel selection which means solutions with higher fitness scores are more likely to be selected. This is done by first taking all the fitness scores $\mathbf{x}^{fit}$ from the previous generation (including the elite member's) and creating the probability mass function (PMF),

$$p(n) = x_n^{fit} \left(\sum_{k=1}^{M} x_k^{fit} \right)^{-1} \tag{3.8}$$

$M - 1$ draws are taken from this distribution (with replacement) and copied to the corresponding individuals of the new generation. This is done by generating $M - 1$ draws from the uniform distribution $U(0, 1)$ stored in some vector $\mathbf{u}$ and then computing,

$$\mathbf{g} = P^{-1}(\mathbf{u}) \tag{3.9}$$

where $\mathbf{g}$ is a vector of $M-1$ draws from $p(n)$ and $P(n)$ is the cumulative mass function (CMF),

$$P(n) = \sum_{k=1}^{n} p(k) \tag{3.10}$$

This is a standard result in probability theory, a proof of which can be found in any standard text such as [83]. There are some nuances to this because $\mathbf{u}$ is continuous and $\mathbf{g}$ is discrete which must be handled using interpolation (see Appendix B).

3.3.7 Mutation

After the replication process, the $M-1$ members of the new population (elite member is excluded) are subjected to the mutation process. Here we set a constant mutation rate R_m and compute the number of affected solutions N_m using

$$N_m = round(R_m(M-1)) \tag{3.11}$$

We next select N_m individuals (without replacement), apply the mutation operator, and return them to the population. If an candidate solution is selected for mutation, 1 or more head and/or tail elements are randomly selected and randomly changed to another value in the relevant set. For example, if a tail element is chosen, only entries from the terminal set can be used for mutation. The number of mutation points is a design parameter of the algorithm.

To get a better idea of how this works, consider our previous example: $(+, a, \times, +, b, a, b, a, b, b, a)$. Assume the algorithm has been configured to do a simple one-point mutation, causing the second element $\{a\}$ to be selected and replaced with the $\{\div\}$ operator. Obviously the new Karva representation will be $(+, \div, \times, +, b, a, b, a, b, b, a)$ which looks very similar to $(+, a, \times, +, b, a, b, a, b, b, a)$. However the parse tree will be completely restructured causing 2 previously unexpressed tail elements to turn on resulting in the following mutation,

$$a + b(a+b) \mapsto \frac{a+b}{b} + ab \tag{3.12}$$

3.3.8 Recombination

After mutation, the population is subjected to the recombination process which is a reproductive process like crossover. There are 3 types of recombination, 1-point, 2-point, and gene. Here we will discuss 1-point and 2-point recombination and defer gene recombination to the section on multigenic systems. Similar to the mutation process, the recombination rate is a design parameter of the algorithm and is chosen before initialization. Each type of recombination typically has it's own rate which for 1-point and 2-point we'll refer to as R_1 and R_2. It should be stressed that 1-point and 2-point recombination are *separate* processes which are done sequentially.

In each case, the number of affected individuals is computed using

$$N_k = 2 \times round(R_k(M-1)), k = 1, 2 \tag{3.13}$$

to ensure there is an even number. Subsequently N_k individuals are randomly selected (without replacement), the recombination process is applied to each sequential pair, and each set of offspring pairs are returned to the population in place of the parents. Consider the case where the following 2 parents are selected:

$$\begin{pmatrix} \mathbf{P}_1 \\ \mathbf{P}_2 \end{pmatrix} = \begin{pmatrix} +, a, \times, +, b, a, b, a, b, b, a \\ -, +, a, b, \times, b, b, a, a, a, b \end{pmatrix} \tag{3.14}$$

In a 1-point recombination, the parent matrix is randomly split into 2 partitions:

$$\begin{pmatrix} \mathbf{P}_1 \\ \mathbf{P}_2 \end{pmatrix} = \left(\begin{array}{cc} \mathbf{P}_{11} & \mathbf{P}_{12} \\ \mathbf{P}_{21} & \mathbf{P}_{22} \end{array} \right) = \left(\begin{array}{c|c} +, a, \times & +, b, a, b, a, b, b, a \\ -, +, a & b, \times, b, b, a, a, a, b \end{array} \right) \tag{3.15}$$

and the children $(\mathbf{Q}_1, \mathbf{Q}_2)^T$ are generated by swapping rows on either the first or second partition:

$$\begin{pmatrix} \mathbf{Q}_1 \\ \mathbf{Q}_2 \end{pmatrix} = \left(\begin{array}{cc} \mathbf{P}_{11} & \mathbf{P}_{22} \\ \mathbf{P}_{21} & \mathbf{P}_{12} \end{array} \right) = \left(\begin{array}{c|c} +, a, \times & b, \times, b, b, a, a, a, b \\ -, +, a & +, b, a, b, a, b, b, a \end{array} \right) \tag{3.16}$$

In a 2-point recombination, the parent matrix is randomly split into 3 partitions:

$$\begin{pmatrix}\mathbf{P}_1\\\mathbf{P}_2\end{pmatrix} = \begin{pmatrix}\mathbf{P}_{11} & \mathbf{P}_{12} & \mathbf{P}_{13}\\\mathbf{P}_{21} & \mathbf{P}_{22} & \mathbf{P}_{23}\end{pmatrix} \tag{3.17}$$

and the children are generated by swapping rows of the center partition:

$$\begin{pmatrix}\mathbf{Q}_1\\\mathbf{Q}_2\end{pmatrix} = \begin{pmatrix}\mathbf{P}_{11} & \mathbf{P}_{22} & \mathbf{P}_{13}\\\mathbf{P}_{21} & \mathbf{P}_{12} & \mathbf{P}_{23}\end{pmatrix} \tag{3.18}$$

Because of the Karva notation recombination always results in a legal expression, thus greatly simplifying the evolutionary process.

3.3.9 Random Numerical Constants

The GEP algorithm handles the constants problem using Random Numerical Constants (RNC) which is an addition to the basic GEP algorithm. In the basic GEP algorithm, numerical constants must be created from scratch which can be a challenge for non integer constants. The RNC approach handles this by adding a third domain after the tail which we will refer to as the C domain. The size is identical to the tailsize, and if we add onto our previous example (headsize of 5, maximum arity of 2), the chromosome map will be written as:

$$(H, H, H, H, H, T, T, T, T, T, T, C, C, C, C, C, C) \tag{3.19}$$

At initialization, the C elements are random filled from the set $(0, 1, 2, \ldots, 8, 9)$ which are indices to a vector of random variables. This vector is generated once at initialization and reused again and again as the algorithm evolves. The way these constants are accessed is by including the ? character in the terminal set which points to each index in order of appearance. For example, consider function:

$$(+, \times, ?, ?, b, a, ?, b, b, ?, a, 9, 4, 0, 7, 2, 4) \tag{3.20}$$

At initialization we will have generated a 10 element vector of random values from our favorite probability distribution which we will denote $\mathbf{z}$. Once the RNCs are taken into account, the above function is mapped to [1]

[1] To be consistent with Matlab syntax, each index of $\mathbf{z}$ has been incremented by one when expressed

$$(+, \times, z_{10}, z_5, b, a, z_1, b, b, z_8, a) \mapsto z_{10} + z_5 b \tag{3.21}$$

The relationship between density selection and achievable constants may be a factor, but has not been addressed in the literature. For example, if the vector $\mathbf{z}$ is populated from $U(0, 1)$ what are the chances we will find the constant 1004.234532? As we will see this is a very difficult problem.

3.3.10 Multiple Genes

For complex problems, single gene systems are typically inadequate so we must at some point turn to multi gene systems. Multi gene systems are quite simply single gene systems linked together by arithmetic operators. These linking operators can be evolvable, however for our implementation we have chosen to use the addition operator $\{+\}$, although it can be easily changed in the configuration script. For example, if we want to extend the template in Equation 3.19 to a 3 gene system, the chromosome map would be written as:

$$(\mathbf{G}_1, \mathbf{G}_2, \mathbf{G}_3) = (\mathbf{H}_5, \mathbf{T}_6, \mathbf{C}_6, \mathbf{H}_5, \mathbf{T}_6, \mathbf{C}_6, \mathbf{H}_5, \mathbf{T}_6, \mathbf{C}_6) \tag{3.22}$$

where

$$\mathbf{H}_5 = (H, H, H, H, H) \tag{3.23}$$

$$\mathbf{T}_6 = (T, T, T, T, T, T) \tag{3.24}$$

$$\mathbf{C}_6 = (C, C, C, C, C, C) \tag{3.25}$$

For multi gene systems, each gene is expressed individually as we have seen previously, then the resulting equations are combined using the linking operator. The mutation and recombination operations work the same, there are now more indices to chose from. The main difference is instead of creating a random vector of constants for the RNCs we now make a random matrix $\mathbf{Z}$ so each gene has a difference set of constants. In the previous example we have $\mathbf{Z} \in \mathbb{R}^{10 \times 3}$, making the C element index point the row and the gene index point to the column of $\mathbf{Z}$.

3.3.11 Transpose

The transpose operation is the first process so far that is unique to the GEP algorithm. There are 2 types that we will consider: Insertion Sequence (IS) transpose and Root Insertion Sequence (RIS) transpose [2]. These operations begin much the same as the previous operations, the number of affected individuals in the population must be identified using the predefined IS and RIS transpose rates. These rates are multiplied by the population size minus 1, rounded, and that number of candidate solutions are randomly selected (without replacement) for processing. As with 1-point and 2-point recombination, the IS and RIS transpose operators are separate and sequential operations.

If an individual is selected for an IS transpose operation, a gene is selected at random and a sequence length L is selected at random from a predefined set, such as $\{2, 3, 4\}$, which is a design parameter of the algorithm. Next, L sequential elements from the $\{H, T\}$ domain (called the transposon) are randomly chosen from the gene and copied to a new randomly selected position in the head. For example, assume the second gene of a multi gene system $\mathbf{G}_2 = (\mathbf{H}_5, \mathbf{T}_6, \mathbf{C}_6)$ has been selected for the case of $L = 2$:

$$(+, \times, -, ?, b, a, ?, \mathbf{b}, \mathbf{b}, ?, a, 9, 4, 0, 7, 2, 4) \tag{3.26}$$

Here, the transposon $(\mathbf{b}, \mathbf{b})$ has been "randomly" selected and bolded. Assume a head index of 3 is randomly selected, and the transposon is inserted in the head to yield:

$$(+, \times, \mathbf{b}, \mathbf{b}, -, ?, b, a, ?, \mathbf{b}, \mathbf{b}, ?, a, 9, 4, 0, 7, 2, 4) \tag{3.27}$$

However, now the gene is of the form $(\mathbf{H}_7, \mathbf{T}_6, \mathbf{C}_6)$ which is illegal, so the last 2 elements of the head $(?, b)$ are discarded to get:

$$(+, \times, \mathbf{b}, \mathbf{b}, -, a, ?, \mathbf{b}, \mathbf{b}, ?, a, 9, 4, 0, 7, 2, 4) \tag{3.28}$$

which is the final result. For this example, the IS transpose operation has done the following mapping:

$$b\mathbf{Z}_{10,2} + (a - \mathbf{Z}_{5,2}) \mapsto b + b(a - \mathbf{Z}_{10,2}) \tag{3.29}$$

[2] There is a third type called gene transpose, however we have omitted this operation from our GEP implementation.

3.3.12 Inversion

The last operator we will discuss is the inversion operator. As before, when its time to apply this operator several candidate solutions are randomly chosen from the pool of available solutions. For each chosen solution, a gene is randomly chosen and the first k elements are flipped from left to right. The number k is randomly chosen from a predefined set that must be configured at initialization. For example consider the following candidate solution discussed previously:

$$(+, \times, -, ?, b, a, ?, b, b, ?, a, 9, 4, 0, 7, 2, 4) \tag{3.30}$$

the inversion operation on this solution for the case of $k = 3$ would yield the following result:

$$(-, \times, +, ?, b, a, ?, b, b, ?, a, 9, 4, 0, 7, 2, 4) \tag{3.31}$$

Observe how the first 3 elements have been flipped from left to right. This implementation differs slightly from that of [35], where the starting point of the inversion sequence can be anywhere in the head of the gene.

3.3.13 Criticism

As with all evolutionary algorithms, the GEP algorithm has a lot of moving parts which makes it difficult to define its theoretical properties exactly. Finding the optimal model configuration for a given problem is challenging and is considered an open problem in evolutionary computing [25]. The GEP algorithm has been criticized by Oltean in [84] where it is pointed out that the GEP algorithm is more sensitive to the algorithm configuration (head size, number of genes, etc.) than other similar approaches. Additionally the linking operator for multi gene systems tends to constrain the search space, and the use of transcendental functions tend to overcomplicate the search space [84].

3.4 Experiment 3-1: Viscosity of Hydrogen

Setup

Here we have repeated the Viscosity of Hydrogen experiment from chapter 2 using the GEP algorithm. We have generated 200 random samples from the equation

$$\mu(T) = \lambda \frac{T^{3/2}}{T + C} \tag{3.32}$$

uniformly over the interval from 0 to 555 degrees Kelvin, where $\lambda = 0.636236562$ and $C = 72$. We've also added Gaussian noise to the generated data with $\sigma = 1$, creating the data set shown in Figure 3-2. The GEP algorithm configuration is shown in Table 3.1, and the MATLAB code used to run this experiment can be found in Appendix B. Due to the noise that we've added, it's unrealistic to expect the algorithm to reach the maximum fitness. Therefore the convergence threshold τ has been compensated using,

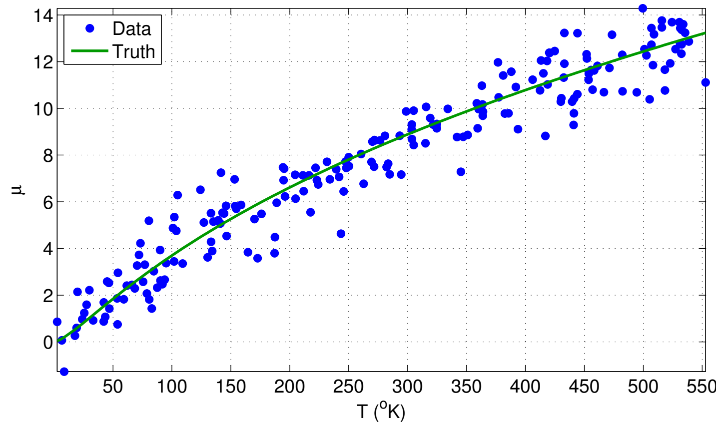


Figure 3-2: Experiment 3-1 Data

Algorithm Parameters	
Terminal Set	$\{a, ?\}$
Function Set	$\{+, -, \times, \div, \sqrt{\lvert \star \rvert}\}$
Genes	2
Head Size	6
Tail Size	7
Gene Size	20
Chrm Size	40
Runs	25
Population Size	200
Max Generations	100
Mutation Rate	0.1
1-Pt Recombination Rate	0.3
2-Pt Recombination Rate	0.3
Gene Recombination Rate	0.3
IST Rate	0.1
RIST Rate	0.1
Inversion Rate	0.1
Min Founders	1
Survival Threshold	10
Convergence Threshold	510
Maximum Fitness	1000

Table 3.1: Experiment 3-1 Algorithm Configuration

$$\tau = \left\lfloor \frac{K}{1 + \mathbb{E}(\mathbf{n}^2)} \right\rfloor \tag{3.33}$$

where K is the maximum fitness (1000 in this case), $\mathbb{E}(\mathbf{n}^2)$ is the second moment of the noise, and $\lfloor \star \rfloor$ means round down or take the floor. For this experiment our estimate of $\mathbb{E}(\mathbf{n}^2)$ was slightly off yielding a convergence threshold of 510 instead of 500.

Results

The GEP algorithm was unable to find the correct function in every run we did in this experiment. The functions that were found are plotted in Figure 3-3 and the learning curves are plotted in Figure 3-4. The algorithm has over fit the data as we can clearly see each solution immediately diverges

as soon as it leaves the training interval. This could be a property of the training data in that there are no defining features in the training interval. Additionally we have not constrained the solution to be zero for negative temperatures thus creating a situation where several decoy solutions meet the convergence criteria, when in fact they diverge outside the training interval. This is confirmed in Figure 3-4 where we can clearly see a handful of runs met the convergence criteria but are incorrect. The resulting solutions from each run (in order of fitness) are in the following section.

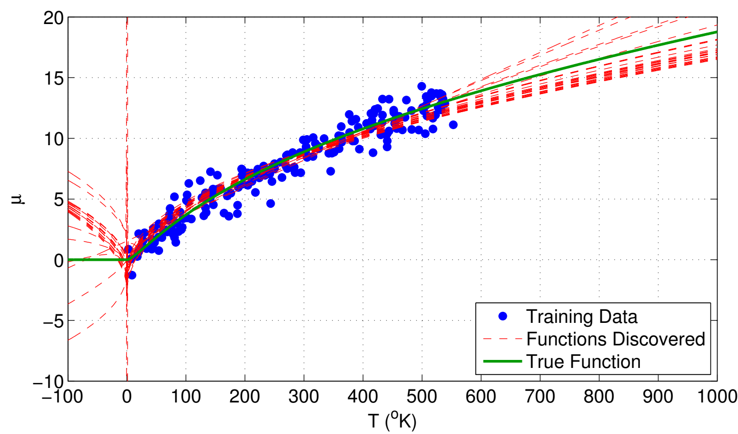


Figure 3-3: Experiment 3-1 Functions Discovered

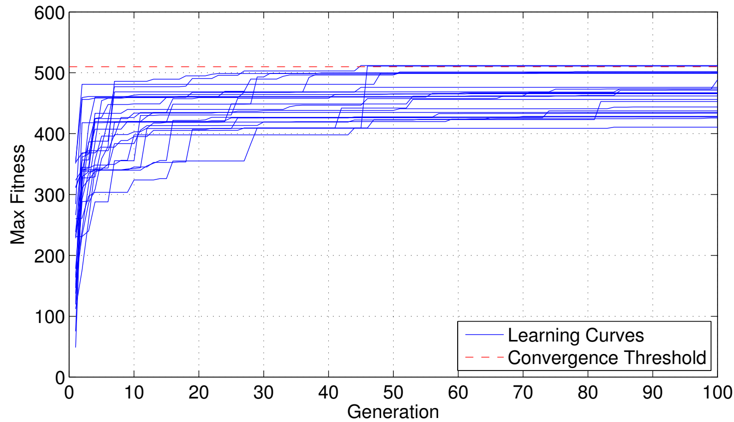


Figure 3-4: Experiment 3-1 Learning Curves

Algorithm Solutions

$$\sqrt{|a| \left|\frac{1}{a^2} + 0.31\right|} - 0.73 \tag{3.34}$$

$$0.55\,\sqrt{|a - 1.0|} - \frac{1.0\,a}{1.2\,a - 1.2\,\sqrt{|a - 0.86|}} \tag{3.35}$$

$$\frac{0.56\,a}{\sqrt{|a|}} - 1.1 \tag{3.36}$$

$$0.55\,(|1.1\,a - 0.73|\;|a|)^{\frac{1}{4}} - 0.97 \tag{3.37}$$

$$\sqrt{|0.31\,a - 1.0|} - \frac{1.0\,a}{a - 0.2} \tag{3.38}$$

$$\sqrt{|0.31\,a + 0.54|} - \frac{1.5\,\sqrt{|a|}}{a} - 0.93 \tag{3.39}$$

$$0.59\,\sqrt{|a|} + \frac{\sqrt{|a|}}{a} - 1.3 \tag{3.40}$$

$$\sqrt{\left|\sqrt{|a|} - 0.31\,a\right|} - \frac{1.5\,\sqrt{|a|}}{a} + \frac{0.19}{a} \tag{3.41}$$

$$0.6\,\sqrt{|a|} - 1.9 \tag{3.42}$$

$$\sqrt{|0.34\,a + 0.033|} - 1.4 \tag{3.43}$$

$$\frac{1.8}{a} + \sqrt{|0.34\,a - 0.92|} - 1.5 \tag{3.44}$$

$$0.59\,\sqrt{\left||a|^{\frac{1}{4}} - 1.0\,a\right|} - 1.5 \tag{3.45}$$

$$0.59\,\sqrt{|a - 0.86|} + \frac{8.5}{a^2} - 1.6 \tag{3.46}$$

$$0.022\,a + 1.5 \tag{3.47}$$

$$0.6\,\sqrt{|a|} - 1.7 \tag{3.48}$$

$$\sqrt{\left|1.8\,\sqrt{|a|} - 0.35\,a\right|} - \frac{0.96}{a} \tag{3.49}$$

$$0.6\,\sqrt{|a|} - \frac{1.0\,a}{a - 0.33} - 0.73 \tag{3.50}$$

$$\sqrt{|0.39\,a + 0.42|} - 2.0 \tag{3.51}$$

$$0.65\,\sqrt{|a|} - 2.4 \tag{3.52}$$

$$\left(6.7 \cdot 10^{-3}\right)\,a + 0.42\,\sqrt{|a - 0.36|} - 0.79 \tag{3.53}$$

$$0.019\,a + \sqrt{\left|0.45\,\sqrt{|a|} - 1.4\right|} \tag{3.54}$$

$$0.65\,\sqrt{|a|} + \frac{0.11}{a} - 2.5 \tag{3.55}$$

$$\sqrt{|0.43\,a + 0.13|} - \frac{0.2}{a} - 2.6 \tag{3.56}$$

$$\frac{0.12\,a}{|a|^{\frac{1}{4}}} + 0.035 \tag{3.57}$$

$$0.011\,a + 0.33\,\sqrt{|a|} - 0.54 \tag{3.58}$$

3.5 Summary

In this chapter we began by giving a high level introduction to evolutionary algorithms using the GP algorithm. The shortcomings of the GP algorithm provided motivation to use the GEP algorithm for our experiments. After discussing the GEP algorithm in detail we repeated the hydrogen viscosity experiment and found that the GEP algorithm performed poorly when faced with the task of mining the true hydrogen viscosity equation from the data. There are several possible reasons for this, which will the topic of the next chapter.

The big problem with evolutionary algorithms is they have so many parameters that must be set, it's hard to know for sure the algorithm is optimally configured for the given problem without exhaustive verification. At this point its hard to say whether our experiment failed because of the problem to be solved or the algorithm was not optimally configured.

Chapter 4

Convergence Experiments

"When things get too complicated, it sometimes makes sense to stop and wonder: Have I asked the right question?"

Enrico Bombieri, 1992 [85]

4.1 Introduction

As we have seen the GEP algorithm could not successfully find the Hydrogen Viscosity formula explicitly in Chapter 3. This was an unexpected result given how simple this formula appears. There are several possible reasons for this result such as,

1. Explicit equations cannot be discovered reliably
2. Noise sensitivity
3. The numerical constants could not be found
4. Insufficient training data
5. Too many local fitness maximums

In this chapter we will explore these possibilities through a series of numerical experiments to see if we can gain some insight and discover where symbolic regression using the GEP algorithm breaks down. We have done a total of 7 experiments ranging from simple to complex which we will briefly discuss before presenting the results.

Experiment 4-1: A Simple Equation

In this experiment we have attempted to find following simple equation to test hypothesis 1:

$$f(x) = \frac{1}{x^2+1}, \quad x \in [-10, +10] \tag{4.1}$$

Experiment 4-2: A Simple Equation + Noise

This experiment is repeat of Experiment 4-1 to test hypothesis 2:

$$f(x) = \frac{1}{x^2+1} + \mathbf{n}, \quad x \in [-10, +10] \tag{4.2}$$

Experiment 4-3: Unnecessary Constants

This experiment is a repeat of Experiment 4-2 where we have enabled random numerical constants to see if this plays a role in creating local fitness maximums in the search space (hypothesis 5):

$$f(x) = \frac{1}{x^2+1} + \mathbf{n}, \quad x \in [-10, +10] \tag{4.3}$$

Experiment 4-4: Necessary Constants

Here we have repeated the conditions of Experiment 4-3 to test hypothesis 3 with the equation

$$f(x) = \frac{\pi}{x^2 + 1} + \mathbf{n}, \quad x \in [-10, +10] \tag{4.4}$$

Experiment 4-5: Information Removal

In this experiment we have repeated Experiment 4-4 eleven times, each time shortening the interval of x to determine at which point there is not enough information for the algorithm to reliably find the correct solution (hypothesis 4):

$$(1) \quad f(x) = \frac{\pi}{x^2 + 1} + \mathbf{n}, \quad x \in [-8, +10] \tag{4.5}$$

$$(2) \quad f(x) = \frac{\pi}{x^2 + 1} + \mathbf{n}, \quad x \in [-6, +10] \tag{4.6}$$

$$(3) \quad f(x) = \frac{\pi}{x^2 + 1} + \mathbf{n}, \quad x \in [-4, +10] \tag{4.7}$$

$$(4) \quad f(x) = \frac{\pi}{x^2 + 1} + \mathbf{n}, \quad x \in [-2, +10] \tag{4.8}$$

$$(5) \quad f(x) = \frac{\pi}{x^2 + 1} + \mathbf{n}, \quad x \in [-1, +10] \tag{4.9}$$

$$(6) \quad f(x) = \frac{\pi}{x^2 + 1} + \mathbf{n}, \quad x \in [0, +10] \tag{4.10}$$

$$(7) \quad f(x) = \frac{\pi}{x^2 + 1} + \mathbf{n}, \quad x \in [1, +10] \tag{4.11}$$

$$(8) \quad f(x) = \frac{\pi}{x^2 + 1} + \mathbf{n}, \quad x \in [2, +10] \tag{4.12}$$

$$(9) \quad f(x) = \frac{\pi}{x^2 + 1} + \mathbf{n}, \quad x \in [4, +10] \tag{4.13}$$

$$(10) \quad f(x) = \frac{\pi}{x^2 + 1} + \mathbf{n}, \quad x \in [6, +10] \tag{4.14}$$

$$(11) \quad f(x) = \frac{\pi}{x^2 + 1} + \mathbf{n}, \quad x \in [8, +10] \tag{4.15}$$

Experiment 4-6: Simplified Viscosity of Hydrogen

In this experiment we have done a simplified viscosity of hydrogen experiment over a larger range of temperatures and set the constant C equal to 1 instead of 72 to determine if finding the correct solution is any easier for the algorithm (revisiting failure hypothesis 3):

$$\mu(T) = 0.636 \frac{T^{3/2}}{T+1}, \quad T \in [-5 \times 10^5, 10 \times 10^6] \tag{4.16}$$

We have considered only the real part of the viscosity, thus enforcing the condition that for negative temperatures the viscosity must be zero.

Experiment 4-7: Imaginary Viscosity of Hydrogen

In this experiment we have let the algorithm try to fit the imaginary part of the viscosity for negative temperatures to see if the pole a $T = -1$ provides any additional information (revisiting failure hypothesis 4):

$$\mu(T) = 0.636 \frac{T^{3/2}}{T+1}, \quad T \in [-20, +20] \tag{4.17}$$

4.2 Experiment 4-1: A Simple Equation

Setup

Let's assume for a moment that the Hydrogen Viscosity equation was too difficult for the GEP algorithm to find. For this first experiment, we have asked the algorithm to learn:

$$f(x) = \frac{1}{x^2 + 1} \tag{4.18}$$

based on 200 random x values from the uniform distribution $U(-10, +10)$ as shown in Figure 4-1 and configured the GEP algorithm as shown in Table 4.1.

Results

We ran 10 independent runs, all of which converged to Equation 4.18 within 20 generations as shown by the learning curves in Figure 4-2. This proves the GEP algorithm is capable of finding the true formula behind a set of data points when the conditions are right, and that explicit equations can be discovered reliably. Why then did Experiment 3-1 fail?

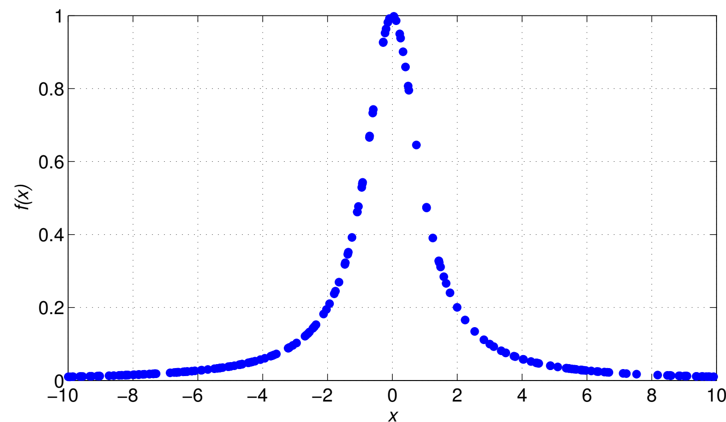


Figure 4-1: Experiment 4-1 Data

Algorithm Parameters	
Terminal Set	x
Function Set	$\{+, -, \times, \div\}$
Genes	1
Head Size	10
Tail Size	11
Gene Size	21
Chrm Size	21
Runs	10
Population Size	200
Max Generations	100
Mutation Rate	0.1
1-Pt Recombination Rate	0.3
2-Pt Recombination Rate	0.3
Gene Recombination Rate	0.3
IST Rate	0.1
RIST Rate	0.1
Inversion Rate	0.1
Min Founders	1
Survival Threshold	10
Convergence Threshold	999
Maximum Fitness	1000
Random Numerical Constants	Off
Fitness Function	$1000/(1 + MSE)$

Table 4.1: Experiment 4-1 Algorithm Configuration

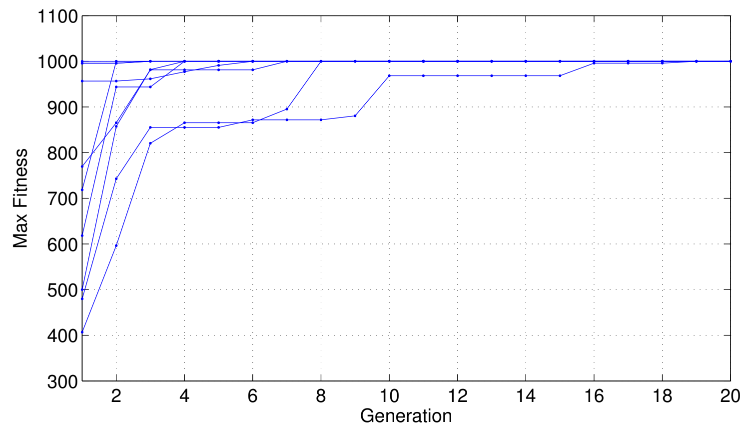


Figure 4-2: Experiment 4-1 Learning Curves

4.3 Experiment 4-2: A Simple Equation + Noise

Setup

The outcome of Experiment 4-1 was promising, however we must address the noise sensitivity question. Here we have repeated Experiment 4-1 with noise as follows:

$$f(x) = \frac{1}{x^2 + 1} + \mathbf{n} \tag{4.19}$$

where is $\mathbf{n} \sim N(\mu, \sigma)$ (i.i.d.) with $\mu = 0$ and $\sigma = 0.1$. We used the same 200 x values from Experiment A as well as the same algorithm parameters (see Table 4.1) aside from the convergence threshold. As we have seen when noise is present, a perfect fitness score is unrealistic, so the convergence threshold τ has been compensated using,

$$\tau = \left\lfloor \frac{K}{1 + \mathbb{E}(\mathbf{n}^2)} \right\rfloor \tag{4.20}$$

where K is the maximum fitness (1000 in this case) and $\mathbb{E}(\mathbf{n}^2)$ is the second moment of the noise.

Results

We did 10 independent runs using the data shown in Figure 4-3 and the algorithm again converged to Equation 4.18 in every case. What came as a surprise was the algorithm converged roughly twice as fast with the addition of noise as shown in Figure 4-4.

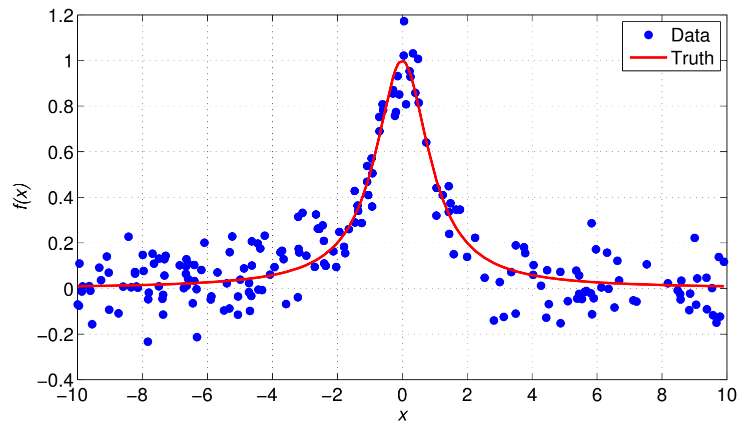


Figure 4-3: Experiment 4-2 Data

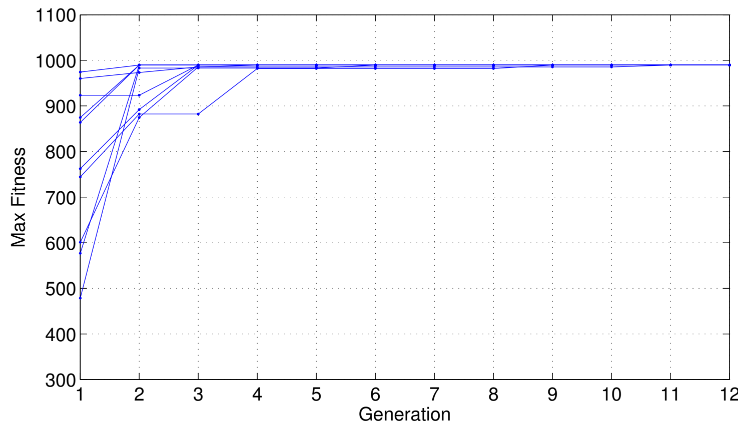


Figure 4-4: Experiment 4-2 Learning Curves

4.4 Experiment 4-3: Unnecessary Constants

Setup

So far we have not used any Random Numerical Constants (RNC) which has substantially reduced the search space. In this experiment we have repeated Experiment 4-2 with the addition of RNCs even though the formula we are trying to find does not have any numerical constants other than the number 1. The algorithm parameters are the same as those shown in Table 4.1 accept Random Numerical Constants have been set to 'On'. In practice it will be unknown if numerical constants exist in the true solution, however typically it is more likely than not.

Results

With the addition of RNCs to the configuration, the GEP algorithm had a significantly more difficult time converging. It found the correct solution 2 out of 10 times, while the others were close approximations. Because we have limited the number of generations to 100, it's very likely the algorithm did not have enough time to converge. This argument is supported by the learning curves shown in Figure 4-5, which did not achieve maximum fitness as fast as Experiments 4-1 and 4-2. With additional of RNCs, there appears to be many more local fitness maximums which are giving the algorithm some trouble finding the correct solution.

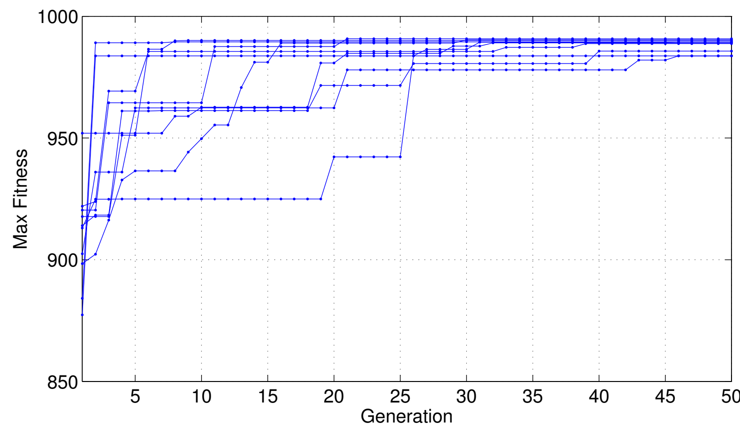


Figure 4-5: Experiment 4-3 Learning Curves

Algorithm Solutions

$$f_1(a) = \frac{1.1}{a^2 + 0.16\,a + 1.1} \tag{4.21}$$

$$f_2(a) = \frac{0.16}{0.16\,a^2 + 0.16} + 0.025 \tag{4.22}$$

$$f_3(a) = \frac{1.8}{1.8\,a^2 + 1.8} \tag{4.23}$$

$$f_4(a) = \frac{1.8}{2.1\,a^2 + 1.9} \tag{4.24}$$

$$f_5(a) = \frac{0.47}{0.22\,a^2 + 0.56} \tag{4.25}$$

$$f_6(a) = \frac{a}{a^3 + 1.1\,a} \tag{4.26}$$

$$f_7(a) = \frac{0.86}{1.1\,a^2 - 0.16\,a + 1.1} + 0.063 \tag{4.27}$$

$$f_8(a) = \frac{0.068}{0.03\,a^2 + 0.093} \tag{4.28}$$

$$f_9(a) = \frac{0.063}{0.063\,a^2 + 0.063} \tag{4.29}$$

$$f_{10}(a) = \frac{1.8}{2.6\,a^2 + 1.8} \tag{4.30}$$

4.5 Experiment 4-4: Necessary Constants

Setup

As we saw in Experiment 4-3, the addition of RNCs to the algorithm impeded convergence. Additionally, it's possible the GEP algorithm had trouble finding the correct solution because there were no numerical constants in the true equation. In this experiment we have repeated Experiment 4-3, but modified the true equation to include a numerical constant:

$$f(x) = \frac{\pi}{x^2 + 1} \tag{4.31}$$

To account for the slower convergence, we have also increased the maximum number of generations to 200.

Results

Upon analyzing the learning curves plotted in Figure 4-6, we can see the convergence behavior is roughly the same as Experiment 4-3, but by allowing an extra 100 generations, the algorithm was able to fine tune the solutions and reach the correct solution (approximately) on every run. The solutions generated on each run were as follows:

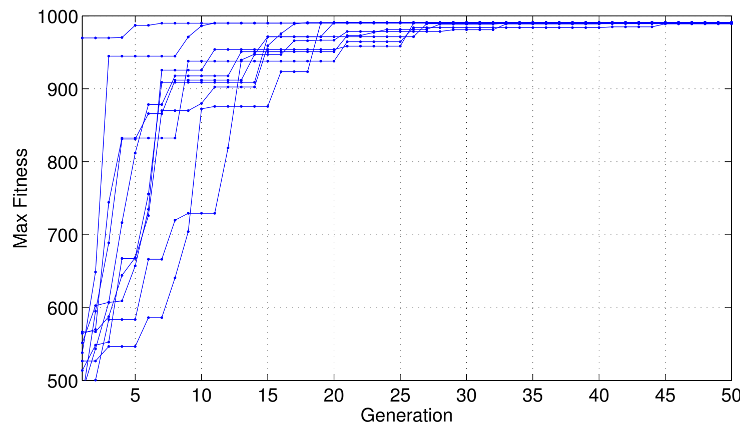


Figure 4-6: Experiment 4-4 Learning Curves

Algorithm Solutions

$$f_1(a) = \frac{1.9}{0.62\,a^2 + 0.59} \tag{4.32}$$

$$f_2(a) = \frac{1.9}{0.69\,a^2 + 0.58} \tag{4.33}$$

$$f_3(a) = \frac{1.9}{0.62\,a^2 + 0.61} \tag{4.34}$$

$$f_4(a) = \frac{3.1}{0.96\,a^2 + 0.97} \tag{4.35}$$

$$f_5(a) = \frac{3.1}{0.96\,a^2 + 0.96} \tag{4.36}$$

$$f_6(a) = \frac{1.9}{0.62\,a^2 + 0.6} \tag{4.37}$$

$$f_7(a) = \frac{3.2}{a^2 + 1.0} \tag{4.38}$$

$$f_8(a) = \frac{1.9}{0.62\,a^2 + 0.59} \tag{4.39}$$

$$f_9(a) = \frac{3.1}{a^2 + 0.97} \tag{4.40}$$

$$f_{10}(a) = \frac{1.8}{0.57\,a^2 + 0.59} \tag{4.41}$$

4.6 Experiment 4-5: Information Removal

Setup

So far we have seen the GEP algorithm find the correct solution to a pair of simple equations under a variety of conditions (i.e. with/without noise, with/without RNCs, etc.). This is promising evidence that it is possible for the GEP algorithm to find the true formula of a simple system explicitly. However, the question of why the Hydrogen Viscosity experiment failed still remains unanswered. In this experiment, we will try to better understand the effects of information removal.

So far we have randomly sampled the independent variable on the interval of $[-10, +10]$, which for Equation 4.18 is essentially the entire region where dependent variable is non-zero. However, in the case of the Hydrogen Viscosity formula we only sampled the independent variable on the interval of $[0, 555]$. To capture the complete non zero region of the dependent variable we would have to sample the entire right side of the real number line $[0, \infty]$. Additionally we have ignored the fact that, for negative temperatures, the viscosity is complex. As we saw, the algorithm was oblivious to this and was able to find multiple relations that fit the data.

Based on what we've seen so far in this chapter, it's reasonable to conclude the reason for the algorithm's success is because there are few solutions, aside from the true solution, that adequately fit the data. In the Hydrogen Viscosity experiment, this was clearly not the case. To test this theory we have rerun Experiment 4-4 eleven times and continuously decreased the independent variable sample interval to determine if there's a correlation with the algorithm's ability to find the correct solution.

Results

The results for this experiment are shown in Table 4.2 and Figures 4-7 through 4-17. In each case, 200 samples for the independent variable were randomly drawn from $U(x_{min}, x_{max})$. The score for each run was determined by counting the number of solutions of the form:

$$f(x) = \frac{c_1}{c_2\, x^2 + c_3} \tag{4.42}$$

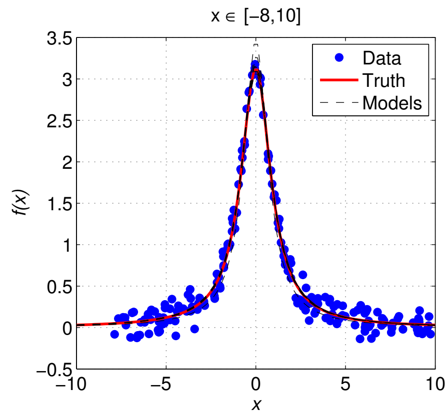
at convergence or the 200th generation, whichever came first. The constants c_k were ignored when comparing solutions. We have assumed that if the solution

run	x_{min}	x_{max}	score	solutions
1	-8	$+10$	10/10	1
2	-6	$+10$	9/10	2
3	-4	$+10$	10/10	1
4	-2	$+10$	8/10	3
5	-1	$+10$	8/10	3
6	0	$+10$	5/10	4
7	$+1$	$+10$	2/10	5
8	$+2$	$+10$	0/10	4
9	$+4$	$+10$	0/10	5
10	$+6$	$+10$	0/10	7
11	$+8$	$+10$	0/10	6

Table 4.2: Experiment 4-5 Results

is of the correct form, the true formula would have benn found eventually. The number of different solution forms are shown in the forms column, again ignoring constants.

These results are not surprising, and confirm that the training data must be properly sampled to find the correct solution. Once we hit the zero crossing in run 6 there was not enough information to find the true solution reliably, which caused the algorithm to become overly creative. This implies there are ”features” in the training data that help lead the algorithm to the correct solution by making it unique from the other solutions in the search space.



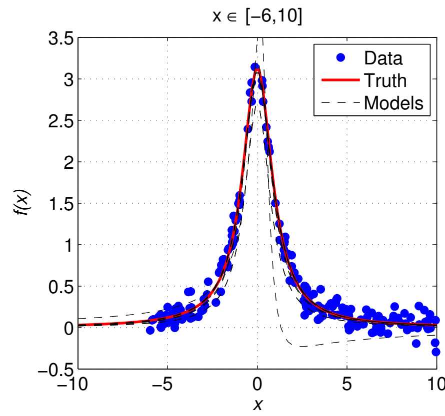
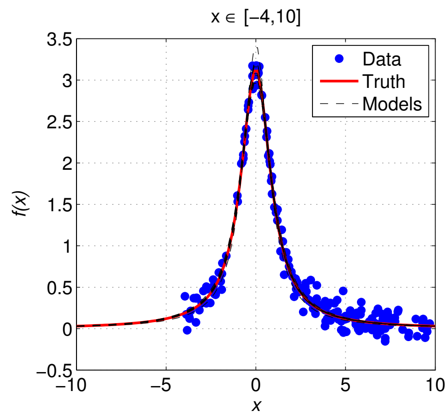


Figure 4-7: Experiment 4-5 Results from run 1 (left)

Figure 4-8: Experiment 4-5 Results from run 2 (right)



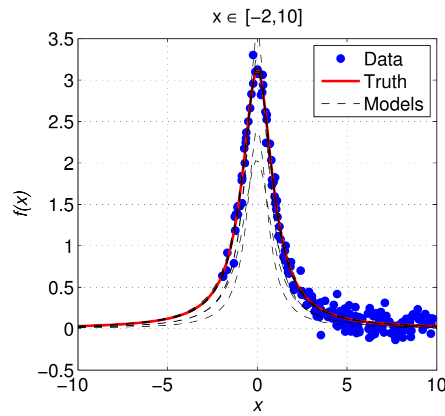
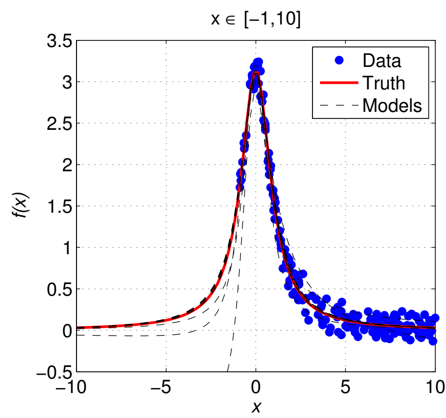


Figure 4-9: Experiment 4-5 Results from run 3 (left)

Figure 4-10: Experiment 4-5 Results from run 4 (right)



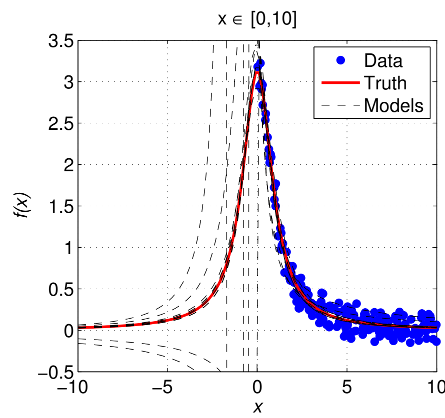
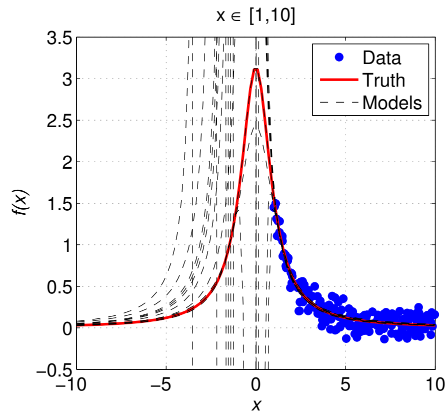


Figure 4-11: Experiment 4-5 Results from run 5 (left)

Figure 4-12: Experiment 4-5 Results from run 6 (right)



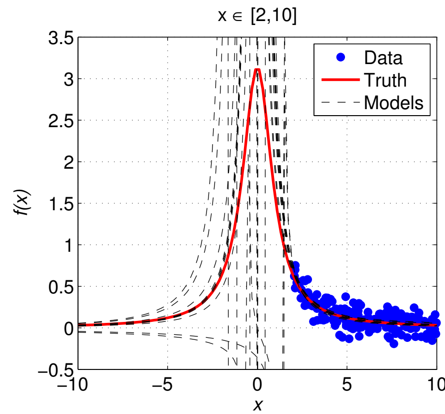


Figure 4-13: Experiment 4-5 Results from run 7 (left)

Figure 4-14: Experiment 4-5 Results from runs 8 (right)

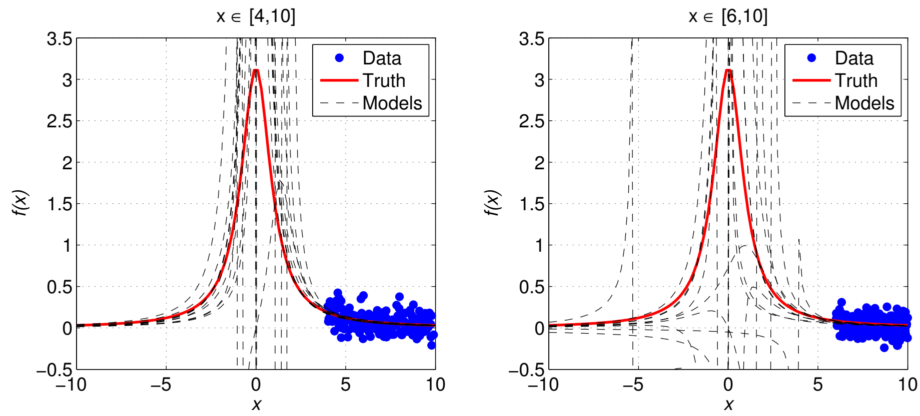


Figure 4-15: Experiment 4-5 Results from runs 9 (left)

Figure 4-16: Experiment 4-5 Results from runs 10 (right)

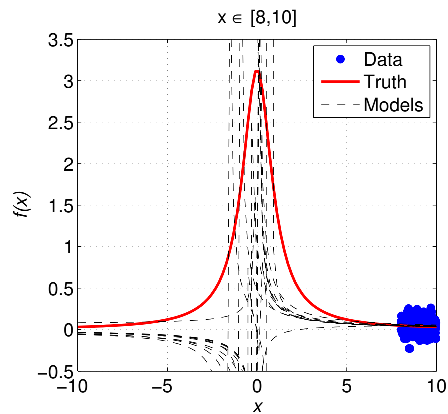


Figure 4-17: Experiment 4-5 Results from run 11

4.7 Experiment 4-6: Simplified Viscosity of Hydrogen

Setup

As we saw in Chapter 3, the convergence behavior of the GEP algorithm in the Hydrogen Viscosity experiment was erratic which suggests our training data did not have the correct "features" to find the true solution reliably. In the previous experiment we saw a similar as result as we began to decrease the size of the independent variable sampling interval. In this experiment we will attempt to solve the Hydrogen Viscosity problem again by sampling on the interval $[-5 \times 10^5, 10 \times 10^6]$ degrees Kelvin and hope the training data has the correct features to uniquely determine the correct solution. Additionally, instead of random sampling we have taken 200 linearly spaced samples with a noise $\sigma = 0.01$ to simplify the problem (see Figure 4-18). Recall, the hydrogen viscosity equation is written as,

$$\mu(T) = \lambda \frac{T^{3/2}}{T + C} \tag{4.43}$$

where λ and C were empirical constants equal to 0.636 and 72 respectively. However finding the constant 72 is likely to be difficult and should be treated separately, so for this experiment we have also let $C = 1$.

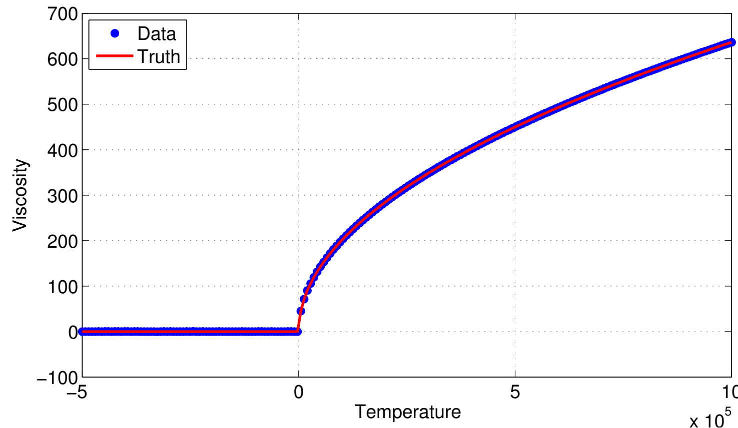


Figure 4-18: Experiment 4-6 Data

Algorithm Parameters	
Terminal Set	$\{a, ?\}$
Function Set	$\{+, -, \times, \div, \Re(\sqrt{\star})\}$
Genes	1
Head Size	10
Tail Size	11
Gene Size	32
Chrm Size	32
Runs	10
Population Size	200
Max Generations	200
Mutation Rate	0.1
1-Pt Recombination Rate	0.3
2-Pt Recombination Rate	0.3
Gene Recombination Rate	0.3
IST Rate	0.1
RIST Rate	0.1
Inversion Rate	0.1
Min Founders	1
Survival Threshold	1
Convergence Threshold	991
Maximum Fitness	1000

Table 4.3: Experiment 4-6 Algorithm Configuration

Note that this is the real part of the training data which is zero for negative temperatures. The algorithm parameters are shown in Table 4.3. We have added the function $\Re(\sqrt{\star})$ to the function set to avoid complex outputs.

Results

Unfortunately the algorithm failed to find the correct solution on every run. The learning curves are shown in Figure 4-19 followed by the algorithm solutions, sorted in order of fitness. The convergence behavior here was not consistent which suggests the training data does have the correct features for the algorithm to succeed. Upon examining the most fit solution, $f_1(a)$ we found that it actually exceeded the convergence threshold, meaning

$$\mathbb{E}(\mathbf{n}^2) \approx \frac{1}{M} \sum_{k=1}^{M} \| f_1(T^{(k)}) - \mu(T^{(k)}) \|^2 \tag{4.44}$$

Since $f_1(a)$ is clearly the wrong solution, we can conclude the algorithm is being fooled by decoy solutions.

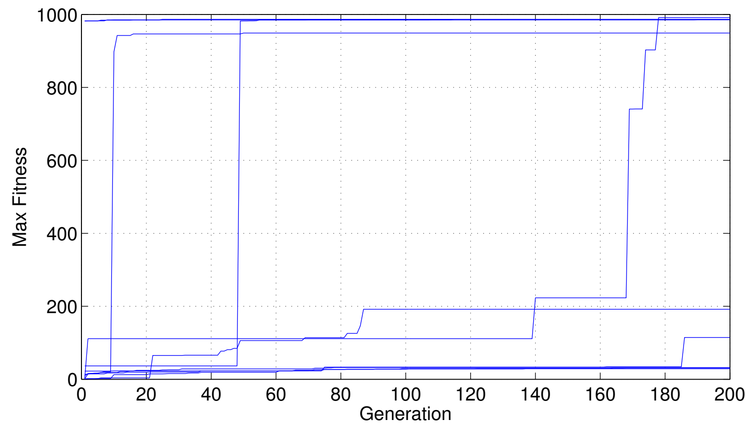


Figure 4-19: Experiment 4-6 Learning Curves

Algorithm Solutions

$$f_1(a) = 0.64\, a\, \Re\left(\sqrt{\frac{1}{a}}\right) \tag{4.45}$$

$$f_2(a) = \Re\big(\sqrt{0.4\, a - 1.7}\big) - 0.049 \tag{4.46}$$

$$f_3(a) = 0.65\, \Re\big(\sqrt{0.95\, a - 2.3}\big) \tag{4.47}$$

$$f_4(a) = \Re\big(\sqrt{0.4\, a - 1.4}\big) - 0.024 \tag{4.48}$$

$$f_5(a) = \Re\big(\sqrt{0.41\, a - 2.8}\big) - 0.27 \tag{4.49}$$

$$f_6(a) = 0.64\, \Re\big(\sqrt{a}\big) - 2.5 \tag{4.50}$$

$$f_7(a) = 0.64\, \Re\big(\sqrt{a}\big) + 0.024 \tag{4.51}$$

$$f_8(a) = 0.65\, \Re\left(\sqrt{a - 1.0\, \Re\big(\sqrt{a}\big) + 0.024}\right) - 6.6 \tag{4.52}$$

$$f_9(a) = 0.65\, \Re\big(\sqrt{a}\big) - 6.8 \tag{4.53}$$

$$f_{10}(a) = 0.65\, \Re\big(\sqrt{a}\big) - 5.3 \tag{4.54}$$

4.8 Experiment 4-7: Imaginary Viscosity of Hydrogen

Setup

In order to differentiate the hydrogen viscosity equation from other solutions in the search space, we have allowed for negative temperatures in this experiment. This is meant to see if the pole on the imaginary axis at $T = C$ will provide enough information for the algorithm to find the correct solution, where again we have taken $C = 1$. This is basically a repeat of Experiment 4-6 with the function set $\{+, -, \times, \div, \sqrt{\star}\}$ and $N = 200$ linearly sampled data points in the interval $[-20, +20]$ used as training data. Additionally the fitness function has been modified to account for the real and imaginary parts of the data using,

$$\Phi_{MS}(f) = K\left(1 + \frac{1}{M}\sum_{k=1}^{M}(\Re(e^{(k)})^2 + \Im(e^{(k)})^2)\right)^{-1} \tag{4.55}$$

where the errors $e^{(k)}$ are the residuals between the training examples and the solution under test f. In this case the noise on the training data was complex ($\sigma = 0.01$) so the convergence threshold τ was set using,

$$\tau = \left\lfloor \frac{K}{1 + \mathbb{E}(\mathbf{n}_R^2) + \mathbb{E}(\mathbf{n}_I^2)} \right\rfloor = 980 \tag{4.56}$$

Results

The resulting solutions are shown in Figures 4-20 through 4-21[1] and the learning curves are shown in Figure 4-22, followed the solutions for each run sorted by fitness. Out of the 10 runs we did, $f_1(a)$, $f_2(a)$, and $f_4(a)$ found the true solution (approximately). On the other hand $f_3(a)$ was not the correct solution form, but was able to score a higher fitness than $f_4(a)$. The pole at $T = 1$ has clearly provided a unique feature the algorithm can use to find the correct solution.

[1] Here we have only shown the data form $[-10, +10]$ to show more detail around the pole on the imaginary axis.

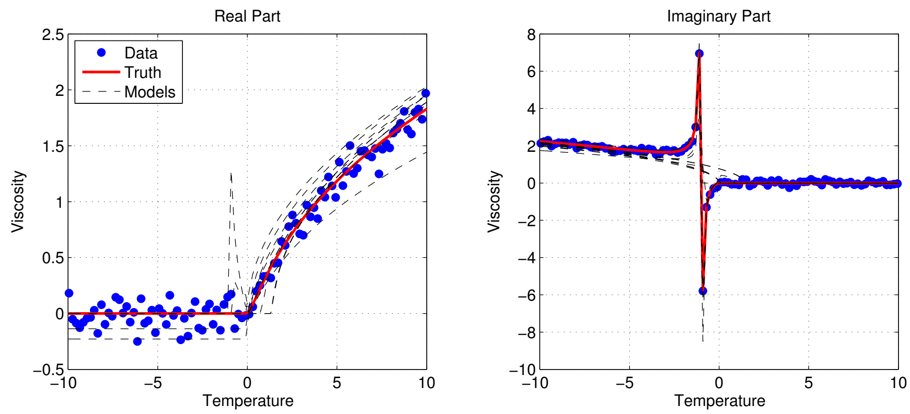


Figure 4-20: Experiment 4-7 Results (real part)

Figure 4-21: Experiment 4-7 Results (imaginary part)

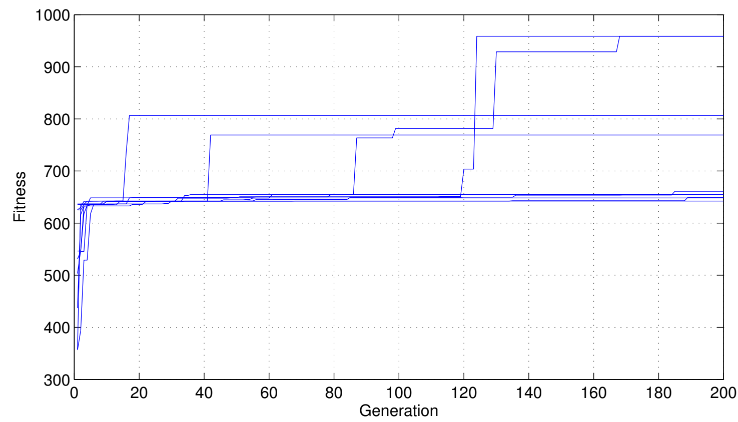


Figure 4-22: Experiment 4-7 Learning Curves

Algorithm Solutions

$$f_1(a) = \frac{\sqrt{a}}{\frac{1.5}{a} + 1.5} \tag{4.57}$$

$$f_2(a) = \frac{\sqrt{a}}{\frac{1.5}{a} + 1.5} \tag{4.58}$$

$$f_3(a) = 0.62\,\sqrt{\frac{a^2}{a + 1.1}} \tag{4.59}$$

$$f_4(a) = \frac{a^{\frac{3}{2}}}{2.0\,a + 1.9} \tag{4.60}$$

$$f_5(a) = \sqrt{0.35\,a - 0.35\left(\frac{a}{a + 0.74}\right)^{\frac{1}{4}}} \tag{4.61}$$

$$f_6(a) = \sqrt{0.42\,a - 0.58} \tag{4.62}$$

$$f_7(a) = \sqrt{0.42\,a - 0.59} \tag{4.63}$$

$$f_8(a) = 0.68\,\sqrt{a} - 0.14 \tag{4.64}$$

$$f_9(a) = 0.045\,\sqrt{244.0\,a} - 0.23 \tag{4.65}$$

$$f_10(a) = 0.059\,\sqrt{122.0\,a} \tag{4.66}$$

4.9 Summary

In these experiments we have discovered that

1. Explicit formulas can be discovered *if* the training examples have descriptive features which are yet to be defined.
2. Noise does not affect convergence when the above condition is satisfied.
3. Numerical constants complicate the search space.

The simple equation we began with obviously had the right features, but it would appear hydrogen viscosity training examples did not. However, by considering negative temperatures and accounting for the imaginary part of the viscosity we were able to improve our probability of success. Unfortunately this is unrealistic in practice. In fact, by considering the imaginary part, this was arguably a different data set.

At this point, the question is: can additional information be extracted from a data set which lacks the correct features? Schmidt used partial derivative ratios for implicit models of the form $f(\mathbf{x}, y) = 0$ as way to detect trivial solutions (i.e. $\mathbf{x} - \mathbf{x} = 0$) in dynamical systems [21, 20, 23] which, in a sense, guided the search down the correct path. Here we have the problem of detecting decoy solutions, which approximate the true solution very closely, but are incorrect. While this seems like this would be a problem with the data set, it may be possible to constrain the search space to penalize decoys.

For all the experiments in this chapter we have been using single gene systems with a head size of 10 (tail size of 11), and function sets with the correct operators. By doing this, we have already constrained the search space. This configuration was not chosen to fit the problem, but it's possible we were lucky for the simple equation we chose. Selecting the optimal algorithm configuration is a common problem in evolutionary computing, which has also been addressed by Schmidt in [22] for the problem of domain alphabet learning.

Chapter 5

Feature Spaces

"How can computers be made to do what needs to be done, without being told exactly how to do it?"

Arther Samuel, 1959 [86]

5.1 Introduction

In an attempt to separate the decoy solutions from the true solution in the symbolic regression problem (assuming one exists), we will next consider mappings to higher dimensional feature spaces. Kernel mappings for regression and pattern recognition are a standard and considered state of the art in the literature [87, 88, 89, 90, 91], however using kernel based fitness functions for evolutionary algorithms is somewhat less common. Hu et. al [92] discusses using Gaussian kernel based fitness functions to recognize native protein sequences in the field of computational biology, however we were unable to find any work in the area of symbolic regression. Most studies are going in the other direction using evolutionary algorithms to *design* kernel functions for support vector machine problems [93, 94, 95, 96]. In this chapter we will first discuss some basic kernel mappings, and then derive a generaized feature extraction process that will allow us recognize true solutions from empirical. The efficacy of this process will be tested in two simple experiments.

5.2 Kernel Functions

5.2.1 The Kernel Trick

The inspiration for mapping a set of training examples from the measurement space to a higher dimensional feature space begins with kernel functions. The general form of a kernel function is the following,

$$k(\mathbf{x}, \mathbf{x}') = \phi(\mathbf{x})^T \phi(\mathbf{x}') \tag{5.1}$$

Kernels functions are often substituted into algorithms involving scalar products, which is commonly referred to as the *kernel trick* [15]. For example, anytime the inner product $\mathbf{x}^T\mathbf{x}$ is computed, a kernel function can be substituted in its place. In fact $\mathbf{x}^T\mathbf{x}$ is itself kernel function for the identity $\phi(\mathbf{x}) = \mathbf{x}$ [15]. Kernels are used in several machine learning algorithms such as support vector machines and Gaussian processes.

We can describe the response of a kernel function to a set of training examples by creating a matrix called the *Gram* matrix,

$$\mathbf{K}_{m,n} = k\left(\mathbf{x}^{(m)}, \mathbf{x}^{(n)}\right) \tag{5.2}$$

With this concept, there is an opportunity for an improved cost function robust to decoys.

5.2.2 Cost Functions Based on the Gram Matrix

In the case where the desired response of the formula we are trying to discover is 1-dimensional, the Gram matrix is nothing more than the outer product,

$$\mathbf{K}_{m,n} = \mathbf{\Phi}_y \mathbf{\Phi}_y^T \tag{5.3}$$

where

$$\mathbf{\Phi}_y = \begin{bmatrix} \phi\left(y^{(1)}\right) & \phi\left(y^{(2)}\right) & \cdots & \phi\left(y^{(M)}\right) \end{bmatrix}^T \tag{5.4}$$

The Gram matrix based on model predictions is then:

$$\hat{\mathbf{K}}_{m,n} = \mathbf{\Phi}_{\hat{y}} \mathbf{\Phi}_{\hat{y}}^T \tag{5.5}$$

where

$$\mathbf{\Phi}_{\hat{y}} = \begin{bmatrix} \phi\left(f\left(\mathbf{x}^{(1)}\right)\right) & \phi\left(f\left(\mathbf{x}^{(2)}\right)\right) & \cdots & \phi\left(f\left(\mathbf{x}^{(M)}\right)\right) \end{bmatrix}^T \tag{5.6}$$

based of these matrices we can measure the similarity of the training examples and the proposed solution using:

$$\Phi(f) = \sum_{m,n} (\mathbf{K}_{m,n} - \hat{\mathbf{K}}_{m,n})^2 \tag{5.7}$$

However, if the mean-square error between the training examples and the candidate function is small, this cost function will still be fooled by decoy solutions. We need to impose a constraint beyond minimizing the error which will penalize decoy solutions. As it turns out, this can be achieved by taking the curvature of the training examples into account.

5.3 Feature Signatures

5.3.1 Derivative Approximations

Using partial derivatives, Schmidt has shown that implicit physical laws can be mined from dynamical systems [23]. This is a slightly different problem,

however a similar approach may work in our case. Currently we are dealing with 1-dimensional data sets which do not have partial derivatives. However, what if we could compare the *curvature* of the entire candidate function to the training examples simultaneously? This imposes a far more strict constraint than simply minimizing the error and may be the basis for a unique formula signature. To do this, we first define:

$$\dot{y}^{(k)} = \frac{dy^{(k)}}{dx^{(k)}} = \frac{y^{(k+1)} - y^{(k)}}{x^{(k+1)} - x^{(k)}} \tag{5.8}$$

which is the approximate derivative of the dependent variable with respect to the independent variable. Before we proceed we will make the following assumptions about the training examples, (1) the independent variable is uniformly sampled, and (2) the dependent variable is smooth. This simplifies the above approximation to:

$$\dot{y}^{(k)} = \frac{dy^{(k)}}{dx^{(k)}} = \frac{y^{(k+1)} - y^{(k)}}{\Delta x} \tag{5.9}$$

For notational simplicity, the set of $M-1$ numerical training derivatives will be denoted by the vector $\dot{\mathbf{y}}$.

5.3.2 Mapping to Higher Dimensions

To completely characterize the curvature of the ordered, uniformly sampled training examples, we next cancel out the Δx term in every sample by computing the following *signature* matrix:

$$\mathbf{S} = \dot{\mathbf{y}}(\dot{\mathbf{y}}^T)^{-1} = \begin{pmatrix} \frac{\dot{y}_1}{\dot{y}_1} & \frac{\dot{y}_1}{\dot{y}_2} & \cdots & \frac{\dot{y}_1}{\dot{y}_{M-1}} \\ \frac{\dot{y}_2}{\dot{y}_1} & \frac{\dot{y}_2}{\dot{y}_2} & \cdots & \frac{\dot{y}_2}{\dot{y}_{M-1}} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\dot{y}_M}{\dot{y}_1} & \frac{\dot{y}_M}{\dot{y}_2} & \cdots & \frac{\dot{y}_{M-1}}{\dot{y}_{M-1}} \end{pmatrix} \tag{5.10}$$

where,

$$\left(\dot{y}^{(k)}\right)^{-1} = 1/\dot{y}^{(k)} \tag{5.11}$$

This effectively applies a mapping from $\mathbb{R} \mapsto \mathbb{R}^{M-1}$ for each training example. The matrix $\hat{\mathbf{S}}$ can be computed using the predictions from candidate solutions

in the same fashion and the average distance from $\mathbf{S}$ to $\hat{\mathbf{S}}$ can be computed using:

$$\Phi_{FS}(f) = \frac{K}{1 + \mathbb{E}(\mathbf{d})} \tag{5.12}$$

where K is again the maximum fitness, and

$$\mathbb{E}(\mathbf{d}) = \left(\frac{1}{N} \sum_{n=1}^{N} \sum_{m=1}^{M} (\mathbf{S}_{m,n} - \hat{\mathbf{S}}_{m,n})^2 \right)^{1/2} \tag{5.13}$$

5.3.3 Invariance

Applying this mapping has following desirable properties, (1) invariance to the scale of the training examples, (2) invariance to linear tranformations on candidate solutions. The first property is obvious because the $\mathbf{S}$ matrix is based on ratios of differences. However the second property may not be so obvious, and if so it may not be clear why this property is desirable. To prove linear transformation invariance, consider any element of the signature matrix based on an arbitrary candidate solution f:

$$\hat{\mathbf{S}}^{(f)}_{m,n} = \frac{f\left(\mathbf{x}^{(m+1)}\right) - f\left(\mathbf{x}^{(m)}\right)}{f\left(\mathbf{x}^{(n+1)}\right) - f\left(\mathbf{x}^{(n)}\right)} \tag{5.14}$$

Define the function g which is a linear transform on the function f:

$$g(\mathbf{x}) = af(\mathbf{x}) + b \tag{5.15}$$

where coefficients a and b are arbitrary. If we compute the signature matrix based on g,

$$\hat{\mathbf{S}}^{(g)}_{m,n} = \frac{g\left(\mathbf{x}^{(m+1)}\right) - g\left(\mathbf{x}^{(m)}\right)}{g\left(\mathbf{x}^{(n+1)}\right) - g\left(\mathbf{x}^{(n)}\right)} \tag{5.16}$$

it's obvious that,

$$\hat{\mathbf{S}}^{(g)}_{m,n} = \frac{af\left(\mathbf{x}^{(m+1)}\right) + b - af\left(\mathbf{x}^{(m)}\right) - b}{af\left(\mathbf{x}^{(n+1)}\right) + b - af\left(\mathbf{x}^{(n)}\right) - b} \tag{5.17}$$

which is the same as,

$$\frac{f\left(\mathbf{x}^{(m+1)}\right)-f\left(\mathbf{x}^{(m)}\right)}{f\left(\mathbf{x}^{(n+1)}\right)-f\left(\mathbf{x}^{(n)}\right)}=\hat{\mathbf{S}}_{m,n}^{(f)} \tag{5.18}$$

5.3.4 Post Optimization

Now that we have shown invariance to linear transforms, the question to ask is: why would we want this? The consequence of this property is that any candidate function f will have the same fitness score as any linear transformation of f. Because of this, any solution found using the signature fitness criterion $\Phi_{FS}(f)$ must be assumed to have unknown coefficients which we will refer to as c_1 and c_2. The benefit of this property is the problem of finding the true data generating solution has been divided into 2 simpler problems, (1) find any formula f that maximizes $\Phi_{FS}(f)$ using the GEP algorithm and (2) solve for c_1 and c_2 by minimizing the error between $\mathbf{y}$ and $c_1 f(\mathbf{x}) + c_2$ using the preferential optimization method. Next we will investigate the efficacy of this approach through 2 simple experiments.

5.4 Experiment 5-1: Penalization Analysis

Setup

To understand the sensitivity of the signaure fitness criterion, we have rerun the hydrogen viscosity experiment 10 times with the mean square error fitness function and recomputed the fitness of the final answers with $\Phi_{FS}(f)$. This will tell us if decoy solutions with high $\Phi_{MS}(f)$ scores are penalized by $\Phi_{FS}(f)$. For this experiment the algorithm has been run according to the configuration in Table 5.1.

Algorithm Parameters	
Terminal Set	$\{a, ?\}$
Function Set	$\{+, -, \times, \div, \sqrt{\star}\}$
Genes	1
Head Size	10
Tail Size	11
Gene Size	32
Chrm Size	32
Runs	10
Population Size	200
Max Generations	200
Mutation Rate	0.1
1-Pt Recombination Rate	0.3
2-Pt Recombination Rate	0.3
Gene Recombination Rate	0.3
IST Rate	0.1
RIST Rate	0.1
Inversion Rate	0.1
Min Founders	1
Survival Threshold	1
Convergence Threshold	999
Maximum Fitness	1000
Random Numerical Constants	On
Constants Range	[0,100]

Table 5.1: Experiment 5-1 Algorithm Configuration

In this experiment we have used the complete hydrogen viscosity equation, repeated here for convenience:

$$\mu(T) = 0.663 \frac{T^{3/2}}{T + 72} \tag{5.19}$$

and have asked the algorithm the find this equation from 200 linearly spaced samples from 1 to 10,000 degrees Kelvin (no noise).

Results

As expected, the algorithm was unable to find the correct solution and instead returned the following:

$$f_1(a) = 0.64\sqrt{a} - 0.59 \tag{5.20}$$

$$f_2(a) = 0.63\sqrt{a} \tag{5.21}$$

$$f_3(a) = 0.63\sqrt{a} \tag{5.22}$$

$$f_4(a) = 0.63\sqrt{a} \tag{5.23}$$

$$f_5(a) = 0.63\sqrt{a} \tag{5.24}$$

$$f_6(a) = 0.63\sqrt{a} \tag{5.25}$$

$$f_7(a) = 0.17\sqrt{(1.1 \cdot 10^4)\left(\sqrt{211.0 - 1.0\,a} + 83.0\right)^{\frac{1}{4}}} \tag{5.26}$$

$$f_8(a) = -122.0 \tag{5.27}$$

$$f_9(a) = \sqrt{34.0\sqrt{a + 133.0} - 122.0} \tag{5.28}$$

$$f_{10}(a) = \frac{a}{\sqrt{a} + 144.0} + 1.0 \tag{5.29}$$

Solutions 1-6 are actually good approximations when $T \gg 72$ which may be due to the large temperatures we included in the training data. The question is now, given these solutions and fitness scores, how would $\Phi_{FS}(f)$ score these solutions? The answer is plotted in Figure 5-1, where it's clear that $\Phi_{FS}(f)$ had a very different interpretation of the solutions found by $\Phi_{MS}(f)$. An interesting result is the solution with lowest $\Phi_{MS}(f)$ score had the *highest* $\Phi_{FS}(f)$ score (i.e. $f_{10}(a)$). Upon closer inspection of $f_{10}(a)$, it's the only solution with a ratio and is the closest in its form to the true solution.

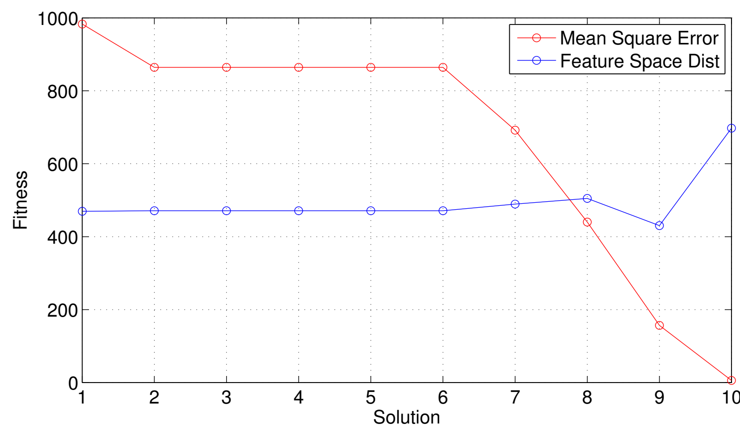


Figure 5-1: Experiment 5-1 Fitness Scores

5.5 Experiment 5-2: Signature Learning

Setup

We have repeated the hydrogen viscosity experiment yet again using the configuration from the previous section using $\Phi_{FS}(f)$.

Results

The algorithm found the correct form of the solution 2 out of 10 runs as shown in the boxes below (sorted in order of fitness):

$$\boxed{f_1(a) = c_1 \frac{a^{\frac{3}{2}}}{a + 72.1} + c_2} \tag{5.30}$$

$$\boxed{f_2(a) = c_1 \frac{a^{\frac{3}{2}}}{a + 71.77} + c_2} \tag{5.31}$$

$$f_3(a) = c_1 \frac{a}{\sqrt{a + 150.6}} + c_2 \tag{5.32}$$

$$f_4(a) = c_1 \frac{a}{\sqrt{a + 150.6}} + c_2 \tag{5.33}$$

$$f_5(a) = c_1 \frac{a}{\sqrt{10.44\, a + 1567.0}} + c_2 \tag{5.34}$$

$$f_6(a) = c_1 \frac{a}{\sqrt{a + 149.6}} + c_2 \tag{5.35}$$

$$f_7(a) = c_1 \frac{a}{\sqrt{2.0\, a + 298.2}} + c_2 \tag{5.36}$$

$$f_8(a) = c_1 \frac{a}{\sqrt{a + 148.6}} + c_2 \tag{5.37}$$

$$f_9(a) = c_1 \frac{a}{\sqrt{a + 147.6}} + c_2 \tag{5.38}$$

$$f_{10}(a) = c_1 \frac{a \sqrt{2.0\, a + 9.44}}{a + 83.01} + c_2 \tag{5.39}$$

5.6 Summary

In this chapter we have introduced a new fitness function which was able to deliver the correct solution form of the hydrogen viscosity formula using the GEP algorithm. This fitness function works by mapping the training examples to a higher dimensional space which can be interpreted as creating a unique signature. From here we will take a step back from the hydrogen viscosity problem and start to examine other formulae to (1) determine if this result was reached by chance and (2) find where this new approach breaks down.

Chapter 6

The Analytic World vs. the Real World

"An algorithm must be seen to be believed"

Donald Knuth, 2011 [97]

6.1 Introduction

In this chapter we have conducted 2 sets of experiments: (1) Analytic Experiments and (2) Real World Experiments. The first group covers the common analytic function types namely: polynomial functions, ration functions, trigonometric functions, logarithmic functions, and exponential functions. In each case we have selected a simple example from each function type which we have asked the GEP algorithm to discover it using $\Phi_{MS}(f)$ and $\Phi_{FS}(f)$. The second group is from a set of experiments pulled from the UCI machine learning repository in the areas of physics and chemistry for which the true data generating function is assumed to be unknown. For each data set we pulled from the UCI repository there was a reference function provided however these are based on regression models that may not necessarily be true. Therefore we have chosen to ignore these solutions and score each solution based on the error due to predictions on hold-out data points.

6.2 Analytic Experiments

Here we have done a total of 5 experiments where we have asked the GEP algorithm to discover several common analytic function types. These functions were as follows:

$$\textbf{Polynomial} : f(x) = x^3 - 4x \tag{6.1}$$

$$\textbf{Rational} : f(x) = \frac{0.5x^3 - 5}{x^2 + 1} \tag{6.2}$$

$$\textbf{Trigonometric} : f(x) = 98.6 \sin(x) \tag{6.3}$$

$$\textbf{Logarithmic} : f(x) = \ln(7846(1 + x)) \tag{6.4}$$

$$\textbf{Exponential} : f(x) = \exp(-2(x - 1)) \tag{6.5}$$

In each case we have only given the algorithm access to a limited domain of the function's data thus creating an ambiguity. Each experiment was run with the algorithm configuration shown in Table 6.1. Parameters that were varied

based on experiment are labeled as such and will be given in the relevant sections.

Algorithm Parameters			
Terminal Set	$\{a, ?\}$	2-Pt Recombination Rate	0.3
Function Set	Varies	Gene Recombination Rate	0.3
Genes	1	IST Rate	0.1
Head Size	10	RIST Rate	0.1
Tail Size	11	Inversion Rate	0.1
Gene Size	32	Survival Threshold	1
Chrm Size	32	Convergence Threshold	999.9
Runs	10	Maximum Fitness	1000
Population Size	200	Training Samples	50
Mutation Rate	0.1	Max Generations	250
1-Pt Recombination Rate	0.3	Min Founders	1

Table 6.1: Analytic Experiments Configuration

6.2.1 Experiment 6-1: Polynomial

Setup

In this experiment we begin with the simple polynomial:

$$f(x) = x^3 - 4x, x \in [-1, +3] \tag{6.6}$$

which should be very easy for both fitness criterions to find the correct answer. A plot of the training examples for this experiment is shown in Figure 6-1. The function set is the following:

$$F = \{+, -, \times, \div\} \tag{6.7}$$

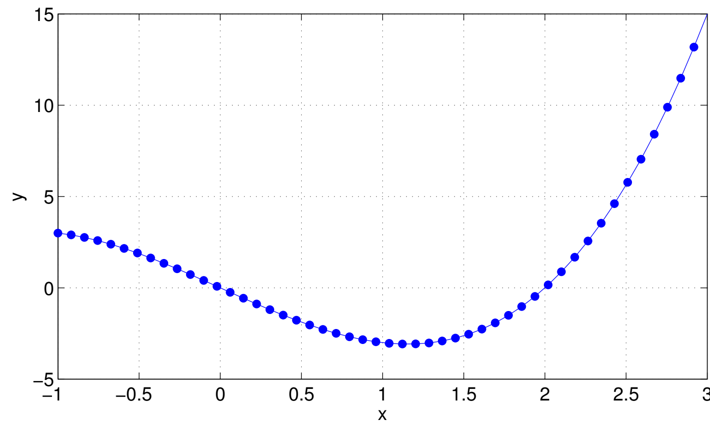


Figure 6-1: Experiment 6-1 Training Examples

Results

The resulting fitness scores for each of the 10 runs we did in this experiment are shown below in Table 6.2. Both $\Phi_{MS}(f)$ and $\Phi_{FS}(f)$ were able to find the correct solution at least once although it appears $\Phi_{FS}(f)$ struggled slightly in that it found the correct answer only 1 out of 10 runs. However, despite this result $\Phi_{FS}(f)$ gave very low scores to the incorrect solutions, whereas $\Phi_{MS}(f)$ gave far more optimistic scores. This result is consistent with our expectations that $\Phi_{FS}(f)$ can recognize truth, however the low probability of success implies more trials or extended convergence time may be necessary to find the correct solution. The solution for each run is given in the following two sections in order of fitness score.

Run	$\Phi_{MS}(f)$	$\Phi_{FS}(f)$
1	1000	1000
2	1000	11
3	372	12
4	429	8
5	372	10
6	1000	45
7	1000	16
8	372	13
9	372	11
10	141	17

Table 6.2: Experiment 6-1 Scores

$\Phi_{MS}(f)$ **Outcomes:**

$$\Phi_{MS}\left(a^3 - 4.0\,a\right) = 1000 \tag{6.8}$$

$$\Phi_{MS}\left(a^3 - 4.0\,a\right) = 1000 \tag{6.9}$$

$$\Phi_{MS}\left(a^3 - 4.0\,a\right) = 1000 \tag{6.10}$$

$$\Phi_{MS}\left(a^3 - 4.0\,a\right) = 1000 \tag{6.11}$$

$$\Phi_{MS}\left(a^3 - 0.24\,a^2 - 2.8\,a\right) = 429.2086 \tag{6.12}$$

$$\Phi_{MS}\left(a^3 - 1.0\,a^2 - 2.0\,a\right) = 371.8584 \tag{6.13}$$

$$\Phi_{MS}\left(a^3 - 1.0\,a^2 - 2.0\,a\right) = 371.8584 \tag{6.14}$$

$$\Phi_{MS}\left(a^3 - 1.0\,a^2 - 2.0\,a\right) = 371.8584 \tag{6.15}$$

$$\Phi_{MS}\left(a^3 - 1.0\,a^2 - 2.0\,a\right) = 371.8584 \tag{6.16}$$

$$\Phi_{MS}\left(0.079\,a^5\right) = 140.5618 \tag{6.17}$$

$\Phi_{FS}(f)$ **Outcomes:**

$$\Phi_{FS}\left(c_1\left(4.0\,a - 1.0\,a^3\right) + c_2\right) = 1000 \tag{6.18}$$

$$\Phi_{FS}\left(c_1\left(a^3 + a^2 - 6.25a\right) + c_2\right) = 44.8651 \tag{6.19}$$

$$\Phi_{FS}\left(c_1\left(-13.0\,a^3 + a^2 + 48.0\,a\right) + c_2\right) = 16.589 \tag{6.20}$$

$$\Phi_{FS}\left(c_1\left(144.0\,a^3 - 71.0\,a^2 - 400.0\,a\right) + c_2\right) = 16.0698 \tag{6.21}$$

$$\Phi_{FS}\left(c_1\left(-0.349\,a^3 + 0.182a^2 + a\right) + c_2\right) = 13.4581 \tag{6.22}$$

$$\Phi_{FS}\left(c_1\left(a^3 + 20.6a^2 - 52.4a\right) + c_2\right) = 12.2249 \tag{6.23}$$

$$\Phi_{FS}\left(c_1\left(0.26\,a^3 - 1.0\,a\right) + c_2\right) = 11.3396 \tag{6.24}$$

$$\Phi_{FS}\left(c_1\left(-74.0\,a^2 + 177.0\,a\right) + c_2\right) = 10.7279 \tag{6.25}$$

$$\Phi_{FS}\left(c_1\left(a^3 + 23.0\,a^2 - 57.0\,a\right) + c_2\right) = 9.614 \tag{6.26}$$

$$\Phi_{FS}\left(c_1\left(2.0\,a - \frac{86.0\,a^2}{a + 94.0}\right) + c_2\right) = 8.1504 \tag{6.27}$$

6.2.2 Experiment 6-2: Rational

Setup

In this experiment we generated samples from the following rational function:

$$f(x) = \frac{0.5x^3 - 5}{x^2 + 1}, x \in [-5, 0] \tag{6.28}$$

A plot of the training examples for this experiment is shown in Figure 6-2. The function set we used for this experiment was as follows:

$$F = \{+, -, \times, \div\} \tag{6.29}$$

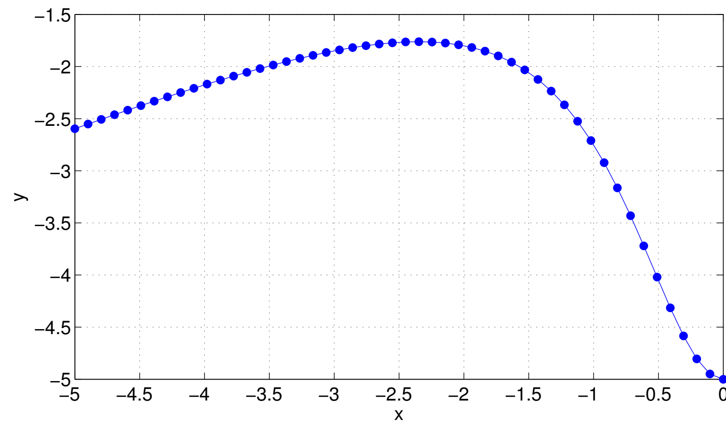


Figure 6-2: Experiment 6-2 Training Examples

Results

In this particular experiment neither $\Phi_{MS}(f)$ nor $\Phi_{FS}(f)$ was able to guide the algorithm to the correct solution. Further experimentation with more convergence time and/or different algorithm configurations is necessary in this case. The solutions for each run is given in the following two sections in order of fitness score. From a visual inspection of the returned solutions, they are all lacking the complexity of the target formula which suggests the target formula was not in the search space of the algorithm.

Run	$\Phi_{MS}(f)$	$\Phi_{FS}(f)$
1	735	19
2	735	14
3	735	20
4	654	20
5	733	20
6	749	21
7	712	20
8	780	16
9	748	21
10	595	21

Table 6.3: Experiment 6-2 Scores

$\Phi_{MS}(f)$ **Outcomes:**

$$\Phi_{MS}\left(\frac{a}{\frac{96.0}{\frac{29.0}{a}-7.2}-7.2}+\frac{30.0}{\frac{96.0}{\frac{29.0}{a}-7.2}-7.2}\right)=780.171 \tag{6.30}$$

$$\Phi_{MS}\left(\frac{30.0}{3.3\,a-6.1}\right)=748.966 \tag{6.31}$$

$$\Phi_{MS}\left(\frac{30.0}{2.6\,a-7.2}\right)=747.6229 \tag{6.32}$$

$$\Phi_{MS}\left(\frac{29.0}{2.0\,a-7.2}\right)=734.5204 \tag{6.33}$$

$$\Phi_{MS}\left(\frac{29.0}{2.0\,a-7.2}\right)=734.5204 \tag{6.34}$$

$$\Phi_{MS}\left(\frac{29.0}{2.0\,a-7.2}\right)=734.5204 \tag{6.35}$$

$$\Phi_{MS}\left(\frac{29.0}{2.0\,a-7.2}\right)=733.1892 \tag{6.36}$$

$$\Phi_{MS}\left(\frac{7.2}{a-1.5}\right)=712.4692 \tag{6.37}$$

$$\Phi_{MS}\left(-\left(4.8\cdot 10^{-3}\right)\,a^2-0.37\,a-3.3\right)=654.3691 \tag{6.38}$$

$$\Phi_{MS}\left(0.034\,a^2-0.034\,a-2.9\right)=594.7123 \tag{6.39}$$

$\Phi_{FS}(f)$ **Outcomes:**

$$\Phi_{FS}\left(c_1\left(29.0\,a^2+144.0\,a\right)+c_2\right)=20.5801 \tag{6.40}$$

$$\Phi_{FS}\left(c_1\left(3.0\,a^2+14.0\,a\right)+c_2\right)=20.5253 \tag{6.41}$$

$$\Phi_{FS}\left(c_1\left(3.0\,a^2+14.0\,a\right)+c_2\right)=20.5253 \tag{6.42}$$

$$\Phi_{FS}\left(c_1\left(50.0\,a^2+244.0\,a\right)+c_2\right)=20.486 \tag{6.43}$$

$$\Phi_{FS}\left(c_1\left(0.96\,a^2+4.5\,a\right)+c_2\right)=20.2835 \tag{6.44}$$

$$\Phi_{FS}\left(c_1\left(14.0\,a^2+68.0\,a\right)+c_2\right)=20.0574 \tag{6.45}$$

$$\Phi_{FS}\left(c_1\left(45.0\,a^2+211.0\,a\right)+c_2\right)=19.6612 \tag{6.46}$$

$$\Phi_{FS}\left(c_1\left(46.0\,a^2+222.0\,a\right)+c_2\right)=19.3367 \tag{6.47}$$

$$\Phi_{FS}\left(c_1\left(a^3-a^2-\left(1.9\cdot 10^2\right)\,a\right)+c_2\right)=15.9873 \tag{6.48}$$

$$\Phi_{FS}\left(32.0c_1+c_2\right)=14.2168 \tag{6.49}$$

6.2.3 Experiment 6-3: Trigonometric

Setup

In this experiment we have used a simple sine function with a non-trivial amplitude:

$$f(x) = 98.6 \sin(x), x \in \left[0, \frac{3\pi}{2}\right] \tag{6.50}$$

This will test to see if either $\Phi_{MS}(f)$ or $\Phi_{FS}(f)$ over think the solution to this problem even when the sine and cosine operators are provided in the function set. A plot of the training examples for this experiment is shown in Figure 6-3. The function set we used for this experiment was as follows:

$$F = \{+, -, \times, \div, \sin(\star), \cos(\star)\} \tag{6.51}$$

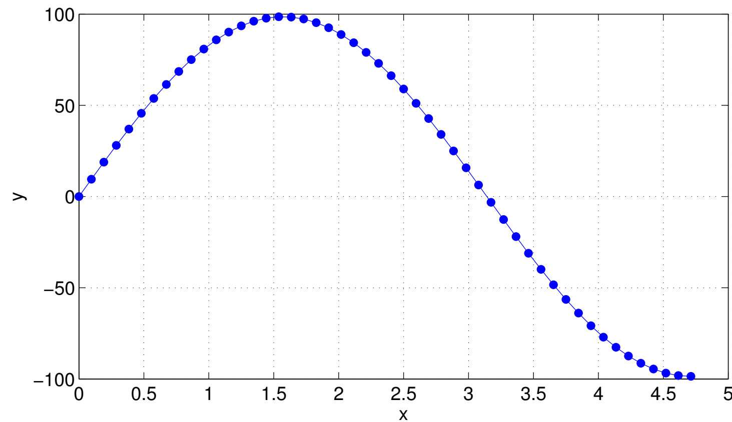


Figure 6-3: Experiment 6-3 Training Examples

Results

Here $\Phi_{MS}(f)$ struggled to find the correct solution, while $\Phi_{FS}(f)$ succeeded on every run. In the case of $\Phi_{MS}(f)$, the top 3 solutions are all very similar yet the fitness score varies considerably. This goes to show that small differences in the amplitude have a large effect of the fitness score. On the other hand, the $\Phi_{FS}(f)$ was unaffected because of its invariance to linear transformations. As a result the correct form of the solution was found on every run.

Run	$\Phi_{MS}(f)$	$\Phi_{FS}(f)$
1	1000	1000
2	243	1000
3	363	1000
4	721	1000
5	507	1000
6	919	1000
7	293	1000
8	41	1000
9	257	1000
10	495	1000

Table 6.4: Experiment 6-3 Scores

$\Phi_{MS}(f)$ **Outcomes:**

$$\Phi_{MS}\left(98.601\,\sin(a)\right) = 999.9996 \tag{6.52}$$

$$\Phi_{MS}\left(99.019\,\sin(a)\right) = 919.3031 \tag{6.53}$$

$$\Phi_{MS}\left(99.48\,\sin(a)\right) = 720.6495 \tag{6.54}$$

$$\Phi_{MS}\left(99.0\,\sin(a) - 1.5\,\cos(a) - 0.62\right) = 507.1791 \tag{6.55}$$

$$\Phi_{MS}\left(\sin(a)^2 + 99.0\,\sin(a) + \cos(a)\right) = 494.6506 \tag{6.56}$$

$$\Phi_{MS}\left(92.0\,\sin(a) + 2.0\,a\,\sin(a) + 1.8\right) = 363.39 \tag{6.57}$$

$$\Phi_{MS}\left(92.0\,\sin(a) + 2.0\,a\,\sin(a) + 1.0\right) = 292.9888 \tag{6.58}$$

$$\Phi_{MS}\left(92.0\,\sin(a) + \sin(a)^2 + 2.0\,a\,\sin(a)\right) = 257.3214 \tag{6.59}$$

$$\Phi_{MS}\left(94.0\,\sin(a) + a\,\sin(a)\right) = 243.2872 \tag{6.60}$$

$$\Phi_{MS}\left(33.0\,\cos(0.4\,\cos(0.02\,a)\,\cos(0.02))\right) = 40.6812 \tag{6.61}$$

$\Phi_{FS}(f)$ **Outcomes:**

$$\Phi_{FS}\left(c_1 \sin(a) + c_2\right) = 1000 \tag{6.62}$$

$$\Phi_{FS}\left(c_1 \sin(a) + c_2\right) = 1000 \tag{6.63}$$

$$\Phi_{FS}\left(c_1 \sin(a) + c_2\right) = 1000 \tag{6.64}$$

$$\Phi_{FS}\left(c_1 \sin(a) + c_2\right) = 1000 \tag{6.65}$$

$$\Phi_{FS}\left(c_1 \sin(a) + c_2\right) = 1000 \tag{6.66}$$

$$\Phi_{FS}\left(c_1 \sin(a) + c_2\right) = 1000 \tag{6.67}$$

$$\Phi_{FS}\left(c_1 \sin(a) + c_2\right) = 1000 \tag{6.68}$$

$$\Phi_{FS}\left(c_1 \sin(a) + c_2\right) = 1000 \tag{6.69}$$

$$\Phi_{FS}\left(c_1 \sin(a) + c_2\right) = 1000 \tag{6.70}$$

$$\Phi_{FS}\left(c_1 \sin(a) + c_2\right) = 1000 \tag{6.71}$$

6.2.4 Experiment 6-4: Logarithmic

Setup

In this experiment have used the following logarithmic function with a non-trivial shift in amplitude:

$$f(x) = \ln(7846(1 + x)), x \in [-0.8, +1] \tag{6.72}$$

A plot of the training examples for this experiment is shown in Figure 6-4. The function set we used for this experiment was as follows:

$$F = \{+, -, \times, \div, \ln(\star)\} \tag{6.73}$$

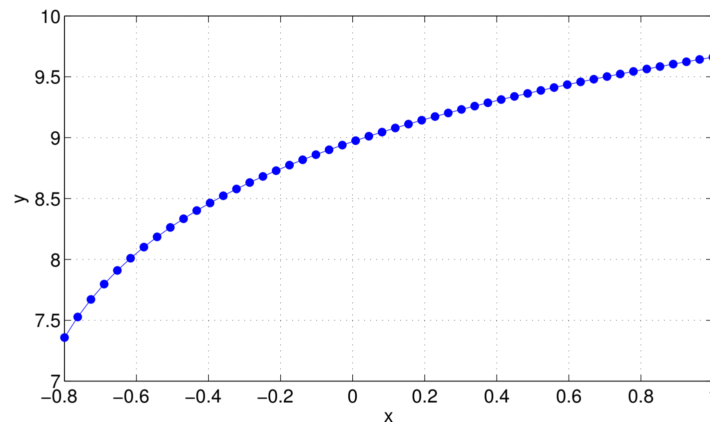


Figure 6-4: Experiment 6-4 Training Examples

Results

The scores for this experiment are shown below in Table 6.5. For the first time in this set of experiments, both $\Phi_{MS}(f)$ and $\Phi_{FS}(f)$ scored highly. In examining the the solutions found by $\Phi_{MS}(f)$, the 1st and 2nd place runs both found the correct answer, however the remaining solutions scored almost as well yet were incorrect. However, in nearly every case the solutions found by $\Phi_{FS}(f)$ delivered the correct form of the solution.

Run	$\Phi_{MS}(f)$	$\Phi_{FS}(f)$
1	993	1000
2	972	986
3	972	1000
4	997	1000
5	971	1000
6	970	1000
7	968	784
8	921	1000
9	999	1000
10	972	1000

Table 6.5: Experiment 6-4 Scores

$\Phi_{MS}(f)$ **Outcomes:**

$$\Phi_{MS}\left(\log(a + 1.0) + 8.9\right) = 998.6848 \tag{6.74}$$

$$\Phi_{MS}\left(\log\left(\left(7.4 \cdot 10^{3}\right)\ a + 7.4 \cdot 10^{3}\right)\right) = 996.7127 \tag{6.75}$$

$$\Phi_{MS}\left(\log\left(82.0\, a^{2} + \left(6.8 \cdot 10^{3}\right)\ a + 7.3 \cdot 10^{3}\right)\right) = 992.556 \tag{6.76}$$

$$\Phi_{MS}\left(1.1\, a + 8.8\right) = 972.2012 \tag{6.77}$$

$$\Phi_{MS}\left(a + \log(a + 6.0) + 7.0\right) = 972.0812 \tag{6.78}$$

$$\Phi_{MS}\left(a + \log\left(3.0\, a^{2} + 333.0\, a + 6.7 \cdot 10^{3}\right)\right) = 971.5509 \tag{6.79}$$

$$\Phi_{MS}\left(1.1\, a + 8.8\right) = 971.4592 \tag{6.80}$$

$$\Phi_{MS}\left(a + \log\left(\frac{\left(3.8 \cdot 10^{18}\right)\ a + 2.5 \cdot 10^{15}}{a}\right) - 34.0\right) = 969.8767 \tag{6.81}$$

$$\Phi_{MS}\left(a + 8.8\right) = 968.2035 \tag{6.82}$$

$$\Phi_{MS}\left(a + 9.0\right) = 921.2855 \tag{6.83}$$

$\Phi_{FS}(f)$ **Outcomes:**

$$\Phi_{FS}\left(c_1 \log(c_2\left(a + 1.0\right))\right) = 1000 \tag{6.84}$$

$$\Phi_{FS}\left(c_1 \log(c_2\left(a + 1.0\right))\right) = 1000 \tag{6.85}$$

$$\Phi_{FS}\left(c_1 \log(c_2\left(a + 1.0\right))\right) = 1000 \tag{6.86}$$

$$\Phi_{FS}\left(c_1 \log(c_2\left(a + 1.0\right))\right) = 1000 \tag{6.87}$$

$$\Phi_{FS}\left(c_1 \log(c_2\left(a + 1.0\right))\right) = 1000 \tag{6.88}$$

$$\Phi_{FS}\left(c_1 \log(c_2\left(a + 1.0\right))\right) = 1000 \tag{6.89}$$

$$\Phi_{FS}\left(c_1 \log(c_2\left(a + 1.0\right))\right) = 1000 \tag{6.90}$$

$$\Phi_{FS}\left(c_1 \log(c_2\left(a + 1.0\right))\right) = 1000 \tag{6.91}$$

$$\Phi_{FS}\left(c_1 \log(c_2\left(a + 1.005\right))\right) = 985.5742 \tag{6.92}$$

$$\Phi_{FS}\left(c_1 \log\left(c_2 \frac{a^2 + 31.8a + 30.9}{(0.127a + 4.1)}\right)\right) = 783.5202 \tag{6.93}$$

6.2.5 Experiment 6-5: Exponential

Setup

In this experiment have used the following exponential function:

$$f(x) = \exp(-2(x-1)), x \in [-1.5, 0] \tag{6.94}$$

A plot of the training examples for this experiment is shown in Figure 6-5. The function set for this experiment was as follows:

$$F = \{+, -, \times, \div, \exp(\star)\} \tag{6.95}$$

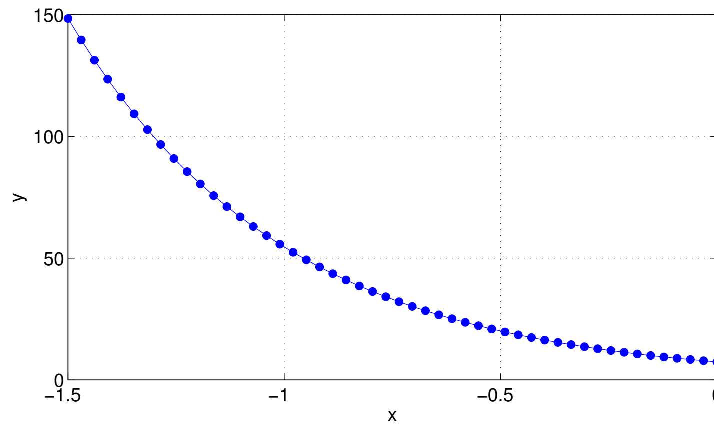


Figure 6-5: Experiment 6-5 Training Examples

Results

Here again we see in Table 6.6 $\Phi_{MS}(f)$ struggled to find its way, while $\Phi_{FS}(f)$ recognized the correct formula on nearly every run.

Run	$\Phi_{MS}(f)$	$\Phi_{FS}(f)$
1	30	1000
2	20	1000
3	32	427
4	4	606
5	46	1000
6	32	1000
7	31	551
8	43	1000
9	28	1000
10	44	427

Table 6.6: Experiment 6-5 Scores

$\Phi_{MS}(f)$ **Outcomes:**

$$\Phi_{MS}\left(a + \mathrm{e}^{\mathrm{e}^{\mathrm{e}^{a}}}\,\mathrm{e}^{a} + 61.0\,a^2\right) = 45.6437 \tag{6.96}$$

$$\Phi_{MS}\left(\mathrm{e}^{2.4\,\mathrm{e}^{a}} + 58.0\,a^2 - 0.097\right) = 44.4815 \tag{6.97}$$

$$\Phi_{MS}\left(14.0\,a^4 + 0.89\,a^3 - 0.89\,a^2 - 47.0\,a\right) = 43.1889 \tag{6.98}$$

$$\Phi_{MS}\left(a^6 + 58.0\,a^2\right) = 32.4447 \tag{6.99}$$

$$\Phi_{MS}\left(54.0\,a^2 + 0.097\,a + 5.8\right) = 31.8 \tag{6.100}$$

$$\Phi_{MS}\left(58.0\,a^2 - 2.0\,a^3 - 1.0\,a^2\,\mathrm{e}^{a} + 0.89\right) = 31.2 \tag{6.101}$$

$$\Phi_{MS}\left(60.0\,a^2 + 0.89\right) = 29.757 \tag{6.102}$$

$$\Phi_{MS}\left(58.0\,a^2\,\mathrm{e}^{-0.037\,a}\right) = 28.1432 \tag{6.103}$$

$$\Phi_{MS}\left(29.0\,\mathrm{e}^{-1.0\,\mathrm{e}^{\mathrm{e}^{a}}}\,\mathrm{e}^{-2.0\,a} - 1.0\,a\right) = 20.336 \tag{6.104}$$

$$\Phi_{MS}\left(\mathrm{e}^{0.035\,a\,\mathrm{e}^{\frac{a^2}{a+100.0}}}\,\mathrm{e}^{-3.5\,a}\right) = 3.7195 \tag{6.105}$$

$\Phi_{FS}(f)$ **Outcomes:**

$$\Phi_{FS}\left(c_1 \mathrm{e}^{-2.0\,a} + c_2\right) = 1000 \tag{6.106}$$

$$\Phi_{FS}\left(c_1 \mathrm{e}^{-2.0\,a} + c_2\right) = 1000 \tag{6.107}$$

$$\Phi_{FS}\left(c_1 \mathrm{e}^{-2.0\,a} + c_2\right) = 1000 \tag{6.108}$$

$$\Phi_{FS}\left(c_1 \mathrm{e}^{-2.0\,a} + c_2\right) = 1000 \tag{6.109}$$

$$\Phi_{FS}\left(c_1 \mathrm{e}^{-2.0\,a} + c_2\right) = 1000 \tag{6.110}$$

$$\Phi_{FS}\left(c_1\left(a + \left(5.5 \cdot 10^5\right)\, \mathrm{e}^{-2.0\,a}\right) + c_2\right) = 999.9871 \tag{6.111}$$

$$\Phi_{FS}\left(c_1\left(a^4 + a^2 - 0.92\,a\right) + c_2\right) = 605.5516 \tag{6.112}$$

$$\Phi_{FS}\left(c_1\left(a + \frac{1.2 \cdot 10^4}{a\,\mathrm{e}^{-1.0\,a} + 26.0}\right) + c_2\right) = 550.7 \tag{6.113}$$

$$\Phi_{FS}\left(c_1 a\,\mathrm{e}^{\frac{59.0}{a - \frac{48.0}{a}}} + c_2\right) = 427.145 \tag{6.114}$$

$$\Phi_{FS}\left(c_1 + c_2 \frac{1.0\,a\,\mathrm{e}^{-1.0\,a}}{2.0\,a + 9.4}\right) = 426.7693 \tag{6.115}$$

6.3 Real Experiments

In the real world the fundamental law describing how observed data is generated is typically unknown for new experiments. Here we have asked the GEP algorithm to infer the formula describing how the input is mapped to the output for the following set of experiments acquired from the UCI machine learning repository[1]:

1. Vapor pressure of bromine
2. Thermal conductivity of air at low pressures
3. Emission of electrons from heated tantulum
4. Magnetic flux after torsion
5. Index of refraction of ethyl alcohol
6. Resistance vs. centigrade temperature

These experiments are subset of what's available and were chosen because the independent variable was sampled linearly at equal intervals and the number of data points was adequate (there were several data sets where this is not the case (i.e. very few data points, and/or uneven sampling)). The data sets we used can be found in Appendix E formatted into a callable MATLAB routine returning each data set in a single structure.

The format of the results in this section will differ from the set of analytical experiments where we knew the correct solution ahead of time. The analysis approach was to:

1. Partition the experimental data into training and test data.
2. Run the algorithm multiple times on the training data using $\Phi_{MS}(f)$ and $\Phi_{FS}(f)$.
3. Select the solution with the highest fitness score from each criterion function.
4. Estimate if the correct solution was found by computing the error between test data and the predictions from each solution and through a visual inspection.

[1] http://archive.ics.uci.edu/ml/datasets/Function+Finding

Algorithm Parameters			
Terminal Set	$\{a, ?\}$	1-Pt Recombination Rate	0.3
Function Set	$\{+, -, \times, \div, \sqrt{(\star)}, \exp(\star)\}$	2-Pi Recombination Rate	0.3
Genes	1	Gene Recombination Rate	0.3
Head Size	10	IST Rate	0.1
Tail Size	11	RIST Rate	0.1
Gene Size	32	Survival Threshold	1
Chrm Size	32	Convergence Threshold	999.9
Runs	10	Maximum Fitness	1000
Population Size	200	Training Samples	Varies
Mutation Rate	0.1	Max Generations	250
Inversion Rate	0.1	Min Founders	1

Table 6.7: Real Experiments Configuration

The algorithm parameters for each experiment were the same in every case and are shown in Table 6.7. To determine the unknown coefficients for the solutions found by $\Phi_{FS}(f)$ we have used the Nelder-Mead simplex algorithm to minimize the mean square error between the solution and the training examples as follows:

$$(\hat{c_1}, \hat{c_2})^T = \underset{(c_1,c_2)^T}{\arg\min} \left(\frac{1}{N} \sum_{k=1}^{N} \left(y^{(k)} - \left(c_1 f \left(x^{(k)} \right) + c_2 \right) \right)^2 \right) \tag{6.116}$$

6.3.1 Experiment 6-6: Vapor Pressure

Setup

In this experiment we have asked the GEP algorithm to find an equation that describes the relationship between the vapor pressure on Bromine (in millimeters of mercury) versus temperature Centigrade. The data used from this experiment is shown below in Table 6.8, courtesy of the UCI machine learning repository[2].

Temperature	Vapor Pressure of Bromine
-95	-6.1193
-90	-5.2591
-85	-4.4482
-80	-3.6849
-75	-2.9701
-70	-2.2828
-65	-1.6503
-60	-1.03
-55	-0.46522
-50	0.086178
-45	0.60432
-40	1.0919
-35	1.5623
-30	2.0082
-25	2.4336
-20	2.8391
-15	3.2268
-10	3.6

Table 6.8: Experiment 6-6 Data Set (courtesy of the UCI machine learning repository)

[2]We have taken the natural log of the original vapor pressure data

Results

The most fit solutions returned by $\Phi_{MS}(f)$ and $\Phi_{FS}(f)$ were as follows:

$$\Phi_{MS}\left(-\left(6.7 \cdot 10^{-4}\right)\, a^2 + 0.033\, a + 3.4\right) = 983 \tag{6.117}$$

$$\Phi_{FS}\left(c_1\left(177.0\, \mathrm{e}^{-0.013\, a} - 1.0\, a\right) + c_2\right) = 947 \tag{6.118}$$

where $c_1 = -0.0197$ and $c_2 = 7.7415$. Although both had strong scores, $\Phi_{FS}(f)$ yielded a better explanation of the data as shown in Figure 6-6 where it faithfully captured the curvature of the training examples. In this case the function returned by $\Phi_{MS}(f)$ was not really a decoy in that it did not explain the data very well in the training interval. If more generations were available, a better solution could have possibly been reached, however the scale of the data resulted a small mean square error which created an overly optimistic fitness score. The test errors for each solution were:

$$e_{MS} = 0.1616 \tag{6.119}$$

$$e_{FS} = 1.74 \times 10^{-4} \tag{6.120}$$

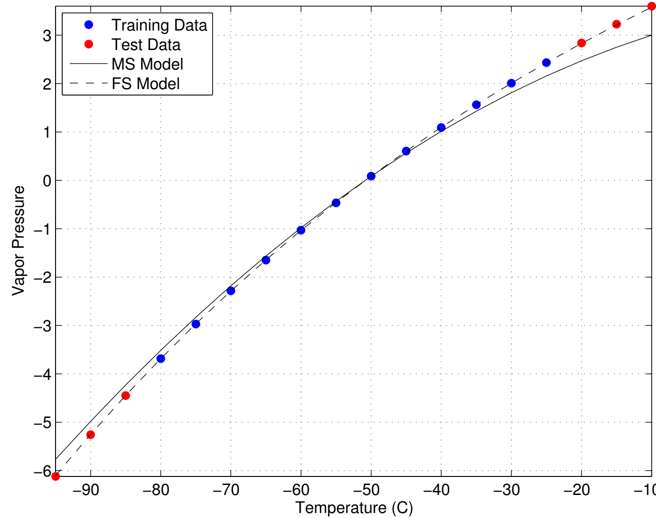


Figure 6-6: Experiment 6-6 Results

6.3.2 Experiment 6-7: Thermal Conductivity

Setup

In this experiment, we have asked the GEP algorithm to tell us the relation between the temperature rate of change with respect to the current squared in a heating element to air pressure in millimeters of mercury. The data set we have used is shown below in Table 6.9, courtesy of the UCI machine learning repository.

Air Pressure	Temperature Change/100
0.01	16.7
0.02	13.55
0.03	11.55
0.04	10
0.05	8.93
0.06	8.2
0.07	7.7
0.08	7.35
0.09	6.95
0.1	6.63
0.11	6.29
0.12	6.1
0.13	5.91
0.14	5.71
0.15	5.52
0.16	5.38
0.17	5.23
0.18	5.11
0.19	5
0.2	4.85

Table 6.9: Experiment 6-7 Data Set (courtesy of the UCI machine learning repository)

Results

The most fit solutions returned by $\Phi_{MS}(f)$ and $\Phi_{FS}(f)$ were as follows:

$$\Phi_{MS}\left(\left(\frac{50.0}{a}+\frac{14.0}{a^2}+\frac{0.41}{a^{\frac{3}{2}}}-1.0\right)^{\frac{1}{4}}\right)=998 \tag{6.121}$$

$$\Phi_{FS}\left(c_1\left(\sqrt{\mathrm{e}^{\frac{0.33}{\sqrt{a}}}}-1.0\,a\right)+c_2\right)=490 \tag{6.122}$$

where $c_1 = 5.18$ and $c_2 = -1.6$. Here we saw the MS criterion provide a better explanation of the test data than the FS criterion. However, because the fitness score on the FS solution was so low, it's clear that the true formula was not found even before analyzing the test examples. The plot of each solution is shown below in Figure 6-7, the resulting mean squared error of the test data for each model was:

$$e_{MS} = 1.25 \tag{6.123}$$

$$e_{FS} = 12.6 \tag{6.124}$$

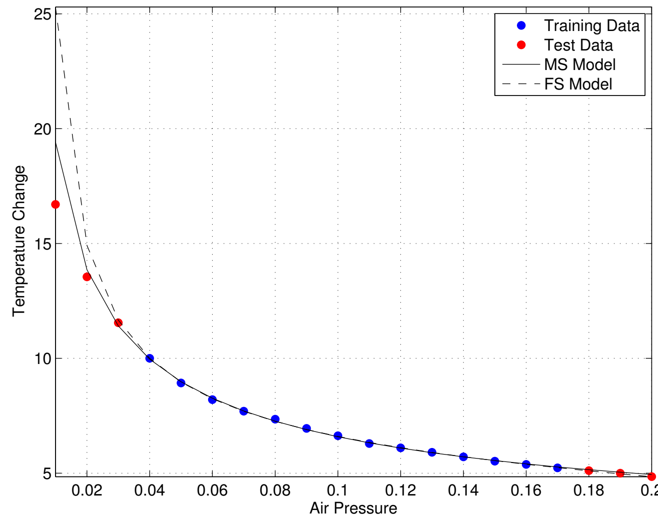


Figure 6-7: Experiment 6-7 Results

6.3.3 Experiment 6-8: Emission of Electrons

Setup

In this experiment we have asked the GEP algorithm to discover the relation between the emission of electrons from heated Tantulum and absolute temperature. The data set for this experiment is shown in Table 6.10, courtesy of the UCI machine learning repository.[3].

Absolute Temperature	ln(Electrons Amps/cm^2)
1000	-29.2658
1100	-24.7919
1200	-21.0504
1300	-17.8726
1400	-15.9111
1500	-12.7543
1600	-10.6209
1700	-8.7982
1800	-7.1384
1900	-5.624
2000	-4.2831
2100	-3.0748
2200	-1.959
2300	-0.93649
2400	0
2500	0.8671

Table 6.10: Experiment 6-8 Data Set (courtesy of the UCI machine learning repository)

[3]We have taken the natural log of the original electron current data

Results

The most fit solutions returned by $\Phi_{MS}(f)$ and $\Phi_{FS}(f)$ were as follows:

$$\Phi_{MS}\left(\sqrt{2.0\,a + \sqrt{1.4\,\sqrt{a}} - 93.0 - 67.0}\right) = 723 \tag{6.125}$$

$$\Phi_{FS}\left(c_1\sqrt{83.0\,a - 1.0\,a^{\frac{3}{2}} + 7.3} + c_2\right) = 575 \tag{6.126}$$

where $c_1 = 0.463$ and $c_2 = -132.19$. Interestingly $\Phi_{FS}(f)$ returned a lower score than the$\Phi_{MS}(f)$, yet the $\Phi_{FS}(f)$ clearly provided a better explanation of the data. $\Phi_{FS}(f)$ is quite conservative when scoring its solutions and only returns a high score when it's justified. $\Phi_{MS}(f)$ was fooled in this experiment by the scale of the data which led the algorithm to believe it had found a good solution erroneously. Both solutions are plotted below in Figure 6-8, and the resulting mean squared errors were,

$$e_{MS} = 10.44 \tag{6.127}$$

$$e_{FS} = 0.76 \tag{6.128}$$

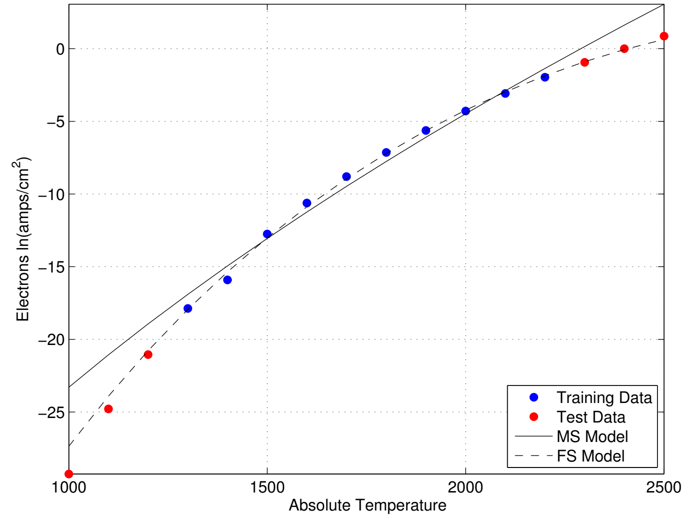


Figure 6-8: Experiment 6-8 Results

6.3.4 Experiment 6-9: Magnetic Flux After Torsion

Setup

In this experiment we have asked the GEP algorithm to discover the relation between magnetic flux after torsion versus position on a 112cm rod. The data set we used for this experiment is shown in Table 6.11, courtesy of the UCI machine learning repository.

Position on Rod (cm)	Magnetic Flux (cgs)/100
2	5.53
8	17.54
14	26.55
20	33.61
26	39.05
32	43.08
38	45.99
44	47.46
50	47.92
56	47.12
62	45.5
68	43.37
74	40.49
80	37.01
86	32.75
92	27.59
98	21.34
104	13.71
110	3.92

Table 6.11: Experiment 6-9 Data Set (courtesy of the UCI machine learning repository)

Results

The most fit solutions returned by $\Phi_{MS}(f)$ and $\Phi_{FS}(f)$ were as follows:

$$\Phi_{MS}\left(\sqrt{-1.0\,a^2 + 100.0\,a - 200.0}\right) = 219.781 \tag{6.129}$$

$$\Phi_{FS}\left(c_1\left(\left(7.6 \cdot 10^3\right)\,a - 76.0\,a^2\right) + c_2\right) = 229.352 \tag{6.130}$$

where $c_1 = 0.0002$ and $c_2 = 17.78$. Here both $\Phi_{MS}(f)$ and $\Phi_{FS}(f)$ struggled to find a good explanation of the data. It's unlikely that more generations would have helped to find better solutions. This appears to be a problem of model order, meaning the true explanation either does not exist or was not in the in search space of the algorithm. Each solution is plotted in Figure 6-9 and the mean squared errors were,

$$e_{MS} = 264.06 \tag{6.131}$$

$$e_{FS} = 53.87 \tag{6.132}$$

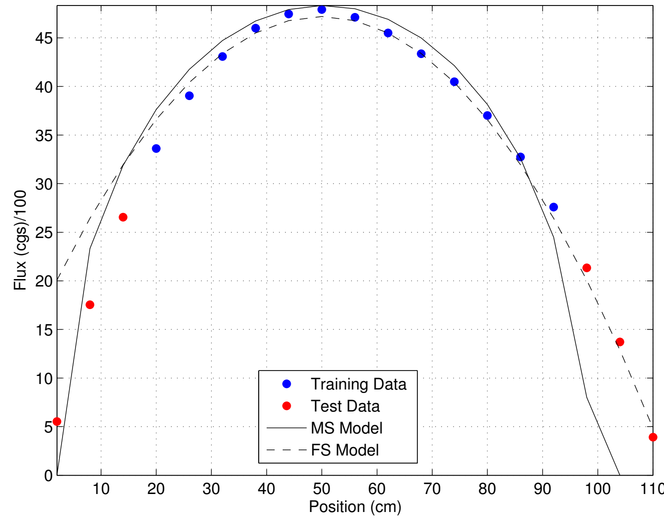


Figure 6-9: Experiment 6-9 Results

6.3.5 Experiment 6-10: Index of Refraction

Setup

In this experiment the dependent variable we have asked the algorithm to predict is the index of refraction of ethyl alchohol (relative to air) for sodium light and the independent variable is Centigrade temperature. The data set we have used is shown in Table 6.12, courtesy of the UCI machine learning repository.

Temperature	Index of Refraction	Temperature	Index of Refraction
14	1.3629	46	1.3497
16	1.3621	48	1.3489
18	1.3613	50	1.348
20	1.3605	52	1.3472
22	1.3597	54	1.3463
24	1.3588	56	1.3454
26	1.358	58	1.3446
28	1.3572	60	1.3437
30	1.3564	62	1.3428
32	1.3556	64	1.3419
34	1.3547	66	1.341
36	1.3539	68	1.34
38	1.3531	70	1.3391
40	1.3522	72	1.3382
42	1.3514	74	1.3373
44	1.3505	76	1.3363

Table 6.12: Experiment 6-10 Data Set (courtesy of the UCI machine learning repository)

Results

The most fit solutions returned by the MS and FS criterions we as follows:

$$\Phi_{MS}\left(\frac{1.4}{a^{\frac{1}{64}}}\right) = 999.9 \tag{6.133}$$

$$\Phi_{FS}\left(c_1\left(a+\sqrt{79\,\mathrm{e}^{0.032\,a}+3700}\right)+c_2\right) = 940 \tag{6.134}$$

where $c_1 = -0.0004$ and $c_2 = 1.3925$. Clearly $\Phi_{MS}(f)$ was tricked into thinking it had a good solution due to the magnitude of the dependent variable. On the other hand, the solution found by $\Phi_{FS}(f)$ was near perfect. The high score given by $\Phi_{FS}(f)$ is a strong indicator that the solution is likely to be correct. Solutions are plotted in Figure 6-10 and the mean squared errors were,

$$e_{MS} = 9.76 \times 10^{-5} \tag{6.135}$$

$$e_{FS} = 1.98 \times 10^{-9} \tag{6.136}$$

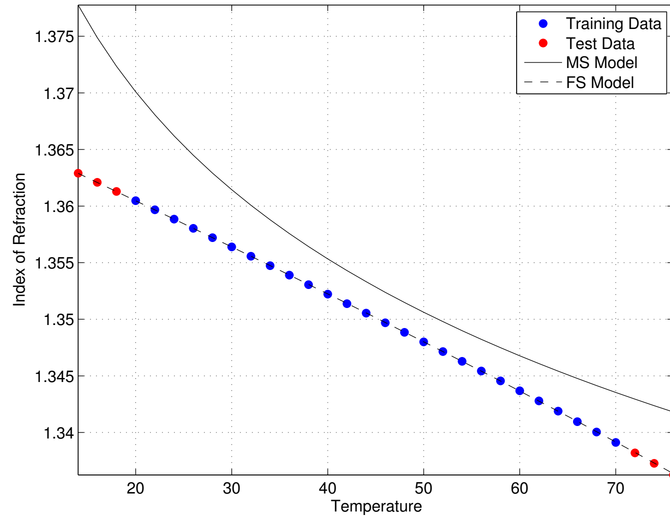


Figure 6-10: Experiment 6-10 Results

6.3.6 Experiment 6-11: Resistance vs. Temperature

Setup

In this final experiment we have tried to automatically discover the relation between the resistance of nickel and Centigrade temperature. We have made use the of the data set shown in Table 6.13, courtesy of the UCI machine learning repository.

Temperature	Resistance of Nickel
-25	11.03
-20	11.25
-15	11.475
-10	11.7
-5	11.935
0	12.173
5	12.42
10	12.66
15	12.92
20	13.173
25	13.435
30	13.7
35	13.965
40	14.24
45	14.52
50	14.8
55	15.08
60	15.385
65	15.69
70	16
75	16.32

Table 6.13: Experiment 6-11 Data Set (courtesy of the UCI machine learning repository)

Results

The most fit solutions returned by $\Phi_{MS}(f)$ and $\Phi_{FS}(f)$ were as follows:

$$\Phi_{MS}\left(0.051\,a + 12.0\right) = 997.5764 \tag{6.137}$$

$$\Phi_{FS}\left(c_1\sqrt{177.0 - 1.0\,a} + c_2\right) = 907.0701 \tag{6.138}$$

where $c_1 = -1.29$ and $c_2 = 29.3$. Here $\Phi_{MS}(f)$ was unable to fit the subtle curvature of the data, yet was convinced it had a near perfect solution. On the other hand, $\Phi_{FS}(f)$ returned a strong score and fit the test data near perfectly. Each solution is plotted in Figure 6-11 and the mean squared errors were,

$$e_{MS} = 0.0357 \tag{6.139}$$

$$e_{FS} = 2.3 \times 10^{-4} \tag{6.140}$$

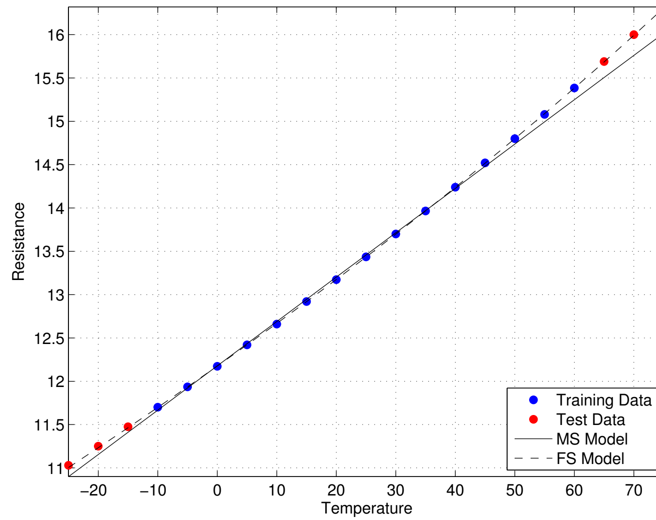


Figure 6-11: Experiment 6-11 Results

6.4 Summary

In this chapter we have presented the results from a total of 11 numerical experiments. The first 5 experiments were analytical in nature and based on several common function types. The last 6 were based on real world data courtesy of the UCI machine learning repository where the true underlying form of the data generating equation was assumed to be unknown. In each experiment we did 10 trials using the mean-square error fitness function $\Phi_{MS}(f)$ and 10 trials using the feature signature fitness function $\Phi_{FS}(f)$. The results of these experiments can be summarized as follows:

- **Experiment 6-1: Polynomial**, Both fitness functions were able to find the correct form of the solution at least once. $\Phi_{MS}(f)$ came out ahead (4 out of 10) of the $\Phi_{FS}(f)$ (1 out of 10) however it was unclear exactly why. Despite this result, we have concluded the recognition performance of $\Phi_{FS}(f)$ is valid, however more trials may be necessary in practice.

- **Experiment 6-2: Rational**, This data set presented a real challenge to both fitness functions as neither was able to discover the true form of the solution. It was concluded from the simplicity of the solutions found by $\Phi_{MS}(f)$ and $\Phi_{FS}(f)$, that the true formula was not in the search space and that additional experiments with different algorithm configurations is necessary.

- **Experiment 6-3: Trigonometric**, In this experiment we saw $\Phi_{MS}(f)$ struggle to find the correct answer while $\Phi_{FS}(f)$ succeeded on every trial thanks to its invariance to linear transformations.

- **Experiment 6-4: Logarithmic**, This was the first experiment where both fitness criterions consistently scored highly in the allowed learning interval. We did see $\Phi_{MS}(f)$ deliver the correct solution 2 out of 10 times however we saw several incorrect solutions scoring almost as well. This is an important general result in that the scaling of the dependent variable needs to be carefully considered when implementing $\Phi_{MS}(f)$. On the other hand, $\Phi_{FS}(f)$ is invariant to scale and was successful 8 out of 10 times.

- **Experiment 6-5: Exponential**, This was the final analytical experiment where we saw $\Phi_{MS}(f)$ struggle to find the correct answer again, and $\Phi_{FS}(f)$ found the correct solution form 5 out of 10 times, again thanks to its invariance to linear transformations.

- **Experiment 6-6: Vapor Pressure**, In this experiment the best solution reached using $\Phi_{MS}(f)$ had a high fitness score yet did a poor explanation of the data (again a scaling problem). Additionally, more generations could have been beneficial. The solution reached using $\Phi_{FS}(f)$ did an excellent job explaining the test data, returned a strong score, and was very confident it found the correct formula.

- **Experiment 6-7: Thermal Conductivity**, In this experiment both fitness criterions failed to adequately explain the data. Interestingly the errors in the training interval were very close yet the score returned by $\Phi_{FS}(f)$ was far more conservative than the score returned by $\Phi_{MS}(f)$.

- **Experiment 6-8: Emission of Electrons**, The solution found using $\Phi_{MS}(f)$ in this experiment returned a poor solution which under fit the data. More generations would again have been beneficial here. The solution found by $\Phi_{FS}(f)$ did a better job but had a low fitness score indicating it had not quite explained the curvature of the data. This is evident in the error on the first few data points in the test set.

- **Experiment 6-9: Magnetic Flux**, This was a very challenging data set for both fitness criterions, both scored poorly, and it was concluded the true formula either did not exist or was not in the search space.

- **Experiment 6-10: Index of Refraction**, This was yet another example of where the scaling of the dependent variable tricked $\Phi_{MS}(f)$ into returning a poor solution with a high fitness score. The data was quite simple and nearly linear in nature, yet $\Phi_{FS}(f)$ was able to correctly find an excellent solution describing the data with a strong score indicating the algorithm's confidence in the result.

- **Experiment 6-11: Resistance vs. Temperature**, In this final experiment, the data exhibited a subtle curvature which $\Phi_{FS}(f)$ was quick to pick up on, delivering a very low test error and strong score.$\Phi_{MS}(f)$ was unable to pick up on this and returned a simple linear model which clearly under fit the test data, again due to the scaling of the training examples.

Chapter 7

Conclusion

Alas, we have reached the end of our journey through the world of machine learning, evolutionary algorithms, and the problem of inferring mathematical truths. Our brief visit to machine learning only scratched the surface of what's out there, but provided the necessary motivation to look for the solution to our problem elsewhere. The ideas set forth by the GEP algorithm and symbolic regression gave us the momentum we needed to move to the concepts of features spaces and signatures. Then finally everything came together through a series of analytical and real world experiments.

Now that we are here, what can be said? Obviously black box models are approximations of the real world, although in many types of problems this approach works exceptionally well. However, if we are looking for something more, white box models are much more attractive. This was established by the results of Experiment 2-1.

Minimizing the squared error between the training examples and the model is pretty standard in most optimization problems. For the case of symbolic regression, we found this approach to be appropriate in a limited number of cases, namely in Experiments 4-1, 4-2, 4-3, 4-4 and 6-1. However many other experiments revealed that this cost function is inefficient at driving the evolutionary process and is easily fooled by the scaling of the training examples (i.e. Experiments 3-1, 4-6, 5-1, 6-3, 6-4, 6-5, 6-6, 6-8, 6-10, and 6-11).

The aha moment came after Experiment 4-5 where we found that there needs to be sufficient information in the training examples to reach the correct formula. This was corroborated by Experiment 4-7. The question then became: what do you do when the training examples lack the necessary information. This is what ultimately led us to the idea of feature signatures, which automates the process of extracting information.

The breakthrough came when we realized that the problem of formula recognition could be sub-divided into 2 smaller problems namely (1) find the correct form of the solution and (2) optimize the coefficients. Through the series of experiments conducted in chapter 6, it's clear that we can find the data generating formula behind a set of data points if it exists and we know how to go looking.

Additionally, the invariance to linear transformations came to be a powerful feature of $\Phi_{FS}(f)$ which can be viewed as shrinking the search space because the true formula or any linear transformation of the true formula yields the correct answer. In the case of $\Phi_{MS}(f)$ there is only 1 correct answer, which is like finding a needle in a haystack.

There are still some unanswered questions concerning model order selection and/or solution existence which we discovered in Experiments 6-2, 6-7, and 6-9. However, we found that $\Phi_{FS}(f)$ is very conservative when it comes to

scoring candidate solutions. Constant failure and low fitness scores are a sign that a formula does not exist in the current search space or in general. Or possibly there are several variables that we are not observing (which we did not discuss).

Based on this, is the work of teaching computers how to do science done? Not even close. Here we have only scratched the surface by solving a very small problem. Computers need to get much smarter if we plan to have fighting chance against the amount of data we are swimming in. Hopefully this will make a small dent in this area and be a source of inspiration to the future.

Future Work

The new cost function we have developed $\Phi_{FS}(f)$ has not been tested with noisy data. In fact, because it relies on derivatives it will most certainly run into trouble. Therefore the next obvious step is to study the effects of noise and whether the correct solution can be inferred if noisy data is smoothed. Additionally in this work we have only considered univariate problems, so in the future multivariate problems must be addressed. The multivariate problem is far more complex and it the size of the search space is much bigger. Another interesting area to explore is in how well the algorithm can prune irrelevant variables from multivariate data sets. Finally, dynamic systems need to be addressed where the functional relationship between the input(s) and output(s) vary over time. All of these additional problems are very complex and the results obtained through further research in this area will undoubtedly be fascinating.

Bibliography

[1] D. Hilbert. "Mathematical Problems,". *An address to the International Congress of Mathematicians at Paris (1900), translated by Maby Winton Newson for the Bulletin of the American Mathematical Society 8 (1902)*, 1902.

[2] R. Penrose. *The Road to Reality: A Complete Guide to the Laws of the Universe*. New York, NY: Random House, 2004.

[3] B. B. Poppe and K. P. Jorden. *Sentinels of the Sun: Forecasting Space Weather*. Boulder, CO: Big Earth Publishing, 2006.

[4] N. N. Taleb. *The Black Swan: The Impact of the Highly Improbable*. New York, NY: Random House, 2007.

[5] K. Elkinson and A. McGrail. "The Risk of Too Much Data". *IEEE Power and Energy Society General Meeting*, 2012.

[6] M. C. Schatz and B. Langmead. "The DNA Data Deluge". *IEEE Spectrum, July*, 2013.

[7] R. P. Norris. "Data Challenges for the Next-Generation Radio Telescopes". *2010 Sixth IEEE International Conference on e-Science Workshops*, 2010.

[8] D. Zhang. "Inconsistencies of Big Data". *Proceedings of the 12th IEEE Conference on Cognitive Informatics and Cognitive Computing*, 2013.

[9] A. Katal et. al. "Big data: Issues, challenges, tools and Good practices". *Sixth International IEEE Conference on Contemporary Computing (IC3)*, 2013.

[10] D. Garlasu et. al. "A big data implementation based on Grid computing". *11th IEEE Roedunet International Conference (RoEduNet)*, 2013.

[11] S. Sagiroglu and D. Sinanc. “Big data: A review”. *International Conference on Collaboration Technologies and Systems (CTS)*, 2013.

[12] T. Condie et. al. “Machine learning on Big Data”. *IEEE 29th International Conference on Data Engineering (ICDE)*, 2013.

[13] F. Galton. “Regression towards mediocrity in hereditary stature”. *The Journal of the Anthropological Institute of Great Britain and Ireland, Vol. 15*, 1886.

[14] L. Ljung. “Black-box models from input-output measurements”. *Proceedings of the 18th IEEE Instrumentation and Measurement Technology Conference*, 2001.

[15] C. M. Bishop. *Pattern Recognition and Machine Learning.* New York, NY: Springer, 2006.

[16] V. N. Vapnik. *The nature of statistical learning theory.* New York, NY: Springer, 1995.

[17] I. Icke and J. C. Bongard. “Improved Genetic Programming Based Symbolic Regression Using Deterministic Machine Learning”. *IEEE Congress on Evolutionary Computation*, 2013.

[18] J. R. Koza. *“Genetic Programming: On the Programming of Computers by Means of Natural Selection,”.* MIT Press, 1992.

[19] J. R. Koza. *“Genetic Programming II: Automatic Discovery of Reusable Programs,”.* MIT Press, 1994.

[20] M. Schmidt and H. Lipson. “Data-Mining Dynamical Systems: Automated Symbolic System Identification for Exploratory Analysis,”. *Proceedings of the 9th Biennial ASME Conference on Engineering Systems Design and Analysis*, 2008.

[21] M. Schmidt and H. Lipson. “Distilling Free-Form Natural Laws form Experimental Data,”. *Science, Vol 324*, 2009.

[22] M. Schmidt and H. Lipson. “Discovering a Domain Alphabet,” . *GECCO ’09, July 8-12*, 2008.

[23] M. Schmidt and H. Lipson. “Symbolic Regression of Implicit Equations,” . *in Genetic Programming Theory and Practice VII, Genetic and Evolutionary Computation, Springer Science + Business Media*, 2008.

[24] J. Koza. "Human-Competitive results produced by genetic programming". *Genetic Programming and Evolvable Machines, vol. 11*, 2010.

[25] M. O'Neill et al. "Open Issues in Genetic Programming,". *Genetic Programming and Evolvable Machines, Springer*, 2010.

[26] I. Kushchu. "An evaluation of evolutionary generalization in genetic programming". *Artificial Intelligence Review, 18(1)*, 2002.

[27] J. Rosca. *"Generality versus size in genetic programming"*. in GP 1996, J. R. Koza et al., editors, MIT Press, 1996.

[28] N. M. Amil et al. "A statistical learning perspective of genetic programming". *EuroGP Conference Proceedings*, 2009.

[29] E. J. Vladislavleva et al. "Order of Nonlinearity as a Complexity Measure for Models Generated by Symbolic Regression via Pareto Genetic Programming". *IEEE Transactions on Evolutionary Computation*, 2009.

[30] M. Castelli et al. *"A quantitative study of learning and generalization in genetic programming"*. in Genetic Programming, S. Silva et al. editors, Springer Berlin Heidelberg, 2011.

[31] M. Keijer and V. Babovic. "Dimensionally aware genetic programming". *Proceedings of the First Genetic and Evolutionary Conference GECCO '99*, 1999.

[32] Z. Zitzler et al. *"A tutorial on evolutionary multiobjective optimization"*. in Metaheuristics for Multiobjective Optimisation, X. Gandibleux et al. editors, Springer Verlag, 2004.

[33] G. Smits and M. Kotanchek. *"Pareto-front exploitation in symbolic regression"*. in Genetic Programming Theory and Practice II, U. O'Reilly et al. editors, Springer Verlag, 2004.

[34] C. Ferreira. "Gene Expression Programming: A New Adaptive Algorithm for Solving Problems,". *Complex Systems, Vol. 13, Issue 2*, 2001.

[35] C. Ferreira. *Gene Expression Programming: Mathematical Modeling by an Artificial Intelligence.* 2nd Ed., Studies in Computational Intelligence 21, Springer, 2006.

[36] A. M. Turing. "Computing Machinery and Intelligence". *Mind, 59(236)*, 1950.

[37] S. Stigler. *The History of Statistics: The Measurement of Uncertainty Before 1900*. Cambridge, MA: Belknap Press of Harvard University Press, 1991.

[38] F. G. Gustavson and J. Wasniewski. "Gausss, Choleskys and Banachiewiczs Contributions to Least Squares". *Technical University of Denmark, Kgs. Lyngby, Denmark, IMM-Technical Report-2011*, 2011.

[39] C. Moler. *Numerical Computing with MATLAB*. Natick, MA: The MathWorks, Inc., 2004.

[40] N. R. Draper and H. Smith. *Applied Regression Analysis*. Hoboken, NJ: Wiley-Interscience, 1998.

[41] S. Chatterjee and A. S. Hadi. "Influential Observations, High Leverage Points, and Outliers in Linear Regression". *Statistical Science, Vol. 1*, 1986.

[42] J. O. Street et al. "A Note on Computing Robust Regression Estimates via Iteratively Reweighted Least Squares". *The American Statistician, Vol. 42, No. 2*, 1988.

[43] D. W. Marquardt and R. D. Snee. "Ridge Regression in Practice,". *The American Statistician, Vol. 29, No. 1*, 1975.

[44] M. J. D. Powell. *"Radial basis functions for multivariable interpolation: a review"*. In J. C. Mason and M. G. Cox Eds., Algorithms for Approximation, Oxford: Clarendon Press, 1987.

[45] D. S. Broomhead and D. Lowe. "Multivariable functional interpolation and adaptive networks,". *Complex Systems, Vol. 2*, 1988.

[46] J. Moody and C. J. Darken. "Fast learning in networks of locally-tuned processing units,". *Neural Computation, Vol. 1, No. 2*, 1989.

[47] J. Park and I. W. Sandberg. "Universal approximation using radial basis function networks,". *Neural Computation, Vol. 3, No. 2*, 1991.

[48] J. Park and I. W. Sandberg. "Approximation and radial basis function networks,". *Neural Computation, Vol. 5, No. 2*, 1991.

[49] C. M. Bishop. *Neural Networks for Pattern Recognition.* New York, NY: Oxford University Press, 1995.

[50] M. E. Tipping. "Sparse Bayesian learning and the relevance vector machine,". *Journal of Machine Learning Research, Vol. 1*, 2001.

[51] R. O. Duda et al. *Pattern Classification.* 2nd Ed., New York, NY: Wiley-Interscience, 2001.

[52] S. P. Lloyd. "Least squares quantization in PCM,". *IEEE Transactions on Information Theory, Vol. 28, No. 2*, 1982.

[53] Y. Linde et al. "An algorithm for vector quantizer design,". *IEEE Transactions on Communications, Vol. 28, No. 1*, 1980.

[54] J. MacQueen. "Some methods for classification and analysis of multivariate observations,". *Proceedings of the Fifth Berkeley Symposium on Mathematical Statistics and Probability, Vol. 1, Berkeley, CA: University of California Press.*, 1967.

[55] F. Rosenblatt. "The perceptron: a probabilistic model for information storage and organization in the brain,". *Psychological Review, 65(6)*, 1958.

[56] F. Rosenblatt. *Principles of Neurodynamics.* Washington DC: Spartan Books, 1962.

[57] A. E. Bryson et al. "Optimal programming problem with inequality constraints. I: Necessary conditions for extremal solutions,". *American Institute of Aeronautics and Astronautics Journal, 1(11)*, 1963.

[58] P. J. Werbos. *Beyond Regression: New tools for prediction and analysis in the behavioral sciences.* Ph.D. Dissertation, Harvard University, Cambridge, MA, 1974.

[59] P. J. Werbos. *The Roots of Backpropagation: From ordered derivatives to neural networks and political forecasting.* New York, NY: Wiley, 1994.

[60] Y. LeCun. "A learning scheme for asymmetric threshold networks,". *Proceedings of Cognitiva 85, Paris, France*, 1985.

[61] D. E. Rumelhart et al. "Learning internal representations by backpropagating errors,". *Nature, 323(99)*, 1986.

[62] S. Grossberg. "Competitive Learning: From interactive activation to adaptive resonance,". *Cognitive Science, 11(1)*, 1987.

[63] D. G. Stork. "Is backpropagation biologically plausible?,". *Proceedings of the International Joint Conference on Neural Networks (IJCNN), New York*, 1989.

[64] M. B. Menhaj M. T. Hagan. "Training Feedforward networks with the Marquardt algorithm,". *IEEE Transactions on Neural Networks, Vol. 5, No. 6*, 1994.

[65] P. E. Gill et al. *Practical Optimization*. Academic Press, Inc, 1981.

[66] J. E. Dennis Jr. and R. B. Schnabel. *Numerical Methods for Unconstrained Optimization and Nonlinear Equations*. Prentice Hall, 1983.

[67] R. Reed. "Pruning Algorithms - A Survey,". *IEEE Transactions on Neural Networks, Vol. 4, No. 5*, 1994.

[68] C. Darwin. *"The origin of species by means of natural selection"*. John Murray, 1859.

[69] J. H. Holland. *"Adaption in Natural and Artificial Systems: An Introductory Analysis with Applications to Biology, Control, and Artificial Intelligence,"*. 2nd ed., MIT Press, 1992.

[70] R. Poli et al. *"A Field Guide to Genetic Programming,"*. Published via http://lulu.com and freely available at http://www.gp-field-guide.org.uk, 2008.

[71] P. J. Angeline. "Two self-adaptive crossover operators for genetic programming,". *In Advances in Genetic Programming 2, P. J. Angeline and K. E. Kinnear, Jr., Eds. MIT Press*, 1996.

[72] C. Ryan and M. Keijzer. "An analysis of diversity of constants of genetic programming,". *In Genetic Programming, Proceedings of EuroGP2003 (Essex, 14-16 Apr. 2003), C. Ryan, T. Soule, M. Keijzer, E. Tsang, R. Poli, and E. Costa, Eds., vol. 2610 of LNCS, Springer*, 2003.

[73] G. F. Spencer. "Automatic generation of programs for crawling and walking,". *In Advances in Genetic Programming, K. E. Kinnear, Jr., Ed. MIT Press,*, 1994.

[74] I. Dempsey et al. “Constant creation with grammatical evolution,”. *International Journal of Innovative Computing and Applications 1, 1,*, 2007.

[75] C. Ferreira. “Function Finding and the Creation of Numerical Constants in Gene Expression Programming,”. *7th Online World Conference on Soft Computing in Industrial Applications*, 2002.

[76] C. Ferreira. “Genetic Representation and Genetic Neutrality in Gene Expression Programming,”. *Advances in Complex Systems, Vol. 5, No. 4*, 2002.

[77] C. Ferreira. “Automatically Defined Functions in Gene Expression Programming,”. *In N. Nedjah, L. de M. Mourelle, A. Abraham, eds., Genetic Systems Programming: Theory and Experiences, Studies in Computational Intelligence, Vol. 13, Springer-Verlag*, 2006.

[78] C. Ferreira. “Gene Expression Programming and the Evolution of Computer Programs,”. *In Leandro N. de Castro and Fernando J. Von Zuben, eds., Recent Developments in Biologically Inspired Computing*, 2004.

[79] C. Ferreira. “Gene Expression Programming in Problem Solving,”. *In R. Roy, M. Kppen, S. Ovaska, T. Furuhashi, and F. Hoffmann, eds., Soft Computing and Industry: Recent Applications,*, 2001.

[80] C. Ferreira. “Analyzing the Founder Effect in Simulated Evolutionary Processes Using Gene Expression Programming,”. *In A. Abraham, J. Ruiz-del-Solar, and M. Kppen (eds), Soft Computing Systems: Design, Management and Applications,*, 2002.

[81] C. Ferreira. “Combinatorial Optimization by Gene Expression Programming: Inversion Revisited,”. *In J. M. Santos and A. Zapico, eds., Proceedings of the Argentine Symposium on Artificial Intelligence*, 2002.

[82] C. Ferreira. “Designing Neural Networks Using Gene Expression Programming,”. *9th Online World Conference on Soft Computing in Industrial Applications*, 2004.

[83] A. Papoulis and S. U. Pillai. *“Probability, Random Variables and Stochastic Processes,”*. 4th ed., McGraw Hill, 2002.

[84] M. Oltean and C. Grosan. "A Comparison of Several Linear Genetic Programming Techniques,". *Complex Systems, 14, 1-1*, 2003.

[85] E. Bombieri. "Prime Territory: Exploring the infinite landscape at the base of the number system,". *The Sciences, Vol. 32 No. 5*, 1992.

[86] A. L. Samuel. "Some studies in machine learning using the game of checkers". *IBM Journal of Research Development, 3(3)*, 1959.

[87] J. A. Garcia et al. "Kernel Multivariate Analysis Framework for Supervised Subspace Learning," . *IEEE Signal Processing Magazine, July*, 2013.

[88] R. Jenssen. "Entropy-Relevant Dimensions in the Kernel Feature Space," . *IEEE Signal Processing Magazine, July*, 2013.

[89] G. Montavon et al. "Analyzing Local Structure in Kernel-Based Learning," . *IEEE Signal Processing Magazine, July*, 2013.

[90] Z. Harchaoui et al. "Kernel-Based Methods for Hypothesis Testing," . *IEEE Signal Processing Magazine, July*, 2013.

[91] L. Song et al. "Kernel Embeddings of Conditional Distributions," . *IEEE Signal Processing Magazine, July*, 2013.

[92] C. Hu X. Li and J. Liang. "On design of optimal nonlinear kernel potential function for protein folding and protein design," . *e-Print, arXiv:cond-mat/0302002, www.arxiv.org*, 2005.

[93] H. Liu et al. "A Method to Choose Kernel Function and its Parameters for Support Vector Machines," . *Machine Learning and Cybernetics, 2005. Proceedings of 2005 International Conference on*, 2005.

[94] K. Thadani Ashutosh et al. "Evolutionary Selection of Kernels in Support Vector Machines," . *Advanced Computing and Communications, 2006. ADCOM 2006. International Conference on*, 2006.

[95] Y. Y. Meng. "Parameters Selection of Hybrid Kernel Based on GA," . *Intelligent Systems and Applications, 2009. ISA 2009. International Workshop on*, 2009.

[96] Y. Zhang and L. Li. "Model Selection in Support Vector Machines Using Self-Adaptive Genetic Algorithm," . *Computational Intelligence and Design (ISCID), 2010 International Symposium on, Vol. 1*, 2010.

[97] D. Knuth et al. *"Leaders in Computing-Changing the Digital World,"*. BCS The Chartered Institute for IT, 2011.

Appendix A

Chapter 2 Programs

A.1 LS.m

```
ffunction mdl = LS(X,y)
%*****************************************************************
% FUNCTION:     w = LS(X,y)
% INFO:         Designs least squares regression model
% INPUT:        -X = Observations matrix
%               -y = Response vector
% OUTPUT:       -mdl = Least squares model struct w/ fields:
%                   .fun = model function handle
%                   .fit = training data fit
%                   .weights = weight vector
% AUTHOR:       A. Hensley, 13-Apr-2013
%*****************************************************************

%Setup design matrix
X = [X(:,1).^0 X];

%Solve for weights
w = X\y;

%Configure Model
mdl.fun = @(z)[z(:)*0+1 z(:)]*w;
mdl.fit = X*w;
mdl.weights = w;
```

A.2 RBF.m

```
function mdl = RBF(X,y,N,alpha)
%****************************************************************
% FUNCTION:     w = RBF(X,y,N,alpha)
% INFO:         Designs radial basis function regression model
% INPUT:        -X = Observations matrix
%               -y = Response vector
%               -N = Number of basis functions
%               -alpha = Smoothing parameter
% OUTPUT:       -mdl = RBF model struct w/ fields
%                   .fun = function handle
%                   .fit = training data fit
%                   .clust = cluster indices
%                   .weights = weight vector
% AUTHOR:       A. Hensley, 13-Apr-2013
%****************************************************************


%Setup Basis Functions
I = kmeans(X,N);
phi = cell(1,N);
ms = zeros(1,N);
Cs = cell(1,N);
for kk = 1:N
    m = mean(X(I==kk,:))';
    C = alpha*eye(size(X,2))+cov(X(I==kk,:));
    phi{kk} = @(x)exp(-0.5*((x(:)-m)'/C)*(x(:)-m));
    ms(kk) = m;
    Cs{kk} = C;
end

%Setup function handle for PHI vector
PHI = '@(x)[';
for kk = 1:N
    PHI = [PHI ' phi{' num2str(kk) '}(x)'];
end
PHI = eval([PHI ']']);

%Setup RBF Design Matrix
[Mx,Nx] = size(X);
Xn = zeros(Mx,N);
hw = waitbar(0,'');
for kk = 1:Mx
   Xn(kk,:) = PHI(X(kk,:));
   waitbar(kk/Mx,hw,'Setting Up RBF Design Matrix')
end
Xn = [Xn(:,1).^0 Xn];
```

```
delete(hw)

%Solve for Weights
w = Xn\y;

%Configure Output
mdl.fun = @(z)[1 PHI(z)]*w;
mdl.fit = Xn*w;
mdl.clust = I;
mdl.weights = w;
mdl.mu = ms;
mdl.C = Cs;
```

A.3 NN.m

```
function mdl = NN(X,y,nH)
%****************************************************************
% FUNCTION:      w = NN(X,y)
% INFO:          Designs neural network regression model
% INPUT:         -X = Observations vector
%                -y = Response vector
% OUTPUT:        -mdl = Neural Network model struct w/ fields:
%                    .obj = NN object
%                    .fit = training data fit
%                    .mX = mean of input data X
%                    .sX = standard deviation of input data X
%                    .my = mean of output data y
%                    .sy = standard deviation of output data y
% AUTHOR:        A. Hensley, 13-Apr-2013
%****************************************************************

%Normalize
mX = mean(X);
my = mean(y);
sX = std(X);
sy = std(y);
X = (X-mX)/sX;
y = (y-my)/sy;

%Create a Fitting Network
net = fitnet(nH);

%Train the Network
[net,tr] = train(net,X',y');

%Process Test Data
mdl.fit = my + sy*net(X')';
mdl.obj = net;
mdl.mX = mX;
mdl.my = my;
mdl.sX = sX;
mdl.sy = sy;
```

A.4 Experiment 2-1: Viscosity of Hydrogen

```
%Experiment 2-1: Viscosity of Hydrogen
%*************************************************************************
%Setup
close all hidden
clc

%Sutherland's Viscosity Model (Hydrogen)
lamda = 0.636236562e-6;
C = 72;
ufun = @(T)lamda*T.^(3/2)./(T+C);

%Generate Data Set
nsamp = 200;
Tmax = 555;
Tmin = 0;
t = sort(Tmin+(Tmax-Tmin)*rand(nsamp,1));
n = 1e-6*randn(nsamp,1);
umeas = ufun(t)+n;

%Least Squares
mdlA = LS(t,umeas);

%Radial Basis Functions
mdlB = RBF(t,umeas,3,1e6);

%Neural Network
mdlC = NN(t,umeas,3);

%Plot Results
ti = -100:1000;
lw = 2;
figure,
subplot(1,3,1),hold on,grid on
plot(t,umeas*1e6,'bo')
plot(ti,ufun(ti)*1e6,'color',[0 0.6 0],'linewidth',lw)
plot(ti,mdlA.fun(ti)*1e6,'r','linewidth',lw)
set(gca,'box','on','fontsize',12)
axis tight
axis([-100 1000 -5 30])
title('Least Squares')
ylabel('Viscosity (\muPa-s)')
xlabel('Temperature (^oK)')
legend('Data','Truth','Model','location','northwest')

subplot(1,3,2),hold on,grid on
fitB = zeros(1,length(ti));
```

```
for kk = 1:length(fitB)
   fitB(kk) = mdlB.fun(ti(kk));
end
plot(t,umeas*1e6,'bo')
plot(ti,ufun(ti)*1e6,'color',[0 0.6 0],'linewidth',lw)
plot(ti,fitB*1e6,'r','linewidth',lw)
set(gca,'box','on','fontsize',12)
axis tight
axis([-100 1000 -5 30])
title('Radial Basis Functions')
xlabel('Temperature (^oK)')

subplot(1,3,3),hold on,grid on
fitC = zeros(1,length(ti));
hw = waitbar(0,'Running Neural Network Model...');
for kk = 1:length(fitC)
    temp = (ti(kk)-mdlC.mX)/mdlC.sX;
   fitC(kk) = mdlC.my+mdlC.sy*mdlC.obj(temp);
   waitbar(kk/length(fitC),hw,'Running Neural Network Model...');
end
delete(hw)
plot(t,umeas*1e6,'bo')
plot(ti,ufun(ti)*1e6,'color',[0 0.6 0],'linewidth',lw)
plot(ti,fitC*1e6,'r','linewidth',lw)
set(gca,'box','on','fontsize',12)
axis tight
axis([-100 1000 -5 30])
title('Neural Network')
xlabel('Temperature (^oK)')
set(gcf,'paperpositionmode','auto','outerposition',[164 342 814 403])

%*********************************************************************
```

Appendix B

Chapter 3 Programs

B.1 GEP Algorithm

B.1.1 Call Structure

- reg_data.m
- setup_config.m
- gepfun.m
  - popgen.m
    - rand_idx.m
  - popexp.m
    - symconv.m
  - popeval.m
    - fitfun.m
  - fit_chk.m
  - nextgen.m
    - rep_fun.m
    - mutate_fun.m
    - recomb1_fun.m
    - recomb2_fun.m
    - recombG_fun.m
    - ist_fun.m
    - rist_fun.m
    - inv_fun.m
    - rnctrnsp_fun.m
    - rncinv_fun.m

B.1.2 reg_data.m

```
function reg = reg_data(fun,nsamp,range,nvar)
%**************************************************************************
% FUNCTION: reg = reg_data(fun,nsamp)
% INFO:     Generates regression data for model defined by @fun
% INPUT:    fun = model function (handles)
%           nsamp = number of samples to generate (scalar)
%           range = range to generate sample (vector)
% OUTPUT:   reg = regression data (struct)
% AUTHOR:   A. Hensley, 28-Nov-2012
% HISTORY:
%**************************************************************************
% Rev 1.0       28-Nov-2012       Hensley       Initial Release
%**************************************************************************

%Generate Input Samples
data = range(1)+diff(range)*rand(nsamp,nvar);

%Configure Output
reg.X = data;
reg.y = fun(data);
```

B.1.3 setup_config.m

```
function config = setup_config(parm)
%**********************************************************************
% FUNCTION: config = setup_config(nvars)
% INFO:     Sets up config structure for GEP algorithm
% INPUT:    parm = struct of parameters with the following fields:
%               .nvars = number of input variables (mandatory)
%               .genes = number of genes (optional, default = 10)
%               .headsize = chromosome headsize (optional, default = 10)
%               .rnc = random numerical constant flag (optional, default =
%                   false).
%               .popsize = population size (optional, default = 100);
%               .maxgen = maximum number of generations (optional, default
%                   = 100).
%               .convcrit = convergence criteria between 0 and 1 which is
%                   percent of maximum fitness (1000) required to declare
%                   convergence (optional, default = 0.999)
%               .selthr = minimum finess value for survival (optional,
%                   default = 1e-2).
%               .library = function library cell array (optional, default =
%                   {'+','-','*','/'}). Supported library functions are as
%                   follows:
%                       '+' = addition
%                       '-' = subtraction
%                       '*' = multiplication
%                       '/' = division
%                       'E' = exp()
%                       'L' = log()
%                       'S' = sin()
%                       'C' = cos()
%                       'T' = tan()
%                       'P' = ()^()
%                       'Q' = sqrt()
%               .founders = minimim number of survivors for generation 0.
%               .trials = number of times to run GEP algorithm.
%               .linkop = linking operation(s)
%               .mutate = mutation rate
%               .rncfun = numerical constants function handle
%               .fitfun = fitness function(default = mean-square)
%                       {'numhits','mean-square','r-square','complex-mse'}
%
% OUTPUT:   config = configuration struct for GEP algorithm
% AUTHOR:   A. Hensley, 11-Jan-2013
% HISTORY:
%**********************************************************************
% Rev 1.0       11-Jan-2013       Hensley       Initial Release
%**********************************************************************
```

```
%Set Defaults
nvars = nan;
genes = 10;
headsize = 10;
rnc = false;
popsize = 100;
maxgen = 100;
convcrit = 0.9999;
selthr = 1e-2;
library = {'+','-','*','/'};
trials = 1;
founders = 1;
linkop = '+';
mutate = 0.1;
rncfun = @(m,n)2*rand(m,n);
fitfun = 'mean-square';

%Extract Info From Parameter Struct
fields = fieldnames(parm);
for kk = 1:length(fields)

    cur = parm.(fields{kk});

    switch fields{kk}

        case 'nvars'
            nvars = cur;

        case 'genes'
            genes = cur;

        case 'headsize'
            if headsize<4
                error('Minimum Headsize is 4')
            end
            headsize = cur;

        case 'rnc'
            rnc = cur;

        case 'popsize'
            popsize = cur;

        case 'maxgen'
            maxgen = cur;

        case 'convcrit'
```

```
            convcrit = cur;

        case 'selthr'
            if selthr==0
                error('selthr must be > 0')
            end
            selthr = cur;

        case 'library'
            library = cur;

        case 'founders'
            founders = cur;

        case 'trials'
            trials = cur;

        case 'linkop'
            linkop = cur;

        case 'mutate'
            mutate = cur;

        case 'rncfun'
            rncfun = cur;

        case 'fitfun'
            fitfun = cur;

        otherwise
            error('Unrecognized Field >> setup_config.m')

    end
end

%Error Chk
if isnan(nvars)
    error('Number of variables is undefined >> setup_config.m')
end
if nvars>26
    errordlg('Unable to handle more than 26 preditor variables')
end

%Switches
config.switch.mutate = true;
config.switch.recomb1 = true;
config.switch.recomb2 = true;
config.switch.recombG = true;
```

```
config.switch.IST = true;
config.switch.RIST = true;
config.switch.geneT = false;
config.switch.inv = true;
config.switch.rnc = rnc;
config.switch.rnc_trnsp = true;
config.switch.rnc_inv = true;

%Setup Variables & Operators
config.nvars = nvars;
input_args = num2cell(char(97:97+nvars-1));
config.input_args = input_args;
if config.switch.rnc
    config.T = [input_args '?'];
else
    config.T = input_args;
end
config.F = library;
[config.Fmap,maxarg] = set_library(library,'#');
config.Ftemp = set_library(library,'$');
config.TF = [config.F config.T];
config.TFmap = [config.Fmap config.T];
config.TFtemp = [config.Ftemp config.T];
config.maxarg = maxarg;
config.arg = [cell2mat(input_args') repmat(',',size(input_args',1),1)]';
config.arg = config.arg(:)';
config.arg = config.arg(1:end-1);
config.empty = '#';
config.empty_temp = '$';

%Gene Preferences
config.genes = genes;
config.headsize = headsize;
config.tailsize = config.headsize*(config.maxarg-1)+1;
if config.switch.rnc
    config.constsize = config.tailsize;
else
    config.constsize = 0;
end
config.genesize = config.headsize+config.tailsize+config.constsize;
config.chrmsize = config.genesize*config.genes;
config.totalsize = config.chrmsize;
config.linkop = linkop;

%Make Chrm Map
H = repmat('H',1,config.headsize);
T = repmat('T',1,config.tailsize);
C = repmat('C',1,config.constsize);
```

```
config.chrm_map = repmat([H T C],1,config.genes);
config.gene_start = 1:config.genesize:config.chrmsize;

%Evolution Preferences
config.trials = trials;
config.popsize = popsize;
config.maxgen = maxgen;
config.mutate = mutate;
config.mutate_pts = 2;
config.recomb1 = 0.3;
config.recomb2 = 0.3;
config.recombG = 0.3;
config.IST_rate = 0.1;
config.ISE_set = 1:3;
config.RIST_rate = 0.1;
config.RIST_set = 1:3;
config.geneT_rate = 0.1;
config.inv_rate = 0.1;
config.inv_set = 2:4;
config.rnctrnsp_rate = 0.1;
config.rnctrnsp_set = 1:3;
config.rncinv_rate = 0.1;
config.rncinv_set = 2:4;
config.founder_pop = founders;

%Convergence/Selection Preferences
config.fitfun = fitfun; %{'numhits','mean-square','r-square','complex-mse'};
config.fitfunmax = 1000;
config.hit_prec = 0.01;
config.selthr = selthr;
config.convcrit = convcrit*config.fitfunmax;

%Constant Setup
config.rncfun = rncfun;
cnum = 10;
config.const_set = num2str(0:cnum-1);
config.const_set(isspace(config.const_set)) = [];
config.empty_const = '?';
if config.switch.rnc
    config.const = config.rncfun(cnum,config.genes);
    %onfig.const = config.crng(1) + config.crng(2)*randn(cnum,config.genes);
else
    config.const = [];
end

%Support Function
function [Fout,varargout] = set_library(Fin,chr)
```

```
N = length(Fin);
Fout = cell(1,N);
maxarg = 0;
for kk = 1:N
    cur = Fin{kk};
    switch cur

        case '+'
            Fout{kk} = ['(' chr '+' chr ')'];
            maxarg = max(maxarg,2);

        case '-'
            Fout{kk} = ['(' chr '-' chr ')'];
            maxarg = max(maxarg,2);

        case '*'
            Fout{kk} = ['(' chr '*' chr ')'];
            maxarg = max(maxarg,2);

        case '/'
            Fout{kk} = ['(' chr '/' chr ')'];
            maxarg = max(maxarg,2);

        case 'E'
            Fout{kk} = ['exp(' chr ')'];
            maxarg = max(maxarg,1);

        case 'L'
            Fout{kk} = ['log(' chr ')'];
            maxarg = max(maxarg,1);

        case 'S'
            Fout{kk} = ['sin(' chr ')'];
            maxarg = max(maxarg,1);

        case 'C'
            Fout{kk} = ['cos(' chr ')'];
            maxarg = max(maxarg,1);

        case 'T'
            Fout{kk} = ['tan(' chr ')'];
            maxarg = max(maxarg,1);

        case 'P'
            Fout{kk} = ['(2)^(' chr ')'];
            maxarg = max(maxarg,1);

        case 'Q'
```

```
            Fout{kk} = ['sqrt(abs(' chr '))'];
            maxarg = max(maxarg,1);

        case 'Y'
            Fout{kk} = ['real(sqrt(' chr '))'];
            maxarg = max(maxarg,1);

        case 'Z'
            Fout{kk} = ['sqrt(' chr ')'];
            maxarg = max(maxarg,1);

        case 'A'
            Fout{kk} = ['(' chr ')^2'];
            maxarg = max(maxarg,0);

        case 'B'
            Fout{kk} = 'const(2)';
            maxarg = max(maxarg,0);

        case 'D'
            Fout{kk} = 'const(3)';
            maxarg = max(maxarg,0);

        otherwise
            error('Unrecognized Function')
    end
end

if nargout>1
    varargout{1} = maxarg;
end
```

B.1.4 gepfun.m

```
function result = gepfun(X,y,config)
%*************************************************************************
% FUNCTION: result = gepfun(X,y,config)
% INFO:     Run GEP algorithm
% INPUT:    X = design matrix with colums corresponding to variables and
%               rows corresponding to observations.
%           y = response vector
%           config = configuration structure
% OUTPUT:   result = configuration struct for GEP algorithm
% AUTHOR:   A. Hensley, 11-Jan-2013
% HISTORY:
%*************************************************************************
% Rev 1.0       11-Jan-2013       Hensley       Initial Release
%*************************************************************************

%Storage
result.fit = zeros(1,config.trials);
result.fun = cell(1,config.trials);
result.it = zeros(1,config.trials);
result.expr = cell(1,config.trials);
result.mdl = cell(1,config.trials);
result.history = cell(1,config.trials);
result.lastgen = cell(1,config.trials);
result.lastfit = cell(1,config.trials);

for kk = 1:config.trials

    %Init Population
    for jj = 1:100
        gen = popgen(config.popsize,config);
        [fun,symb,parsimony] = popexp(gen,config);
        fit = popeval(fun,X,y,config);
        unfit = fit_chk(fit,config);
        %disp(fit)
        if sum(~unfit)>=config.founder_pop
           break
        end
        disp(['Founder Pop ' num2str(jj) ' Failed'])
    end
    if jj==100
        error('Unable to Init Founder Population')
    end

    %Setup
    history.fit = zeros(1,config.maxgen);
    history.fun = cell(1,config.maxgen);
```

```
history.std = zeros(1,config.maxgen);
converge = false;
hw = waitbar(0,'Initializing...');
figure,

for it = 1:config.maxgen;

    %Update
    if it>1
        [fun,symb,parsimony] = popexp(gen,config);
        fit = popeval(fun,X,y,config);
    end

    %Most Fit
    history.fit(it) = max(fit);
    history.std(it) = std(fit);
    mostfit = find(fit==max(fit),1);
    mdl = fun(mostfit);
    [~,mdl_eval] = popeval(mdl,X,y,config);

    %Update Figure
    subplot(2,2,1)
    plot(y),hold on,plot(mdl_eval,'r'),grid on,hold off
    title('Model')
    subplot(2,2,2),hold on
    plot(-parsimony,fit,'o','markersize',4),grid on
    set(gca,'box','on')
    subplot(2,2,3)
    plot(y-mdl_eval),grid on
    title('Error')

    %Current Solution
    syms a
    eval(['f = ' symb{mostfit} ';']);
    history.fun{kk} = vpa(expand(f),3);
    disp(history.fun{kk})
    disp(max(fit))

    %Update Waitbar
    waitbar(it/config.maxgen,hw,...
        ['Generation ' num2str(it) ' (Fitness: ' num2str(max(fit)) ')']);

    %Chk for Convergence
    if max(fit)>=config.convcrit
        converge = true;
        disp('Algorithm Converged')
        break
    end
```

```
        %Next Generation
        if it<config.maxgen
            gen = nextgen(gen,fit,config);
        end
    end
    delete(hw)
    close

    %Check for Convergence
    if ~converge
        disp('Algorithm Could Not Converge')
    end

    %Save Results/Simplfy Expression
    syms a
    eval(['f = ' symb{mostfit} ';']);
    result.expr{kk} = symb{mostfit};
    result.fit(kk) = max(fit);
    result.fun{kk} = vpa(expand(f),3);
    result.it(kk) = it;
    result.mdl{kk} = mdl;
    result.history{kk} = history;
    result.lastgen{kk} = gen;
    result.lastfit{kk} = fit;
    disp('Final Answer:')
    disp(result.expr{kk})
    disp(result.fun{kk})

end
```

B.1.5 popgen.m

```
function x = popgen(n,config)
%**********************************************************************
% FUNCTION: x = popgen(n,config)
% INFO:     Generates n candidtate solutions based config preferences
% INPUT:    n = number of candidate solutions to generate (scalar)
%           config = configuration parameters for GEP algorithm (struct)
% OUTPUT:   x = candiate solution chromosomes (cell array)
% AUTHOR:   A. Hensley, 28-Nov-2012
% HISTORY:
%**********************************************************************
% Rev 1.0      28-Nov-2012      Hensley      Initial Release
%**********************************************************************

%Init
x = cell(1,n);
for kk = 1:n
    for jj = 1:config.genes

        %Generate Head
        h = config.headsize;
        nh = length(config.TF);
        cur_head = [config.TF{rand_idx(nh,h)}];

        %Generate Tail
        t = config.tailsize;
        nt = length(config.T);
        cur_tail = [config.T{rand_idx(nt,t)}];

        %Generate Constants
        if config.switch.rnc
            nc = length(config.const_set);
            cur_const = config.const_set(rand_idx(nc,t));
        else
            cur_const = [];
        end

        %Assemble Chromosome
        x{kk} = [x{kk} cur_head cur_tail cur_const];
    end
end
```

B.1.6 rand_idx.m

```
function idx = rand_idx(m,varargin)
%**************************************************************************
% FUNCTION: idx = rand_idx(m)
% INFO:     Selects random index
% INPUT:    m = largest possible index (scalar)
%           n = number of indices to generate (scalar)
% OUTPUT:   idx = idicies (scalar/vector)
% AUTHOR:   A. Hensley, 28-Nov-2012
% HISTORY:
%**************************************************************************
% Rev 1.0       28-Nov-2012       Hensley       Initial Release
%**************************************************************************
if nargin>1
    n = varargin{1};
else
    n = 1;
end
idx = floor(1+m*rand(1,n));
```

B.1.7 popexp.m

```
function [f s parsimony] = popexp(x,config)
%***************************************************************************
% FUNCTION: f = popexp(x,config)
% INFO:     Converts chromosomes to functional expressions
% INPUT:    x = chromosomes (cell array)
%           config = configuration parameters for GEP algorithm (struct)
% OUTPUT:   f = function handles (cell array)
% AUTHOR:   A. Hensley, 28-Nov-2012
% HISTORY:
%***************************************************************************
% Rev 1.0       28-Nov-2012       Hensley       Initial Release
%***************************************************************************

%Setup
N = length(x);
f = cell(1,N);
s = cell(1,N);
parsimony = zeros(1,N);
for kk = 1:N

    %Segment/Reshape
    curG = reshape(x{kk}(1:config.chrmsize)',[],config.genes)';
    expG = cell(1,config.genes);

    for jj = 1:config.genes

        %Current Gene
        z = curG(jj,:);

        %Set Counter
        ii = 1;

        %Begin Gene Expression
        while ii<=config.headsize+config.tailsize

            %Lookup T/F
            cursym = symconv(z(ii),config);

            if ii==1

                %Init
                ft = ['@' cursym];
                ii = ii+1;

            else
```

```
            %Fill Empty Slots
            idxA = find(ft==config.empty,1);
            if ~isempty(idxA)
                ft = [ft(1:idxA-1) cursym ft(idxA+1:end)];
                ii = ii+1;
            else

                %Update Place Holders
                idxB = strfind(ft,config.empty_temp);
                if ~isempty(idxB)
                    ft(idxB) = config.empty;
                else
                   break
                end

            end
        end

    end
    parsimony(kk) = parsimony(kk)+ii-1;

    %Chk for Constants
    const_loc = strfind(ft,config.empty_const);
    if ~isempty(const_loc)
        c_idx = curG(jj,config.headsize+config.tailsize+1:end);
        c_val = config.const(:,jj);
        for uu = 1:length(const_loc)
            const_loc = const_loc(1);
            c = c_val(str2double(c_idx(uu))+1);
            c = ['(' num2str(c) ')'];
            ft = [ft(1:const_loc-1) c ft(const_loc+1:end)];
            const_loc = strfind(ft,'?');
        end
    end

    %Save Current Gene
    expG{jj} = ft(2:end);

end

%Assemble Final Expression
expL = [];
for ii = 1:config.genes-1
    expL = [expL expG{ii} config.linkop];
end
expL = [expL expG{end}];
f{kk} = eval(['@(' config.arg ')' vectorize(expL)]);
```

```
    s{kk} = expL;

    %Error Chk
    if nargin(f{kk})~=config.nvars
       error('Bad Expression Generated')
    end
end
```

B.1.8 symconv.m

```
function expr = symconv(sym,config)
%***************************************************************************
% FUNCTION: fexpr = symconv(sym,map)
% INFO:     Converts symbol to function/terminal using map
% INPUT:    sym = symbol to convert (char)
%           config = configuration parameters for GEP algorithm (struct)
% OUTPUT:   expr = function/terminal ready for evaluation
% AUTHOR:   A. Hensley, 28-Nov-2012
% HISTORY:
%***************************************************************************
% Rev 1.0       28-Nov-2012       Hensley       Initial Release
%***************************************************************************
idx = strcmp(sym,config.TF);
expr = config.TFmap{idx};
```

B.1.9 popeval.m

```
function [fit varargout] = popeval(fun,X,y,config)
%**********************************************************************
% FUNCTION: f = popeval(fun,X,y,config)
% INFO:     Evaluates fitness of current population
% INPUT:    fun = current population (cell array)
%           X = predictors (matrix)
%           y = response (vector)
%           config = configuration parameters for GEP algorithm (struct)
% OUTPUT:   f = fitness results(vector)
% AUTHOR:   A. Hensley, 28-Nov-2012
% HISTORY:
%**********************************************************************
% Rev 1.0       28-Nov-2012       Hensley       Initial Release
%**********************************************************************

%Error Chk
if nargout>1 && length(fun)>1
    error('Error in popeval: fun variable has too many elements')
end

%Assign Inputs
for jj = 1:length(config.input_args)
    eval([config.input_args{jj} '=X(:,jj);'])
end

%Setup Eval
N = length(fun);
fit = zeros(1,N);
implicitSearch = false;
if all(y==0)
    implicitSearch = true;
end

%Compute Fitness
for kk = 1:N
    eval(['cur = fun{kk}(' config.arg ');']);
    if imag(sum(cur))~=0
        fit(kk) = 0;
        continue
    end
    if implicitSearch
        fit(kk) = fitfun(X,fun{kk},config);
    else
        fit(kk) = fitfun(y,cur,config);
    end
```

```
end

%Handle Varargout
if nargout>1
    varargout{1} = cur;
end
```

B.1.10 fitfun.m

```
function f = fitfun(T,P,config)
%**************************************************************************
% FUNCTION: f = fitfun(T,P,type)
% INFO:     Executes chosen fitness function
% INPUT:    T = Target values (vector)
%           X = Predicted Values (matrix)
%           config = configuration parameters for GEP algorithm (struct)
% OUTPUT:   f = fitness score (vector)
% AUTHOR:   A. Hensley, 06-Dec-2012
% HISTORY:
%**************************************************************************
% Rev 1.0       06-Dec-2012       Hensley       Initial Release
%**************************************************************************

%Setup
K = config.fitfunmax;

%Implicit
if isa(P,'function_handle')

    %Only 2 Vars For Now
    if size(T,2)>2
        error('Implicit proc not configured for more than 2 vars')
    end

    dT = diff(T);
    x = T(1:end-1,1);
    dx = dT(:,1);
    y = T(1:end-1,2);
    dy = dT(:,2);
    dP_da = (P(x+dx,y)-P(x,y))./dx;
    dP_db = -(P(x,y+dy)-P(x,y))./dy;

    e1 = dT(:,2)./dT(:,1)-dP_da./dP_db;
    e2 = dT(:,1)./dT(:,2)-dP_db./dP_da;

    if all(isnan(e1))||all(isnan(e2))
        f = 0;
        return
    end

    f = K/(1+nansum(e1.^2)+nansum(e2.^2));

    if imag(f)~=0
        f = 0;
```

```
    end

    return
end

%Error Chk
if any(isnan(T)) || any(isnan(P))
    f = 0;
    return
end
if any(isinf(T)) || any(isinf(P))
    f = 0;
    return
end

%Explicit
switch config.fitfun

    case 'numhits'

        E = abs(T-P)<=config.hit_prec;
        f = sum(E/length(T))*K;

    case 'mean-square'

        E = mean((P-T).^2);

        if isreal(E)
            f = K./(1+E);
        else
            f = 0;
        end

    case 'r-square'

        R = corrcoef(T,P);
        f = R(2,1)^2*K;

    case 'complex-mse'

        Ei = mean((real(P)-real(T)).^2);
        Er = mean((imag(P)-imag(T)).^2);
        f = K./(1+Er+Ei);

    case 'complex-mse-mag'

        E = mean(abs(P-T).^2);
        f = K./(1+E);
```

```
    case 'hamming-mse'

        H = hamming(length(T));
        H = H/sum(H);
        E = H'*(abs(P-T).^2);
        f = K./(1+E);

    case 'kernel'

        N = length(T);

        if length(P)==1
            P = P*ones(N,1);
        end

        dT = diff(T(:));
        dP = diff(P(:));

        Kt = dT*(1./dT');
        Kp = dP*(1./dP');

        E = sqrt(nanmean(nansum((Kt-Kp).^2)));
        f = K/(1+E);

        if ~isreal(f)
            f = 0;
        end

    otherwise
        errordlg('Fitness Method Not Implemented')
end

r3 = @(z)round(z*1000)/1000;
if length(unique(r3(P)))==1;
    f = 0;
end
```

B.1.11 fit_chk.m

```
function unfit = fit_chk(fit,config)
%**************************************************************************
% FUNCTION: unfit = fit_chk(fit,config)
% INFO:     Checks fitness of current population.
% INPUT:    fit = current generation fitness (struct)
%           config = configuration parameters for GEP algorithm (struct)
% OUTPUT:   unfit = fitness mask (cell array)
% AUTHOR:   A. Hensley, 06-Dec-2012
% HISTORY:
%**************************************************************************
% Rev 1.0       06-Dec-2012       Hensley       Initial Release
%**************************************************************************
unfit = fit<config.selthr | isnan(fit) | isinf(fit) | imag(fit)~=0;
```

B.1.12 nextgen.m

```
function ngen = nextgen(gen,fit,config)
%**********************************************************************
% FUNCTION: ngen = nextgen(gen,fit,config)
% INFO:     Updates generation
% INPUT:    gen = current population (cell array)
%           fit = current generation fitness (struct)
%           config = configuration parameters for GEP algorithm (struct)
% OUTPUT:   ngen = next generation (cell array)
% AUTHOR:   A. Hensley, 28-Nov-2012
% HISTORY:
%**********************************************************************
% Rev 1.0      28-Nov-2012      Hensley      Initial Release
%**********************************************************************

%Elitism
mostfit = find(fit==max(fit),1);
safe = gen(mostfit);
leastfit = find(fit==min(fit),1);
gen(leastfit) = [];
fit(leastfit) = [];

%Replication
ngen = rep_fun(gen,fit,config);
clear gen

%Mutation
if config.switch.mutate
    ngen = mutate_fun(ngen,config);
end

%Recombination (1-pt)
if config.switch.recomb1
    ngen = recomb1_fun(ngen,config);
end

%Recombination (2-pt)
if config.switch.recomb2
    ngen = recomb2_fun(ngen,config);
end

%Recombination (Gene)
if config.switch.recombG
    ngen = recombG_fun(ngen,config);
end
```

```
%Insertion Sequence Transpose
if config.switch.IST
    ngen = ist_fun(ngen,config);
end

%Root Insertion Sequence Transpose
if config.switch.RIST
    ngen = rist_fun(ngen,config);
end

%Gene Transpose (NOT IMPLEMENTED)
% if config.switch.geneT
%     new = genet_fun(surv,config);
%     ngen = [ngen new];
% end

%Sequence Inversion
if config.switch.inv
    ngen = inv_fun(ngen,config);
end

%Constant Transpose
if config.switch.rnc_trnsp && config.switch.rnc
    ngen = rnctrnsp_fun(ngen,config);
end

%Constant Inversion
if config.switch.rnc_inv && config.switch.rnc
    ngen = rncinv_fun(ngen,config);
end

%Add Most Fit
ngen = [ngen safe];
```

B.1.13 rep_fun.m

```
function ngen = rep_fun(gen,fit,config)
%**********************************************************************
% FUNCTION: ngen = rep_fun(gen,fit,config)
% INFO:     Applies replication operator
% INPUT:    gen = current population (cell array)
%           fit = current generation fitness (struct)
%           config = configuration parameters for GEP algorithm (struct)
% OUTPUT:   ngen = next generation (cell array)
% AUTHOR:   A. Hensley, 06-Dec-2012
% HISTORY:
%**********************************************************************
% Rev 1.0       06-Dec-2012       Hensley       Initial Release
%**********************************************************************

%Prune
unfit = fit_chk(fit,config);
fitF = fit;
genF = gen;
fitF(unfit) = [];
genF(unfit) = [];

if isempty(fitF)
   disp('Population Extinct')
   return
end

%Replicate
fitpdf = fitF/sum(fitF);
fitcdf = [0 cumsum(fitpdf)];
fitsup = 0:length(fitpdf);
new = rand(1,length(fit));
idx = ceil(interp1(fitcdf,fitsup,new));
ngen = genF(idx);
```

B.1.14 mutate_fun.m

```
function ngen = mutate_fun(surv,config)
%**************************************************************************
% FUNCTION: new = mutate_fun(surv,config)
% INFO:     Applies mutation operator to survivors
% INPUT:    surv = survivor chromosomes (cell array)
%           config = configuration parameters for GEP algorithm (struct)
% OUTPUT:   ngen = updated generation (cell array)
% AUTHOR:   A. Hensley, 01-Dec-2012
% HISTORY:
%**************************************************************************
% Rev 1.0       01-Dec-2012       Hensley       Initial Release
%**************************************************************************

%Setup
N = length(surv);
nmut = round(config.mutate*N);
idx = rand_idx(N,nmut);
new = cell(1,nmut);

%Begin
for kk = 1:nmut

    %Determine Mutation Points
    cur = surv{idx(kk)};
    pts = rand_idx(length(cur),config.mutate_pts);
    type = config.chrm_map(pts);

    %Apply
    for jj = 1:length(pts)

        switch type(jj)

            case 'H'

                ii = rand_idx(length(config.TF));
                cur(pts(jj)) = config.TF{ii};

            case 'T'

                ii = rand_idx(length(config.T));
                cur(pts(jj)) = config.T{ii};

            case 'C'

                ii = rand_idx(length(config.const_set));
```

```
                cur(pts(jj)) = config.const_set(ii);

            otherwise

                error('Bad Chrm Map')

        end

    end

    %Update Output Variable
    new{kk} = cur;

end

%Update Population
ngen = surv;
ngen(idx) = new;
```

B.1.15 recomb1_fun.m

```
function ngen = recomb1_fun(surv,config)
%**************************************************************************
% FUNCTION: new = recomb1_fun(surv,config)
% INFO:     Applies 1-point recombination operator to survivors
% INPUT:    surv = survivor chromosomes (cell array)
%           config = configuration parameters for GEP algorithm (struct)
% OUTPUT:   new = recombined chromosomes (cell array)
% AUTHOR:   A. Hensley, 01-Dec-2012
% HISTORY:
%**************************************************************************
% Rev 1.0       01-Dec-2012       Hensley       Initial Release
%**************************************************************************

%Setup
N = length(surv);
nrecomb = 2*round(config.recomb1*N);
[~,idx] = sort(rand(length(surv),1));
idx = idx(1:nrecomb);
new = cell(1,nrecomb);

%Begin
for kk = 1:2:nrecomb

    %Select Parents
    parentA = surv{idx(kk)};
    parentB = surv{idx(kk+1)};

    %Recombination
    xpt = rand_idx(config.totalsize,1);
    childA = [parentA(1:xpt) parentB(xpt+1:end)];
    childB = [parentB(1:xpt) parentA(xpt+1:end)];

    %Update Output Variable
    new{kk} = childA;
    new{kk+1} = childB;

end

%Update
ngen = surv;
ngen(idx) = new;
```

B.1.16 recomb2_fun.m

```
function ngen = recomb2_fun(surv,config)
%**************************************************************************
% FUNCTION: new = recomb2_fun(surv,config)
% INFO:     Applies 2-point recombination operator to survivors
% INPUT:    surv = survivor chromosomes (cell array)
%           config = configuration parameters for GEP algorithm (struct)
% OUTPUT:   new = recombined chromosomes (cell array)
% AUTHOR:   A. Hensley, 01-Dec-2012
% HISTORY:
%**************************************************************************
% Rev 1.0       01-Dec-2012       Hensley       Initial Release
%**************************************************************************

%Setup
N = length(surv);
nrecomb = 2*round(config.recomb2*N);
[~,idx] = sort(rand(length(surv),1));
idx = idx(1:nrecomb);
new = cell(1,nrecomb);

%Begin
for kk = 1:2:nrecomb

    %Select Parents
    parentA = surv{idx(kk)};
    parentB = surv{idx(kk+1)};

    %Recombination
    xpt = sort(rand_idx(config.totalsize,2));
    childA = [parentA(1:xpt(1)) parentB(xpt(1)+1:xpt(2)) parentA(xpt(2)+1:end)];
    childB = [parentB(1:xpt(1)) parentA(xpt(1)+1:xpt(2)) parentB(xpt(2)+1:end)];

    %Update Output Variable
    new{kk} = childA;
    new{kk+1} = childB;

end

%Update
ngen = surv;
ngen(idx) = new;
```

B.1.17 recombG_fun.m

```
function ngen = recombG_fun(surv,config)
%**************************************************************************
% FUNCTION: new = recombG_fun(surv,config)
% INFO:     Applies gene recombination operator to survivors
% INPUT:    surv = survivor chromosomes (cell array)
%           config = configuration parameters for GEP algorithm (struct)
% OUTPUT:   ngen = updated generation (cell array)
% AUTHOR:   A. Hensley, 02-Dec-2012
% HISTORY:
%**************************************************************************
% Rev 1.0       02-Dec-2012       Hensley       Initial Release
%**************************************************************************

%Setup
N = length(surv);
nrecomb = 2*round(config.recombG*N);
[~,idx] = sort(rand(length(surv),1));
idx = idx(1:nrecomb);
new = cell(1,nrecomb);

%Begin
for kk = 1:2:nrecomb

    %Select Parents
    parentA = surv{idx(kk)};
    parentB = surv{idx(kk+1)};

    %Recombination
    temp = config.gene_start(rand_idx(config.genes,1));
    xpt = [temp temp+config.genesize];
    childA = [parentA(1:xpt(1)-1) parentB(xpt(1):xpt(2)-1) parentA(xpt(2):end)];
    childB = [parentB(1:xpt(1)-1) parentA(xpt(1):xpt(2)-1) parentB(xpt(2):end)];

    %Update Output Variable
    new{kk} = childA;
    new{kk+1} = childB;

end

%Update
ngen = surv;
ngen(idx) = new;
```

B.1.18 ist_fun.m

```
function ngen = ist_fun(surv,config)
%**************************************************************************
% FUNCTION: new = ist_fun(surv,config)
% INFO:     Applies insertion sequence transpose operator to survivors
% INPUT:    surv = survivor chromosomes (cell array)
%           config = configuration parameters for GEP algorithm (struct)
% OUTPUT:   new = transposed chromosomes (cell array)
% AUTHOR:   A. Hensley, 02-Dec-2012
% HISTORY:
%**************************************************************************
% Rev 1.0       02-Dec-2012       Hensley       Initial Release
%**************************************************************************

%Setup
N = length(surv);
ntransp = round(config.IST_rate*N);
[~,idx] = sort(rand(length(surv),1));
idx = idx(1:ntransp);
new = cell(1,ntransp);
setsize = length(config.ISE_set);

%Begin
for kk = 1:ntransp

    %Select Chromosome
    cur = surv{idx(kk)};

    %Select Gene
    gene_idx = rand_idx(config.genes);
    a = config.gene_start(gene_idx);
    gene_mask = a:a+config.headsize+config.tailsize-1;
    gene = cur(gene_mask);

    %Select Transposon
    mask = (1:rand_idx(setsize,1))-1;
    head_tail = config.headsize+config.tailsize;
    pnt = rand_idx(head_tail-length(mask)+1);
    is = gene(pnt+mask);

    %Select Insertion Point (Not allowed to be 1)
    ins = rand_idx(config.headsize-1)+1;

    %Apply
    head = gene(1:config.headsize);
    headT = [head(1:ins-1) is head(ins:end)];
    headT = headT(1:config.headsize);
```

```
    %Update Chromosome
    gene(1:config.headsize) = headT;
    cur(gene_mask) = gene;
    new{kk} = cur;

end

%Update
ngen = surv;
ngen(idx) = new;
```

B.1.19 rist_fun.m

```
function ngen = rist_fun(surv,config)
%**************************************************************************
% FUNCTION: new = rist_fun(surv,config)
% INFO:     Applies root insertion sequence transpose operator to survivors
% INPUT:    surv = survivor chromosomes (cell array)
%           config = configuration parameters for GEP algorithm (struct)
% OUTPUT:   new = transposed chromosomes (cell array)
% AUTHOR:   A. Hensley, 02-Dec-2012
% HISTORY:
%**************************************************************************
% Rev 1.0       02-Dec-2012       Hensley       Initial Release
%**************************************************************************

%Setup
N = length(surv);
ntransp = round(config.RIST_rate*N);
[~,idx] = sort(rand(length(surv),1));
idx = idx(1:ntransp);
new = cell(1,ntransp);
setsize = length(config.RIST_set);

%Begin
for kk = 1:ntransp

    %Select Chromosome
    cur = surv{idx(kk)};

    %Select Gene
    gene_idx = config.gene_start(rand_idx(config.genes));
    head_mask = gene_idx:gene_idx+config.headsize-1;
    head = cur(head_mask);

    %Select Insertion Point (Not allowed to be 1)
    pnt = rand_idx(config.headsize-1,1)+1;

    %Find Next Function
    temp = head;
    temp(1:pnt-1)='#';
    pnt = find(ismember(temp,cell2mat(config.F)),1);
    if isempty(pnt)
        new{kk} = cur;
        continue
    end

    %Get Transposon
```

```
    mask = pnt+((1:rand_idx(setsize,1))-1);
    mask(mask>config.headsize) = [];
    ris = head(mask);

    %Apply
    headT = [ris head];
    headT = headT(1:config.headsize);

    %Update Output Variable
    cur(head_mask) = headT;
    new{kk} = cur;

end

%Update
ngen = surv;
ngen(idx) = new;
```

B.1.20 inv_fun.m

```
function ngen = inv_fun(surv,config)
%*************************************************************************
% FUNCTION: new = inv_fun(surv,config)
% INFO:     Applies inversion operator to survivors
% INPUT:    surv = survivor chromosomes (cell array)
%           config = configuration parameters for GEP algorithm (struct)
% OUTPUT:   ngen = updated generation (cell array)
% AUTHOR:   A. Hensley, 02-Dec-2012
% HISTORY:
%*************************************************************************
% Rev 1.0       02-Dec-2012       Hensley       Initial Release
%*************************************************************************

%Setup
N = length(surv);
ninv = round(config.inv_rate*N);
[~,idx] = sort(rand(length(surv),1));
idx = idx(1:ninv);
new = cell(1,ninv);
setsize = length(config.inv_set);

%Begin
for kk = 1:ninv

    %Select Chromosome
    cur = surv{idx(kk)};

    %Select Gene
    gene_idx = rand_idx(config.genes);
    a = config.gene_start(gene_idx);
    head_mask = a:a+config.headsize;
    head = cur(head_mask);

    %Select Inversion Sequence
    seq_str = 1;
    seq_end = config.inv_set(rand_idx(setsize));
    mask = seq_str:seq_end;
    mask(mask>length(head)) = [];

    %Apply
    head(mask) = fliplr(head(mask));

    %Update Chromosome
    cur(head_mask) = head;
    new{kk} = cur;
```

```
end

%Update
ngen = surv;
ngen(idx) = new;
```

B.1.21 rnctrnsp_fun.m

```
function ngen = rnctrnsp_fun(surv,config)
%***************************************************************************
% FUNCTION: ngen = rnctrnsp_fun(surv,config)
% INFO:     Applies transpose to constants domina
% INPUT:    surv = survivor chromosomes (cell array)
%           config = configuration parameters for GEP algorithm (struct)
% OUTPUT:   new = transposed chromosomes (cell array)
% AUTHOR:   A. Hensley, 02-Dec-2012
% HISTORY:
%***************************************************************************
% Rev 1.0       02-Dec-2012       Hensley       Initial Release
%***************************************************************************

%Setup
N = length(surv);
ntransp = round(config.rnctrnsp_rate*N);
[~,idx] = sort(rand(length(surv),1));
idx = idx(1:ntransp);
new = cell(1,ntransp);
setsize = length(config.rnctrnsp_set);

%Begin
for kk = 1:ntransp

    %Select Chromosome
    cur = surv{idx(kk)};

    %Select Gene
    gene_idx = config.gene_start(rand_idx(config.genes));
    a = gene_idx+config.headsize+config.tailsize;
    b = a+config.constsize-1;
    const_mask = a:b;
    const = cur(const_mask);

    %Select Insertion Point (Not allowed to be 1)
    pnt = rand_idx(config.constsize-1,1)+1;

    %Get Transposon
    mask = pnt+((1:rand_idx(setsize,1))-1);
    mask(mask>config.constsize) = [];
    seq = const(mask);

    %Apply
    constT = [seq const];
    constT = constT(1:config.constsize);
```

```
    %Update Output Variable
    cur(const_mask) = constT;
    new{kk} = cur;

end

%Update
ngen = surv;
ngen(idx) = new;
```

B.1.22 rncinv_fun.m

```
function ngen = rncinv_fun(surv,config)
%*************************************************************************
% FUNCTION: ngen = rncinv_fun(surv,config)
% INFO:     Applies inversion to constants domain
% INPUT:    surv = survivor chromosomes (cell array)
%           config = configuration parameters for GEP algorithm (struct)
% OUTPUT:   new = transposed chromosomes (cell array)
% AUTHOR:   A. Hensley, 02-Dec-2012
% HISTORY:
%*************************************************************************
% Rev 1.0       02-Dec-2012       Hensley       Initial Release
%*************************************************************************

%Setup
N = length(surv);
ninv = round(config.rncinv_rate*N);
[~,idx] = sort(rand(length(surv),1));
idx = idx(1:ninv);
new = cell(1,ninv);
setsize = length(config.rncinv_set);

%Begin
for kk = 1:ninv

    %Select Chromosome
    cur = surv{idx(kk)};

    %Select Gene
    gene_idx = config.gene_start(rand_idx(config.genes));
    a = gene_idx+config.headsize+config.tailsize;
    b = a+config.constsize-1;
    const_mask = a:b;
    const = cur(const_mask);

    %Select Insertion Point (Not allowed to be 1)
    pnt = rand_idx(config.constsize,1);

    %Get Sequence
    seq_len = config.rncinv_set(rand_idx(setsize));
    invmask = pnt+(1:seq_len);
    invmask(invmask>config.constsize) = [];
    seq = const(invmask);

    %Apply
    const_inv = const;
```

```
    const_inv(invmask) = fliplr(const_inv(invmask));

    %Update Output Variable
    cur(const_mask) = const_inv;
    new{kk} = cur;

end

%Update
ngen = surv;
ngen(idx) = new;
```

B.2 Experiment 3-1: Viscosity of Hydrogen

```
%Experiment 3-1: Viscosity of Hydrogen
%************************************************************************
%Setup
close all hidden
clc

%Sutherland's Viscosity Model (Hydrogen)
lambda = 0.636236562e-6;
C = 72;
ufun = @(T)lambda*T.^(3/2)./(T+C);

%Generate Data Set
nsamp = 200;
Tmax = 555;
Tmin = 0;
nvars = 1;
reg = reg_data(ufun,nsamp,[Tmin Tmax],nvars);
n = 1e-6*randn(nsamp,1);
reg.y = reg.y+n;
reg.y = reg.y*1e6;

figure,plot(reg.X,reg.y,'.')
[~,I] = sort(reg.X);
hold on,plot(reg.X(I),ufun(reg.X(I))*1e6,'r')

%Configure Algorithm
N = 25;
parm.nvars = nvars;
parm.genes = 2;
parm.library = {'+','-','*','/','Q'};
parm.selthr = 10;
parm.maxgen = 100;
parm.trials = N;
parm.popsize = 200;
parm.headsize = 6;
parm.rnc = true;
parm.convcrit = floor(1000/(1+mean((n*1e6).^2)))/1000;
parm.rncfun = @(m,n)rand(m,n);

%Run
config = setup_config(parm);
result = gepfun(reg.X,reg.y,config);
save Experiment3_1.mat
%************************************************************************
```

Appendix C

Chapter 4 Programs

C.1 Experiment 4-1: A Simple Equation

```
%Experiment 4-1: A Simple Equation
%****************************************************************

%Make Data Set
fc = @(x)1./(1+x.^2);
nsamp = 200;
xmax = 10;
xmin = -10;
nvars = 1;
reg = reg_data(fc,nsamp,[xmin xmax],nvars);
reg.y = reg.y;

%Save to MAT File
save simple.mat

%Configure Algorithm
N = 10;
parm.nvars = nvars;
parm.genes = 1;
parm.library = {'+','-','*','/'};
parm.selthr = 10;
parm.maxgen = 100;
parm.trials = N;
parm.popsize = 200;
parm.headsize = 10;
parm.rnc = false;
parm.convcrit = 0.999;
parm.rncfun = @(m,n)2*rand(m,n);

%Run
config = setup_config(parm);
result = gepfun(reg.X,reg.y,config);
save Experiment4_1.mat

%****************************************************************
```

C.2 Experiment 4-2: A Simple Equation + Noise

```
%Experiment 4-2: A Simple Equation + Noise
%*****************************************************************

%Load Data Set
load simple.mat

%Add Noise
n = 0.1*randn(nsamp,1);
reg.y = reg.y+n;

%Configure Algorithm
N = 10;
parm.nvars = nvars;
parm.genes = 1;
parm.library = {'+','-','*','/'};
parm.selthr = 10;
parm.maxgen = 100;
parm.trials = N;
parm.popsize = 200;
parm.headsize = 10;
parm.rnc = false;
parm.convcrit = floor(1000/(1+mean(n.^2)))/1000;
parm.rncfun = @(m,n)2*rand(m,n);

%Run
config = setup_config(parm);
result = gepfun(reg.X,reg.y,config);
save Experiment4_2.mat

%*****************************************************************
```

C.3 Experiment 4-3: Unnecessary Constants

```
%Experiment 4-3: Unnecessary Constants
%**************************************************************

%Load Data Set
load simple.mat

%Add Noise
n = 0.1*randn(nsamp,1);
reg.y = reg.y+n;

%Configure Algorithm
N = 10;
parm.nvars = nvars;
parm.genes = 1;
parm.library = {'+','-','*','/'};
parm.selthr = 10;
parm.maxgen = 100;
parm.trials = N;
parm.popsize = 200;
parm.headsize = 10;
parm.rnc = true;
parm.convcrit = floor(1000/(1+mean(n.^2)))/1000;
parm.rncfun = @(m,n)2*rand(m,n);

%Run
config = setup_config(parm);
result = gepfun(reg.X,reg.y,config);
save Experiment4_3.mat

%**************************************************************
```

C.4 Experiment 4-4: Necessary Constants

```
%Experiment 4-4: Necessary Constants
%****************************************************************

%Load Data Set
load simple.mat

%Add Noise & Constant
n = 0.1*randn(nsamp,1);
reg.y = pi*reg.y+n;

%Configure Algorithm
N = 10;
parm.nvars = nvars;
parm.genes = 1;
parm.library = {’+’,’-’,’*’,’/’};
parm.selthr = 10;
parm.maxgen = 100;
parm.trials = N;
parm.popsize = 200;
parm.headsize = 10;
parm.rnc = true;
parm.convcrit = floor(1000/(1+mean(n.^2)))/1000;
parm.rncfun = @(m,n)2*rand(m,n);

%Run
config = setup_config(parm);
result = gepfun(reg.X,reg.y,config);
save Experiment4_4.mat

%****************************************************************
```

C.5 Experiment 4-5: Information Removal

```
%Experiment 4-5: Information Removal
%*****************************************************************

%Load Data Set
load simple.mat

%Setup Loop
nsamp = 200;
xmax = 10;
xmin = [-8,-6,-4,-2,-1,0,1,2,4,6,8];
ext = 'abcdefghijk';
nvars = 1;

%Configure Algorithm
N = 10;
parm.nvars = nvars;
parm.genes = 1;
parm.library = {'+','-','*','/'};
parm.selthr = 10;
parm.maxgen = 200;
parm.trials = N;
parm.popsize = 200;
parm.headsize = 10;
parm.rnc = true;
parm.rncfun = @(m,n)2*rand(m,n);

%Loop
for kk = 1:length(xmin)

    %Reconfigure Input Data
    reg = reg_data(fc,nsamp,[xmin(kk) xmax],nvars);
    n = 0.1*randn(nsamp,1);
    reg.y = pi*reg.y+n;

    %Run Algorithm
    parm.convcrit = floor(1000/(1+mean(n.^2)))/1000;
    config = setup_config(parm);
    result = gepfun(reg.X,reg.y,config);

    %Save Results
    save(['Experiment8' ext(kk) '.mat'])

end

%NOTES
```

```
% a: [-8 10]
% b: [-6 10]
% c: [-4 10]
% d: [-2 10]
% e: [-1 10]
% f: [0 10]
% g: [1 10]
% h: [2 10]
% i: [4 10]
% j: [6 10]
% k: [8 10]

%**************************************************************
```

C.6 Experiment 4-6: Simplified Viscosity of Hydrogen

```
%Experiment 4-6: Simplified Viscosity of Hydrogen
%*****************************************************************

%Sutherland's Viscosity Model (Hydrogen)
lambda = 0.636236562e-6; C = 1;
ufun = @(T)lambda*T.^(3/2)./(T+C);

%Generate Data Set
nsamp = 200;
Tmin = -500000; Tmax = 1000000;
nvars = 1;
reg.X = linspace(Tmin,Tmax,nsamp)';
reg.y = real(ufun(reg.X));
sigm = 1e-7;
n = sigm*randn(length(reg.X),1);
reg.y = reg.y+n;
reg.y = reg.y*1e6;

%Configure Algorithm
N = 10;
parm.nvars = nvars;
parm.genes = 1;
parm.library = {'+','-','*','/','Y'};
parm.selthr = 1;
parm.maxgen = 200;
parm.trials = N;
parm.popsize = 200;
parm.headsize = 10;
parm.rnc = true;
parm.convcrit = floor(1000/( 1+mean(real(n*1e6).^2)+...
    mean(imag(n*1e6).^2) ))/1000;
parm.rncfun = @(m,n)2*rand(m,n);

%Run
config = setup_config(parm);
result = gepfun(reg.X,reg.y,config);
save Experiment4_6.mat

%*****************************************************************
```

C.7 Experiment 4-7: Imaginary Viscosity of Hydrogen

```
%Experiment 4-7: Imaginary Viscosity of Hydrogen
%*****************************************************************

%Sutherland's Viscosity Model (Hydrogen)
lambda = 0.636236562e-6; C = 1;
ufun = @(T)lambda*T.^(3/2)./(T+C);

%Generate Data Set
nsamp = 200;
Tmin = -20; Tmax = 20;
nvars = 1;
reg.X = linspace(Tmin,Tmax,nsamp)';
reg.y = ufun(reg.X);
j = sqrt(-1); sigm = 1e-7;
n = sigm*(randn(length(reg.X),1)+j*randn(length(reg.X),1));
reg.y = reg.y+n;
reg.y = reg.y*1e6;

%Configure Algorithm
N = 10;
parm.nvars = nvars;
parm.genes = 1;
parm.library = {'+','-','*','/','Z'};
parm.selthr = 1;
parm.maxgen = 200;
parm.trials = N;
parm.popsize = 200;
parm.headsize = 10;
parm.rnc = true;
parm.convcrit = floor(1000/( 1+mean(real(n*1e6).^2)+...
    mean(imag(n*1e6).^2) ))/1000;
parm.rncfun = @(m,n)2*rand(m,n);
parm.fitfun = 'complex-mse';

%Run
config = setup_config(parm);
result = gepfun(reg.X,reg.y,config);
save Experiment4_7.mat

%*****************************************************************
```

Appendix D

Chapter 5 Programs

D.1 Experiment 5-1: Sensitivity Analysis

```
%Experiment 5-1: Sensitivity Analysis
%**************************************************************************

%Sutherland’s Viscosity Model (Hydrogen)
lambda = 0.636236562;
C = 72;
ufun = @(T)lambda*T.^(3/2)./(T+C);

%Generate Data Set
nsamp = 200;
Tmax = 1e5;
Tmin = 1;
nvars = 1;
reg.X = linspace(Tmin,Tmax,nsamp)’;
reg.y = ufun(reg.X);

%Configure Algorithm
N = 10;
parm.nvars = nvars;
parm.genes = 1;
parm.library = {’+’,’-’,’*’,’/’,’Z’};
parm.selthr = 1;
parm.maxgen = 200;
parm.trials = N;
parm.popsize = 200;
parm.headsize = 10;
parm.rnc = true;
parm.rncfun = @(m,n)100*rand(m,n);
parm.fitfun = ’mean-square’;

%Run
config = setup_config(parm);
result = gepfun(reg.X,reg.y,config);
save Experiment5_1.mat

%Feature Space Mapping
mdlEval = zeros(nsamp,N);
err = zeros(1,N);
K1 = (diff(reg.y)*(1./diff(reg.y)’));
K1 = K1-eye(size(K1))+diag(diff(reg.y));
for kk = 1:N
    mdlEval(:,kk) = result.mdl{kk}{1}(reg.X);
    K2 = diff(mdlEval(:,kk))*(1./diff(mdlEval(:,kk))’);
    K1 = K2-eye(size(K2))+diag(diff(mdlEval(:,kk)));
    err(kk) = sqrt(mean(sum((K1-K2).^2)));
```

```
end
new = 1000./(1+err);

%Plot Results
figure,
[~,I] = sort(result.fit,'descend');
plot(result.fit(I),'ro-'),hold on
plot(new(I),'bo-')
set(gcf,'paperpositionmode','auto','outerposition',[360 362 628 386])
set(gca,'fontsize',14)
xlabel('Solution')
ylabel('Fitness')
grid on
legend('Mean Square Error','Feature Space Dist')

%***************************************************************************
```

D.2 Experiment 5-2: Feature Space Learning

```
%Experiment 5-2: Feature Space Learning
%***************************************************************************

%Sutherland's Viscosity Model (Hydrogen)
lambda = 0.636236562;
C = 72;
ufun = @(T)lambda*T.^(3/2)./(T+C);

%Generate Data Set
nsamp = 200;
Tmax = 1e5;
Tmin = 1;
nvars = 1;
reg.X = linspace(Tmin,Tmax,nsamp)';
reg.y = ufun(reg.X);

%Configure Algorithm
N = 10;
parm.nvars = nvars;
parm.genes = 1;
parm.library = {'+','-','*','/','Z'};
parm.selthr = 1;
parm.maxgen = 200;
parm.trials = N;
parm.popsize = 200;
parm.headsize = 10;
parm.rnc = true;
parm.rncfun = @(m,n)100*rand(m,n);
parm.fitfun = 'kernel';

%Run
config = setup_config(parm);
result = gepfun(reg.X,reg.y,config);
save Experiment5_2.mat

%***************************************************************************
```

Appendix E

Chapter 6 Programs

E.1 Analytic Experiments

```
function AnalyticExperiments()
%**********************************************************************
%Analytic Experiments Master Function
%**********************************************************************
%Setup
close all
clc
rng('shuffle')

%Run Experiments
AnalyticExperimentsSub('poly','mean-square')
AnalyticExperimentsSub('poly','kernel')
AnalyticExperimentsSub('rat','mean-square')
AnalyticExperimentsSub('rat','kernel')
AnalyticExperimentsSub('trig','mean-square')
AnalyticExperimentsSub('trig','kernel')
AnalyticExperimentsSub('trig-taylor','mean-square')
AnalyticExperimentsSub('trig-taylor','kernel')
AnalyticExperimentsSub('log','mean-square')
AnalyticExperimentsSub('log','kernel')
AnalyticExperimentsSub('log-taylor','mean-square')
AnalyticExperimentsSub('log-taylor','kernel')
AnalyticExperimentsSub('exp','mean-square')
AnalyticExperimentsSub('exp','kernel')
AnalyticExperimentsSub('exp-taylor','mean-square')
AnalyticExperimentsSub('exp-taylor','kernel')


function AnalyticExperimentsSub(type,fitfun)
%**********************************************************************
%Analytic Experiments Sub-Function
%**********************************************************************

%Preferences
fset = {'+','-','*','/'};
hsize = 10;
nsamp = 50;
nvars = 1;
legloc = 'northwest';

switch type

    case 'poly'

        f = @(x)x.^3-4*x;
```

```
    trnDomain = [-1,+3];
    pltDomain = [-2,+4];
    fstr = 'x^3-4x';

case 'rat'

    f = @(x)(0.5*x.^3-5)./(x.^2+1);
    trnDomain = [-5,0];
    pltDomain = [-6,+5];
    fstr = '(0.5x^3-5)/(x^2+1)';

case 'trig'

    f = @(x)98.6*sin(x);
    fset = {'+','-','*','/','S','C'};
    trnDomain = [0,3*pi/2];
    pltDomain = [-pi,+2*pi];
    fstr = '98.6sin(x)';

case 'trig-taylor'

    f = @(x)98.6*sin(x);
    trnDomain = [0,3*pi/2];
    pltDomain = [-pi,+2*pi];
    fstr = '98.6sin(x)';
    hsize = 20;

case 'log'

    f = @(x)log(7846*(1+x));
    fset = {'+','-','*','/','L'};
    trnDomain = [-0.8,+1];
    pltDomain = [-1,+2];
    fstr = 'log(7846(1+x))';

case 'log-taylor'

    f = @(x)log(7846*(1+x));
    trnDomain = [-0.8,+1];
    pltDomain = [-1,+2];
    fstr = 'log(7846(1+x))';
    hsize = 20;

case 'exp'

    f = @(x)exp(-2*(x-1));
    fset = {'+','-','*','/','E'};
    trnDomain = [-1.5,0];
```

```
        pltDomain = [-1.7,+1];
        fstr = 'exp(-2(x-1))';
        legloc = 'northeast';

    case 'exp-taylor'

        f = @(x)exp(-2*(x-1));
        trnDomain = [-1.5,0];
        pltDomain = [-1.7,+1];
        fstr = 'exp(-2(x-1))';
        legloc = 'northeast';
        hsize = 20;

    otherwise
        error('Bad Function Type')

end

%Generate Data Set
trnX = linspace(trnDomain(1),trnDomain(2),nsamp)';
pltX = linspace(pltDomain(1),pltDomain(2),nsamp)';
reg.X = trnX;
reg.y = f(reg.X);

%Plot Data Set
figure,
plot(pltX,f(pltX),'k--')
hold on
plot(reg.X,reg.y,'.-'),hold on
set(gca,'fontsize',14)
xlabel('x'),ylabel('f(x)')
legend(fstr,'Training Data','location',legloc)
set(gcf,'paperpositionmode','auto','outerposition',[360 362 628 386])
grid on
axis tight
drawnow
print(gcf,'-depsc',['Figure-AnalyticExp-' type '-TrnData.eps'])
close

%Configure Algorithm
N = 10;
parm.nvars = nvars;
parm.genes = 1;
parm.library = fset;
parm.selthr = 1;
parm.maxgen = 250;
parm.trials = N;
parm.popsize = 200;
```

```
parm.headsize = hsize;
parm.rnc = true;
parm.rncfun = @(m,n)100*rand(m,n);
parm.fitfun = fitfun;

%Run Algorithm
config = setup_config(parm);
result = gepfun(reg.X,reg.y,config);

%Save Results
save([type '-' fitfun '.mat'])
```

E.2 Real Experiments

```
function RealExperiments()
%**************************************************************************
%Real Experiments Master Function
%**************************************************************************
%Setup
close all
clc
rng('shuffle')

%Run Experiments
RealExperimentsSub('case12','mean-square')
RealExperimentsSub('case12','kernel')
RealExperimentsSub('case46a','mean-square')
RealExperimentsSub('case46a','kernel')
RealExperimentsSub('case50','mean-square')
RealExperimentsSub('case50','kernel')
RealExperimentsSub('case56b','mean-square')
RealExperimentsSub('case56b','kernel')
RealExperimentsSub('case105','mean-square')
RealExperimentsSub('case105','kernel')
RealExperimentsSub('case161a','mean-square')
RealExperimentsSub('case161a','kernel')
RealExperimentsSub('case171a','mean-square')
RealExperimentsSub('case171a','kernel')


function RealExperimentsSub(type,fitfun)
%**************************************************************************
%Real Experiments Sub-Function
%**************************************************************************

%Preferences
fset = {'+','-','*','/','Z','E'};
hsize = 10;
nvars = 1;
legloc = 'northwest';
if ismember(type,{'case50','case105','161a'})
    legloc = 'northeast';
end

%Prep Data Set
data = getRealData();
reg.X = data.(type)(4:end-3,1);
reg.y = data.(type)(4:end-3,2);
test.X = [data.(type)(1:3,1); data.(type)(end-2:end,1)];
```

```
test.y = [data.(type)(1:3,2); data.(type)(end-2:end,2)];

%Plot Data Set
figure,
plot(reg.X,reg.y,'b.')
hold on
plot(test.X,test.y,'ro'),hold on
set(gca,'fontsize',14)
xlabel('x'),ylabel('f(x)')
legend('Training Data','Test Data','location',legloc)
set(gcf,'paperpositionmode','auto','outerposition',[360 362 628 386])
grid on
axis tight
drawnow
print(gcf,'-depsc',['Figure-PhysicsExp-' type '-Data.eps'])
close

%Configure Algorithm
N = 10;
parm.nvars = nvars;
parm.genes = 1;
parm.library = fset;
parm.selthr = 1;
parm.maxgen = 250;
parm.trials = N;
parm.popsize = 200;
parm.headsize = hsize;
parm.rnc = true;
parm.rncfun = @(m,n)100*rand(m,n);
parm.fitfun = fitfun;

%Run Algorithm
config = setup_config(parm);
result = gepfun(reg.X,reg.y,config);

%Save Results
save([type '-' fitfun '.mat'])
```

E.3 Real Experiment Data Sets

```
function data = getRealData()
%*********************************************************************
% Experimental Data Sets
% Source: UCI Machine Learning Repository
%*********************************************************************
% Data Sets:
% 1  CASE 46A: Vapor pressure of bromine
% 2  CASE 50: Thermal Conductivity of Air at Low Pressures
% 3  CASE 56B: Emission of electrons from heated tantulum
% 4  CASE 105: Magnetic flux after torsion
% 5  CASE 161A: Index of refraction of ethyl alcohol
% 6  CASE 171A: Resistance vs. Centigrade temperature
%*********************************************************************
% CASE 46a:
% Source: National Research Council of the United States of America,
% {\em International Critical Tables of Numerical Data: Physics,
%     Chemistry and Technology}, McGraw-Hill, 1926, Vol. III, p. 201.
%
% Description: Vapor pressure of bromine in mm of mercury vs.
% temperature in Centrigrade.
%
% Reference Relation: $\log y =k_{1}/(x+273.1)+k_{2}$
%*********************************************************************

data.case46a = [...
    -95.0   0.0022
    -90.0   0.0052
    -85.0   0.0117
    -80.0   0.0251
    -75.0   0.0513
    -70.0   0.1020
    -65.0   0.1920
    -60.0   0.3570
    -55.0   0.6280
    -50.0   1.0900
    -45.0   1.8300
    -40.0   2.9800
    -35.0   4.7700
    -30.0   7.4500
    -25.0  11.4000
    -20.0  17.1000
    -15.0  25.2000
    -10.0  36.6000];

data.case46a(:,2) = log(data.case46a(:,2));
```

```
%*************************************************************************
% CASE 50:
% Source: {\em Physical Review}, Vol. II, 1913, ``Thermal Conductivity
% of Air at Low Pressures,'' A. Trowbridge, p. 61.
%
% Description: Slope of a line relating temperature rise and the square
% of the current in a heating element vs. pressure in mm of mercury.
%
% Reference Relation: $1/y^{2}=k_{1}x+k_{2}$
%
% Comments: The relation $1/y^{2.25}=k_{1}x+k_{2}$ fits much better.
%*************************************************************************

data.case50 = [...
    0.01  1670
    0.02  1355
    0.03  1155
    0.04  1000
    0.05   893
    0.06   820
    0.07   770
    0.08   735
    0.09   695
    0.10   663
    0.11   629
    0.12   610
    0.13   591
    0.14   571
    0.15   552
    0.16   538
    0.17   523
    0.18   511
    0.19   500
    0.20   485];

data.case50(:,2) = data.case50(:,2)/100;

%*************************************************************************
% CASE 56b:
% Source: National Research Council of the United States of America,
% {\em International Critical Tables of Numerical Data: Physics,
% Chemistry and Technology}, McGraw-Hill, 1926, Vol. VI, p. 55.
%
% Description: Emission of electrons from heated tantulum in
% amps/cm$^{2}$ vs. absolute temperature.
%
% Reference Relation: $y=k_{1}x^{2}e^{-k_{2}/x}$
```

```
%
% Comments: The reference relation is Richardson's equation.
%*************************************************************************

data.case56b = [...
    1000  1.95e-13
    1100  1.71e-11
    1200  7.21e-10
    1300  1.73e-08
    1400  1.23e-07
    1500  2.89e-06
    1600  2.44e-05
    1700  1.51e-04
    1800  7.94e-04
    1900  3.61e-03
    2000  1.38e-02
    2100  4.62e-02
    2200  1.41e-01
    2300  3.92e-01
    2400  1.00e+00
    2500  2.38e+00];

data.case56b(:,2) = log(data.case56b(:,2));

%*************************************************************************
% CASE 105:
% Source: {\em Physical Review}, Vol. VIII, 1916, ``On the
% Demagnetization of Iron and Steel Rods by Strain and Impact,'' Guy G.
% Becknell, p. 515.
%
% Description: Magnetic flux after torsion in cgs units vs. position in
% cm where measurement was taken on a 112 cm rod.
%
% Reference Relation: $(x+k_{1}y)^{2}+k_{2}x+k_{3}y+k_{4}=0$
%
% Comments: The reference relation is a parabola with the axes rotated;
% the tangent of the angle of rotation ($k_{1}$), according to the
% source, is .00147.
%*************************************************************************

data.case105 = [...
    2   553
    8  1754
    14  2655
    20  3361
    26  3905
    32  4308
    38  4599
```

```
    44  4746
    50  4792
    56  4712
    62  4550
    68  4337
    74  4049
    80  3701
    86  3275
    92  2759
    98  2134
    104  1371
    110   392];

data.case105(:,2) = data.case105(:,2)/100;

%***************************************************************************
% CASE 161a:
% Source: {\em Physical Review}, Vol. XX, 1922, ``The Variation of the
% Index of Refraction of Water, Ethyl Alcohol, and Carbon Bisulphide,
% with the Temperature,'' Elmer E. Hall and Arthur R. Payne, p. 257.
%
% Description: Index of refraction of ethyl alcohol, relative to air,
% forr sodium light vs. Centigrade temperature.
%
% Reference Relation: $y=k_{1}x^{3}+k_{2}x^{2}+k_{3}x+k_{4}$
%
% Comments: The source calls the reference relation empirical.  A
% residual plot shows that the data is not linear.
%***************************************************************************

data.case161a = [...
14  1.36290
16  1.36210
18  1.36129
20  1.36048
22  1.35967
24  1.35885
26  1.35803
28  1.35721
30  1.35639
32  1.35557
34  1.35474
36  1.35390
38  1.35306
40  1.35222
42  1.35138
44  1.35054
46  1.34969
```

```
48  1.34885
50  1.34800
52  1.34715
54  1.34629
56  1.34543
58  1.34456
60  1.34368
62  1.34279
64  1.34189
66  1.34096
68  1.34004
70  1.33912
72  1.33820
74  1.33728
76  1.33626];

%**********************************************************************
% CASE 171a:
% Source: {\em Physical Review}, Vol. XXX, 1910, ``Note on the Relation
% between the Temperature and the Resistance of Nickel,'' C. F. Martin,
% p. 523.
%
% Description: Resistance vs. Centigrade temperature.
%
% Reference Relation: $\log y =k_{1}x+k_{2}$
%
% Comments: I have taken only two of three data sets tabulated in the
% source, since Martin says the third fits the reference relation
% poorly.  Note that Martin explictly states that he does not yet have
% enough evidence to claim a general law.  He does seem to believe that
% the reference relation holds for the two data sets I have collected,
% however.  The reference relation shows clear lack of fit.
%**********************************************************************

data.case171a = [...
    -25  11.030
    -20  11.250
    -15  11.475
    -10  11.700
     -5  11.935
      0  12.173
      5  12.420
     10  12.660
     15  12.920
     20  13.173
     25  13.435
     30  13.700
     35  13.965
```

```
40  14.240
45  14.520
50  14.800
55  15.080
60  15.385
65  15.690
70  16.000
75  16.320];
```